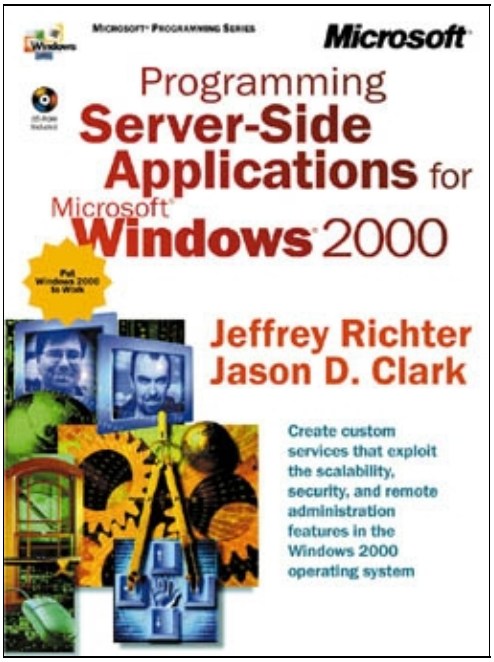
[Next]



[Previous] [Next]

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

Library of Congress Cataloging-in-Publication Data
Richter, Jeffrey.
Programming Server-Side Applications for Microsoft Windows 2000 / Jeffrey Richter,
Jason D. Clark.
p. cm.
Includes index.
ISBN 0-7356-0753-2
1. Microsoft Windows (Computer File) 2. Computer Programming. 3. Application
software--Development. I. Title.

QA76.76.O63 R5453 2000
005.26'8--dc21 99-089908

Printed and bound in the United States of America.

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

Distributed in Canada by Penguin Books Canada Limited.

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

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

Acquisitions Editor: Ben Ryan

Project Editor: Victoria Thulman

Technical Editor: Robert Lyon

[Previous] [Next]

To Kristin,
Nothing means more to me than our journey together. Your support and lovis are the most valuable gifts you share with me. Thanks for accompanying me on my sky car as we head for the highest GPS satellites orbiting above. Nessum will stand between us.
—Jeff

Dearest Annette,
Thank you for filling in the spaces and keeping me well rounded, refreshed, encouraged, and happy while undertaking this enormous task. Thank you for giving me your understanding, your flexibility, your help, and most of all, thank you for giving me your love. Because of you, I feel complete,
—Jason

Dear God,
Thank you for all that I have. I feel blessed.
—Jason

[Previous] [Next]

Introduction

Microsoft Windows 2000 offers many features and subsystems designed specifically to handle an enterprise's mission-critical data-processing needs. These features and subsystems are not available on client operating systems such as Microsoft Windows 98. Only Windows 2000 offers the Service Control Manager (SCM), performance monitoring, event logging, security, asynchronous I/O, and so on. This book describes these features, explains the motivation to use them, and gives you the information you need to best leverage them.

This book does not attempt to explain basic Windows programming and assumes that you are already quite familiar with many Windows topics such as processes, threads, thread synchronization, DLLs, Unicode, structured exception handling, and memory management. If you need a refresher on any of these topics, I encourage you to consult *Programming Applications for Microsoft Windows, Fourth Edition* (Jeffrey Richter, Microsoft Press, 1999). The sample source code in the book you are reading requires you to be well

acquainted with the C++ programming language.

Throughout this book, emphasis is placed on writing high-performance and robust services that are expected to stay running 24 hours a day, 7 days a week. Also, Microsoft is hard at work developing 64-bit Windows 2000. It is expected that many companies will eventually use 64-bit Windows to host their services since this system will offer greater performance and scalability. At the time of this writing, a 64-bit version of Windows has not been released. However, 64-bit Windows has been considered while developing all the source code in this book. The sample applications will build and run with little or no modification on 64-bit Windows once Microsoft makes it available.

[Previous] [Next]

What's in This Book

This book explains the features offered by Windows 2000 that are available to service developers. Here is a partial list of what this book has to offer:

- **Performance and scalability** Throughout the book, programming techniques are discussed that will make your software scale better than the majority of server software running on Windows today. Techniques for improving your device I/O and interthread communication (ITC), which are common scalability bottlenecks, will improve your server's performance and cut costs, making your software more viable and competitive.

- **Security** One section is dedicated entirely to designing service software that takes advantage of the security features of Windows 2000. This section includes an exhaustive treatment of the ins and outs of integrating your service with the security features of the operating system.

- **Kerberos and the SSPI** The security section includes a chapter covering the powerful Kerberos security provider, new to Windows 2000. It also completely describes how to take advantage of it and other security providers such as NT LAN Manager (NTLM) and Secure Sockets Layer (SSL) by using the flexible Security Support Provider Interface (SSPI) functions. This information will help you develop software that communicates in a secure manner in your enterprise's intranet as well as on the global Internet!

- **Securing private objects** Chapter 10, "Access Control," completely covers private object security. The chapter includes text and sample code demonstrating how to use the powerful security features of the operating system to secure custom objects in your software.

- **64-bit Windows readiness** The text addresses 64-bit-specific issues; samples will build with little or no modification on 64-bit Windows (when it becomes available).

- **Practical sample applications** The sample applications on the companion CD describe a wealth of useful programming techniques while providing some very useful tools.

- **Fault tolerance** Unlike other books on programming, which commonly omit error handling from their discussions and code samples, this book focuses on fault tolerance in both the sample code and the chapter text. We have done this because we know that fault tolerance is of critical importance to the service developer.

- **Use of C++** The sample applications use C++ since many readers have requested it. As a result, the sample applications require fewer lines of code and their logic is easier to follow and understand.

- **Reusable code** Whenever possible, we created the source code to be generic and reusable. This should allow you to take individual functions or entire C++ classes and drop them into your own applications with little or no modification. See Appendix B for a brief discussion of some of the classes found in this book.

- **The SuperSCP utility** This utility allows you to explore all the services installed on a local or remote machine. Using this utility, you can also change the configuration of these services, control them, and monitor their execution. In essence, this utility allows you to manipulate a service in every conceivable way allowed by the operating system.

- **The TokenMaster utility** Using this security utility, you can discover and manipulate the user context of processes running on your system. Doing this is very useful for learning the intricacies of security in Windows and for testing security on a system. Having control over the user context can be very helpful to an administrator who is testing different security features of the operating system with the goal of tightening the security on his system.

- **The AccessMaster utility** You can use this utility to modify the access rights on nearly every securable object in the system. Windows ships with editors for file security and registry security, and also with an editor for security on Active Directory. However, AccessMaster allows you to interactively modify security of these objects as well as named pipes, window stations and desktops, synchronization objects, process and threads, and many other objects. This utility can be very useful in learning security in Windows and is a very practical tool.

- **The TrusteeMan utility** This utility allows you to fully administer local user and group trustee accounts on a system, as well as assign and revoke privileges for these accounts.

- **The MsgTableDump utility** This utility allows you to view the message resources in EXE or DLL files, including the system messages found in Kernel32.dll.

- **The Windows 2000 Platform SDK (which includes the WMI SDK)** A complete *x*86 version of the Platform SDK for Windows 2000 is available on the companion CD.

- **The Performance Counter class** This is a C++ class that makes it extremely easy to expose performance counters.

- **Specifications** The Windows 2000 distributed application specification and the Microsoft BackOffice logo specification are included on the companion CD.

- **Windows Installer** Oh yeah, before we forget, the sample applications on the companion CD take advantage of the new Windows Installer built into Windows 2000. The Windows Installer gives you fine control over the parts you want to install and also allows you to easily uninstall the book's sample applications and executable files using the Add/Remove Programs Control Panel applet. Of course, you can always just access the source files and executable files directly from the companion CD if you prefer.

[Previous] [Next]

This Book Has No Mistakes

This section's title clearly states what we want to say. But, of course, we all know that it is a flat-out lie. My editors and I have worked hard to bring you the most accurate, up-to-date, in-depth, easy-to-read, painless-to-understand, bug-free information. Even with the fantastic team assembled, we all know that things slip through the cracks. If you find any mistakes in this book (especially bugs), we would greatly appreciate it

if you would send the mistakes to Jeff via his Web site, *http://www.JeffreyRichter.com*, or to Jason via e-mail at jclark@microsoft.com.

[Previous] [Next]

The CD-ROM and System Requirements

The companion CD contains the source code and executable files for all the sample applications presented in the book. All sample applications were written and compiled with Microsoft Visual C++ 6.0 and the Windows 2000 Platform SDK. Most of the sample applications require features that exist only in Windows 2000; no attempt has been made to build or test the applications on any other version of Windows.

In the root directory of the companion CD you will find the Microsoft Visual Studio workspace file ("Programming Server-Side Apps.dsw") and the common header file ("CmnHdr.h"). Under the root directory is a separate directory for each sample application. The *x*86 directory contains the debug versions of all the sample applications so that you can run them directly from the companion CD.

When you insert the companion CD into the drive, the welcome screen will present itself automatically. If the screen does not appear, go to the drive's Setup directory and execute the StartCD.exe application.

[Previous] [Next]

Support

Microsoft Press provides corrections for this book at the following address:

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

If you have comments, questions, or ideas regarding this book, please send them to Microsoft Press using postal mail or e-mail:

Microsoft Press
Attn: *Programming Server-Side Applications for Microsoft Windows 2000* editor
One Microsoft Way
Redmond, WA 98052-6399
mspinput@microsoft.com

[Previous] [Next]

Thanks for Your Help

We could not have written this book without the help and technical assistance of several people. In particular, we'd like to thank the following people:

- Members of the Microsoft Press editorial team: Carl Diltz, Stephen Guty, Robert Lyon, Joel Panchot, Jocelyn Paul, John Pierce, Ben Ryan, Eric Stroo, Crystal Thomas, and Victoria Thulman.

- Members of the Windows 2000 team: Scott Field, Mark Lucovsky, Michael Parkes, Dmitry Robsman, Jeffrey Saathoff, Jon Schwartz, Rick Vicik, Landy Wang, Brad Waters, and Bob Watson.

- Members of the WMI team: Irena Hudis, Michael Maston, Raymond McCollum, Steve Menzies, Simon Muzio, Lev Novik, Sanj Surati, Patrick Thompson, and Stephen Todd.

- Members of the Developer Support team: Richard Ault, Robin Caron, Frank Kim, David Mowers, Gary Peluso, and the entire Kernel Base and Networking teams. We would especially like to thank Jonathan Russ and Dave McPherson for their tireless help with the samples.

Jason would personally like to thank the following people:

- The Group: You folks keep me entertained and interested while continuing to be wonderfully supportive. I appreciate your friendship: Jelani Alexander, Richard (Wito) Bietz, Tina Fields, Jacob Rogers, Jon Rogers, Stephanie Taitano, and Jon Wiesman.

- The Fam: I am lucky to be blessed with great relations—Duane, Peggy, Andy, Mindy, and Smokey. I am also very happy to be recently accepted into another great family: Jim, Carolyn, and "The Cousins." (Special thanks to Jeff K. for his constant moral support on this project.)

- The One: Annette Lynn Takemoto Clark, you are the reason behind everything I do. Your love and support make everything good. Thank you.

Jeffrey would personally like to thank the following people:

- Members of the Entertainment and Festivities Party: Jeff Cooperstein and Stephanie, Keith Pleas and Susan Wells, Susan Ramee and Sanjeev Surati, Scott Ludwig and Val Horvath and their son, Nicolas, Darrin and Shaula Massena, Neil Fishman and Kristin Palmer, John and Pam Robbins, David Solomon, and Jeff Prosise.

- Members of the Brotherhood: Ron, Maria, Joey (Hoops of Fire), and Brandy Richter.

- Members of the Raising Jeff Squad: Arlene and Sylvan Richter.

- Member of the Fleece Faction: Max.

- Member of the Devotion Division: Kristin Trace.

[Previous] [Next]

Chapter 1

The Discipline of Service Development

We all agree that good programming requires a great deal of talent. Implementing proper error checking, anticipating the resources available on a system, predicting all the potential inputs from various users—this is the work that makes programming an art form. Writing services requires total mastery of this art form.

Neglecting to handle every nuance is not catastrophic for application software; failures usually affect a single user, not the entire enterprise. But *server* software *is* mission critical and therefore requires strict attention to all details. The disciplined developer of servers writes code to address these details. The following sections describe some of the disciplines that a server designer must pay strict attention to.

NOTE

> Throughout this book, I frequently use the terms "server" and "service." When I use the term "server," I am referring to a machine or an application that performs duties for a client. When I use the term "service," I am referring to a special Microsoft Windows application that performs duties for a client but also contains additional infrastructure that enables it to receive special treatment by the operating system. These terms obviously overlap, and I sometime use them interchangeably, but if I am specifically discussing a Windows service, I will use the term "service."

[Previous] [Next]

Fault Tolerance and Tidy Code

Today's software is so complex that anticipating every execution environment is impossible. By "environment" I mean the contents of your process's address space, the values of your function's parameters, and the effects of other processes running on the same system. Because of the complexity, services, which run continuously for months on end, *must* be fault tolerant.

Most of us have had our share of college professors who adamantly expressed their views on how to perform proper error checking and recovery from within our functions. We developers know that writing error-tolerant code is what we should do, but frequently we view the required attention to detail as tedious and so omit it. We've become complacent, thinking that the operating system will "just take care of us." Many developers out there actually believe that memory is endless, and that leaking various resources is OK because they know that the operating system will clean up everything automatically when the process dies.

Certainly many applications are implemented in this way, and the results are not devastating because the applications tend to run for short periods of time and then are restarted. However, services run *forever,* and omitting the proper error-recovery and resource-cleanup code is catastrophic!

In my opinion the only way to write an application capable of running 24 hours a day, seven days a week, is to use exception handling. For this reason, I highly recommend that you become familiar with exception handling techniques. There are two types of exception handling: structured exception handling, a mechanism offered by Windows operating systems; and C++ exception handling, a mechanism offered by the compiler. Both of these mechanisms allow you to write code capable of recovering from unanticipated failures and hard errors such as access violations, divisions by zero, and stack overflows.

Both exception handling mechanisms are useful in different scenarios, and fortunately Microsoft Visual C++ allows us to use them interchangeably within a single application. The sample applications in this book demonstrate the concepts involved in creating robust server applications, and many of them use exception handling liberally. If you desire more information about exception handling, see *Programming Applications for Microsoft Windows, Fourth Edition* (Jeffrey Richter, Microsoft Press, 1999).

Proper use of C++ in a service can make your coding life substantially easier. I find that wrapping simple Windows objects in C++ classes is useful for several reasons, which follow. Many of the sample applications in this book use C++ classes for exactly these reasons.

- By placing the code to close the object in the C++ class's destructor, the compiler ensures that the object is destroyed.
- Wrapping calls to Windows functions inside C++ class methods allows you to enforce the proper calling of Windows functions.
- Calling Windows functions via a C++ class method allows you to place certain checks and verify assertions in one place. This makes it substantially easier to find bugs in your code.

- C++ classes reduce the amount of code you write, making your code more readable and maintainable.
- You get code reuse by using C++ classes. If you use C++ template classes, you can create generic solutions while preserving type safety.

[Previous] [Next]

Scalability and Performance

In the early days of programming, developers had limited system resources. This forced developers to implement crafty and un-maintainable algorithms to eke out as much system performance as possible. Today, computer system performance and storage capabilities have increased enormously, allowing developers to design simpler and more maintainable algorithms.

Unfortunately, these advances have also allowed developers to become lazy. I know developers who don't even think twice about allocating megabytes of storage for tasks that need no more than a few kilobytes, tops. I also know developers who use mutex objects where critical sections would more than suffice. These developers simply don't care that functions that reference a mutex require a user-mode to kernel-mode round-trip transition, which requires about 1000 CPU cycles—and that doesn't even include the code that must execute once in kernel mode. In contrast, critical sections usually stay entirely in user mode and require about 100 CPU cycles to execute.

More and more people are using computers because servers offer the information that improves our quality of living. Studies show that users become easily frustrated with an unresponsive server and seek out desired information elsewhere, which translates into loss of business and revenue. Similarly, an enterprise with an unresponsive server frustrates employees and ultimately affects productivity—this also translates into a loss of business and revenue.

In some situations, you can improve server responsiveness by adding more computers. However, for many reasons you are usually better off running your server on a single machine when you can. First, writing server software that has various parts of itself executing on different machines is much harder than writing software that executes on a single machine. Second, the complexity of administering multiple machines often increases at a greater-than-linear rate. Third, you introduce several more potential points of failure, making errors much more difficult to locate and correct. Using only a single-machine server is not always achievable, but you should keep this ideal in mind while you implement your server's code.

On a single-machine server, performance can be gained by adding RAM, processors, disk storage, network cards, and so on, but only if your server's code is implemented to use these resources in an efficient manner. For example, the performance of a server using a one-thread-per-client connection usually won't double simply by doubling the amount of RAM in the machine. However, if the same server were designed using an efficient thread-pooling algorithm, the server's performance would scale well with additional resources.

Writing highly scalable applications requires a great deal of discipline. At every step, you must consider the ramifications of the code you write:

- Am I writing this code so that a user-mode-to-kernel-mode transition is avoided?
- Am I aligning my memory access so that it doesn't span a cache-line boundary?
- Am I ensuring that my variables are properly aligned?
- Am I replacing processes with threads to reduce the use of system resources?

- Am I reducing the number of runnable threads to avoid wasteful context switching?
- Am I having threads do useful work while waiting for device I/O operations to complete?
- Am I abandoning ANSI strings in favor of Unicode strings to improve the performance of Windows functions?
- Am I exploiting the cool features that Windows has to offer?

I mention these questions here in the hope that you will keep them in mind while you implement your code. Every item is discussed in detail either in *Programming Applications for Microsoft Windows, Fourth Edition,* or in this book.

[Previous] [Next]

Administration

Different organizations have different needs. Software developers try to meet these needs by designing systems and application software that expose myriad configuration settings. These multiple options overwhelm most users and system administrators. The problem is compounded by the fact that most systems consist of many computers networked together. In addition, the person with the expertise to configure a machine is usually not physically sitting in front of it, causing that machine to have prolonged downtime.

Server applications in particular usually run on computers that live in rooms that are basically closets, so your server software should be capable of being remotely administered. In other words, an administrator should be able to have one machine start and stop the execution of your service while she sits at another machine. The administrator should also be able to configure your server's numerous settings remotely.

From a server's perspective, your service might need to communicate information to the administrator. For example, you might want your server to report the number of clients connected and the amount of time required to process the average client request. Your server might also need to report exceptional events such as low disk space or an attempted security breach.

Microsoft has added a lot of infrastructure to Windows to make remote administration easy. The Service Control Manager (or SCM, pronounced "scum") can control a service's execution. The registry is used for storing a server's configuration settings. The performance-monitoring facility allows a server to report run-time statistics, and the event log database allows a server to notify an administrator of exceptional events. All these facilities have programmatic interfaces and can be accessed remotely. The proper use of these facilities will make an administrator's life easier. Each facility is discussed in this book.

The Microsoft Management Console (MMC) is a tool implemented in MMC.exe that provides one-stop shopping for the system administrator. It integrates access into the various components of the system, including installed services. You should also be able to configure your service via the MMC.

[Previous] [Next]

Security

Each day, more and more computers are being connected together, and the benefits are outstanding. Now, computers (and people) can communicate with one another, combining pieces of information from different sources to produce results that were not imaginable just a few years ago. I'm sure that everyone reading this

book believes that sharing information is a good thing for all humankind.

But I'm just as sure we all have some information about ourselves that we'd like to keep private. And we live in a world where some people perform malicious acts such as deleting important pieces of information from disk drives or databases. Even when malice isn't the intent, someone can delete or change some important piece of information by accident. To address security issues, the servers we implement must verify the client making the information request and must ensure that the client is restricted from performing unauthorized operations. If the server software does not handle the security validations correctly, the results are disastrous. Unfortunately, we hear stories about this every day.

Although the need for security has grown over the years, many developers still have the luxury of all but ignoring security. Today, operating systems such as Microsoft Windows 2000 offer a tremendous number of built-in security features, freeing application developers from dealing directly with security issues in their code. The built-in features give the individual user the ability to secure files in ways that were previously available only in server operating environments. Application code needs only to deal gracefully with an "Access Denied" result when attempting to open a secured file.

Whereas this built-in security makes the application developer's life simpler, it does not afford the service developer the same luxury. Your server is a doorway that opens your system to the outside world. If you do not place capable guards at this doorway, you will be making your server software and its data vulnerable to attack. In the worst-case scenario, you might reveal your entire system or local network to the outside world. To guard against attacks, your server software must contain security-conscious code. Fortunately, Windows 2000 offers a sound security architecture and functions that can greatly simplify the work of a server developer.

Windows offers mechanisms that allow your software to detect who is connecting to your server. It also allows your software to set access permissions for data controlled by your service. Once you know which client desires access to what data, Windows can perform an access check to validate this client's request. Using the system's built-in mechanisms makes your development easier, but you still have to intentionally consider security issues when designing your service. In addition to deciding who has access to what, Windows provides features that allow the encryption and decryption of data as it transfers from one machine to another.

Part IV of this book is dedicated to Windows security features. The information in these chapters will explain the Windows security architecture and give you the tools necessary to develop a server that is secure from attacks and human errors.

NOTE

To be eligible for Microsoft's "Certified for Windows 2000 Server" logo and BackOffice Logo programs, your service must support *Single Sign-On* (SSO). Single Sign-On means that the user enters his password just once when he first approaches the machine; the user is never asked for his password again. Once authenticated, all services just use the user's authenticated information. Windows' built-in security mechanisms (in particular, *impersonation*) make it easy for your service to support unified logon. The following Web sites have additional information on Microsoft certification and logo programs:

http://msdn.microsoft.com/certification/
http://msdn.microsoft.com/winlogo/
http://www.microsoft.com/backoffice/designed/

[Previous] [Next]

Why Develop a Service?

When designing your server application, you can implement it as a simple Windows executable in the same way you would implement any application. However, Windows supports a special type of application named a *service*. A service is a normal Windows executable application that contains additional infrastructure. This additional infrastructure allows the executable application to be controlled and monitored using the Service Control Manager (a component built into every machine running Windows).

The SCM allows an administrator to start, stop, pause, and continue a service executable locally or remotely. The SCM also monitors the service and can automatically restart it or reboot the machine if the service terminates unexpectedly. All server applications that ship with Windows are implemented as services. The MMC provides a Services snap-in that allows an administrator to control all the installed services by using a common interface.

For all these reasons, I strongly suggest that you include the additional infrastructure code in your application to turn it into a service. The details of how to do this are described in Chapter 3 of this book. The standard interface that allows you to control the SCM is described in Chapter 4.

> **NOTE**
>
> To be eligible for the Microsoft BackOffice Logo program, your server application must be implemented as a Windows service.

[Previous] [Next]

Network Communication

Many services are stand-alone applications that sit on a single machine and monitor various actions. Here are some services that ship with Windows:

- **Uninterruptible Power Supply** Responds to notifications from a UPS device in the case of a power failure
- **Indexing Service** Monitors changes to files on your hard disk so that it can update a master catalog, allowing for fast searches
- **Windows Installer** Manages the installation, repair, and removal of software on the machine
- **RunAs Service** Allows a user to run an application using a different security context
- **Task Scheduler** Spawns applications at specific times

None of the services just mentioned require any network communication, but many services do. Here are some examples of network-aware services that ship with Windows:

- **Event Log** Sends requested event log records to remote machines, allowing for remote administration
- **NetMeeting Remote Desktop Sharing** Allows remote users to view and control your machine's desktop by using Microsoft NetMeeting
- **Server** Allows remote users to access shared folders, printers, and named pipes

- **Workstation** Allows your machine to remotely connect to another machine's shared folders, printers, and named pipes

If your service requires network communication, you have many mechanisms available to you:

- **Mailslot** Allows one-way interprocess communication (IPC) of untyped data between machines running Windows
- **Named pipe** Allows two-way interprocess communication of untyped data between machines running Windows
- **Socket** Allows two-way interprocess communication of untyped data between machines running any operating system that supports sockets
- **Remote Procedure Call (RPC)** Allows two-way interprocess communication of typed data between machines running any operating system that supports RPC
- **Component Object Model (COM)** Allows two-way interprocess communication of typed data between machines running any operating system that supports COM

All network communication mechanisms can be quite complex, each having its own nuances. In fact, whole books that discuss sockets, RPC, and COM are available. For this reason, I have decided not to address communication mechanisms directly in this book and instead encourage you to seek out other sources for information about them. Of all the mechanisms I mentioned in the preceding list, named pipes are the easiest to understand and use, and so most of the sample applications in this book that require network communication use named pipes.

NOTE

You can also use native network protocols (IPX/SPX/NetBIOS, NetBEUI, TCP/IP, AppleTalk, and so on) as your communication mechanism. However, Microsoft strongly discourages this because it ties your application to a specific protocol, whereas the other higher level mechanisms (mailslots, pipes, sockets, RPC, and COM) work over all the native protocols, allowing your application to run in more varied environments.

[Previous] [Next]

Chapter 2

Device I/O and Interthread Communication

I can't stress enough the importance of this chapter, which covers two topics that are essential when implementing high-performance, scalable applications: device I/O and interthread communication. A scalable application handles a large number of concurrent operations as efficiently as it handles a small number of concurrent operations. For a service application, typically these operations are processing client requests that arrive at unpredictable times and require an unpredictable amount of processing power. These requests usually arrive from I/O devices such as network adapters; processing the requests frequently requires additional I/O devices such as disk files.

In Microsoft Windows applications, threads are the best facility available to help you partition work. Each thread is assigned to a processor, which allows a multiprocessor machine to execute multiple operations simultaneously, increasing throughput. When a thread issues a device I/O request, the thread is temporarily suspended until the device completes the I/O request. This suspension hurts performance because the thread is

unable to do useful work such as initiate another client's request for processing. So, in short, you want to keep your threads doing useful work all the time.

To help keep threads busy, you will need to make your threads communicate with one another about the operations they will perform. Microsoft has spent years researching and testing in this area and has developed a finely tuned mechanism to create this communication. This mechanism, called the *I/O completion port,* can help you create high-performance, scalable applications. By using the I/O completion port, you can make your application's threads achieve phenomenal throughput by reading and writing to devices without waiting for the devices to respond.

The I/O completion port was originally designed to handle device I/O, but over the years, Microsoft has architected more and more operating system facilities that fit seamlessly into the I/O completion port model. One example is the new job kernel object introduced in Microsoft Windows 2000: as a job object monitors its processes, it sends event notifications to an I/O completion port. The JobLab sample application, which can be found in *Programming Applications for Microsoft Windows, Fourth Edition* (Jeffrey Richter, Microsoft Press, 1999), demonstrates how I/O completion ports and job objects work together.

Throughout my many years as a Windows developer, I have found more and more uses for the I/O completion port, and I feel that every Windows developer must fully understand how the I/O completion port works. Many of the sample applications in this book use the I/O completion port. Even though I present the I/O completion port in this chapter about device I/O, be aware that the I/O completion port doesn't have to be used with device I/O at all—simply put, it is an awesome interthread communication mechanism with an infinite number of uses.

From this fanfare, you can probably tell that I'm a huge fan of the I/O completion port. My hope is that by the end of this chapter, you will be too. But instead of jumping right into the details of the I/O completion port, I'm going to explain what Windows originally offered developers for interthread communication and device I/O. This will give you a much greater appreciation for the I/O completion port. Toward the end of the chapter, in the section "I/O Completion Ports," I'll discuss the I/O completion port.

[Previous] [Next]

Opening and Closing Devices

One of the strengths of Windows is the sheer number of devices that it supports. In the context of this discussion, I define a device to be anything that allows communication. Table 2-1 lists some devices and their most common uses.

Table 2-1. *Various devices and their common uses*

Device	Most Common Use
File	Persistent storage of arbitrary data
Directory	Attribute and file compression settings
Logical disk drive	Drive formatting
Physical disk drive	Partition table access
Serial port	Data transmission over a phone line

Parallel port	Data transmission to a printer
Mailslot	One-to-many transmission of data, usually over a network to a machine running Windows
Named pipe	One-to-one transmission of data, usually over a network to a machine running Windows
Anonymous pipe	One-to-one transmission of data on a single machine (never over the network)
Socket	Datagram or stream transmission of data, usually over a network to any machine supporting sockets (The machine need not be running Windows.)
Console	A text window screen buffer

This chapter discusses how an application's threads communicate with these devices without waiting for the devices to respond. Windows tries to hide device differences from the software developer as much as possible. That is, once you open a device, the Windows functions that allow you to read and write data to the device are the same no matter what device you are communicating with. Although only a few functions are available for reading and writing data regardless of the device, devices are certainly different from one another. For example, it makes sense to set a baud rate for a serial port, but a baud rate has no meaning when using a named pipe to communicate over a network (or over the local machine). Devices are subtly different from one another, and I will not attempt to address all their nuances. However, I will spend some time addressing files because files are so common.

To perform any type of I/O, you must first open the desired device and get a handle to it. The way you get the handle to a device depends on the particular device. Table 2-2 lists various devices and the functions you should call to open them.

Table 2-2. *Functions for opening various devices*

Device	Function Used to Open the Device
File	*CreateFile* (*pszName* is path name or UNC path name).
Directory	*CreateFile* (*pszName*is directory name or UNC directory name). Windows 2000 allows you to open a directory if you specify the FILE_FLAG_BACKUP_SEMANTICS flag in the call to *CreateFile*. Opening the directory allows you to change the directory's attributes (to normal, hidden, and so on) and its time stamp.
Logical disk drive	*CreateFile* (*pszName*is "\\.*x*:"). Windows 2000 allows you to open a logical drive if you specify a string in the form of "\\.*x*:" where *x* is a drive letter. For example, to open drive A, you specify "\\.\A:". Opening a drive allows you to format the drive or determine the media size of the drive.
Physical disk drive	*CreateFile* (*pszName* is "\\.\PHYSICALDRIVE*x*"). Windows 2000 allows you to open a physical drive if you specify a string in the form of "\\.\PHYSICALDRIVE*x*" where *x* is a physical drive number. For example, to read or write to physical sectors on the user's first physical hard disk, you specify "\\.\PHYSICALDRIVE0". Opening a physical drive allows you to access the hard drive's partition tables directly. Opening the physical drive is potentially dangerous; an incorrect write to the

	drive could make the disk's contents inaccessible by the operating system's file system.
Serial port	*CreateFile* (*pszName*is "COM*x*").
Parallel port	*CreateFile* (*pszName*is "LPT*x*").
Mailslot server	*CreateMailslot*(*pszName* is "\\.\mailslot*mailslotname*").
Mailslot client	*CreateFile* (*pszName*is "*servername*\mailslot*mailslotname* ").
Named pipe server	*CreateNamedPipe*(*pszName* is "\\.\pipe*pipename*").
Named pipe client	*CreateFile* (*pszName*is "*servername*\pipe*pipename*").
Anonymous pipe	*CreatePipe* client and server.
Socket	*socket*, *accept*, or *AcceptEx*.
Console	*CreateConsoleScreenBuffer* or *GetStdHandle*.

Each function in Table 2-2 returns a handle that identifies the device. You can pass the handle to various functions to communicate with the device. For example, you call *SetCommConfig* to set the baud rate of a serial port:

```
BOOL SetCommConfig(
   HANDLE       hCommDev,
   LPCOMMCONFIG pCC,
   DWORD        dwSize);
```

And you use *SetMailslotInfo* to set the time-out value when waiting to read data:

```
BOOL SetMailslotInfo(
   HANDLE hMailslot,
   DWORD  dwReadTimeout);
```

As you can see, these functions require a handle to a device for their first argument.

When you are finished manipulating a device, you must close it. For most devices, you do this by calling the very popular *CloseHandle* function:

```
BOOL CloseHandle(HANDLE hObject);
```

However, if the device is a socket, you must call *closesocket* instead:

```
int closesocket(SOCKET s);
```

Also, if you have a handle to a device, you can find out what type of device it is by calling *GetFileType*:

```
DWORD GetFileType(HANDLE hDevice);
```

All you do is pass to the *GetFileType* function the handle to a device, and the function returns one of the values listed in Table 2-3.

Table 2-3. *Values returned by the* GetFileType *function*

Value	Description
FILE_TYPE_UNKNOWN	The type of the specified file is unknown.
FILE_TYPE_DISK	The specified file is a disk file.

FILE_TYPE_CHAR	The specified file is a character file, typically an LPT device or a console.
FILE_TYPE_PIPE	The specified file is either a named or an anonymous pipe.

A Detailed Look at *CreateFile*

The *CreateFile* function, of course, creates and opens disk files, but don't let the name fool you—it opens lots of other devices as well:

```
HANDLE CreateFile(
   PCTSTR pszName,
   DWORD  dwDesiredAccess,
   DWORD  dwShareMode,
   PSECURITY_ATTRIBUTES psa,
   DWORD  dwCreationDistribution,
   DWORD  dwFlagsAndAttrs,
   HANDLE hfileTemplate);
```

As you can see, *CreateFile* requires quite a few parameters, allowing for a great deal of flexibility when opening a device. At this point, I'll discuss all these parameters in detail.

When you call *CreateFile*, the *pszName* parameter identifies the device type as well as a specific instance of the device.

The *dwDesiredAccess* parameter specifies how you want to transmit data to and from the device. You can pass four possible values, which are described in Table 2-4.

Table 2-4. *Values that can be passed for* CreateFile's dwDesiredAccess *parameter*

Value	Meaning
0	You do not intend to read or write data to the device. Pass 0 when you just want to change the device's configuration settings—for example, if you want to change only a file's time stamp.
GENERIC_READ	Allows read-only access from the device.
GENERIC_WRITE	Allows write-only access to the device. For example, this value can be used to send data to a printer and by backup software. Note that GENERIC_WRITE does not imply GENERIC_READ.
GENERIC_READ | GENERIC_WRITE	Allows both read and write access to the device. This value is the most common since it allows the free exchange of data.

The *dwShareMode* parameter specifies device-sharing privileges. It is likely that a single device can and will be accessed by several computers at the same time (in a networking environment) or by several processes at the same time (in a multithreaded environment). The potential for device sharing means that you must think about whether you should and how you will restrict other computers or processes from accessing the device's data. Table 2-5 describes the possible values that can be passed for the *dwShareMode* parameter.

Table 2-5. *Values related to I/O that can be passed for* CreateFile's dwShareMode *parameter*

Value	Meaning
0	You require that no other process is reading or writing to the device. If another process has opened the device, your call to *CreateFile* fails. If you

	successfully open the device, another process's call to *CreateFile* always fails.
FILE_SHARE_READ	You require that no other process is writing to the device. If another process has opened the device for write or exclusive access, your call to *CreateFile* fails. If you successfully open the device, another process's call to *CreateFile* fails if GENERIC_WRITE access is requested.
FILE_SHARE_WRITE	You require that no other process is reading from the device. If another process has opened the device for read or exclusive access, your call to *CreateFile* fails. If you successfully open the device, another process's call to *CreateFile* fails if GENERIC_READ access is requested.
FILE_SHARE_READ | FILE_SHARE_WRITE	You don't care if another process is reading from or writing to the device. If another process has opened the device for exclusive access, your call to *CreateFile* fails. If you successfully open the device, another process's call to *CreateFile* fails when exclusive read, exclusive write, or exclusive read/write access is requested.

NOTE

If you are opening a file, you can pass a pathname that is up to _MAX_PATH (defined as 260) characters long. However, you can transcend this limit by calling *CreateFileW* (the Unicode version of *CreateFile*) and precede the pathname with "\\?\". Calling *CreateFileW* removes the prefix and allows you to pass a path that is almost 32,000 Unicode characters long. Remember, however, that you must use fully qualified paths when using this prefix; the system does not process relative directories such as "." and "..". Also, each individual component of the path is still limited to _MAX_PATH characters.

The *psa* parameter points to a SECURITY_ATTRIBUTES structure that allows you to specify security information and whether or not you'd like *CreateFile*'s returned handle to be inheritable. The security descriptor inside this structure is used only if you are creating a file on a secure file system such as NTFS; the security descriptor is ignored in all other cases. Usually, you just pass NULL for the *psa* parameter, indicating that the file is created with default security and that the returned handle is noninheritable.

The *dwCreationDistribution* parameter is most meaningful when *CreateFile* is being called to open a file as opposed to another type of device. Table 2-6 lists the possible values that you can pass for this parameter.

Table 2-6. *Values that can be passed for* CreateFile*'s* dwCreationDistribution *parameter*

Value	Meaning
CREATE_NEW	Tells *CreateFile* to create a new file and to fail if a file with the same name already exists.
CREATE_ALWAYS	Tells *CreateFile* to create a new file regardless of whether a file with the same name already exists. If a file with the same name already exists, *CreateFile* overwrites the existing file.
OPEN_EXISTING	Tells *CreateFile* to open an existing file or device and to fail if the file or device doesn't exist.
OPEN_ALWAYS	Tells *CreateFile* to open the file if it exists and to create a new file if it doesn't exist.
TRUNCATE_EXISTING	Tells *CreateFile* to open an existing file, truncate its size to 0 bytes, and fail if the file doesn't already exist.

NOTE

When you are calling *CreateFile* to open a device other than a file, you must pass OPEN_EXISTING for the *dwCreationDistribution* parameter.

CreateFile's *dwFlagsAndAttrs* parameter has two purposes: it allows you to set flags that fine-tune the communication with the device, and if the device is a file, you also get to set the file's attributes. Most of these communication flags are signals that tell the system how you intend to access the device. The system can then optimize its caching algorithms to help your application work more efficiently. I'll describe the communication flags first and then discuss the file attributes.

CreateFile Cache Flags

FILE_FLAG_NO_BUFFERING This flag indicates not to use any data buffering when accessing a file. To improve performance, the system caches data to and from disk drives. Normally you do not specify this flag, and the cache manager keeps recently accessed portions of the file system in memory. This way, if you read a couple of bytes from a file and then read a few more bytes, the file's data is most likely loaded in memory, and the disk has to be accessed only once instead of twice, greatly improving performance. However, this process does mean that portions of the file's data are in memory twice: the cache manager has a buffer, and you called some function (such as *ReadFile*) that copied some of the data from the cache manager's buffer into your own buffer.

When the cache manager is buffering data, it might also read ahead so that the next bytes you're likely to read are already in memory. Again, speed is improved by reading more bytes than necessary from the file. Memory is potentially wasted if you never attempt to read further in the file. (See the FILE_FLAG_SEQUENTIAL_SCAN and FILE_FLAG_RANDOM_ACCESS flags, discussed next, for more about reading ahead.)

By specifying the FILE_FLAG_NO_BUFFERING flag, you tell the cache manager that you do not want it to buffer any data—you take on this responsibility yourself! Depending on what you're doing, this flag can improve your application's speed and memory usage. Because the file system's device driver is writing the file's data directly into the buffers that you supply, you must follow certain rules:

- You must always access the file by using offsets that are exact multiples of the disk volume's sector size. (Use the *GetDiskFreeSpace* function to determine the disk volume's sector size.)
- You must always read/write a number of bytes that is an exact multiple of the sector size.
- You must make sure that the buffer in your process's address space begins on an address that is integrally divisible by the sector size.

FILE_FLAG_SEQUENTIAL_SCAN and FILE_FLAG_RANDOM_ACCESS These flags are useful only if you allow the system to buffer the file data for you. If you specify the FILE_FLAG_NO_BUFFERING flag, both of these flags are ignored.

If you specify the FILE_FLAG_SEQUENTIAL_SCAN flag, the system thinks you are accessing the file sequentially. When you read some data from the file, the system will actually read more of the file's data than the amount you requested. This process reduces the number of hits to the hard disk and improves the speed of your application. If you perform any direct seeks on the file, the system has spent a little extra time and memory caching data that you are not accessing. This is perfectly OK, but if you do it often, you'd be better off specifying the FILE_FLAG_RANDOM_ACCESS flag. This flag tells the system not to pre-read file data.

To manage a file, the cache manager must maintain some internal data structures for the file—the larger the file, the more data structures required. When working with extremely large files, the cache manager might not be able to allocate the internal data structures it requires and will fail to open the file. To access extremely large files, you must open the file using the FILE_ FLAG_NO_BUFFERING flag.

FILE_FLAG_WRITE_THROUGH This is the last cache-related flag. It disables intermediate caching of file-write operations to reduce the potential for data loss. When you specify this flag, the system writes all file modifications directly to the disk. However, the system still maintains an internal cache of the file's data, and file-read operations use the cached data (if available) instead of reading data directly from the disk. When this flag is used to open a file on a network server, the Windows file-write functions do not return to the calling thread until the data is written to the server's disk drive.

That's it for the buffer-related communication flags. Now let's discuss the remaining communication flags.

Miscellaneous *CreateFile* Flags

FILE_FLAG_DELETE_ON_CLOSE Use this flag to have the file system delete the file after all handles to it are closed. This flag is most frequently used with the FILE_ATTRIBUTE_TEMPORARY attribute. When these two flags are used together, your application can create a temporary file, write to it, read from it, and close it. When the file is closed, the system automatically deletes the file—what a convenience!

FILE_FLAG_BACKUP_SEMANTICS Use this flag in backup and restore software. Before opening or creating any files, the system normally performs security checks to be sure that the process trying to open or create a file has the requisite access privileges. However, backup and restore software is special in that it can override certain file security checks. When you specify the FILE_FLAG_ BACKUP_SEMANTICS flag, the system checks the caller's access token to see whether the Backup/Restore File and Directories privileges are enabled. If the appropriate privileges are enabled, the system allows the file to be opened. You can also use the FILE_FLAG_BACKUP_SEMANTICS flag to open a handle to a directory.

FILE_FLAG_POSIX_SEMANTICS In Windows, filenames are case-preserved whereas filename searches are case-insensitive. However, the POSIX subsystem requires that filename searches be case-sensitive. The FILE_FLAG_ POSIX_SEMANTICS flag causes *CreateFile* to use a case-sensitive filename search when creating or opening a file. Use the FILE_FLAG_POSIX_ SEMANTICS flag with extreme caution—if you use it when you create a file, that file might not be accessible to Windows applications.

FILE_FLAG_OPEN_REPARSE_POINT In my opinion, this flag should have been called FILE_FLAG_IGNORE_REPARSE_POINT since it tells the system to ignore the file's reparse attribute (if it exists). Reparse attributes allow a file system filter to modify the behavior of opening, reading, writing, and closing a file. Usually, the modified behavior is desired, so using the FILE_FLAG_OPEN_REPARSE_POINT flag is not recommended.

FILE_FLAG_OPEN_NO_RECALL This flag tells the system not to restore a file's contents from offline storage (such as tape) back to online storage (such as a hard disk). When files are not accessed for long periods of time, the system can transfer the file's contents to offline storage, freeing up hard disk space. When the system does this, the file on the hard disk is not destroyed; only the data in the file is destroyed. When the file is opened, the system automatically restores the data from offline storage. The FILE_FLAG_OPEN_NO_RECALL flag instructs the system not to restore the data and causes I/O operations to be performed against the offline storage medium.

FILE_FLAG_OVERLAPPED This flag tells the system that you want to access a device asynchronously. You'll notice that the default way of opening a device is synchronous I/O (not specifying FILE_FLAG_OVERLAPPED). Synchronous I/O is what most developers are used to. When you read data from a file, your thread is suspended, waiting for the information to be read. Once the information has been

read, the thread regains control and continues executing.

Because device I/O is slow when compared with most other operations, you might want to consider communicating with some devices asynchronously. Here's how it works: Basically, you call a function to tell the operating system to read or write data, but instead of waiting for the I/O to complete, your call returns immediately, and the operating system completes the I/O on your behalf using its own threads. When the operating system has finished performing your requested I/O, you can be notified. Asynchronous I/O is the key to creating high-performance service applications. Windows offers several different methods of asynchronous I/O, all of which are discussed in this chapter.

File Attribute Flags

Now it's time to examine the attribute flags for *CreateFile*'s *dwFlagsAndAttrs* parameter, described in Table 2-7. These flags are completely ignored by the system unless you are creating a brand new file and you pass NULL for *CreateFile*'s *hfileTemplate* parameter. Most of the attributes should already be familiar to you.

Table 2-7. *File attribute flags that can be passed for* CreateFile'*s* dwFlagsAndAttrs *parameter*

Flag	Meaning
FILE_ATTRIBUTE_ARCHIVE	The file is an archive file. Applications use this flag to mark files for backup or removal. When *CreateFile* creates a new file, this flag is automatically set.
FILE_ATTRIBUTE_ENCRYPTED	The file is encrypted.
FILE_ATTRIBUTE_HIDDEN	The file is hidden. It won't be included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL	The file has no other attributes set. This attribute is valid only when it's used alone.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	The file will not be indexed by the content indexing service.
FILE_ATTRIBUTE_OFFLINE	The file exists, but its data has been moved to offline storage. This flag is useful for hierarchical storage systems.
FILE_ATTRIBUTE_READONLY	The file is read-only. Applications can read the file but can't write to it or delete it.
FILE_ATTRIBUTE_SYSTEM	The file is part of the operating system or is used exclusively by the operating system.
FILE_ATTRIBUTE_TEMPORARY	The file's data will be used only for a short time. The file system tries to keep the file's data in RAM rather than on disk to keep the access time to a minimum.

Use FILE_ATTRIBUTE_TEMPORARY if you are creating a temporary file. When *CreateFile* creates a file with the temporary attribute, *CreateFile* tries to keep the file's data in memory instead of on the disk. This makes accessing the file's contents much faster. If you keep writing to the file and the system can no longer keep the data in RAM, the operating system will be forced to start writing the data to the hard disk. You can improve the system's performance by combining the FILE_ATTRIBUTE_TEMPORARY flag with the FILE_FLAG_DELETE_ON_CLOSE flag (discussed earlier). Normally, the system flushes a file's cached data when

the file is closed. However, if the system sees that the file is to be deleted when it is closed, the system doesn't need to flush the file's cached data.

In addition to all these communication and attribute flags, a number of flags allow you to control the security quality of service when opening a named-pipe device. Since these flags are specific to named pipes only, I will not discuss them here. To learn about them, please read about the *CreateFile* function in the Platform SDK documentation.

CreateFile's last parameter, *hfileTemplate*, identifies the handle of an open file or is NULL. If *hfileTemplate* identifies a file handle, *CreateFile* ignores the attribute flags in the *dwFlagsAndAttrs* parameter completely and uses the attributes associated with the file identified by *hfileTemplate*. The file identified by *hfileTemplate* must have been opened with the GENERIC_READ flag for this to work. If *CreateFile* is opening an existing file (as opposed to creating a new file), the *hfileTemplate* parameter is ignored.

If *CreateFile* succeeds in creating or opening a file or device, the handle of the file or device is returned. If *CreateFile* fails, INVALID_HANDLE_VALUE is returned.

NOTE

Most Windows functions that return a handle return NULL when the function fails. However, *CreateFile* returns INVALID_HANDLE_ VALUE (defined as -1) instead. I have often seen code like this, which is incorrect:

```
HANDLE hfile = CreateFile(...);
if (hfile == NULL) {
   // We'll never get in here
} else {
   // File may or may not be created OK
}
```

Here's the correct way to check for an invalid file handle:

```
HANDLE hfile = CreateFile(...);
if (hfile == INVALID_HANDLE_VALUE) {
   // File not created
} else {
   // File created OK
}
```

[Previous] [Next]

Working with File Devices

Because working with files is so common, I want to spend some time addressing issues that apply specifically to file devices. This section shows how to position a file's pointer and change a file's size.

The first issue you must be aware of is that Windows was designed to work with extremely large files. Instead of representing a file's size using 32-bit values, the original Microsoft designers chose to use 64-bit values. This means that theoretically a file can reach a size of 16 EB (exabytes).

Dealing with 64-bit values in a 32-bit operating system makes working with files a little unpleasant because a lot of Windows functions require you to pass a 64-bit value as two separate 32-bit values. But as you'll see, working with the values is not too difficult and, in normal day-to-day operations, you probably won't need to

work with a file greater than 4 GB. This means that the high 32 bits of the file's 64-bit size will frequently be 0 anyway.

Getting a File's Size

When working with files, quite often you will need to acquire the file's size. The easiest way to do this is by calling *GetFileSizeEx*:

```
BOOL GetFileSizeEx(
   HANDLE         hfile,
   PLARGE_INTEGER pliFileSize);
```

The first parameter, *hfile*, is the handle of an opened file, and the *pliFileSize* parameter is the address of a LARGE_INTEGER union. This union allows a 64-bit signed value to be referenced as two 32-bit values or as a single 64-bit value, and it can be quite convenient when working with file sizes and offsets. Here is (basically) what the union looks like:

```
typedef union _LARGE_INTEGER {
   struct {
      DWORD LowPart;    // Low  32-bit unsigned value
      LONG HighPart;    // High 32-bit signed value
   };
   LONGLONG QuadPart;   // Full 64-bit signed value
} LARGE_INTEGER, *PLARGE_INTEGER;
```

In addition to LARGE_INTEGER, there is a ULARGE_INTEGER structure representing an unsigned 64-bit value:

```
typedef union _ULARGE_INTEGER {
   struct {
      DWORD LowPart;     // Low  32-bit unsigned value
      DWORD HighPart;    // High 32-bit unsigned value
   };
   ULONGLONG QuadPart;   // Full 64-bit unsigned value
} ULARGE_INTEGER, *PULARGE_INTEGER;
```

Another very useful function for getting a file's size is *GetCompressedFileSize*:

```
DWORD GetCompressedFileSize(
   PCTSTR pszFileName,
   PDWORD pdwFileSizeHigh);
```

This function returns the file's physical size, whereas *GetFileSizeEx* returns the file's logical size. For example, consider a 100-KB file that has been compressed to occupy 85 KB. Calling *GetFileSizeEx* returns the logical size of the file—100 KB—whereas *GetCompressedFileSize* returns the actual number of bytes on disk occupied by the file—85 KB.

Unlike *GetFileSizeEx*, *GetCompressedFileSize* takes a filename passed as a string instead of taking a handle for the first parameter. The *GetCompressedFileSize* function returns the 64-bit size of the file in an unusual way: the low 32 bits of the file's size are the function's return value. The high 32 bits of the file's size are placed in the DWORD pointed to by the *pdwFileSizeHigh* parameter. Here the use of the ULARGE_INTEGER structure comes in handy:

```
ULARGE_INTEGER ulFileSize;
ulFileSize.LowPart = GetCompressedFileSize("SomeFile.dat",
   &ulFileSize.HighPart);

// 64-bit file size is now in ulFileSize.QuadPart
```

Positioning a File Pointer

Calling *CreateFile* causes the system to create a file kernel object that manages operations on the file. Inside this kernel object is a file pointer. This file pointer indicates the 64-bit offset within the file where the next synchronous read or write should be performed. Initially, this file pointer is set to 0, so if you call *ReadFile* immediately after a call to *CreateFile*, you will start reading the file from offset 0. If you read 10 bytes of the file into memory, the system updates the pointer associated with the file handle so that the next call to *ReadFile* starts reading at the tenth byte in the file. For example, look at this code, in which the first 10 bytes from the file are read into the buffer, and then the next 10 bytes are read into the buffer:

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hfile = CreateFile("MyFile.dat", ...); // Pointer set to 0
ReadFile(hfile, pb, 10, &dwNumBytes, NULL);   // Reads bytes  0 - 9
ReadFile(hfile, pb, 10, &dwNumBytes, NULL);   // Reads bytes 10 - 19
```

Because each file kernel object has its own file pointer, opening the same file twice gives slightly different results:

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hfile1 = CreateFile("MyFile.dat", ...); // Pointer set to 0
HANDLE hfile2 = CreateFile("MyFile.dat", ...); // Pointer set to 0
ReadFile(hfile1, pb, 10, &dwNumBytes, NULL);   // Reads bytes 0 - 9
ReadFile(hfile2, pb, 10, &dwNumBytes, NULL);   // Reads bytes 0 - 9
```

In this example, two different kernel objects manage the same file. Since each kernel object has its own file pointer, manipulating the file with one file object has no effect on the file pointer maintained by the other object, and the first 10 bytes of the file are read twice.

I think one more example will help make all this clear:

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hfile1 = CreateFile("MyFile.dat", ...); // Pointer set to 0
HANDLE hfile2;
DuplicateHandle(
   GetCurrentProcess(), hfile1,
   GetCurrentProcess(), &hfile2,
   0, FALSE, DUPLICATE_SAME_ACCESS);
ReadFile(hfile1, pb, 10, &dwNumBytes, NULL);   // Reads bytes  0 - 9
ReadFile(hfile2, pb, 10, &dwNumBytes, NULL);   // Reads bytes 10 - 19
```

In this example, one file kernel object is referenced by two file handles. Regardless of which handle is used to manipulate the file, the one file pointer is updated. As in the first example, different bytes are read each time.

If you need to access a file randomly, you will need to alter the file pointer associated with the file's kernel object. You do this by calling *SetFilePointerEx*:

```
BOOL SetFilePointerEx(
   HANDLE         hfile,
   LARGE_INTEGER  liDistanceToMove,
   PLARGE_INTEGER pliNewFilePointer,
   DWORD          dwMoveMethod);
```

The *hfile* parameter identifies the file kernel object whose file pointer you wish to change. The *liDistanceToMove* parameter tells the system by how many bytes you want to move the pointer. The number you specify is added to the current value of the file's pointer, so a negative number has the effect of stepping backward in the file. The last parameter of *SetFilePointerEx*, *dwMoveMethod*, tells *SetFilePointerEx* how to

interpret the *liDistanceToMove* parameter. Table 2-8 describes the three possible values you can pass via *dwMoveMethod* to specify the starting point for the move.

Table 2-8. *Values that can be passed for* SetFilePointerEx'*s* dwMoveMethod *parameter*

Value	Meaning
FILE_BEGIN	The file object's file pointer is set to the value specified by the *liDistanceToMove* parameter. Note that *liDistanceToMove* is interpreted as an unsigned 64-bit value.
FILE_CURRENT	The file object's file pointer has the value of *liDistanceToMove* added to it. Note that *liDistanceToMove* is interpreted as a signed 64-bit value, allowing you to seek backward in the file.
FILE_END	The file object's file pointer is set to the logical file size plus *liDistanceToMove* parameter. Note that *liDistanceToMove* is interpreted as a signed 64-bit value, allowing you to seek backward in the file.

After *SetFilePointerEx* has updated the file object's file pointer, the new value of the file pointer is returned in the LARGE_INTEGER pointed to by the *pliNewFilePointer* parameter. You can pass NULL for *pliNewFilePointer* if you're not interested in the new pointer value.

Here are a few facts to note about *SetFilePointerEx*:

- Setting a file's pointer beyond the end of the file's current size is legal. Doing so does not actually increase the size of the file on disk unless you write to the file at this position or call *SetEndOfFile*.
- When using *SetFilePointerEx* with a file opened with FILE_FLAG_NO_BUFFERING, the file pointer can be positioned only on sector-aligned boundaries. The FileCopy sample application later in this chapter demonstrates how to do this properly.
- Windows does not offer a *GetFilePointerEx* function, but you can use *SetFilePointerEx* to get the desired effect:

```
LARGE_INTEGER liCurrentPosition = { 0 };
SetFilePointerEx(hfile, liCurrentPosition, &liCurrentPosition,
   FILE_CURRENT);
```

Setting the End of a File

Usually, the system takes care of setting the end of a file when the file is closed. However, you might sometimes want to force a file to be smaller or larger. On those occasions, call

```
BOOL SetEndOfFile(HANDLE hfile);
```

This *SetEndOfFile* function truncates or extends a file's size to the size indicated by the file object's file pointer. For example, if you wanted to force a file to be 1024 bytes long, you'd use *SetEndOfFile* this way:

```
HANDLE hfile = CreateFile(...);
LARGE_INTEGER liDistanceToMove;
liDistanceToMove.QuadPart = 1024;
SetFilePointerEx(hfile, liDistanceToMove, NULL, FILE_BEGIN);
SetEndOfFile(hfile);
CloseHandle(hfile);
```

Using Windows Explorer to examine the properties of this file reveals that the file is exactly 1024 bytes long.

[Previous] [Next]

Performing Synchronous Device I/O

This section discusses the Windows functions that allow you to perform synchronous device I/O. Keep in mind that a device can be a file, mailslot, pipe, socket, and so on. No matter which device is used, the I/O is performed using the same functions.

Without a doubt, the easiest and most commonly used functions for reading from and writing to devices are *ReadFile* and *WriteFile*:

```
BOOL ReadFile(
   HANDLE      hfile,
   PVOID       pvBuffer,
   DWORD       nNumBytesToRead,
   PDWORD      pdwNumBytes,
   OVERLAPPED* pOverlapped);

BOOL WriteFile(
   HANDLE      hfile,
   CONST VOID  *pvBuffer,
   DWORD       nNumBytesToWrite,
   PDWORD      pdwNumBytes,
   OVERLAPPED* pOverlapped);
```

The *hfile* parameter identifies the handle of the device you want to access. When the device is opened, you must not specify the FILE_FLAG_ OVERLAPPED flag, or the system will think that you want to perform asynchronous I/O with the device. The *pvBuffer* parameter points to the buffer to which the device's data should be read or to the buffer containing the data that should be written to the device. The *nNumBytesToRead* and *nNumBytesToWrite* parameters tell *ReadFile* and *WriteFile* how many bytes to read from the device and how many bytes to write to the device, respectively.

The *pdwNumBytes* parameters indicate the address of a DWORD that the functions fill with the number of bytes successfully transmitted to and from the device. The last parameter, *pOverlapped*, should be NULL when performing synchronous I/O. You'll examine this parameter in more detail shortly when asynchronous I/O is discussed.

Both *ReadFile* and *WriteFile* return TRUE if successful. By the way, *ReadFile* can be called only for devices that were opened with the GENERIC_ READ flag. Likewise, *WriteFile* can be called only when the device is opened with the GENERIC_WRITE flag.

Flushing Data to the Device

Remember from our look at the *CreateFile* function that you can pass quite a few flags to alter the way in which the system caches file data. Some other devices, such as serial ports, mailslots, and pipes, also cache data. If you want to force the system to write cached data to the device, you can call *FlushFileBuffers*:

```
BOOL FlushFileBuffers(HANDLE hfile);
```

The *FlushFileBuffers* function forces all the buffered data associated with a device that is identified by the *hfile* parameter to be written. For this to work, the device has to be opened with the GENERIC_WRITE flag. If the function is successful, TRUE is returned.

[Previous] [Next]

Basics of Asynchronous Device I/O

Compared to most other operations carried out by a computer, device I/O is one of the slowest and most unpredictable. The CPU performs arithmetic operations and even paints the screen much faster than it reads data from or writes data to a file or across a network. However, using asynchronous device I/O enables you to better utilize resources and thus create more efficient applications.

Consider a thread that issues an asynchronous I/O request to a device. This I/O request is passed to a device driver, which assumes the responsibility of actually performing the I/O. While the device driver waits for the device to respond, the application's thread is not suspended as it waits for the I/O request to complete. Instead, this thread continues executing and performs other useful tasks.

At some point, the device driver finishes processing the queued I/O request and must notify the application that data has been sent, data has been received, or an error has occurred. You'll learn how the device driver notifies you of I/O completions in the next section, "Receiving Completed I/O Request Notifications." For now, let's concentrate on how to queue asynchronous I/O requests. Queuing asynchronous I/O requests is the essence of designing a high-performance, scalable application, and it is what the remainder of this chapter is all about.

To access a device asynchronously, you must first open the device by calling *CreateFile*, specifying the FILE_FLAG_OVERLAPPED flag in the *dwFlagsAndAttrs* parameter. This flag notifies the system that you intend to access the device asynchronously.

To queue an I/O request for a device driver, you use the *ReadFile* and *WriteFile* functions that you already learned about in the section "Performing Synchronous Device I/O." For convenience, I'll list the function prototypes again:

```
BOOL ReadFile(
   HANDLE       hfile,
   PVOID        pvBuffer,
   DWORD        nNumBytesToRead,
   PDWORD       pdwNumBytes,
   OVERLAPPED*  pOverlapped);

BOOL WriteFile(
   HANDLE       hfile,
   CONST VOID   *pvBuffer,
   DWORD        nNumBytesToWrite,
   PDWORD       pdwNumBytes,
   OVERLAPPED*  pOverlapped);
```

When either of these functions is called, the function checks to see if the device, identified by the *hfile* parameter, was opened with the FILE_ FLAG_OVERLAPPED flag. If this flag is specified, the function performs asynchronous device I/O. By the way, when calling either function for asynchronous I/O, you can (and usually do) pass NULL for the *pdwNumBytes* parameter. After all, you expect these functions to return before the I/O request has completed, so examining the number of bytes transferred is meaningless at this time.

The OVERLAPPED Structure

When performing asynchronous device I/O, you must pass the address to an initialized OVERLAPPED structure via the *pOverlapped* parameter. The word "overlapped" in this context means that the time spent performing the I/O request overlaps the time your thread spends performing other tasks. Here's what an OVERLAPPED structure looks like:

```
typedef struct _OVERLAPPED {
   DWORD  Internal;     // [out] Error code
   DWORD  InternalHigh; // [out] Number of bytes transferred
   DWORD  Offset;       // [in]  Low  32-bit file offset
   DWORD  OffsetHigh;   // [in]  High 32-bit file offset
   HANDLE hEvent;       // [in]  Event handle or data
} OVERLAPPED, *LPOVERLAPPED;
```

This structure contains five members. Three of these members, *Offset*, *OffsetHigh*, and *hEvent*, must be initialized prior to calling *ReadFile* or *WriteFile*. The other two members, *Internal* and *InternalHigh*, are set by the device driver and can be examined when the I/O operation completes. Here is a more detailed explanation of these member variables:

- ***Offset* and *OffsetHigh*** When a file is being accessed, these members indicate the 64-bit offset in the file where you want the I/O operation to begin. Recall that each file kernel object has a file pointer associated with it. When issuing a synchronous I/O request, the system knows to start accessing the file at the location identified by the file pointer. After the operation is complete, the system updates the file pointer automatically so that the next operation can pick up where the last operation left off.

  When performing asynchronous I/O, this file pointer is ignored by the system. Imagine what would happen if your code placed two asynchronous calls to *ReadFile* (for the same file kernel object). In this scenario, the system wouldn't know where to start reading for the second call to *ReadFile*. You probably wouldn't want to start reading the file at the same location used by the first call to *ReadFile*. You might want to start the second read at the byte in the file that followed the last byte that was read by the first call to *ReadFile*. To avoid the confusion of multiple asynchronous calls to the same object, all asynchronous I/O requests must specify the starting file offset in the OVERLAPPED structure.

  Note that the *Offset* and *OffsetHigh* members are *not* ignored for nonfile devices—you must initialize both members to 0 or the I/O request will fail and *GetLastError* will return ERROR_INVALID_PARAMETER.

- ***hEvent*** This member is used by one of the four methods available for receiving I/O completion notifications. When using the alertable I/O notification method, this member can be used for your own purposes. I know many developers who store the address of a C++ object in *hEvent*. (This member will be discussed more in the section "Signaling an Event Kernel Object.")

- ***Internal*** This member holds the processed I/O's error code. As soon as you issue an asynchronous I/O request, the device driver sets *Internal* to STATUS_PENDING, indicating that no error has occurred because the operation has not started. In fact, the macro *HasOverlappedIoCompleted*, which is defined in WinBase.h, allows you to check whether an asynchronous I/O operation has completed. If the request is still pending, FALSE is returned; if the I/O request is completed, TRUE is returned. Here is the macro's definition:

  ```
  #define HasOverlappedIoCompleted(pOverlapped) \
     ((pOverlapped)->Internal != STATUS_PENDING)
  ```

- ***InternalHigh*** When an asynchronous I/O request completes, this member holds the number of bytes transferred.

When first designing the OVERLAPPED structure, Microsoft decided not to document the *Internal* and *InternalHigh* members (which explains their names). As time went on, Microsoft realized that the information contained in these members would be useful to developers and so documented them. However, Microsoft didn't change the names of the members because the operating system source code referenced them frequently, and Microsoft didn't want to modify the code.

NOTE

When an asynchronous I/O request completes, you will receive the address of the OVERLAPPED structure that was used when the request was initiated. Having more contextual information passed around with an OVERLAPPED structure is frequently useful—for example, if you wanted to store the handle of the device used to initiate the I/O request inside the OVERLAPPED structure. The OVERLAPPED structure doesn't offer a device handle member or other potentially useful members for storing context, but you can solve this problem quite easily.

I frequently create a C++ class that is derived from an OVERLAPPED structure. This C++ class can have any additional information in it that I want. When my application receives the address of an OVERLAPPED structure, I simply cast the address to a pointer of my C++ class. Now I have access to the OVERLAPPED members and any additional context information my application needs. The FileCopy sample application at the end of this chapter demonstrates this technique. See my CIOReq C++ class in the FileCopy sample application for the details.

Asynchronous Device I/O Caveats

You should be aware of a couple of issues when performing asynchronous I/O. First, the device driver doesn't have to process queued I/O requests in a first-in first-out (FIFO) fashion. For example, if a thread executes the following code, the device driver will quite possibly write to the file and then read from the file:

```
OVERLAPPED o1 = { 0 };
OVERLAPPED o2 = { 0 };
BYTE bBuffer[100];
ReadFile (hfile, bBuffer, 100, NULL, &o1);
WriteFile(hfile, bBuffer, 100, NULL, &o2);
```

A device driver typically executes I/O requests out of order if doing so helps performance. For example, to reduce head movement and seek times, a file system driver might scan the queued I/O request list looking for requests that are near the same physical location on the hard drive.

The second issue you should be aware of is the proper way to perform error checking. Most Windows functions return FALSE to indicate failure or nonzero to indicate success. However, the *ReadFile* and *WriteFile* functions behave a little differently. An example might help to explain.

When attempting to queue an asynchronous I/O request, the device driver might choose to process the request synchronously. This can occur if you're reading from a file and the system checks whether the data you want is already in the system's cache. If the data is available, your I/O request is not queued to the device driver; instead, the system copies the data from the cache to your buffer, and the I/O operation is complete.

ReadFile and *WriteFile* return a nonzero value if the requested I/O was performed synchronously. If the requested I/O is executing asynchronously, or if an error occurred while calling *ReadFile* or *WriteFile*, FALSE is returned. When FALSE is returned, you must call *GetLastError* to determine specifically what happened. If *GetLastError* returns ERROR_IO_PENDING, the I/O request was successfully queued and will complete later.

If *GetLastError* returns a value other than ERROR_IO_PENDING, the I/O request could not be queued to the device driver. Here are the most common error codes returned from *GetLastError* when an I/O request can't be queued to the device driver:

- **ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY** Each device driver maintains a fixed-size list (in a nonpaged pool) of outstanding I/O requests. If this list is full, the system can't queue your request, *ReadFile* and *WriteFile* return FALSE, and *GetLastError* reports

one of these two error codes (depending on the driver).

- **ERROR_NOT_ENOUGH_QUOTA** Certain devices require that your data buffer's storage be page locked so that the data cannot be swapped out of RAM while the I/O is pending. This page-locked storage requirement is certainly true of file I/O when using the FILE_FLAG_NO_BUFFERING flag. However, the system restricts the amount of storage that a single process can page lock. If *ReadFile* and *WriteFile* cannot page lock your buffer's storage, the functions return FALSE, and *GetLastError* reports ERROR_NOT_ENOUGH_QUOTA. You can increase a process's quota by calling *SetProcessWorkingSetSize*.

How should you handle these errors? Basically, these errors occur because a number of outstanding I/O requests have not yet completed, so you need to allow some pending I/O requests to complete and then reissue the calls to *ReadFile* and *WriteFile*.

The third issue you should be aware of is that the data buffer and OVERLAPPED structure used to issue the asynchronous I/O request must not be moved or destroyed until the I/O request has completed. When queuing an I/O request to a device driver, the driver is passed the *address* of the data buffer and the *address* of the OVERLAPPED structure. Notice that just the address is passed, not the actual block. The reason for this should be quite obvious: memory copies are very expensive and waste a lot of CPU time.

When the device driver is ready to process your queued request, it transfers the data referenced by the *pvBuffer* address, and it accesses the file's offset member and other members contained within the OVERLAPPED structure pointed to by the *pOverlapped* parameter. Specifically, the device driver updates the *Internal* member with the I/O's error code and the *InternalHigh* member with the number of bytes transferred.

NOTE

It is absolutely essential that these buffers not be moved or destroyed until the I/O request has completed, or memory will be corrupted. Also, you must allocate and initialize a unique OVERLAPPED structure for each I/O request.

The preceding note is very important and is one of the most common bugs developers introduce when implementing an asynchronous device I/O architecture. Here's an example of what *not* to do:

```
VOID ReadData(HANDLE hfile) {
   OVERLAPPED o = { 0 };
   BYTE b[100];
   ReadFile(hfile, b, 100, NULL, &o);
}
```

This code looks fairly harmless, and the call to *ReadFile* is perfect. The only problem is that the function returns after queuing the asynchronous I/O request. Returning from the function essentially frees the buffer and the OVERLAPPED structure from the thread's stack, but the device driver is not aware that *ReadData* returned. The device driver still has two memory addresses that point to the thread's stack. When the I/O completes, the device driver is going to modify memory on the thread's stack, corrupting whatever happens to be occupying that spot in memory at the time. This bug is particularly difficult to find because the memory modification occurs asynchronously. Sometimes the device driver might perform I/O synchronously, in which case you won't see the bug. Sometimes the I/O might complete right after the function returns, or it might complete over an hour later, and who knows what the stack is being used for then?

Canceling Queued Device I/O Requests

Sometimes you might want to cancel a queued device I/O request before the device driver has processed it. Windows offers a few ways to do this:

- You can call *CancelIo* to cancel all I/O requests queued by the calling thread for the specified handle:

```
BOOL CancelIo(HANDLE hfile);
```

- You can cancel all queued I/O requests, regardless of which thread queued the request, by closing the handle to a device itself.

- When a thread dies, the system automatically cancels all I/O requests issued by the thread.

As you can see, there is no way to cancel a single, specific I/O request.

NOTE

Canceled I/O requests complete with an error code of ERROR_OPERATION_ABORTED.

[Previous] [Next]

Receiving Completed I/O Request Notifications

At this point you know how to queue an asynchronous device I/O request, but I haven't discussed how the device driver notifies you after the I/O request has completed.

Windows offers four different methods (briefly described in Table 2-9) for receiving I/O completion notifications, and this chapter covers all of them. The methods are shown in order of complexity, from easiest to understand and implement (signaling a device kernel object) to hardest to understand and implement (I/O completion ports).

Table 2-9. *Methods for receiving I/O completion notifications*

Technique	Summary
Signaling a device kernel object	Not useful for performing multiple simultaneous I/O requests against a single device. Allows one thread to issue an I/O request and another thread to process it.
Signaling an event kernel object	Allows multiple simultaneous I/O requests against a single device. Allows one thread to issue an I/O request and another thread to process it.
Using alertable I/O	Allows multiple simultaneous I/O requests against a single device. The thread that issued an I/O request must also process it.
Using I/O completion ports	Allows multiple simultaneous I/O requests against a single device. Allows one thread to issue an I/O request and another thread to process it. This technique is highly scalable and has the most flexibility.

As stated at the beginning of this chapter, the I/O completion port is the hands-down best method of the four for receiving I/O completion notifications. By studying all four, you'll learn why Microsoft added the I/O completion port to Windows and how the I/O completion port solves all the problems that exist for the other methods.

Signaling a Device Kernel Object

Once a thread issues an asynchronous I/O request, the thread continues executing, doing useful work. Eventually, the thread needs to synchronize with the completion of the I/O operation. In other words, you'll hit a point in your thread's code at which the thread can't continue to execute unless the data from the device is fully loaded into the buffer.

In Windows, a device kernel object can be used for thread synchronization, so the object can either be in a signaled or nonsignaled state. The *ReadFile* and *WriteFile* functions set the device kernel object to the nonsignaled state just before queuing the I/O request. When the device driver completes the request, the driver sets the device kernel object to the signaled state.

A thread can determine whether an asynchronous I/O request has completed by calling either *WaitForSingleObject* or *WaitForMultipleObjects*. Here is a simple example:

```
HANDLE hfile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);
BYTE bBuffer[100];
OVERLAPPED o = { 0 };
o.Offset = 345;

BOOL fReadDone = ReadFile(hfile, bBuffer, 100, NULL, &o);
DWORD dwError = GetLastError();

if (!fReadDone && (dwError == ERROR_IO_PENDING)) {
   // The I/O is being performed asynchronously; wait for it to complete
   WaitForSingleObject(hfile, INFINITE);
   fReadDone = TRUE;
}

if (fReadDone) {
   // o.Internal contains the I/O error
   // o.InternalHigh contains the number of bytes transferred
   // bBuffer contains the read data
} else {
   // An error occurred; see dwError
}
```

This code issues an asynchronous I/O request and then immediately waits for the request to finish, defeating the purpose of asynchronous I/O! Obviously you would never actually write code similar to this, but the code does demonstrate important concepts, which I'll summarize here:

- The device must be opened for asynchronous I/O by using the FILE_FLAG_OVERLAPPED flag.
- The OVERLAPPED structure must have its *Offset*, *OffsetHigh*, and *hEvent* members initialized. In the code example, I set them all to 0 except for *Offset,* which I set to 345 so that *ReadFile* reads data from the file starting at byte 346.
- *ReadFile*'s return value is saved in *fReadDone*, which indicates whether the I/O request was performed synchronously.
- If the I/O request was not performed synchronously, I check to see whether an error occurred or whether the I/O is being performed asynchronously. Comparing the result of *GetLastError* with ERROR_IO_PENDING gives me this information.
- To wait for the data, I call *WaitForSingleObject*, passing the handle of the device kernel object. Calling the function suspends the thread until the kernel object becomes signaled. The device driver signals the object when it completes the I/O. After *WaitForSingleObject* returns, the I/O is complete and I set *fReadDone* to TRUE.
- After the read completes, you can examine the data in *bBuffer*, the error code in the OVERLAPPED structure's *Internal* member, and the number of bytes transferred in the OVERLAPPED structure's

InternalHigh member.

- If a true error occurred, *dwError* contains the error code giving more information.

Signaling an Event Kernel Object

The method for receiving I/O completion notifications just described is very simple and straightforward, but it turns out not to be all that useful because it does not handle multiple I/O requests well. For example, suppose you were trying to carry out multiple asynchronous operations on a single file at the same time. Say that you wanted to read 10 bytes from the file and write 10 bytes to the file simultaneously. The code might look like this:

```
HANDLE hfile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);

BYTE bBuffer[10];
OVERLAPPED oRead = { 0 };
oRead.Offset = 0;
ReadFile(hfile, bBuffer, 10, NULL, &oRead);

OVERLAPPED oWrite = { 0 };
oWrite.Offset = 10;
WriteFile(hfile, "Jeff", 5, NULL, &oWrite);


WaitForSingleObject(hfile, INFINITE);

// We don't know what completed: Read? Write? Both?
```

You can't synchronize your thread by waiting for the device to become signaled because the object becomes signaled as soon as either of the operations completes. If you call *WaitForSingleObject*, passing it the device handle, you will be unsure whether the function returned because the read operation completed, the write operation completed, or both operations completed. Clearly, there needs to be a better way to perform multiple, simultaneous asynchronous I/O requests so that you don't run into this predicament—fortunately, there is.

The last member of the OVERLAPPED structure, *hEvent*, identifies an event kernel object. You must create this event object by calling *CreateEvent*. When an asynchronous I/O request completes, the device driver checks to see whether the *hEvent* member of the OVERLAPPED structure is NULL. If *hEvent* is not NULL, the driver signals the event by calling *SetEvent*. The driver also sets the device object to the signaled state just as it did before. However, if you are using events to determine when a device operation has completed, you shouldn't wait for the device object to become signaled—wait for the event instead.

If you want to perform multiple asynchronous device I/O requests simultaneously, you must create a separate event object for each request, initialize the *hEvent* member in each request's OVERLAPPED structure, and then call *ReadFile* or *WriteFile*. When you reach the point in your code at which you need to synchronize with the completion of the I/O request, simply call *WaitForMultipleObjects*, passing in the event handles associated with each outstanding I/O request's OVERLAPPED structures. With this scheme, you can easily and reliably perform multiple asynchronous device I/O operations simultaneously and use the same device object. The following code demonstrates this approach:

```
HANDLE hfile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);

BYTE bBuffer[10];
OVERLAPPED oRead = { 0 };
oRead.Offset = 0;
oRead.hEvent = CreateEvent(...);
ReadFile(hfile, bBuffer, 10, NULL, &oRead);
```

```
OVERLAPPED oWrite = { 0 };
oWrite.Offset = 10;
oWrite.hEvent = CreateEvent(...);
WriteFile(hfile, "Jeff", 5, NULL, &oWrite);


HANDLE h[2];
h[0] = oRead.hEvent;
h[1] = oWrite.hEvent;
DWORD dw = WaitForMultipleObjects(2, h, FALSE, INFINITE);
switch (dw _ WAIT_OBJECT_0) {
   case 0:   // Read completed
      break;

   case 1:   // Write completed
      break;
}
```

This code is somewhat contrived and is not *exactly* what you'd do in a real-life application, but it does illustrate my point. Typically, a real-life application would have a loop that waits for I/O requests to complete. As each request completes, the thread would perform the desired task, queue another asynchronous I/O request, and loop back around, waiting for more I/O requests to complete.

GetOverlappedResult

Recall that originally Microsoft was not going to document the OVERLAPPED structure's *Internal* and *InternalHigh* members, which meant it needed to provide another way for you to know how many bytes were transferred during the I/O processing and get the I/O's error code. To make this information available to you, Microsoft created the *GetOverlappedResult* function:

```
BOOL GetOverlappedResult(
   HANDLE      hfile,
   OVERLAPPED* pOverlapped,
   PDWORD      pdwNumBytes,
   BOOL        fWait);
```

Microsoft now documents the *Internal* and *InternalHigh* members, so the *GetOverlappedResult* function is not very useful. However, when I was first learning asynchronous I/O, I decided to reverse engineer the function to help solidify concepts in my head. The following code shows how *GetOverlappedResult* is implemented internally:

```
BOOL GetOverlappedResult(
   HANDLE hfile,
   OVERLAPPED* po,
   PDWORD pdwNumBytes,
   BOOL fWait) {

   if (po->Internal == STATUS_PENDING) {
      DWORD dwWaitRet = WAIT_TIMEOUT;
      if (fWait) {
         // Wait for the I/O to complete
         dwWaitRet = WaitForSingleObject(
            (po->hEvent != NULL) ? po->hEvent : hfile, INFINITE);
      }

      if (dwWaitRet == WAIT_TIMEOUT) {
         // I/O not complete and we're not supposed to wait
         SetLastError(ERROR_IO_INCOMPLETE);
         return(FALSE);
      }
```

```
      if (dwWaitRet != WAIT_OBJECT_0) {
         // Error calling WaitForSingleObject
         return(FALSE);
      }
   }

   // I/O is complete; return number of bytes transferred
   *pdwNumBytes = po->InternalHigh;

   if (SUCCEEDED(po->Internal)) {
      return(TRUE);   // No I/O error
   }

   // Set last error to I/O error
   SetLastError(po->Internal);
   return(FALSE);
}
```

Alertable I/O

The third method available to you for receiving I/O completion notifications is called *alertable I/O*. At first, Microsoft touted alertable I/O as the absolute best mechanism for developers who wanted to create high-performance, scalable applications. But as developers started using alertable I/O, they soon realized that it was not going to live up to the promise.

I have worked with alertable I/O quite a bit, and I'll be the first to tell you that alertable I/O is horrible and should be avoided. However, to make alertable I/O work, Microsoft added some infrastructure into the operating system that I have found to be extremely useful and valuable. As you read this section, concentrate on the infrastructure that is in place and don't get bogged down in the I/O aspects.

Whenever a thread is created, the system also creates a queue that is associated with the thread. This queue is called the asynchronous procedure call (APC) queue. When issuing an I/O request, you can tell the device driver to append an entry to the calling thread's APC queue. To have completed I/O notifications queued to your thread's APC queue, you call the *ReadFileEx* and *WriteFileEx* functions:

```
BOOL ReadFileEx(
   HANDLE      hfile,
   PVOID       pvBuffer,
   DWORD       nNumBytesToRead,
   OVERLAPPED* pOverlapped,
   LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine);

BOOL WriteFileEx(
   HANDLE      hfile,
   CONST VOID  *pvBuffer,
   DWORD       nNumBytesToWrite,
   OVERLAPPED* pOverlapped,
   LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine);
```

Like *ReadFile* and *WriteFile, ReadFileEx* and *WriteFileEx* issue I/O requests to a device driver, and the functions return immediately. The *ReadFileEx* and *WriteFileEx* functions have the same parameters as the *ReadFile* and *WriteFile* functions, with two exceptions. First, the **Ex* functions are not passed a pointer to a DWORD that gets filled with the number of bytes transferred; this information can be retrieved only by the callback function. Second, the **Ex* functions require that you pass the address of a callback function, called a *completion routine*. This routine must have the following prototype:

```
VOID WINAPI CompletionRoutine(
   DWORD       dwError,
```

```
   DWORD       dwNumBytes,
   OVERLAPPED* po);
```

When you issue an asynchronous I/O request with *ReadFileEx* and *WriteFileEx*, the functions pass the address of this function to the device driver. When the device driver has completed the I/O request, it appends an entry in the issuing thread's APC queue. This entry contains the address of the completion routine function and the address of the OVERLAPPED structure used to initiate the I/O request.

NOTE

By the way, when an alertable I/O completes, the device driver will not attempt to signal an event object. In fact, the device does not reference the OVERLAPPED structure's *hEvent* member at all. Therefore, you can use the *hEvent* member for your own purposes if you like.

When the thread is in an alertable state (discussed shortly), the system examines its APC queue and, for every entry in the queue, the system calls the completion function, passing it the I/O error code, the number of bytes transferred, and the address of the OVERLAPPED structure. Note that the error code and number of bytes transferred can also be found in the OVERLAPPED structure's *Internal* and *InternalHigh* members. (As I mentioned earlier, Microsoft originally didn't want to document these, so it passed them as parameters to the function.)

I'll get back to this completion routine function shortly. First let's look at how the system handles the asynchronous I/O requests. The following code queues three different asynchronous operations:

```
hfile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);

ReadFileEx(hfile, ...);    // Perform first ReadFileEx
WriteFileEx(hfile, ...);   // Perform first WriteFileEx
ReadFileEx(hfile, ...);    // Perform second ReadFileEx

SomeFunc();
```

If the call to *SomeFunc* takes some time to execute, the system completes the three operations before *SomeFunc* returns. While the thread is executing the *SomeFunc* function, the device driver is appending completed I/O entries to the thread's APC queue. The APC queue might look something like this:

```
first WriteFileEx completed
second ReadFileEx completed
first ReadFileEx completed
```

The APC queue is maintained internally by the system. You'll also notice from the list that the system can execute your queued I/O requests in any order, and that the I/O requests that you issue last might be completed first and vice versa. Each entry in your thread's APC queue contains the address of a callback function and a value that is passed to the function.

As I/O requests complete, they are simply queued to your thread's APC queue—the callback routine is not immediately called because your thread might be busy doing something else and cannot be interrupted. To process entries in your thread's APC queue, the thread must put itself in an alertable state. This simply means that your thread has reached a position in its execution where it can handle being interrupted. Windows offers five functions that can place a thread in an alertable state:

```
DWORD SleepEx(
   DWORD dwTimeout,
   BOOL  fAlertable);
DWORD WaitForSingleObjectEx(
   HANDLE hObject,
   DWORD  dwTimeout,
   BOOL   fAlertable);
```

```
DWORD WaitForMultipleObjectsEx(
   DWORD   cObjects,
   PHANDLE phObjects,
   BOOL    fWaitAll,
   DWORD   dwTimeout,
   BOOL    fAlertable);

BOOL SignalObjectAndWait(
   HANDLE hObjectToSignal,
   HANDLE hObjectToWaitOn,
   DWORD  dwMilliseconds,
   BOOL   fAlertable);

DWORD MsgWaitForMultipleObjectsEx(
   DWORD   nCount,
   PHANDLE pHandles,
   DWORD   dwMilliseconds,
   DWORD   dwWakeMask,
   DWORD   dwFlags);
```

The last argument to the first four functions is a Boolean value indicating whether the calling thread should place itself in an alertable state. For *MsgWaitForMultipleObjectsEx*, you must use the MWMO_ALERTABLE flag to have the thread enter an alertable state. If you're familiar with the *Sleep*, *WaitForSingleObject*, and *WaitForMultipleObjects* functions, you shouldn't be surprised to learn that, internally, these non-*Ex* functions call their **Ex* counterparts, always passing FALSE for the *fAlertable* parameter.

When you call one of the five functions just mentioned and place your thread in an alertable state, the system first checks your thread's APC queue. If at least one entry is in the queue, the system does not put your thread to sleep; instead, the system pulls the entry from the APC queue and your thread calls the callback routine, passing the routine the completed I/O request's error code, number of bytes transferred, and address of the OVERLAPPED structure. When the callback routine returns to the system, the system checks for more entries in the APC queue. If more entries exist, they are processed. However, if no more entries exist, your call to the alertable function returns. Something to keep in mind is that if any entries are in your thread's APC queue when you call any of these functions, your thread never sleeps!

The only time these functions suspend your thread is when no entries are in your thread's APC queue at the time you call the function. While your thread is suspended, the thread will wake up if the kernel object (or objects) that you're waiting on becomes signaled or if an APC entry appears in your thread's queue. Since your thread is in an alertable state, as soon as an APC entry appears, the system wakes your thread and empties the queue (by calling the callback routines). Then the functions immediately return to the caller—your thread does not go back to sleep waiting for kernel objects to become signaled.

The return value from these five functions indicates why they have returned. If the return value is WAIT_IO_COMPLETION, you know that the thread is continuing to execute because at least one entry was processed from the thread's APC queue. If the return value is anything else, the thread woke up because the sleep period expired, the specified kernel object or objects became signaled, or a mutex was abandoned.

The Bad and the Good of Alertable I/O

At this point, we've discussed the mechanics of performing alertable I/O. Now you need to know about the two issues that make alertable I/O a horrible method for doing device I/O.

- **Callback functions** Alertable I/O requires that you create callback functions, which makes implementing your code much more difficult. These callback functions typically don't have enough contextual information about a particular problem to guide you, so you end up placing a lot of information in global variables. Fortunately, these global variables don't need to be synchronized

because the thread calling one of the five alterable functions is the same thread executing the callback functions. A single thread can't be in two places at one time, so the variables are safe.

- **Threading issues** The real big problem with alertable I/O is this: The thread issuing the I/O request must also handle the completion notification. If a thread issues several requests, that thread must respond to each request's completion notification, even if other threads are sitting completely idle. Since there is no load balancing, the application doesn't scale well.

Both of these problems are pretty severe, so I strongly discourage the use of alertable I/O for device I/O. I'm sure you guessed by now that the I/O completion port mechanism, discussed in the next section, solves both of the problems I just described. I promised to tell you some good stuff about the alertable I/O infrastructure, so before I move on to the I/O completion port, I'll do that.

Windows offers a function that allows you to manually queue an entry to a thread's APC queue:

```
DWORD QueueUserAPC(
   PAPCFUNC  pfnAPC,
   HANDLE    hThread,
   ULONG_PTR dwData);
```

The first parameter is a pointer to an APC function that must have the following prototype:

```
VOID WINAPI APCFunc(ULONG_PTR dwParam);
```

The second parameter is the handle of the thread for which you want to queue the entry. Note that this thread can be any thread in the system. If *hThread* identifies a thread in a different process's address space, *pfnAPC* must specify the memory address of a function that is in the address space of the target thread's process. The last parameter to *QueueUserAPC*, *dwData*, is a value that simply gets passed to the callback function.

Even though *QueueUserAPC* is prototyped as returning a DWORD, the function actually returns a BOOL indicating success or failure. You can use *QueueUserAPC* to perform extremely efficient interthread communication, even across process boundaries. Unfortunately, however, you can pass only a single value.

QueueUserAPC can also be used to force a thread out of a wait state. Suppose you have a thread calling *WaitForSingleObject*, waiting for a kernel object to become signaled. While the thread is waiting, the user wants to terminate the application. You know that threads should cleanly destroy themselves, but how do you force the thread waiting on the kernel object to wake up and kill itself? *QueueUserAPC* is the answer.

The following code demonstrates how to force a thread out of a wait state so that the thread can exit cleanly. The main function spawns a new thread, passing it the handle of some kernel object. While the secondary thread is running, the primary thread is also running. The secondary thread (executing the *ThreadFunc* function) calls *WaitForSingleObjectEx*, which suspends the thread, placing it in an alertable state. Now, say that the user tells the primary thread to terminate the application. Sure, the primary thread could just exit, and the system would kill the whole process, but this approach is not very clean, and in many scenarios, you'll just want to kill an operation without terminating the whole process.

So the primary thread calls *QueueUserAPC,* which places an APC entry in the secondary thread's APC queue. Since the secondary thread is in an alertable state, it now wakes and empties its APC queue by calling the *APCFunc* function. This function does absolutely nothing and just returns. Since the APC queue is now empty, the thread returns from its call to *WaitForSingleObjectEx* with a return value of WAIT_IO_COMPLETION. The *ThreadFunc* function checks specifically for this return value, knowing that it received an APC entry indicating that the thread should exit.

```
// The APC callback function has nothing to do
VOID WINAPI APCFunc(ULONG_PTR dwParam) {
   // Nothing to do in here
```

```
}


UINT WINAPI ThreadFunc(PVOID pvParam) {
   HANDLE hEvent = (HANDLE) pvParam;   // Handle is passed to this thread

   // Wait in an alertable state so that we can be forced to exit cleanly
   DWORD dw = WaitForSingleObjectEx(hEvent, INFINITE, TRUE);
   if (dw == WAIT_OBJECT_0) {
      // Object became signaled
   }
   if (dw == WAIT_IO_COMPLETION) {
      // QueueUserAPC forced us out of a wait state
      return(0);   // Thread dies cleanly
   }

   return(0);
}


void main() {
   HANDLE hEvent = CreateEvent(...);
   HANDLE hThread = (HANDLE) _beginthreadex(NULL, 0,
      ThreadFunc, (PVOID) hEvent, 0, NULL);

   // Force the secondary thread to exit cleanly
   QueueUserAPC(APCFunc, hThread, NULL);
   WaitForSingleObject(hThread, INFINITE);
   CloseHandle(hThread);
   CloseHandle(hEvent);
}
```

I know that some of you are thinking that this problem could have been solved by replacing the call to *WaitForSingleObjectEx* with a call to *WaitForMultipleObjects* and by creating another event kernel object to signal the secondary thread to terminate. For my simple example, your solution would work. However, if my secondary thread called *WaitForMultipleObjects* to wait until *all* objects became signaled, *QueueUserAPC* would be the only way to force the thread out of a wait state.

For good examples showing how to use *QueueUserAPC* this way, please see the RegNotify sample application in Chapter 5 of this book and the WaitForMultipleExpressions sample application in Chapter 11 of *Programming Applications for Microsoft Windows, Fourth Edition* (Jeffrey Richter, Microsoft Press, 1999).

I/O Completion Ports

Windows 2000 is designed to be a secure, robust operating system running applications that service literally thousands of users. Historically, you've been able to architect a service application by following one of two models:

- **Serial model** A single thread waits for a client to make a request (usually over the network). When the request comes in, the thread wakes and handles the client's request.

- **Concurrent model** A single thread waits for a client request and then creates a new thread to handle the request. While the new thread is handling the client's request, the original thread loops back around and waits for another client request. When the thread that is handling the client's request is completely processed, the thread dies.

The problem with the serial model is that it does not handle multiple, simultaneous requests well. If two clients make requests at the same time, only one can be processed at a time; the second request must wait for the first request to finish processing. A service that is designed using the serial approach cannot take

advantage of multiprocessor machines. Obviously, the serial model is good only for the simplest of server applications, in which few client requests are made and requests can be handled very quickly. A Ping server is a good example of a serial server.

Because of the limitations in the serial model, the concurrent model is extremely popular. In the concurrent model, a thread is created to handle each client request. The advantage is that the thread waiting for incoming requests has very little work to do. Most of the time, this thread is sleeping. When a client request comes in, the thread wakes, creates a new thread to handle the request, and then waits for another client request. This means that incoming client requests are handled expediently. Also, because each client request gets its own thread, the server application scales well and can easily take advantage of multiprocessor machines. So if you are using the concurrent model and upgrade the hardware (add another CPU), the performance of the server application improves.

Service applications using the concurrent model were implemented using Windows. The Windows team noticed that application performance was not as high as desired. In particular, the team noticed that handling many simultaneous client requests meant that many threads were running in the system concurrently. Because all these threads were runnable (not suspended and waiting for something to happen), Microsoft realized that the Windows kernel spent too much time context switching between the running threads, and the threads were not getting as much CPU time to do their work. To make Windows an awesome server environment, Microsoft needed to address this problem. The result is the I/O completion port kernel object.

Creating an I/O Completion Port

The theory behind the I/O completion port is that the number of threads running concurrently must have an upper bound; that is, 500 simultaneous client requests cannot allow 500 runnable threads to exist. What, then, is the proper number of concurrent, runnable threads? Well, if you think about this question for a moment, you'll come to the realization that if a machine has two CPUs, having more than two runnable threads—one for each processor—really doesn't make sense. As soon as you have more runnable threads than CPUs available, the system has to spend time performing thread context switches, which wastes precious CPU cycles—a potential deficiency of the concurrent model.

Another deficiency of the concurrent model is that a new thread is created for each client request. Creating a thread is cheap when compared to creating a new process with its own virtual address space, but creating threads is far from free. The service application's performance can be improved if a pool of threads is created when the application initializes, and these threads hang around for the duration of the application. I/O completion ports were designed to work with a pool of threads.

An I/O completion port is probably the most complex kernel object. To create an I/O completion port, you call *CreateIoCompletionPort*:

```
HANDLE CreateIoCompletionPort(
   HANDLE    hfile,
   HANDLE    hExistingCompPort,
   ULONG_PTR CompKey,
   DWORD     dwNumberOfConcurrentThreads);
```

This function performs two different tasks: it creates an I/O completion port, and it associates a device with an I/O completion port. This function is overly complex, and in my opinion, Microsoft should have split it into two separate functions. When I work with I/O completion ports, I separate these two capabilities by creating two tiny functions that abstract the call to *CreateIoCompletionPort*. The first function I write is called *CreateNewCompletionPort*, and I implement it as follows:

```
HANDLE CreateNewCompletionPort(DWORD dwNumberOfConcurrentThreads) {

   return(CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0,
```

```
      dwNumberOfConcurrentThreads));
}
```

This function takes a single argument, *dwNumberOfConcurrentThreads*, and then calls the Windows *CreateIoCompletionPort* function, passing in hard-coded values for the first three parameters and *dwNumberOfConcurrentThreads* for the last parameter. You see, the first three parameters to *CreateIoCompletionPort* are used only when you are associating a device with a completion port. (I'll talk about this shortly.) To create just a completion port, I pass INVALID_HANDLE_VALUE, NULL, and 0, respectively, to *CreateIoCompletionPort*'s first three parameters.

The *dwNumberOfConcurrentThreads* parameter tells the I/O completion port the maximum number of threads that should be runnable at the same time. If you pass 0 for the *dwNumberOfConcurrentThreads* parameter, the completion port defaults to allowing as many concurrent threads as there are CPUs on the host machine. This is usually exactly what you want so that extra context switching is avoided. You might want to increase this value if the processing of a client request requires a lengthy computation that rarely blocks, but increasing this value is strongly discouraged. You might experiment with the *dwNumberOfConcurrentThreads* parameter by trying different values and comparing your application's performance.

You'll notice that *CreateIoCompletionPort* is about the only Windows function that creates a kernel object but does not have a parameter that allows you to pass the address of a SECURITY_ATTRIBUTES structure. This is because completion ports are intended for use within a single process only. The reason will be clear to you when I explain how to use completion ports.

Associating a Device with an I/O Completion Port

When you create an I/O completion port, the kernel actually creates five different data structures, as shown in Figure 2-1. You should refer to this figure as you continue reading.

Figure 2-1. *The internal workings of an I/O completion port*

The first data structure is a device list indicating the device or devices associated with the port. You associate a device with the port by calling *CreateIoCompletionPort*. Again, I created my own function, *AssociateDeviceWithCompletionPort*, which abstracts the call to *CreateIoCompletionPort*:

```
BOOL AssociateDeviceWithCompletionPort(
   HANDLE hCompPort, HANDLE hDevice, DWORD dwCompKey) {

   HANDLE h = CreateIoCompletionPort(hDevice, hCompPort, dwCompKey, 0);
   return(h == hCompPort);
}
```

AssociateDeviceWithCompletionPort appends an entry to an existing completion port's device list. You pass to the function the handle of an existing completion port (returned by a previous call to *CreateNewCompletionPort*), the handle of the device (this can be a file, a socket, a mailslot, a pipe, and so on), and a completion key (a value that has meaning to you; the operating system doesn't care what you pass here). Each time you associate a device with the port, the system appends this information to the completion port's device list.

NOTE

The *CreateIoCompletionPort* function is complex, and I recommend that you mentally separate the two reasons for calling it. There is one advantage to having the function be so complex: you can create an I/O completion port and associate a device with it at the same time. For example, the following code opens a file and creates a new completion port, associating the file with it. All I/O requests to the file will complete with a completion key of CK_FILE, and the port will allow as many as two threads to execute concurrently.

```
#define CK_FILE   1
HANDLE hfile = CreateFile(...);
HANDLE hCompPort = CreateIoCompletionPort(hfile, NULL, CK_FILE, 2);
```

The second data structure is an I/O completion queue. When an asynchronous I/O request for a device completes, the system checks to see whether the device is associated with a completion port and, if it is, the system appends the completed I/O request entry to the end of the completion port's I/O completion queue. Each entry in this queue indicates the number of bytes transferred, the completion key value that was set when the device was associated with the port, the pointer to the I/O request's OVERLAPPED structure, and an error code. I'll discuss how entries are removed from this queue shortly.

NOTE

Issuing an I/O request to a device and not having an I/O completion entry queued to the I/O completion port is possible. This is not usually necessary, but it can come in handy occasionally—for example, when you send data over a socket and you don't care whether the data actually makes it or not.

To issue an I/O request without having a completion entry queued, you must load the OVERLAPPED structure's *hEvent* member with a valid event handle and bitwise-OR this value with 1, like this:

```
Overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
Overlapped.hEvent = (HANDLE) ((DWORD_PTR) Overlapped.hEvent | 1);
ReadFile(..., &Overlapped);
```

Now you can issue your I/O request, passing the address of this OVERLAPPED structure to the desired function (such as *ReadFile* above).

It would be nice if you didn't have to create an event just to stop the queuing of the I/O completion. I would like to be able to do the following, but it doesn't work:

```
Overlapped.hEvent = 1;
ReadFile(..., &Overlapped);
```

Also, don't forget to reset the low-order bit before closing this event handle:

```
CloseHandle((HANDLE) ((DWORD_PTR) Overlapped.hEvent & ~1));
```

Architecting Around an I/O Completion Port

When your service application initializes, it should create the I/O completion port by calling a function like *CreateNewCompletionPort*. The application should then create a pool of threads to handle client requests. The question you ask now is, "How many threads should be in the pool?" This is a tough question to answer, and I will address it in more detail later in the section "How Many Threads in the Pool?" For right now, a standard rule of thumb is to take the number of CPUs on the host machine and multiply it by 2. So on a dual-processor machine, you should create a pool of four threads.

All the threads in the pool should execute the same function. Typically, this thread function performs some sort of initialization and then enters a loop that should terminate when the service process is instructed to stop. Inside the loop, the thread puts itself to sleep waiting for device I/O requests to complete to the completion port. Calling *GetQueuedCompletionStatus* does this:

```
BOOL GetQueuedCompletionStatus(
   HANDLE       hCompPort,
   PDWORD       pdwNumBytes,
   PULONG_PTR   CompKey,
   OVERLAPPED** ppOverlapped,
   DWORD        dwMilliseconds);
```

The first parameter, *hCompPort*, indicates which completion port the thread is interested in monitoring. Many service applications will use a single I/O completion port and have all I/O request notifications complete to this one port. Basically, the job of *GetQueuedCompletionStatus* is to put the calling thread to sleep until an entry appears in the specified completion port's I/O completion queue or until the specified time-out occurs (as specified in the *dwMilliseconds* parameter).

The third data structure associated with an I/O completion port is the waiting thread queue. As each thread in the thread pool calls *GetQueuedCompletionStatus*, the ID of the calling thread is placed in this waiting thread queue, enabling the I/O completion port kernel object to always know which threads are currently waiting to handle completed I/O requests. When an entry appears in the port's I/O completion queue, the completion port wakes one of the threads in the waiting thread queue. This thread gets the pieces of information that make up a completed I/O entry: the number of bytes transferred, the completion key, and the address of the OVERLAPPED structure. This information is returned to the thread via the *pdwNumBytes*, *pCompKey*, and *ppOverlapped* parameters passed to *GetQueuedCompletionStatus*.

Determining the reason that *GetQueuedCompletionStatus* returned is somewhat difficult; the following code demonstrates the proper way to do it:

```
DWORD dwNumBytes;
ULONG_PTR CompKey;
OVERLAPPED* pOverlapped;

// hIOCP is initialized somewhere else in the program
BOOL fOk = GetQueuedCompletionStatus(hIOCP,
   &dwNumBytes, &CompKey, &pOverlapped, 1000);
DWORD dwError = GetLastError();

if (fOk) {
   // Process a successfully completed I/O request
} else {
   if (pOverlapped != NULL) {
      // Process a failed completed I/O request
      // dwError contains the reason for failure
   } else {
      if (dwError == WAIT_TIMEOUT) {
         // Time-out while waiting for completed I/O entry
      } else {
         // Bad call to GetQueuedCompletionStatus
         // dwError contains the reason for the bad call
      }
   }
}
```

As you would expect, entries are removed from the I/O completion queue in a first-in first-out fashion. However, as you might not expect, threads that call *GetQueuedCompletionStatus* are awakened in a last-in first-out (LIFO) fashion. The reason for this is again to improve performance. For example, say that four threads are waiting in the waiting thread queue. If a single completed I/O entry appears, the last thread to call *GetQueuedCompletionStatus* wakes up to process the entry. When this last thread is finished processing the entry, the thread again calls *GetQueuedCompletionStatus* to enter the waiting thread queue. Now if another I/O completion entry appears, the same thread that processed the first entry is awakened to process the new entry.

As long as I/O requests complete so slowly that a single thread can handle them, the system will just keep waking the one thread, and the other three threads will continue to sleep. By using this LIFO algorithm,

threads that don't get scheduled can have their memory resources (such as stack space) swapped out to the disk and flushed from a processor's cache. This means having many threads waiting on a completion port isn't bad. If you do have several threads waiting but few I/O requests completing, the extra threads have most of their resources swapped out of the system anyway.

How the I/O Completion Port Manages the Thread Pool

Now it's time to discuss why I/O completion ports are so useful. First, when you create the I/O completion port, you specify the number of threads that can run concurrently. As I said, you usually set this value to the number of CPUs on the host machine. As completed I/O entries are queued, the I/O completion port wants to wake up waiting threads. However, the completion port will wake up only as many threads as you have specified. So if four I/O requests complete and four threads are waiting in a call to *GetQueuedCompletionStatus*, the I/O completion port will allow only two threads to wake up; the other two threads continue to sleep. As each thread processes a completed I/O entry, the thread again calls *GetQueuedCompletionStatus*. The system sees that more entries are queued and wakes the same threads to process the remaining entries.

If you're thinking about this carefully, you should notice that something just doesn't make a lot of sense: if the completion port only ever allows the specified number of threads to wake up concurrently, why have more threads waiting in the thread pool? For example, suppose I'm running on a machine with two CPUs, and I create the I/O completion port, telling it to allow no more than two threads to process entries concurrently. But I create four threads (twice the number of CPUs) in the thread pool. It seems as though I am creating two additional threads that will never be awakened to process anything.

But I/O completion ports are very smart. When a completion port wakes a thread, the completion port places the thread's ID in the fourth data structure associated with the completion port, a released thread list. (See Figure 2-1.) This allows the completion port to remember which threads it awakened and to monitor the execution of these threads. If a released thread calls any function that places the thread in a wait state, the completion port detects this and updates its internal data structures by moving the thread's ID from the released thread list to the paused thread list (the fifth and final data structure that is part of an I/O completion port).

The goal of the completion port is to keep as many entries in the released thread list as is specified by the concurrent number of threads value used when creating the completion port. If a released thread enters a wait state for any reason, the released thread list shrinks and the completion port releases another waiting thread. If a paused thread wakes, it leaves the paused thread list and reenters the released thread list. This means that the released thread list can now have more entries in it than are allowed by the maximum concurrency value.

NOTE

Once a thread calls *GetQueuedCompletionStatus*, the thread is "assigned" to the specified completion port. The system assumes that all assigned threads are doing work on behalf of the completion port. The completion port wakes threads from the pool only if the number of running assigned threads is less than the completion port's maximum concurrency value.

You can break the thread/completion port assignment in one of three ways:

- Have the thread exit.
- Have the thread call *GetQueuedCompletionStatus*, passing the handle of a different I/O completion port.
- Destroy the I/O completion port that the thread is currently assigned to.

Let's tie all of this together now. Say that we are again running on a machine with two CPUs. We create a completion port that allows no more than two threads to wake concurrently, and we create four threads that are waiting for completed I/O requests. If three completed I/O requests get queued to the port, only two threads are awakened to process the requests, reducing the number of runnable threads and saving context switching time. Now if one of the running threads calls *Sleep*, *WaitForSingleObject*, *WaitForMultipleObjects*, *SignalObjectAndWait*, a synchronous I/O call, or any function that would cause the thread not to be runnable, the I/O completion would detect this and wake a third thread immediately. The goal of the completion port is to keep the CPUs saturated with work.

Eventually, the first thread will become runnable again. When this happens, the number of runnable threads will be higher than the number of CPUs in the system. However, the completion port again is aware of this and will not allow any additional threads to wake up until the number of threads drops below the number of CPUs. The I/O completion port architecture presumes that the number of runnable threads will stay above the maximum for only a short time and will die down quickly as the threads loop around and again call *GetQueuedCompletionStatus*. This explains why the thread pool should contain more threads than the concurrent thread count set in the completion port.

How Many Threads in the Pool?

Now is a good time to discuss how many threads should be in the thread pool. Consider two issues. First, when the service application initializes, you want to create a minimum set of threads so that you don't have to create and destroy threads on a regular basis. Remember that creating and destroying threads wastes CPU time, so you're better off minimizing this process as much as possible. Second, you want to set a maximum number of threads because creating too many threads wastes system resources. Even if most of these resources can be swapped out of RAM, minimizing the use of system resources and not wasting even paging file space is to your advantage, if you can help it.

You will probably want to experiment with different numbers of threads. Most services (including Microsoft Internet Information Services) use heuristic algorithms to manage their thread pools. I recommend that you do the same. For example, you can create the following variables to manage the thread pool:

```
LONG g_nThreadsMin;    // Minimum number of threads in pool
LONG g_nThreadsMax;    // Maximum number of threads in pool
LONG g_nThreadsCrnt;   // Current number of threads in pool
LONG g_nThreadsBusy;   // Number of busy threads in pool
```

When your application initializes, you can create the *g_nThreadsMin* number of threads, all executing the same thread pool function. The following pseudo-code shows how this thread function might look:

```
DWORD WINAPI ThreadPoolFunc(PVOID pv) {

   // Thread is entering pool
   InterlockedIncrement(&g_nThreadsCrnt);
   InterlockedIncrement(&g_nThreadsBusy);

   for (BOOL fStayInPool = TRUE; fStayInPool;) {

      // Thread stops executing and waits for something to do
      InterlockedDecrement(&m_nThreadsBusy);
      BOOL fOk = GetQueuedCompletionStatus(...);
      DWORD dwIOError = GetLastError();

      // Thread has something to do, so it's busy
      int nThreadsBusy = InterlockedIncrement(&m_nThreadsBusy);

      // Should we add another thread to the pool?
      if (nThreadsBusy == m_nThreadsCrnt) {    // All threads are busy
```

```
            if (nThreadsBusy < m_nThreadsMax) {   // The pool isn't full
               if (GetCPUUsage() < 75) {   // CPU usage is below 75%

                  // Add thread to pool
                  CloseHandle(chBEGINTHREADEX(...));
               }
            }
         }

         if (!fOk && (dwIOError == WAIT_TIMEOUT)) {   // Thread timed-out
            if (!ThreadHasIoPending()) {
               // There isn't much for the server to do, and this thread
               // can die because it has no outstanding I/O requests
               fStayInPool = FALSE;
            }
         }

         if (fOk || (po != NULL)) {
            // Thread woke to process something; process it
            ...

            if (GetCPUUsage() > 90) {       // CPU usage is above 90%
               if (!ThreadHasIoPending()) { // No pending I/O requests
                  if (g_nThreadsCrnt > g_nThreadsMin)) { // Pool above min

                     fStayInPool = FALSE;   // Remove thread from pool
                  }
               }
            }
         }
      }

      // Thread is leaving pool
      InterlockedDecrement(&g_nThreadsBusy);
      InterlockedDecrement(&g_nThreadsCurrent);
      return(0);
   }
```

This pseudo-code shows how creative you can get when using an I/O completion port. The *GetCPUUsage* and *ThreadHasIoPending* functions are not part of the Windows API. If you want their behavior, you'll have to implement the functions yourself. In addition, you must make sure that the thread pool always contains at least one thread in it, or clients will never get tended to. Use my pseudo-code as a guide, but your particular service might perform better if structured differently.

NOTE

Earlier in this chapter, in the section "Canceling Queued Device I/O Requests," I said that the system automatically cancels all pending I/O requests issued by a thread when that thread terminates. This is why a function similar to *ThreadHasIoPending* is necessary in the pseudo-code. Do not allow threads to terminate if they have outstanding I/O requests.

Many services offer a management tool that allows an administrator to have some control over the thread pool's behavior—for example, to set the minimum and maximum number of threads, the CPU time usage thresholds, and also the maximum concurrency value used when creating the I/O completion port.

Simulating Completed I/O Requests

I/O completion ports do not have to be used with device I/O at all. This chapter is also about interthread communication techniques, and the I/O completion port kernel object is an awesome mechanism to use to help with this. In the "Alertable I/O" section of this chapter, I presented the *QueueUserAPC* function, which

allows a thread to post an APC entry to another thread. I/O completion ports have an analogous function, *PostQueuedCompletionStatus*:

```
BOOL PostQueuedCompletionStatus(
   HANDLE      hCompPort,
   DWORD       dwNumBytes,
   ULONG_PTR   CompKey,
   OVERLAPPED* pOverlapped);
```

This function appends a completed I/O notification to an I/O completion port's queue. The first parameter, *hCompPort*, identifies the completion port that you wish to queue the entry for. The remaining three parameters, *dwNumBytes*, *CompKey*, and *pOverlapped*, indicate the values that should be returned by a thread's call to *GetQueuedCompletionStatus*. When a thread pulls a simulated entry from the I/O completion queue, *GetQueuedCompletionStatus* returns TRUE, indicating a successfully executed I/O request.

The *PostQueuedCompletionStatus* function is incredibly useful—it gives you a way to communicate with all the threads in your pool. For example, when the user terminates a service application, you want all the threads to exit cleanly. But if the threads are waiting on the completion port and no I/O requests are coming in, the threads can't wake up. By calling *PostQueuedCompletionStatus* once for each thread in the pool, each thread can wake up, examine the values returned from *GetQueuedCompletionStatus*, see that the application is terminating, and clean up and exit appropriately.

You must be careful when using a thread termination technique like the one I just described. My example works because the threads in the pool are dying and not calling *GetQueuedCompletionStatus* again. However, if you want to notify each of the pool's threads of something and have them loop back around to call *GetQueuedCompletionStatus* again, you will have a problem because the threads wake up in a LIFO order. So you will have to employ some additional thread synchronization in your application to ensure that each pool thread gets the opportunity to see its simulated I/O entry. Without this additional thread synchronization, one thread might see the same notification several times.

The FileCopy Sample Application

The FileCopy sample application ("02 FileCopy.exe"), shown in Listing 2-1 at the end of this chapter, demonstrates the use of I/O completion ports. The source code and resource files for the application are in the 02-FileCopy directory on the companion CD. The program simply copies a file specified by the user to a new file called FileCopy.cpy. When the user executes FileCopy, the dialog box shown in Figure 2-2 appears.

Figure 2-2. *The dialog box for the FileCopy sample application*

The user clicks on the Pathname button to select the file to be copied, and the Pathname and File Size fields are updated. When the user clicks on the Copy button, the program calls the *FileCopy* function, which does all the hard work. Let's concentrate our discussion on the *FileCopy* function.

When preparing to copy, *FileCopy* opens the source file and retrieves its size, in bytes. I want the file copy to execute as blindingly fast as possible, so the file is opened using the FILE_FLAG_NO_BUFFERING flag. Opening the file with the FILE_FLAG_NO_BUFFERING flag allows me to access the file directly, bypassing the additional memory copy overhead incurred when allowing the system's cache to "help" access the file. Of course, accessing the file directly means slightly more work for me: I must always access the file using offsets that are multiples of the disk volume's sector size, and I must read and write data that is a

multiple of the sector's size as well. I chose to transfer the file's data in BUFFSIZE (64 KB) chunks, which is guaranteed to be a multiple of the sector size. This is why I round up the source file's size to a multiple of BUFFSIZE. You'll also notice that the source file is opened with the FILE_FLAG_ OVERLAPPED flag so that I/O requests against the file are performed asynchronously.

The destination file is opened similarly: both the FILE_FLAG_NO_ BUFFERING and FILE_FLAG_OVERLAPPED flags are specified. I also pass the handle of the source file as *CreateFile*'s *hfileTemplate* parameter when creating the destination file, causing the destination file to have the same attributes as the source.

NOTE

Once both files are open, the destination file size is immediately set to its maximum size by calling *SetFilePointerEx* and *SetEndOfFile*. Adjusting the destination file's size now is extremely important because NTFS maintains a high-water marker that indicates the highest point at which the file was written. If you read past this marker, the system knows to return zeroes. If you write past the marker, the file's data from the old high-water marker to the write offset is filled with zeroes, your data is written to the file, and the file's high-water marker is updated. This behavior satisfies C2 security requirements pertaining to not presenting prior data. When you write to the end of a file on an NTFS partition, causing the high-water marker to move, NTFS must perform the I/O request synchronously even if asynchronous I/O was desired. If the *FileCopy* function didn't set the size of the destination file, none of the overlapped I/O requests would be performed asynchronously.

Now that the files are opened and ready to be processed, *FileCopy* creates an I/O completion port. To make working with I/O completion ports easier, I created a small C++ class, CIOCP, that is a very simple wrapper around the I/O completion port functions. This class can be found in the IOCP.h file discussed in Appendix B, "Class Library." *FileCopy* creates an I/O completion port by creating an instance (named iocp) of my CIOCP class.

The source file and destination file are associated with the completion port by calling the CIOCP's *AssociateDevice* member function. When associated with the completion port, each device is assigned a completion key. When an I/O request completes against the source file, the completion key is CK_READ, indicating that a read operation must have completed. Likewise, when an I/O request completes against the destination file, the completion key is CK_WRITE, indicating that a write operation must have completed.

Now we're ready to initialize a set of I/O requests (OVERLAPPED structures) and their memory buffers. The *FileCopy* function keeps 4 (MAX_ PENDING_IO_REQS) I/O requests outstanding at any one time. For applications of your own, you might prefer to allow the number of I/O requests to dynamically grow or shrink as necessary. In the FileCopy program, the CIOReq class encapsulates a single I/O request. As you can see, this C++ class is derived from an OVERLAPPED structure but contains some additional context information. FileCopy allocates an array of CIOReq objects and calls the *AllocBuffer* method to associate a BUFFSIZE-sized data buffer with each I/O request object. The data buffer is allocated using the *VirtualAlloc* function. Using *VirtualAlloc* ensures that the block begins on an even allocation-granularity boundary, which satisfies the requirement of the FILE_FLAG_NO_BUFFERING flag: the buffer must begin on an address that is evenly divisible by the volume's sector size.

To issue the initial read requests against the source file, I perform a little trick: I post four CK_WRITE I/O completion notifications to the I/O completion port. When the main loop runs, the thread waits on the port and wakes immediately, thinking that a write operation has completed. This causes the thread to issue a read request against the source file, which really starts the file copy.

The main loop terminates when there are no outstanding I/O requests. As long as I/O requests are outstanding, the interior of the loop waits on the completion port by calling CIOCP's *GetStatus* method (which calls

GetQueuedCompletionStatus internally). This call puts the thread to sleep until an I/O request completes to the completion port. When *GetQueuedCompletionStatus* returns, the returned completion key, *CompKey*, is checked. If *CompKey* is CK_READ, an I/O request against the source file is completed. I then call the CIOReq's *Write* method to issue a write I/O request against the destination file. If *CompKey* is CK_WRITE, an I/O request against the destination file is completed. If I haven't read beyond the end of the source file, I call CIOReq's *Read* method to continue reading the source file.

When there are no more outstanding I/O requests, the loop terminates and cleans up by closing the source and destination file handles. Before *FileCopy* returns, it must do one more task: it must fix the size of the destination file so that it is the same size as the source file. To do this, I reopen the destination file without specifying the FILE_FLAG_NO_BUFFERING flag. Because I am not using this flag, file operations do not have to be performed on sector boundaries. This allows me to shrink the size of the destination file to the same size as the source file.

Listing 2-1. *The FileCopy sample application*

FileCopy.cpp

```
/******************************************************************************
Module: FileCopy.cpp Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
#include "..\CmnHdr.h" // See Appendix A. #include <WindowsX.h> #include "..\ClassLib\IOCP.h" //
See Appendix B. #include "..\ClassLib\EnsureCleanup.h" // See Appendix B. #include "Resource.h"
////////////////////////////////////////////////////////////////////////////// // Each I/O Request needs an OVERLAPPED
structure and a data buffer class CIOReq : public OVERLAPPED { public: CIOReq() { Internal =
InternalHigh = 0; Offset = OffsetHigh = 0; hEvent = NULL; m_nBuffSize = 0; m_pvData = NULL; }
~CIOReq() { if (m_pvData != NULL) VirtualFree(m_pvData, 0, MEM_RELEASE); } BOOL
AllocBuffer(SIZE_T nBuffSize) { m_nBuffSize = nBuffSize; m_pvData = VirtualAlloc(NULL,
m_nBuffSize, MEM_COMMIT, PAGE_READWRITE); return(m_pvData != NULL); } BOOL
Read(HANDLE hDevice, PLARGE_INTEGER pliOffset = NULL) { if (pliOffset != NULL) { Offset
= pliOffset->LowPart; OffsetHigh = pliOffset->HighPart; } return(::ReadFile(hDevice, m_pvData,
m_nBuffSize, NULL, this)); } BOOL Write(HANDLE hDevice, PLARGE_INTEGER pliOffset =
NULL) { if (pliOffset != NULL) { Offset = pliOffset->LowPart; OffsetHigh = pliOffset->HighPart; }
return(::WriteFile(hDevice, m_pvData, m_nBuffSize, NULL, this)); } private: SIZE_T m_nBuffSize;
PVOID m_pvData; }; ////////////////////////////////////////////////////////////////////////////// #define BUFFSIZE (64
* 1024) // The size of an I/O buffer #define MAX_PENDING_IO_REQS 4 // The maximum # of of
I/Os // The completion key values indicate the type of completed I/O. #define CK_READ 1 #define
CK_WRITE 2 ////////////////////////////////////////////////////////////////////////////// BOOL FileCopy(PCTSTR
pszFileSrc, PCTSTR pszFileDst) { BOOL fOk = FALSE; // Assume file copy fails
LARGE_INTEGER liFileSizeSrc = { 0 }, liFileSizeDst; try { { // Open the source file without
buffering & get its size CEnsureCloseFile hfileSrc = CreateFile(pszFileSrc, GENERIC_READ,
FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_FLAG_NO_BUFFERING |
FILE_FLAG_OVERLAPPED, NULL); if (hfileSrc.IsInvalid()) goto leave; // Get the file's size
GetFileSizeEx(hfileSrc, &liFileSizeSrc); // Non-buffered I/O requires sector-sized transfers. // I'll use
buffer-size transfers since it's easier to calculate. liFileSizeDst.QuadPart =
chROUNDUP(liFileSizeSrc.QuadPart, BUFFSIZE); // Open the destination file without buffering &
set its size CEnsureCloseFile hfileDst = CreateFile(pszFileDst, GENERIC_WRITE, 0, NULL,
CREATE_ALWAYS, FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, hfileSrc); if
(hfileDst.IsInvalid()) goto leave; // File systems extend files synchronously. Extend the destination file
// now so that I/Os execute asynchronously improving performance. SetFilePointerEx(hfileDst,
liFileSizeDst, NULL, FILE_BEGIN); SetEndOfFile(hfileDst); // Create an I/O completion port and
associate the files with it. CIOCP iocp(0); iocp.AssociateDevice(hfileSrc, CK_READ); // Read from
source file iocp.AssociateDevice(hfileDst, CK_WRITE); // Write to destination file // Initialize
```

```
record-keeping variables CIOReq ior[MAX_PENDING_IO_REQS]; LARGE_INTEGER
liNextReadOffset = { 0 }; int nReadsInProgress = 0; int nWritesInProgress = 0; // Prime the file copy
engine by simulating that writes have completed. // This causes read operations to be issued. for (int
nIOReq = 0; nIOReq < chDIMOF(ior); nIOReq++) { // Each I/O request requires a data buffer for
transfers chVERIFY(ior[nIOReq].AllocBuffer(BUFFSIZE)); nWritesInProgress++;
iocp.PostStatus(CK_WRITE, 0, &ior[nIOReq]); } // Loop while outstanding I/O requests still exist
while ((nReadsInProgress > 0) || (nWritesInProgress > 0)) { // Suspend the thread until an I/O
completes ULONG_PTR CompKey; DWORD dwNumBytes; CIOReq* pior;
iocp.GetStatus(&CompKey, &dwNumBytes, (OVERLAPPED**) &pior, INFINITE); switch
(CompKey) { case CK_READ: // Read completed, write to destination nReadsInProgress--;
pior->Write(hfileDst); // Write to same offset read from source nWritesInProgress++; break; case
CK_WRITE: // Write completed, read from source nWritesInProgress--; if
(liNextReadOffset.QuadPart < liFileSizeDst.QuadPart) { // Not EOF, read the next block of data from
the source file. pior->Read(hfileSrc, &liNextReadOffset); nReadsInProgress++;
liNextReadOffset.QuadPart += BUFFSIZE; // Advance source offset } break; } } fOk = TRUE; }
leave:; } catch (...) { } if (fOk) { // The destination file size is a multiple of the page size. Open the //
file WITH buffering to shrink its size to the source file's size. CEnsureCloseFile hfileDst =
CreateFile(pszFileDst, GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL); if
(hfileDst.IsValid()) { SetFilePointerEx(hfileDst, liFileSizeSrc, NULL, FILE_BEGIN);
SetEndOfFile(hfileDst); } } return(fOk); } ////////////////////////////////////////////////////////////////////////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
chSETDLGICONS(hwnd, IDI_FILECOPY); // Disable Copy button since no file is selected yet.
EnableWindow(GetDlgItem(hwnd, IDOK), FALSE); return(TRUE); }
//////////////////////////////////////////////////////////////////////////// void Dlg_OnCommand(HWND hwnd, int id,
HWND hwndCtl, UINT codeNotify) { TCHAR szPathname[_MAX_PATH]; switch (id) { case
IDCANCEL: EndDialog(hwnd, id); break; case IDOK: // Copy the source file to the destination file.
Static_GetText(GetDlgItem(hwnd, IDC_SRCFILE), szPathname, sizeof(szPathname));
SetCursor(LoadCursor(NULL, IDC_WAIT)); chMB(FileCopy(szPathname, TEXT("FileCopy.cpy"))
? "File Copy Successful" : "File Copy Failed"); break; case IDC_PATHNAME: OPENFILENAME
ofn = { OPENFILENAME_SIZE_VERSION_400 }; ofn.hwndOwner = hwnd; ofn.lpstrFilter =
TEXT("*.*\0"); lstrcpy(szPathname, TEXT("*.*")); ofn.lpstrFile = szPathname; ofn.nMaxFile =
chDIMOF(szPathname); ofn.lpstrTitle = TEXT("Select file to copy"); ofn.Flags = OFN_EXPLORER |
OFN_FILEMUSTEXIST; BOOL fOk = GetOpenFileName(&ofn); if (fOk) { // Show user the source
file's size Static_SetText(GetDlgItem(hwnd, IDC_SRCFILE), szPathname); CEnsureCloseFile hfile =
CreateFile(szPathname, 0, 0, NULL, OPEN_EXISTING, 0, NULL); if (hfile.IsValid()) {
LARGE_INTEGER liFileSize; GetFileSizeEx(hfile, &liFileSize); // NOTE: Only shows bottom
32-bits of size SetDlgItemInt(hwnd, IDC_SRCFILESIZE, liFileSize.LowPart, FALSE); } }
EnableWindow(GetDlgItem(hwnd, IDOK), fOk); break; } }
//////////////////////////////////////////////////////////////////////////// INT_PTR WINAPI Dlg_Proc(HWND hwnd,
UINT uMsg, WPARAM wParam, LPARAM lParam) { switch (uMsg) {
chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
chHANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand); } return(FALSE); }
//////////////////////////////////////////////////////////////////////////// int WINAPI _tWinMain(HINSTANCE
hinstExe, HINSTANCE, PTSTR pszCmdLine, int) { DialogBox(hinstExe,
MAKEINTRESOURCE(IDD_FILECOPY), NULL, Dlg_Proc); return(0); } ////////////////////////////////
End of File ////////////////////////////////
```

[Previous] [Next]

Chapter 3

Service Applications

The Microsoft Windows operating system offers various facilities that make implementing the server-side portion of a client-server application easier. As you know, a server is a Windows application that performs server-side duties. Microsoft recommends that server applications be implemented as *services*. In fact, the Microsoft BackOffice Logo program *requires* that server applications be implemented as services.

A service is a normal Windows application containing additional infrastructure that enables it to receive special treatment by the operating system—for example, the ability to be remotely administered, allowing an administrator to start or stop the application from a remote machine. By turning your server application into a service, you'll get this and other features for free. This chapter describes what a service application is, how to design a service application, and what additional facilities the operating system offers to services.

Out of the box, Windows includes a number of services. Table 3-1 lists some of the services installed on my machine running Microsoft Windows 2000 and the name of the executable files containing the code for the services.

Table 3-1. *Services and associated executable files*

Service Name	Description	Executable Name
Alerter	Notifies users and computers of administrative alerts	Services.exe
ClipBook	Allows pages to be seen by remote ClipBook viewers	ClipSrv.exe
Computer Browser	Maintains a list of computers on your network	Services.exe
DHCP Client	Registers and updates IP addresses and DNS names	Services.exe
Distributed Transaction Coordinator	Coordinates transactions distributed across two or more databases, message queues, file systems, or other transaction-protected resource managers	MSDTC.exe
Event Log	Logs event messages issued by programs and Windows	Services.exe
Messenger	Sends and receives messages transmitted by administrators or the Alerter service	Services.exe
Net Logon	Supports pass-through authentication of account logon events for computers in a domain	LSASS.exe
Plug and Play	Manages device installation and configuration, and notifies programs of device changes	Services.exe
Remote Procedure Call (RPC)	Provides the endpoint mapper and other miscellaneous RPC services	SvcHost.exe
Remote Procedure Call (RPC) Locator	Manages the RPC name service database	Locator.exe
Server	Provides RPC support and file, print, and named pipe sharing	Services.exe
Task Scheduler	Enables a program to run at a designated time	MSTask.exe
Telephony	Provides Telephony API (TAPI) support	SvcHost.exe

Uninterruptible Power Supply	Manages an uninterruptible power supply (UPS) connected to the computer	UPS.exe
Windows Installer	Installs, repairs, and removes software according to instructions contained in .msi files	MSIExec.exe
Windows Management Instrumentation	Provides system management information	WinMgmt.exe
Workstation	Provides network connections and communications	Services.exe

Microsoft produces many more services than the ones I listed in Table 3-1: Microsoft Exchange Server, Microsoft Merchant Server, and Microsoft SQL Server, to name a few. All these server applications are implemented as services and are part of Microsoft BackOffice Suite.

First and foremost, a service application is just a 32-bit or 64-bit executable, so everything you already know about DLLs, structured exception handling, memory-mapped files, virtual memory, device I/O, thread-local storage, thread synchronization, Unicode, and other Windows facilities is available to a service. And this means that converting an existing server application into a service should be relatively easy and straightforward for you.

NOTE

The Microsoft Windows 2000 Resource Kit contains a utility named SrvAny.exe that allows an existing application to be started remotely, just like a true service. However, SrvAny does not allow an application to be remotely administered in any other way, and therefore should be used as a short-term solution only. You are strongly encouraged to modify your application's code to turn it into a full-fledged service and ignore the SrvAny utility altogether.

Second, you need to know that a service should have absolutely no user interface. Most services run on a server machine locked away in a closet somewhere. So if your service presented any user interface elements, such as message boxes, no user would be in front of the machine to see and then dismiss them. And, as you'll see later in this chapter, any windows created would probably appear on a window station or desktop different from the one the user was sitting in front of, and thus the message wouldn't be visible to the user anyway. Because a service won't have a user interface, it doesn't matter whether you choose to implement your service as a graphical user interface (GUI) application (with *(w)WinMain* as its entry point) or as a console user interface (CUI) application (with *(w)main* as its entry point).

If a service is not supposed to present any user interface, how do you configure the service? How can you start and stop a service? How can the service issue warnings or error messages? How can the service report statistical data about its performance? The answer to all these questions is that a service can be remotely administered. Windows offers a number of administrative tools that allow a service to be managed from other machines connected on the network so that it is not necessary for someone to physically check (or even have physical access to) the computer running the service. You are probably already familiar with many of these tools: the Microsoft Management Console (MMC), with its Services, Event Viewer, and System Monitor snap-ins; the registry editor; and the Net.exe command-line tool.

These facilities are provided by Windows to simplify the development effort of the service writer. They also give an administrator a consistent way to manage machines remotely and locally. Note that these facilities are not exclusive to services: any application (or device driver) can take advantage of them. These facilities are discussed throughout the chapters of this book.

[Previous] [Next]

The Windows Service Communication Architecture

Three types of components are involved in making services work:

- **Service Control Manager (SCM, pronounced *scum*)** Each Windows 2000 system ships with a component called the Service Control Manager. This component lives in the Services.exe file; it is automatically invoked when the operating system boots, and terminates when the system is shut down. The SCM runs with system privileges and provides a unified and secure means of controlling service applications. The SCM is responsible for communicating with the various services, telling them to start, stop, pause, continue, and so on.

- **Service application** A service is simply an application that contains the infrastructure necessary for communicating with the SCM, which sends commands to the service, telling it to start, stop, pause, continue, or shut down. A service also calls special functions that communicate its status back to the SCM.

- **Service Control Program (SCP)** This is an application that usually presents a user interface that allows a user to start, stop, pause, continue, and otherwise control all the services installed on a machine. The service control program calls special Windows functions that let it talk to the SCM.

Figure 3-1 shows how all these components communicate with one another. Notice that SCP applications do not communicate with services directly; all communication goes through the SCM. This architecture is precisely what makes the remote administration transparent to the SCP and service applications. It is possible to implement an architecture and a protocol that enables your SCP application to talk directly with your service application, but you must write the communication code yourself.

Figure 3-1. *Windows service communication architecture*

Of these three components, you will never implement the SCM itself. Microsoft implements the SCM and packages it into every version of Windows 2000. What you will implement are services and SCPs. This chapter will cover what you need to know to design and implement a service, and the next chapter will cover the details of writing an SCP.

[Previous] [Next]

Service Control Programs that Ship with Windows

Before I delve into how to write a service, you must at least know how an SCP can control a service. So I'll start off by examining a few SCP applications that ship with Windows.

Services Snap-In

The SCP application with which you should become most familiar is the Services snap-in, shown in Figure 3-2. This snap-in shows the list of all services installed on the target machine. The Name and Description columns identify each service's name and offer an informative description of the service's function. The Status column indicates whether the service is Started, Paused, or Stopped (blank entries indicate "stopped"). The Startup Type column indicates when the SCM should invoke the service, and the Log On As column indicates the security context used by the service when it is running.

This information is kept in the SCM's database, which lives inside the registry under the following subkey:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services
```

You should never access this subkey directly; instead, an SCP should call the Windows functions (discussed in the next chapter) that manipulate the database in this subkey. Directly modifying the contents of this key will yield unpredictable results. When you install a product that includes a service, the setup program for that product is an SCP that adds the service's information to the SCM's database.

You can view a remote SCM's service database by selecting the Computer Management node in the left pane of the Computer Management console and then choosing Connect To Another Computer from the Action menu.

Figure 3-2. *Services snap-in*

NOTE

All services that ship with Windows log on as the LocalSystem security context. This is a highly privileged account, and it is strongly recommended that any services you write also use the LocalSystem account.

So now that you're looking at the Services snap-in, you're probably wondering about all the tasks you can perform with it. Here are the most common operations:

- **Start a service** The administrator starts a service by selecting the service from the list and clicking the Start toolbar button. Only services with a Startup Type of Automatic or Manual can be started; disabled services cannot. Disabling a service is useful for troubleshooting problems with a machine.

- **Stop a service** The administrator stops a service by selecting the service and clicking the Stop toolbar button. Note that some services do not allow themselves to be stopped after they are started. The Event Log service is an example; it stops only when the machine shuts down.

- **Pause and resume a service** The administrator pauses a service by selecting a running service and clicking the Pause toolbar button. Note that most services do not allow themselves to be paused. Also note that "pause" has no exact definition. For one service, pausing can mean that the service won't accept client requests until it finishes processing the outstanding requests. For another service, pausing can mean that the service can no longer process any of its operations. Paused services can be resumed by clicking the Start toolbar button.

- **Restart a service** The administrator restarts a service by selecting a running or paused service and clicking the Restart toolbar button. Restarting a service causes the snap-in to stop the service and then start the service. This is simply a convenience feature and is very useful when debugging your own service.

The preceding list certainly accounts for 99 percent of what administrators do with the Services snap-in, but the snap-in can also be used to reconfigure a service. To change a service's settings, you select the service and then display its Properties dialog box. This dialog box contains four tabs; each tab allows the administrator to reconfigure parts of the selected service. The configurable settings are discussed in the following sections.

General Properties

The General tab (shown in Figure 3-3) allows the administrator to examine and reconfigure general information about the service. The first fact you need to know is that each service goes by two string names: an internal name (used for programmatic purposes) and a display name (a pretty string presented to administrators and users). After being added to the machine's service database, a service's internal name cannot be altered, but the administrator can modify the service's display name and description. The General tab also shows the service's pathname but does not allow the administrator to change it. (This is a limitation imposed by the tab, not by the system.) The administrator can change the service's Startup Type to one of the following:

- **Automatic** One of the features of a service is that the SCM can automatically start it for you. If the service has a Startup Type of Automatic, the SCM spawns the service when the operating system boots. It is important to note that automatic services run before any user interactively logs on to the machine. In fact, many machines that run Windows are set up to run only services&"temp0032.html">next chapter.

- **Disabled** A disabled service tells the SCM not to start it under any circumstance. You disable the DHCP Client service when you manually assign an IP address to your machine rather than have it dynamically obtain an IP address from a machine running the DHCP Server service. Disabling a service is also quite useful when troubleshooting a system by allowing you to take a specific service out of the equation.

Figure 3-3. *General tab for the Windows Installer service*

Log On Properties

In addition to configuring the actual service, the administrator can reconfigure the security context under which the service will execute on the Log On tab, shown in Figure 3-4. The security context can be one of the following:

- **LocalSystem Account** A service running under the LocalSystem account can do just about anything on the computer: open any file, shut down the machine, change the system time, and so on. A service running under the LocalSystem account can optionally be allowed to interact with the desktop. Most services don't require this option, and you are strongly discouraged from using it.

- **This Account** A service can also execute under a specific security context (identified by a user's name and password). This restricts the service to accessing the resources accessible to the specified account.

I'll talk more about LocalSystem and user accounts in the "Service Issues" section later in this chapter.

The Log On tab also allows the administrator to specify which hardware profiles the service is enabled in. Hardware profiles allow you to configure services according to your hardware configuration. For example, you might want the fax service to run when your laptop computer is docked and not run when it is undocked.

Figure 3-4. *Log On tab for the Distributed Link Tracking service*

Recovery Properties

The Recovery tab, shown in Figure 3-5, allows the administrator to tell the SCM what actions to perform should the service terminate abnormally. Abnormal termination means that the service stopped without reporting a status of SERVICE_STOPPED (discussed later in this chapter). For the first, second, and subsequent attempts, the SCM can do nothing, automatically restart the service, run an executable, or reboot the computer. Note that running an executable and rebooting the computer can fail if the account under which the service is running doesn't have the appropriate privileges or permissions.

Figure 3-5. *Recovery tab for the Fax Service service*

Dependencies Properties

The Dependencies tab, shown in Figure 3-6, shows the services on which the selected service depends and also what services depend on the selected service. In the figure, you'll see six services dependent on the Workstation service. If the administrator attempts to stop the Workstation service and any dependent services are running, the SCM fails the call. Many SCP programs are written to notify the user that dependent services are running, and to allow the user to choose whether to also stop the dependent services. The Dependencies tab does not allow an administrator to modify any of these dependencies. (I'll discuss service dependencies more in the next chapter.)

Figure 3-6. *Dependencies tab for the Workstation service*

Net.exe and SC.exe

In addition to the Services snap-in, Windows ships with a command-line SCP tool named Net.exe. This tool is limited in that it allows you to control only those services residing on the local machine. Using Net.exe, you can start, stop, pause, and continue services using the following syntax:

```
NET START    servicename
NET STOP     servicename
NET PAUSE    servicename
NET CONTINUE servicename
```

You can also use Net.exe to display a list of services running on the local machine by simply typing the following, without specifying a *servicename*:

```
NET START
```

For debugging, the Net.exe tool is quite handy because you can place calls to it in a batch file or other script file.

Another SCP application that Microsoft offers is a command-line tool named SC.exe. This tool ships as part of the Microsoft Windows 2000 Resource Kit. Running this tool without passing it any parameters displays its usage syntax, as shown here:

```
DESCRIPTION:
   SC is a command line program used for communicating with the
   NT Service Controller and services.
USAGE:
   sc <server> [command] [service name] <option1> <option2>...
```

```
   The option <server> has the form \\ServerName
   Further help on commands can be obtained by typing: "sc [command]"
   Commands:
     query-----------Queries the status for a service, or
                     enumerates the status for types of services.
     queryex---------Queries the extended status for a service, or
                     enumerates the status for types of services.
     start-----------Starts a service.
     pause-----------Sends a PAUSE control request to a service.
     interrogate-----Sends an INTERROGATE control request to a service.
     continue--------Sends a CONTINUE control request to a service.
     stop------------Sends a STOP request to a service.
     config----------Changes the configuration of a service (persistant)
     description-----Changes the description of a service.
     failure---------Changes the actions taken by a service upon failure
     qc--------------Queries the configuration information for a service
     qdescription----Queries the description for a service.
     qfailure--------Queries the actions taken by a service upon failure
     delete----------Deletes a service (from the registry).
     create----------Creates a service. (adds it to the registry).
     control---------Sends a control to a service.
     sdshow----------Displays a service's security descriptor.
     sdset-----------Sets a service's security descriptor.
     GetDisplayName--Gets the DisplayName for a service.
     GetKeyName------Gets the ServiceKeyName for a service.
     EnumDepend------Enumerates Service Dependencies.

   The following commands don't require a service name:
   sc <server> <command> <option>
     boot------------(ok | bad) Indicates whether the last boot should
                     be saved as the last-known-good boot configuration
     Lock------------Locks the Service Database
     QueryLock-------Queries the LockStatus for the SCManager Database
EXAMPLE:
   sc start MyService

Would you like to see help for the QUERY and QUERYEX commands?
[ y | n ]: y
QUERY and QUERYEX OPTIONS :
   If the query command is followed by a service name, the status
   for that service is returned.  Further options do not apply in
   this case.  If the query command is followed by nothing or one of
   the options listed below, the services are enumerated.
   type=    Type of services to enumerate (driver, service, all)
            (default = service)
   state=   State of services to enumerate (inactive, all)
            (default = active)
   bufsize= The size (in bytes) of the enumeration buffer
            (default = 1024)
   ri=      The resume index number at which to begin the enumeration
            (default = 0)
   group=   Service group to enumerate
            (default = all groups)
SYNTAX EXAMPLES
sc query                - Enumerates status for active services &
                          drivers
sc query messenger      - Displays status for the messenger service
sc queryex messenger    - Displays extended status for the messenger
                          service
sc query type= driver   - Enumerates only active drivers
sc query type= service  - Enumerates only Win32 services
sc query state= all     - Enumerates all services & drivers
sc query bufsize= 50    - Enumerates with a 50 byte buffer.
sc query ri= 14         - Enumerates with resume index = 14
sc queryex group= ""    - Enumerates active services not in a group
```

```
sc query type= service type= interact - Enumerates all interactive
                                        services
sc query type= driver group= NDIS     - Enumerates all NDIS drivers
```

While developing and debugging a service, this tool can help tremendously since it offers a rich command-line interface to all the service control options and can easily be used in a script file.

[Previous] [Next]

The Windows Service Application Architecture

In this section, I explain the additional infrastructure that turns a server application into a service, thus allowing your application to be remotely administered. I've found Microsoft's service architecture to be a little difficult to understand at first. The difficulty is due to the fact that every service process always contains at least two threads, and these threads must communicate with one another. So you have to deal with thread synchronization issues and interthread communication issues.

Another issue you need to consider is that a single executable file can contain several services. If you look back at Table 3-1, you'll see that many services are contained inside the Services.exe file. Most of these services (such as DHCP Client, Messenger, and Alerter) are fairly simple in their implementation. It would be very inefficient if each of these services had to run as a separate process, with its own address space and additional process overhead. Because of this overhead, Microsoft allows a single executable to contain several services. The Services.exe file actually contains about 20 different services inside of it, including the three just mentioned.

When designing a service executable, you must concern yourself with three kinds of functions:

- **Process's entry-point function** This function is your standard *(w)main* or *(w)WinMain* function with which you should be extremely familiar by now. For a service, this function initializes the process as a whole and then calls a special function that connects the process with the local system's SCM. At this point, the SCM takes control of your primary thread for its own purposes. Your code will regain control only when all of the services in the executable have stopped running.

- **Service's *ServiceMain* function** You must implement a *ServiceMain* function for each service contained inside your executable file. To run a service, the SCM spawns a thread in your process that executes your *ServiceMain* function. When the thread returns from *ServiceMain*, the SCM considers the service stopped. Note that this function does not have to be called *ServiceMain*; you can give it any name you desire.

- **Service's *HandlerEx* function** Each service must have a *HandlerEx* function associated with it. The SCM communicates with the service by calling the service's *HandlerEx* function. The code in the *HandlerEx* function is executed by your process's primary thread. The *HandlerEx* function either executes the necessary action, or it must communicate the SCM's instructions to the thread that is executing the service's *ServiceMain* function by using some form of interthread communication. Note that each service can have its own *HandlerEx* function, or multiple services (in a single executable) can share a single *HandlerEx* function. One of the parameters passed to the *HandlerEx* function indicates which service the SCM wishes to communicate with. Note that this function does not have to be called *HandlerEx*; you can give it any name you desire.

Figure 3-7 will help you put this architecture in perspective. It shows the functions necessary to implement a service executable that houses two services as well as the lines of interprocess communication (IPC) and interthread communication (ITC). In the upcoming sections, I will examine these three functions in detail and flesh out exactly what their responsibilities are. I recommend that you refer to this figure while reading.

Figure 3-7. *Windows service application architecture*

The Process Entry-Point Function: *(w)main* or *(w)WinMain*

When an administrator wants to start a service, the SCM determines whether or not the executable file containing the service is already running. If it is not running, the SCM spawns the executable file. The process's primary thread is responsible for performing the process-wide initialization. (Service-specific initialization should be done in the appropriate service's *ServiceMain* function.)

After the process is initialized, the entry-point function must contact the SCM, which now takes over control of the process. To contact the SCM, the entry-point function must first allocate and initialize an array of SERVICE_TABLE_ENTRY structures:

```
typedef struct _SERVICE_TABLE_ENTRY {
   PTSTR                    lpServiceName;   // Service's internal name
   LPSERVICE_MAIN_FUNCTION lpServiceProc;   // Service's ServiceMain
} SERVICE_TABLE_ENTRY, *LPSERVICE_TABLE_ENTRY;
```

The first member indicates the internal, programmatic name of the service, and the second member is the address of the service's *ServiceMain* callback function. If the executable houses just one service, the array of SERVICE_TABLE_ENTRY structures must be initialized as follows:

```
SERVICE_TABLE_ENTRY ServiceTable[] = {
   { TEXT("ServiceName1"), ServiceMain1 },
```

```
   { NULL, NULL }   // Marks end of array
};
```

If your executable contains three services, you must initialize the array like this:

```
SERVICE_TABLE_ENTRY ServiceTable[] = {
   { TEXT("ServiceName1"), ServiceMain1 },
   { TEXT("ServiceName2"), ServiceMain2 },
   { TEXT("ServiceName3"), ServiceMain3 },
   { NULL, NULL }   // Marks end of array
};
```

The last structure in the array must have both members set to NULL to indicate the end of the array. Now the process connects itself to the SCM by calling *StartServiceCtrlDispatcher*:

```
BOOL StartServiceCtrlDispatcher(
   CONST SERVICE_TABLE_ENTRY* pServiceTable);
```

Calling this function and passing in the address of the service table array is how the executable process indicates which services are contained within the process. At this point, the SCM knows which service it was trying to start and iterates through the array looking for it. Once the service is found, a thread is created and begins executing the service's *ServiceMain* function (whose address is obtained from the SERVICE_TABLE_ENTRY array).

NOTE

The SCM keeps close tabs on how a service is doing. For example, when the SCM spawns a service executable, the SCM waits for the primary thread in the executable to call *StartServiceCtrlDispatcher*. If *StartServiceCtrlDispatcher* is not called within 30 seconds, the SCM thinks that the service executable is malfunctioning and calls *TerminateProcess* to forcibly kill the process. For this reason, if your process requires more than 30 seconds to initialize, you must spawn another thread to handle the initialization so that the primary thread can quickly call *StartServiceCtrlDispatcher*. Note that I'm discussing process-wide initialization here. Individual services should initialize themselves using their own *ServiceMain* functions.

StartServiceCtrlDispatcher does not return until all services in the executable have stopped running. While at least one service is running, the SCM controls what the process's primary thread executes. Usually, this thread has nothing to do and just sleeps, not wasting precious CPU time. If the administrator attempts to start another service implemented in the same executable, the SCM does not spawn another instance of the executable. Instead, the SCM communicates to the executable's primary thread and has it iterate the list of services again, this time looking for the service that is being started. Once found, a new thread is spawned, which executes the appropriate service's *ServiceMain* function.

NOTE

When determining whether to spawn a new service process or a new thread in an existing service process, the SCM performs a strict comparison of the service pathname strings. For example, say that two services are implemented in a single executable file, MyServices.exe. The first service is added to the SCM's database using an executable pathname of "%windir%\System32\MyService.exe", but the second service is added to the database using "C:\WinNT\System32\MyService.exe". If both services are started, the SCM will spawn two separate processes, both of them running the same MyService.exe service application. To ensure the SCM uses a single process for all services in a single executable file, use the same pathname string when adding the services to the SCM's database.

Internally, the system is keeping track of which services within the process are executing. When each service exits (usually because the *ServiceMain* function returns), the system checks to see whether any services are still running. If no services are running, then and only then does the entry-point function's call to *StartServiceCtrlDispatcher* return. Your code should perform any process-wide cleanup, and then the entry-point function should return, causing the process to terminate. Note that you must complete your clean-up code in 30 seconds or the SCM kills the process.

The *ServiceMain* Function

Each service in the executable file must have its own *ServiceMain* function:

```
VOID WINAPI ServiceMain(
   DWORD  dwArgc,
   PTSTR* pszArgv);
```

The SCM starts a service by creating a new thread; this thread begins its execution with the *ServiceMain* function. As I mentioned earlier, I call this particular function *ServiceMain*, but the function can have any name you choose. The name you choose for the function is not important because you pass its address in the SERVICE_TABLE_ENTRY's *lpServiceProc* member. However, you can't have two *ServiceMain* functions with the same name in a single executable file; if you do, the compiler or linker will generate an error when you try to build your project.

Two parameters are passed to a *ServiceMain* function. These parameters create a mechanism that allows an administrator to start a service with some command-line parameters via the *StartService* function (discussed in the next chapter). Personally, I don't know of any service that references these parameters, and I encourage you to ignore them. Having a service configure itself by reading settings out of the following registry subkey is better than using parameters passed to *ServiceMain*. (The *ServiceName* portion of the key should be replaced with the actual name of the service.)

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\ServiceName\Parameters
```

Many services ship with a client application that allows an administrator to configure the service's settings. The client application simply saves these settings in the registry subkey. When the service starts, it retrieves the settings from the registry.

If a service is running when its configuration changes, three options are available to it:

- The service ignores the changed configuration settings until the next time the service starts. This is the simplest choice, and many services existing today have taken this approach.
- The service can be explicitly told that it should reconfigure itself. An SCP does this by calling the *ServiceControl* function, passing the SERVICE_CONTROL_PARAMCHANGE value. Chapter 4 describes how to do this.
- The service can call the *RegNotifyChangeKeyValue* function to receive a notification when an external application has changed its registry settings. This allows a service to reconfigure itself on the fly. The RegNotify sample application in Chapter 5 shows how to accomplish this.

The first task that the *ServiceMain* function must perform is telling the SCM the address of the service's *HandlerEx* callback function. It does this by calling *RegisterServiceCtrlHandlerEx*:

```
SERVICE_STATUS_HANDLE RegisterServiceCtrlHandlerEx(
   PCTSTR                pszServiceName, // Service's internal name
   LPHANDLER_FUNCTION_EX pfnHandler,     // Service's HandlerEx function
   PVOID                 pvContext);     // User-defined value
```

The first parameter indicates the service for which you are setting a *HandlerEx* function, and the second parameter is the address of the *HandlerEx* function. The *pszServiceName* parameter must match the name used when the array of SERVICE_TABLE_ENTRYs was initialized and passed to *StartServiceCtrlDispatcher*. The last parameter, *pvContext*, is a user-defined value that is passed to the service's *HandlerEx* function. I'll discuss the *HandlerEx* function in the next section.

RegisterServiceCtrlHandlerEx returns a SERVICE_STATUS_HANDLE, which is a value that uniquely identifies the service to the SCM. All future communication from the service to the SCM will require this handle instead of the service's internal string name.

> **NOTE**
>
> Unlike most handles in the system, the handle returned from *RegisterServiceCtrlHandlerEx* is never closed by you.

After *RegisterServiceCtrlHandlerEx* returns, the *ServiceMain* thread should immediately tell the SCM that the service is continuing to initialize. It does this by calling the *SetServiceStatus* function:

```
BOOL SetServiceStatus(
   SERVICE_STATUS_HANDLE hService,
   LPSERVICE_STATUS      pServiceStatus);
```

This function requires that you pass it the handle identifying your service (which is returned from the call to *RegisterServiceCtrlHandlerEx*) and the address of an initialized SERVICE_STATUS structure:

```
typedef struct _SERVICE_STATUS {
   DWORD dwServiceType;
   DWORD dwCurrentState;
   DWORD dwControlsAccepted;
   DWORD dwWin32ExitCode;
   DWORD dwServiceSpecificExitCode;
   DWORD dwCheckPoint;
   DWORD dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

The SERVICE_STATUS structure contains seven members that reflect the current status of the service. All of these members, described in the following list, must be set correctly before you pass the structure to *SetServiceStatus*.

- ***dwServiceType*** This member indicates what type of service executable you have implemented. Set this member to SERVICE_WIN32_OWN_PROCESS when your executable houses a single service, or to SERVICE_WIN32_SHARE_PROCESS when your executable houses two or more services. In addition to these two flags, you can OR in the SERVICE_INTERACTIVE_PROCESS flag when your service needs to interact with the desktop. (You should avoid interactive services as much as possible.) The value of *dwServiceType* should never change during the lifetime of your service.

- ***dwCurrentState*** This member is the most important member of the SERVICE_STATUS structure. It tells the SCM the current state of your service. To report that your service is still initializing, you should set this member to SERVICE_START_PENDING. I'll explain the other possible values when we talk about the *HandlerEx* function in the section "Codes Requiring Status Reporting."

- ***dwControlsAccepted*** This member indicates what control notifications the service is willing to accept. If you allow a service control program to pause and continue your service, specify SERVICE_ACCEPT_PAUSE_CONTINUE. Many services do not support pausing and continuing; you have to decide if this functionality makes sense for your service. If you allow a service control program to stop your service, specify SERVICE_ACCEPT_STOP. If you want your service to be notified when the operating system is being shut down, specify SERVICE_ACCEPT_SHUTDOWN.

You can also indicate whether you want to receive parameter change, hardware profile change, and power event notifications by specifying SERVICE_ACCEPT_PARAMCHANGE, SERVICE_ACCEPT_HARDWAREPROFILECHANGE, or SERVICE_ACCEPT_POWEREVENT, respectively.

Use the OR operator to combine the desired set of flags. Note that your service can change the controls it accepts while it is running. For example, I have written services that allowed themselves to be paused as long as no clients were connected to them.

- ***dwWin32ExitCode and dwServiceSpecificExitCode*** These two members allow the service to report error codes. If a service wants to report a Win32 error code (as defined in WinError.h), it sets the *dwWin32ExitCode* member to the desired code. A service can also report errors that are specific to the service and do not map to a predefined Win32 error code. To make the service do this, you must set the *dwWin32ExitCode* member to ERROR_SERVICE_SPECIFIC_ERROR and then set the *dwServiceSpecificExitCode* member to the service-specific error code. Note that a customized SCP would be required to report this error code. Set the *dwWin32ExitCode* member to NO_ERROR when the service is running normally and has no error to report.

- ***dwCheckPoint* and *dwWaitHint*** These members allow a service to report its progress. When you set *dwCurrentState* to SERVICE_START_PENDING, you should set *dwCheckPoint* to 1 and set *dwWaitHint* to the number of milliseconds required for the service to reach its next *SetServiceStatus* call. Once the service is fully initialized, you should re-initialize the SERVICE_STATUS structure's members so that *dwCurrentState* is SERVICE_RUNNING, and then set both *dwCheckPoint* and *dwWaitHint* to 0.

  The *dwCheckPoint* member exists for your own benefit. It allows a service to report how far its processing has progressed. Each time you call *SetServiceStatus*, you should increment *dwCheckPoint* to a number that indicates what "step" your service has executed. It is totally up to you how frequently to report your service's progress. If you do decide to report each step of your service's initialization, the *dwWaitHint* member should be set to indicate how many milliseconds you think you need to reach the next step (checkpoint)—not the number of milliseconds required for the service to complete its processing.

NOTE

The *ServiceMain* function must call *SetServiceStatus* within 80 seconds of starting, or the SCM thinks that the service has failed to start. If no other services are running in the service process, the SCM kills the process.

NOTE

Just before creating your service's thread, the SCM sets your service's status to indicate a current state of START_PENDING, a checkpoint of 0, and a wait hint of 2000 milliseconds. If your *ServiceMain* function requires more than 2000 milliseconds to initialize, the very first time you call *SetServiceStatus*, you should indicate a current state of SERVICE_START_PENDING, a checkpoint of 1, and a wait hint as desired. Notice that the checkpoint should be set to 1. A very common mistake developers make is setting the checkpoint to 0 the first time they call *SetServiceStatus*, which can confuse the administrator's SCP program by causing it to believe the service is not responding properly. If your service requires more initialization, you can continue to report a state of SERVICE_START_PENDING, incrementing the checkpoint and setting the wait hint as desired.

After your service's initialization is complete, your service calls *SetServiceStatus* to indicate SERVICE_RUNNING (with the checkpoint and wait hint both set to 0). Now your service is running. Usually a service runs by placing itself in a loop. Inside the loop, the service thread suspends itself, waiting for a network request or a notification indicating that the service should pause, continue, stop, shut down, and so on. If a network request comes in, the service thread wakes up, processes the request, and loops back around to wait for the next request or notification.

If the service wakes because of a notification, it processes the notification. If the service receives a stop or shutdown notification, the loop is terminated, and the service's *ServiceMain* function returns, killing the thread. If the service is the last service running, the process also terminates.

NOTE

The *SetServiceStatus* function examines the SERVICE_STATUS structure's *dwCurrentState* member. If this member is set to SERVICE_STOPPED, *SetServiceStatus* closes the service's status handle (which is the first parameter to *SetServiceStatus*). This is why you never have to explicitly close the handle returned from *RegisterServiceCtrlHandlerEx* and, even more importantly, why you should never call *SetServiceStatus* after you have called it with a current state of SERVICE_STOPPED. Attempting to do so will raise an invalid handle exception when running the service under a debugger.

The *HandlerEx* Function

Each service in the executable file has to be associated with a *HandlerEx* function:

```
DWORD WINAPI HandlerEx(
   DWORD dwControl,
   DWORD dwEventType,
   PVOID pvEventData,
   PVOID pvContext);
```

I call this function *HandlerEx*, but the function can have any name you choose. The actual name is not important because you pass the function's address as a parameter to the *RegisterServiceCtrlHandlerEx* function. However, you can't have two *HandlerEx* functions with the same name in a single executable file; if you do, the compiler or linker will generate an error when you try to build your project.

Most of the time, the process's primary thread is suspended inside the call to *StartServiceCtrlDispatcher*. When an SCP program wants to control a service, the SCM communicates the control to the process's primary thread. The thread wakes up and calls the appropriate service's *HandlerEx* function. For example, when an administrator uses the Services snap-in to stop a service, the snap-in communicates the administrator's desire to the local or remote SCM. The SCM then wakes the service executable's primary thread, which calls the service's *HandlerEx* function, passing it a SERVICE_CONTROL_STOP control code.

The SCM also sends device, hardware profile, and power event notifications to a service's *HandlerEx* function. These notifications allow the service to reconfigure itself appropriately and to participate in granting or denying system changes.

NOTE

Because the process's primary thread executes every service's *HandlerEx* function, you should implement these *HandlerEx* functions so that they execute quickly. Failure to do so will prevent other services in the same process from receiving their desired actions in a reasonable amount of time.

Since the primary thread executes the *HandlerEx* function but the service is executed by another thread, it might be necessary for *HandlerEx* to communicate actions or notifications to the service thread. There is no standard way to perform this communication; the method really depends on how you implement the service. You can queue an asynchronous procedure call (APC), post an I/O completion status, post a window message, or whatever. I recommend that you choose a queuing mechanism such as the ones just mentioned to avoid synchronizing the *HandlerEx* thread with the *ServiceMain* thread. I always handle this communication by posting an I/O completion status.

The *HandlerEx* function is passed four parameters. The first parameter, *dwControl*, indicates the requested action or notification. If *dwControl* identifies a device, hardware profile, or power event notification, the *dwEventType* and *pvEventData* parameters offer more specific information about the action or notification. The *pvContext* parameter is simply the user-defined value originally passed to the *RegisterServiceCtrlHandlerEx* function. Using this value, you can create a single *HandlerEx* function that is used by all services in a single executable; the *pvContext* value could be used to determine the specific service that the *HandlerEx* function needs to communicate with. In the next section, I'll talk about how to handle these control codes and notifications.

The *HandlerEx* function's return value allows a service's handler to return some information back to the SCM. If the *HandlerEx* function doesn't handle a particular control code, return ERROR_CALL_NOT_IMPLEMENTED. If the *HandlerEx* function handles a device, hardware profile, or power event request, return NO_ERROR. To deny a request, return any other Win32 error code. For any other control codes, the *HandlerEx* function should return NO_ERROR.

[Previous] [Next]

Control Codes and Status Reporting

The *HandlerEx* function is responsible for handling all requested actions of the service and all notifications. *HandlerEx*'s first parameter is a code indicating the action request or notification. Table 3-2 describes the codes that indicate an action request. An action request tells the service to perform some action to alter its execution state.

Table 3-2. *Control codes that indicate an action request*

Control Code	Description
SERVICE_CONTROL_STOP	Requests the service to stop.
SERVICE_CONTROL_PAUSE	Requests the service to pause.
SERVICE_CONTROL_CONTINUE	Requests the paused service to resume.
SERVICE_CONTROL_INTERROGATE	Requests the service to immediately update its current status information to the SCM. This is the only control code that all services must respond to.

Table 3-3 describes the codes that indicate notifications. Notifications inform services of "interesting" events in the system. However, services usually do not alter their execution state in response to notifications (although a service might choose to alter its execution state).

Table 3-3. *Control codes that indicate a notification*

Control Code	Description

SERVICE_CONTROL_PARAMCHANGE	Notifies the service that configuration parameters have changed. A service can ignore this or reconfigure itself while running.
SERVICE_CONTROL_DEVICEEVENT	Notifies the service of a device event. The service must call *RegisterDeviceNotification* to receive these notifications.
SERVICE_CONTROL_HARDWAREPROFILECHANGE	Notifies the service of a hardware profile change.
SERVICE_CONTROL_POWEREVENT	Notifies the service of a power event.
A number between 128 and 255, inclusive	Notifies the service of a user-defined event.

Codes Requiring Status Reporting

The work performed by the *HandlerEx* function differs dramatically depending on the control code it receives. In particular, the action request codes require special attention in your code. When the *HandlerEx* function receives a SERVICE_CONTROL_STOP, SERVICE_CONTROL_SHUTDOWN, SERVICE_CONTROL_PAUSE, or SERVICE_CONTROL_CONTINUE control code, *SetServiceStatus* must be called to acknowledge receipt of the code and to specify how long the service thinks it will take to process the state change. For example, you acknowledge receipt of the control code by setting the SERVICE_STATUS structure's *dwCurrentState* member to SERVICE_STOP_PENDING, SERVICE_PAUSE_PENDING, or SERVICE_CONTINUE_PENDING. In addition, the *HandlerEx* function must return within 30 seconds, or the SCP application will again think that the service has stopped responding. If the SCM thinks that the service has stopped responding, it doesn't kill the service; it just returns failure to the SCP that initiated the service control code.

While a stop, shutdown, pause, or continue operation is pending, you must also specify how long you think the operation will take to complete. Specifying the duration is useful because a service might not be able to change its state immediately—it might have to wait for a network request to complete or for data to be flushed to a drive. You indicate how long the state change will take to complete by using the *dwCheckPoint* and *dwWaitHint* members of the SERVICE_STATUS structure, just as you did when you reported that the service was first starting. If you want, you can report periodic progress by incrementing the *dwCheckPoint* member and setting the *dwWaitHint* member to indicate how long you expect the service to take to get to the next step.

After the service has performed all the actions required to stop, shut down, pause, or continue itself, *SetServiceStatus* should be called again. This time you set the *dwCurrentStatus* member to SERVICE_STOPPED, SERVICE_PAUSED, or SERVICE_RUNNING. When you report any of these three status codes, both the *dwCheckPoint* and *dwWaitHint* members should be 0 because the service has completed its state change.

NOTE

After a service calls *SetServiceStatus* to report SERVICE_STOPPED, the SCM allows that service to run for up to 30 seconds more. If the service is still running after 30 seconds, the SCM terminates the service's process if there are no other services currently running in that process.

When the *HandlerEx* function receives a SERVICE_CONTROL_INTERROGATE control code, the service should simply acknowledge receipt by setting *dwCurrentState* to the service's current state and calling

SetServiceStatus. (Again, set both *dwCheckPoint* and *dwWaitHint* to 0 before making this call to *SetServiceStatus*.)

When the system is shutting down, the *HandlerEx* function receives a SERVICE_CONTROL_SHUTDOWN notification code. The service should perform the minimal set of actions necessary to save any data and should ultimately call *SetServiceStatus* to report SERVICE_STOPPED. To ensure that a machine shuts down in a timely fashion, a service should process this control code only if it absolutely has to. By default, the system gives just 20 seconds for all services to shut down. After 20 seconds, the SCM process (Services.exe) is killed and the machine continues to shut down. This 20-second period is set by the *WaitToKillServiceTimeout* value, which is contained in the following registry subkey:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control
```

NOTE

When the system is shutting down, the SCM notifies all services that accept the SERVICE_CONTROL_SHUTDOWN notification code. Some services might ignore the code, some might save data to disk, some might stop themselves and terminate. You must be very careful not to execute any actions in your service that require the assistance of other services. These other services might be in a "bad" state or might even have terminated. The system completely ignores service dependencies when shutting down. Microsoft's goal was to make the system shut down as quick as possible. In fact, other services might even receive their shutdown notifications before your service receives its notification. The problem with this shutdown notification order is that services you depend on might stop running at any time, and your service must handle this situation gracefully.

For the notification codes listed in Table 3-3, your *HandlerEx* function should handle the notification and return. Do not call the *SetServiceStatus* function unless the notification response forces the service to change its execution state. If the service is going to change its execution state, *SetServiceStatus* should be called to set the *dwCurrentState*, *dwCheckPoint*, and *dwWaitHint* members to the appropriate values as previously discussed.

Dealing with Interthread Communication Issues

A service is awkward to write because although the primary thread executing the *HandlerEx* function receives the action request, often the service thread needs to do the actual work to process the request. For example, you might be writing a service that processes client requests that come in over a named pipe. Your service's thread suspends itself waiting for a client to connect. If your *HandlerEx* thread receives a SERVICE_CONTROL_STOP code, how do you stop the service? I've seen many developers simply call *TerminateThread* from the *HandlerEx* function to kill the service thread forcibly. By now, you should know that *TerminateThread* is one of the worst functions you can possibly call because the thread doesn't get a chance to clean up: the thread's stack is not destroyed, the thread can't release any kernel objects that it might have waited on, DLLs are not notified that the thread has been destroyed, and so on.

The proper way for the service to stop is to somehow wake up, see that it is supposed to stop, clean up properly, and then return from its *ServiceMain* function. To make the service do this, you must implement some form of interthread communication between your *HandlerEx* function and your *ServiceMain* function. You can use any queuing interthread communication mechanism you like, including APC queues, sockets, and window messages. I always use I/O completion ports.

To update its current status, a service must frequently call *SetServiceStatus*. All this status reporting can be another difficult aspect of coding a service. Service implementers often debate about where to place the calls to *SetServiceStatus*. Here are some of the possibilities:

- Have the *HandlerEx* function make the initial call to *SetServiceStatus* to report the pending action, and then use interthread communication to get the code to the *ServiceMain* thread. The *ServiceMain* thread does the work and then uses interthread communication to let the *HandlerEx* function know that the action is complete. At that point, the *HandlerEx* function calls *SetServiceStatus* again to report the service's new execution state.

- Have the *HandlerEx* function use interthread communication to get the code to the *ServiceMain* thread. The *ServiceMain* thread makes the initial call to *SetServiceStatus* to report the pending action, does the work, and then calls *SetServiceStatus* again to report the service's new execution state.

- Have the *HandlerEx* function make the initial call to *SetServiceStatus* to report the pending action, and then use interthread communication to get the code to the *ServiceMain* thread. The *ServiceMain* thread does the work and also calls *SetServiceStatus* again to report the service's new execution state.

All of the above scenarios have pros and cons. I have experimented at great length with all of these possibilities and feel very confident in recommending the last option. Here are my reasons.

First, the SCP calls a function to control a service, and the SCM passes this control to the service. At this point, the SCP is suspended, waiting for the service to call *SetServiceStatus* to indicate that the service has received the control code. If the service's *HandlerEx* function doesn't return within 30 seconds, the SCM allows the SCP to wake, and the SCP's function call to control the service returns failure.

Second, the *HandlerEx* function is executed by the service process's primary thread. (All services in a single process have their *HandlerEx* functions executed by the primary thread.) If the *HandlerEx* function waited for the *ServiceMain* thread to complete the action before returning, any other services in the same process would not be able to receive action requests or notifications. This would make all the other services appear nonresponsive, which is unacceptable (in my opinion).

So I prefer the third method—the *HandlerEx* function makes the initial call to *SetServiceStatus*, interthread communication is used to get the code to the *ServiceMain* thread, and the *ServiceMain* thread performs the work and calls *SetServiceStatus* to report the new execution state. However, this method has a problem: a potential race condition exists. Imagine a service's *HandlerEx* receives a SERVICE_CONTROL_PAUSE code, responds with a SERVICE_PAUSE_PENDING, and then passes the code to the *ServiceMain* thread. The *ServiceMain* thread starts processing the code when, all of a sudden, the *HandlerEx* thread preempts the *ServiceMain* thread and receives a SERVICE_CONTROL_STOP code. The *HandlerEx* function now responds with a SERVICE_STOP_PENDING code and queues the new code to the *ServiceMain* thread. When the *ServiceMain* thread gets CPU time again, it completes its processing of the SERVICE_CONTROL_PAUSE code and reports SERVICE_PAUSED. Then the thread sees the queued SERVICE_CONTROL_STOP code, stops the service, and reports SERVICE_STOPPED. After all of this, the SCM receives the following state updates:

```
SERVICE_PAUSE_PENDING
SERVICE_STOP_PENDING
SERVICE_PAUSED
SERVICE_STOPPED
```

As you can see, these updates are gibberish, and an administrator could become quite confused. Note, however, that the service is running fine. You'd be surprised how many services I've seen that can actually report this sequence. The developers of these services never fix the problem, because it is quite unlikely that an administrator will issue action requests to the service so quickly—but it can happen! To solve this sequence problem, you must use a thread synchronization mechanism. The TimeService sample application at the end of this chapter uses a CGate C++ class to solve this problem efficiently.

When I first started working with services, I thought that the SCM would be responsible for preventing race conditions from occurring. But my experiments show that the SCM does absolutely nothing to time the sending of control codes. In fact, the SCM does absolutely nothing to ensure that a service receives control

codes properly, either. Here's what I mean: While a service is already paused, try sending the service a SERVICE_CONTROL_PAUSE code. You won't be able to do this with the Services snap-in because the snap-in will see that the service is paused and thus disable the Pause button. But if you use the SC.exe command-line utility, nothing is stopping you from sending a pause code to a service that is already paused. I would have expected the SCM to report failure to the SC.exe utility, but the SCM simply calls the service's *HandlerEx* function, passing it the SERVICE_CONTROL_PAUSE code. Your service must be able to handle these erroneous control codes gracefully.

I have seen many services written that don't deal with the possibility of the same control code being sent to the service multiple times in a row. For example, I know of a service that closed the handle to a named pipe when the service was suspended. The service then proceeded to create another kernel object that, coincidentally, got the same handle value as the handle of the original named pipe. Then the service received another pause control code and called *CloseHandle*, passing the handle value of the old pipe. Since this value happened to be the same as another kernel object's handle, the new kernel object was destroyed and the rest of the service started failing in strange and mysterious ways! I can't tell you how much of a pleasure this mess was to debug.

To fix this multiple stop, pause, or continue control code problem, check first to see whether your service is already in the desired state. If it is, don't call *SetServiceStatus*, and don't execute your code to change states—just return.

Here is some logic I've seen used in services quite frequently. When the *HandlerEx* function receives a SERVICE_CONTROL_PAUSE code, *HandlerEx* calls *SetServiceStatus* to report SERVICE_PAUSE_PENDING, calls *SuspendThread* to "pause" the service's thread, and then calls *SetServiceStatus* again to report SERVICE_PAUSED. This series of calls does avoid the race conditions, because all the work is being done by one thread, but what *is* this code doing? Does suspending the service thread pause the service? Well, I guess I have to answer "Yes" to that. However, what does it mean to pause a service? The answer depends on the service.

If I am writing a service that processes client requests over the network, to me, pause means that I'll stop accepting new requests. But what about the request I might be in the middle of processing right now? Maybe I should finish it so that my client doesn't indefinitely hang. If my *HandlerEx* function simply calls *SuspendThread*, the service thread might be in the middle of who knows what. Maybe the thread's inside a call to *malloc*, trying to allocate some memory. If another service running inside the same process also calls *malloc*, this other service gets suspended too (since access to the heap is serialized). This is certainly not what we want happening!

Oh—and what about this: do you think you should be allowed to stop a service that is paused? I do, and apparently Microsoft thinks so too, because the Services snap-in allows me to click on the Stop button even when a service is paused. But how can I stop a service that is paused because its thread has been suspended? Please don't say *TerminateThread*.

These are some of the issues that make service development challenging.

[Previous] [Next]

Service Issues

When you first start developing services, you'll notice that some aspects might not work the way you expect. Services are special beasts that run in special operating environments. In this section, I'll discuss some of the issues you will probably encounter. However, I won't spend too much time on them here because they are all explained much more thoroughly in other chapters of this book. I just want to give you an idea of what to keep an eye out for.

LocalSystem vs. Specific User Account

In this section, I'm going to explain how running a service under the LocalSystem account differs from running the service under a specified user account.

The LocalSystem account is an account given to the operating system itself. The operating system is never restricted from accessing resources. A service running under the LocalSystem account can manipulate any directory or file, change the system's time, start or stop any other service, shut down the machine, and perform a whole slew of other normally restricted actions without any hindrance whatsoever. A LocalSystem service is considered to be part of the system's Trusted Computing Base (TCB).

NOTE

This is the government definition of Trusted Computing Base, which I found at *http://nsi.org/Library/Compsec/compglos.txt*:

"The totality of protection mechanisms within a computer system&"95%">

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\
LanmanServer\Parameters
```

You can also enable all pipes and shares on the machine to be accessed by all NULL session connections by setting the RestrictNullSessionAccess data value (located under the same subkey) to 0. Even though you *can* do this, you should not because it opens a huge security hole on the system.

- Access resources after impersonating a specific user. To do this, you can call the various impersonation functions offered by Windows. (Impersonation is discussed in the chapters in Part IV of this book.) A service can also impersonate a specific user by having the LocalSystem service call the *LogonUser* function, passing a domain, username, and password to be authenticated. Note that the *LogonUser* function requires the TCB privilege (also known as the "Act as Part of the Operating System" privilege) that is granted, by default, to the LocalSystem account only—how convenient!

LocalSystem vs. Specific User Registry Subkeys

The registry is broken down into two main keys. The first key, HKEY_LOCAL_MACHINE, is where all the system-wide settings are stored. A service or an application can always read any settings under the HKEY_LOCAL_MACHINE key.

The second key, HKEY_USERS, is where each user's user-specific settings are stored. This key is further broken down into two *types* of subkeys. The first type of subkey is a specific user subkey. Each user account on the machine has a collection of registry settings that map to a subkey under HKEY_USERS. When the specific user logs on and becomes the interactive user, the familiar HKEY_CURRENT_USER key maps to the specific user's subkey under HKEY_USERS.

The second type of subkey under HKEY_USERS is called .DEFAULT, and it contains a user's default settings. When a new user account is created on the system, a new subkey is created under HKEY_USERS, and the settings in this subkey are populated with the current settings in the .DEFAULT subkey.

Like the settings under HKEY_LOCAL_MACHINE, the settings under HKEY_USERS\.DEFAULT are always available to services and applications, although a service is not likely to need to access it. A specific user's settings are not available under HKEY_USERS until that user has logged on to the system. Since services typically run under the LocalSystem account, a service should not attempt to access any specific user's settings under the HKEY_USERS account. Similarly, a LocalSystem service should not access the registry by using HKEY_CURRENT_USER. For more information about registry settings and user profiles, see Chapter 5.

Kernel Object Security

In this section, I introduce a common problem many developers run into when their client application and service run on the same machine and the client and service try to share a kernel object.

Here's the scenario. Your service starts running and calls *CreateFileMapping* to create a file-mapping object. *CreateFileMapping* creates a kernel object, so one of its parameters is the address of a SECURITY_ATTRIBUTES data structure. If you're like most programmers, you pass NULL for this parameter, which causes the kernel object to be created with *default security*. Notice that I said *default security*; I didn't say *no security*. Default security means that the object's access control is determined by the security context under which the object is created.

For example, a kernel object created by a LocalSystem service will, by default, allow full access to anything else running in the LocalSystem account, and will allow only read and execute access to members of the local administrators group. So if a LocalSystem service creates a file-mapping object with default security, an application running under a local administrators account can read from the file-mapping object but cannot write to it in any way. An application running under any other account will not be able to access the file-mapping object at all!

Right now, I just want to make you aware of this issue. Much more can be said about kernel object security for clients and services, and there are several ways to handle it, but they require that you have a much better understanding of Windows security. So I encourage you to read the security chapters in Part IV of this book to get the whole story.

Interactive Services, Window Stations, and Desktops

In this section, I'm just going to touch issues that affect how services interoperate with window stations and desktops. For more information about window stations and desktops, see Chapter 10.

When you create a kernel object, you can specify how it should be secured by passing the address of a SECURITY_ATTRIBUTES structure. But what about user objects such as windows and menus? User objects have a different model of use; they are not opened and closed—you just access them as you need them, which makes the code easier to write and enhances performance. Further, user objects existed in 16-bit Windows operating systems, which didn't support security in any form whatsoever. If Microsoft had added the SECURITY_ATTRIBUTES structure to *CreateWindow* and *CreateMenu*, developers would have had difficulty porting their 16-bit code.

Microsoft needed a way to secure user objects without affecting the existing functions and without affecting the way you used the objects. This is what window stations and desktops are all about. A window station is a *logical* combination of a keyboard, a mouse, and a display. The word "logical" means that the devices don't actually have to exist. When the system boots, it creates the interactive window station, named "WinSta0"; the physical keyboard, mouse, and display are assigned to this interactive window station. A window station also contains its own clipboard, a set of global atoms, and a group of desktop objects.

A desktop consists of a logical display surface and a set of user objects: windows, menus, and hooks. Threads are also associated with desktops. (See the *SetThreadDesktop* and *GetThreadDesktop* functions in the Platform SDK documentation.) If a thread associated with one desktop attempts to send a message to a window created on another desktop, the system fails the call. This is the security mechanism at work. Likewise, a thread can't install a hook on a thread that's part of another desktop.

Like window stations, desktops are identified by their string names. The WinLogon.exe process creates three desktops:

- **WinLogon** Presets the Logon dialog box. After the user logs on, WinLogon.exe switches to the Default desktop.
- **Default** The location where Explorer.exe and all the user's application windows appear. Whenever an application is run, it executes on this desktop.
- **Screen saver** Runs the system's screen saver when the user has been idle for an extended period of time.

The system itself has its own special user account. So two "users" are effectively accessing a machine at one time: the LocalSystem user and the logged-on user. Certainly, if two users are running applications on a single machine, you don't want all the application windows visible on the single display device; both users would be clamoring for each other's screen real estate.

For this reason, the LocalSystem account is given its own window station, named "Service-0x0-3e7$" (the number is the service's logon SID), with its own desktop, named Default. The window station is noninteractive and does not have a physical keyboard, mouse, and display—the LocalSystem "user" is not an actual human being and therefore has no need to type, click the mouse, or "see" anything. Any application running on a desktop in this window station can create windows, but the windows will not be visible to the logged-on user. This is why services should never present a user interface: no one will ever see it, and the thread will be suspended as it waits for input, which can never be entered.

Using the Services snap-in, you can display the properties for a service. You might recall that the Log On tab contains an Allow Service To Interact With Desktop check box. When checked, this option causes the SCM to start a service on the interactive window station's default desktop: "WinSta0\Default". Notice that you can mark this check box only if your service runs under the LocalSystem account. This is because the LocalSystem account is highly privileged and is able to access the interactive user's window station and desktop.

One problem with a service interacting with the desktop is that the default desktop is not always visible. When the screen saver is running or the user logs off, the service's user interface remains on the Default desktop, but the user interface is not visible until the next user logs in. Another problem is that it makes the system quite vulnerable. For example, a normal application is able to send window messages to the service's window. Well, the application is running with the logged-on user's security context, but the application is now

communicating with a process running as LocalSystem through an insecure channel. A restricted user can easily gain access to resources to which they should not be given access.

Here's an example: The service is running as LocalSystem and is presenting windows on the interactive default desktop. Normally, when the logged-on user attempts to use Task Manager's Processes tab to kill a service, an Access Denied or similar message is displayed, and the service continues running. This functionality is desired, of course. You wouldn't want a user who is logged on as a guest to kill the Server service, preventing other machines on the network from accessing directories, files, and printers (especially since the Server service runs inside Services.exe). However, if the LocalSystem service creates an interactive window, the logged-on user will see the window in Task Manager's Applications tab. Choosing End Task at this point sends a WM_CLOSE message to the window, which *can* kill the service.

NOTE

For all these reasons, Microsoft strongly discourages the use of the Allow Service To Interact With Desktop check box. In fact, an administrator can forbid services from interacting with the desktop by setting the NoInteractiveServices value to a nonzero value in the following registry subkey:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Windows
```

So what about services that run under a specific user account? When the SCM launches a service executable under a specific account, the SCM first authenticates the user, using the supplied user name and password as part of the service's configuration information. (The user name is stored in the registry, and the password is stored in an LSA secret.) This authentication creates a logon session, which gets its very own noninteractive window station and desktop. The service executable is now invoked using this dedicated window station and desktop, and is called something like "UserAccountLogonSID\Default", where "UserAccountLogonSID" is a unique number generated during authentication.

This unique identification means that if you have two or more services set to run under the same user account, each will get its own logon SID, window station, and desktop (provided they're in separate processes)—the threads in these service executables will not be able to communicate by using user objects. This unique identification also means that if a service is running under the same account as the currently logged-on interactive user, the service's user interface will not be visible. In contrast, the LocalSystem account is authenticated just once, so multiple service executables running under the LocalSystem account all share the same window station and desktop and can communicate using user objects.

Microsoft has added some service-specific features to the familiar *MessageBox(Ex)* functions. First, when you pass the MB_SERVICE_NOTIFICATION flag, *MessageBox(Ex)* will display the message box on the interactive window station's active desktop regardless of whether the desktop is WinLogon, Default, or Screen Saver. This guarantees that the message box is visible on the display device.

Second, *MessageBox(Ex)* supports the MB_DEFAULT_DESKTOP_ONLY flag. This flag is similar to MB_SERVICE_NOTIFICATION except that it makes the message box visible only on the interactive window station's default desktop: a user must log on to see the message box. Note that *MessageBox(Ex)* does not return until a user has seen and dismissed the message box. By the way, to use either the MB_DEFAULT_DESKTOP_ONLY or the MB_SERVICE_NOTIFICATION flags, a service does not have to run under the LocalSystem account nor does its Allow Service To Interact With The Desktop option have to be selected. The system ensures that the message box is visible.

If you've been following along carefully, you've noticed that I have discouraged you from using every method I've described for creating an interactive service. If you want to produce a service that offers a user interface,

what should you do? The answer is simple: create a separate, client-side application that uses some IPC mechanism (RPC, COM, named pipes, sockets, memory-mapped files, and so on) to talk to the service. I know that many people don't want to do this because they have to create a whole new project that produces its own executable, but creating a separate application is the correct and best-supported method. Microsoft simply can't break your architecture if you do this.

The client-side application could run on Windows 2000, Windows 98, UNIX, or any other operating system. You could create an HTML-based user interface that communicates with the service using ActiveX controls or Active Server Pages. You should also consider making your user interface a snap-in for MMC. Think of this client-side application as opening possibilities for you rather than restricting them because of all the user interface issues.

Before moving on, I'd like to discuss just one more user interface issue: some Windows functions produce hard-error message boxes. For example, the system will automatically display a message box if you are running an application from a CD-ROM drive and remove the disc. If the system didn't display a message box, its only other option would be to kill the process.

It is possible to modify the system's behavior so that hard errors are logged to the event log and do not cause message boxes to appear. To alter the system's behavior, you must modify the ErrorMode value in the following registry subkey:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Windows
```

Table 3-4 lists the possible values for ErrorMode.

Table 3-4. *ErrorMode values for configuring the display of hard-error message boxes*

Value	Description
0 (the default value)	The system displays error message boxes.
1	For non-system-generated errors, an error message box is displayed. For system-generated errors, an entry is added to the event log and no error message box is displayed.
2 (best for an unattended server)	For system-generated and non-system-generated errors, an entry is added to the event log and no error message is displayed.

[Previous] [Next]

Debugging a Service

Debugging a service is a bit more tricky than debugging a normal application for several reasons. First, the debugger cannot start the service; the SCM must start the service. Second, many services start before a user logs on to the machine. For this reason, setting an automatic service to manual while you're debugging it is a good idea. Third, services run in their own window station and desktop, which is not visible to an interactive user.

So how can you debug a service? The best way is to run it as a regular executable instead of as a service. Inside your service's *(w)main* or *(w)WinMain* function, check for a special command-line switch of your own devising and, if this switch is present, call your service's *ServiceMain* function directly instead of calling *StartServiceCtrlDispatcher*. This technique, of course, has several disadvantages:

- If your executable contains several services, you can debug only one service at a time.

- The executable is running under your (the interactive user's) account instead of the account that the SCM would have used. This might restrict access to resources that normally would be granted.

- You cannot send pause, continue, stop, shut down, or any user-defined notifications to the service, prohibiting the testing of the execution paths.

The approach just described makes debugging your service very easy, but a better approach is to connect a debugger to the service while the service is running. Most debuggers offer the ability to connect to a process while it is running. If you already have a debugger installed on your system, you can open the Task Manager, right-click on the service's process name, and select the Debug option from the context menu. This spawns the debugger and attaches it to your service. You can now set breakpoints, debug the service code, and even test how your service responds to control notifications. Here are a few problems with connecting the debugger to the service:

- The account you've used to log on to the system must have the debug privilege granted. By default, this privilege is granted to administrators only. If you are logged on as Power User or some other account, you must have an administrator grant you the debug privilege.

- You won't be able to debug your initialization code because the debugger will connect after the service is up and running.

If you really want to use this approach to debug your service's initialization code, you can do it easily enough by simply adding a call to the *DebugBreak* function inside your *(w)main* or *(w)WinMain* function. However, this technique works only if your service runs under the LocalSystem account. If you run your service under a different user account, your debugger will not work properly because the system won't allow it to interact with the interactive window station and desktop.

Here is another technique you can use to debug a service: Windows offers the ability to invoke a debugger whenever a process starts. To have the system do this, you must first create a subkey under the following registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\
   CurrentVersion\Image File Execution Options
```

Under this key, create a subkey that is the name of your service's executable (without the path). In the executable name subkey, add a string value whose name is Debugger, and set this value equal to the full pathname of your debugger (for example, "C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin\msdev.exe").

Once you have all this set up, you can go to the Services snap-in and start the service. The SCM will launch the debugger instead of the service executable. At this point, open your source code file, set breakpoints, and then let the service go. Note that you will have just 30 seconds to do this before the SCM forcibly terminates the debugger (since the service will not have called *StartServiceCtrlDispatcher*).

If all of these limitations are bothering you, and you want to be able to debug the service in its most natural environment without having to interact with the desktop or worry about which user account the service is logged on to, you can use a kernel debugger.

[Previous] [Next]

The TimeService Sample Service

The TimeService sample service ("03 TimeService.exe"), shown in Listing 3-1 at the end of this section,

includes all the necessary components for building a service. The source code and resource files for the application are in the 03TimeService directory on the companion CD. This very simple service returns the server machine's date and time when a client connects to the server. The service does assume that you are somewhat familiar with named pipes and I/O completion ports (discussed in Chapter 2).

If you examine the *_tWinMain* function, you'll see that this service has the ability to install and remove itself from the SCM's database depending on whether "-install" or "-remove" is passed as a command-line argument. Once you build the service, run it once from the command line, passing "-install". When you no longer want to keep the service on your machine, run the executable from the command line, passing "-remove" as an argument. I'll discuss the details of the functions used for adding services to and deleting services from the SCM database in the next chapter.

The most important aspect of *_tWinMain* to notice is that an array of SERVICE_TABLE_ENTRY structures is initialized with two members: one for the service and one with NULL entries to identify the last service. The address of this service table array is passed to *StartServiceCtrlDispatcher*, which creates a thread for the service. This new thread begins execution starting with the *TimeServiceMain* function. Note that *StartServiceCtrlDispatcher* will not return to the *_tWinMain* function until the *TimeServiceMain* function exits and its thread terminates.

The *TimeServiceMain* function implements the actual code to process the client requests. It starts by creating an I/O completion port. The service thread sits in a loop waiting for requests to enter the completion port. Two types of requests are possible: a client has connected to the pipe and wants the machine's date and time information; or the service needs to process an action request such as pause, continue, or stop.

Once the completion port is created, I initialize a global CServiceStatus object, *g_ssTime*. The CServiceStatus C++ class is my own creation and greatly simplifies the reporting of service status updates. This class is derived from the Windows SERVICE_STATUS structure and is basically a thin level of abstraction that places some logic over how the member variables are updated. CServiceStatus has a CGate class object as a member, which is used to synchronize the *TimeHandlerEx* thread and the *TimeServiceMain* thread, guaranteeing that the *ServiceMain* thread processes a single action request at a time.

The CServiceStatus's *Initialize* method internally calls *RegisterServiceCtrlHandlerEx* to notify the SCM of the service's *HandlerEx* function (*TimeHandlerEx*), and it passes the address of the I/O completion port C++ class object, *iocp*, to the handler function in its *pvContext* parameter. The *Initialize* method also sets the *dwServiceType* member, which never changes during the lifetime of the service.

Next, I call the *AcceptControls* method, which sets the *dwControlsAccepted* member. The *AcceptControls* method can be called periodically while the service runs to accept or reject the desired controls. The TimeService accepts stop, pause, continue, and shutdown codes during its lifetime.

You'll notice that my *TimeHandlerEx* function passes control codes to the service thread by calling the CIOCP's *PostStatus* method, which internally calls *PostQueuedCompletionStatus* using the handle of the I/O completion port. A completion key of CK_SERVICECONTROL is specified to indicate to the *ServiceMain* thread that it is waking up because of a service action request. The *TimeHandlerEx* function then returns as quickly as possible. The service's thread is responsible for waking up, processing the code, and then waiting for more client requests (if appropriate).

Back inside the *TimeServiceMain* function, a do-while loop starts. Inside this loop, I check the *CompKey* variable to see what action the service needs to respond to next. Since this variable is initialized to CK_SERVICECONTROL and the *dwControl* variable is initialized to SERVICE_CONTROL_CONTINUE, the service's first order of business is to create the named pipe that the client application will use to make requests of the service. This pipe is then associated with the completion port by using a completion key of CK_PIPE, and an asynchronous call to *ConnectNamedPipe* is made. The service now reports to the SCM that it is up and running by calling the *g_ssTime* object's *ReportUltimateState* method, which internally calls

SetServiceStatus to report SERVICE_RUNNING.

The service calls the *iocp* object's *GetStatus* method (which internally calls *GetQueuedCompletionStatus*). This causes the service thread to sleep until an event appears in the completion port. If a service control code appears (because *TimeHandlerEx* called *PostQueuedCompletionStatus*), the service thread wakes, processes the control code (as appropriate), and then reports to the SCM again that the operation is complete. Note that the *TimeHandlerEx* function is responsible for reporting the pending state of the action, and the *TimeServiceMain* function is responsible for reporting the service's final execution state (which is the third method I discussed earlier in the section "Dealing with Interthread Communication Issues").

When the service thread wakes because *GetQueuedCompletionStatus* returns a completion key of CK_PIPE, a client has connected to the pipe. At this point, the service gets the system time and calls *WriteFile* to send the time to the client. Then the service disconnects the client and issues another asynchronous call to *ConnectNamedPipe* so that another client can connect.

When the service thread wakes with a SERVICE_CONTROL_STOP or SERVICE_CONTROL_SHUTDOWN code, it closes the pipe and terminates. This causes the completion port to close, and the *TimeServiceMain* function returns, killing the service thread. At this point, the *StartServiceCtrlDispatcher* returns to the *_tWinMain* function, which also returns, killing the process.

After you build the service, you must execute the program with the "_install" command-line switch to install the service in the SCM's database. Be sure to include quotes around the executable name since it includes a space, "03 TimeService.exe". Also, you'll want to use the Services snap-in to start and administer the "Programming Server-Side Applications Time" service.

Listing 3-1. *The TimeService sample service*

TimeService.cpp

```
/******************************************************************************
Module: TimeService.cpp Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
#include "..\CmnHdr.h" /* See Appendix A. */ #include "..\ClassLib\IOCP.h" /* See Appendix B */
#include "..\ClassLib\EnsureCleanup.h" /* See Appendix B */ #define SERVICESTATUS_IMPL
#include "ServiceStatus.h" ////////////////////////////////////////////////////////////////////////////// TCHAR
g_szServiceName[] = TEXT("Programming Server-Side Applications Time"); CServiceStatus
g_ssTime; ////////////////////////////////////////////////////////////////////////////// // The completion port wakes for 1
of 2 reasons: enum COMPKEY { CK_SERVICECONTROL, // A service control code CK_PIPE // A
client connects to our pipe }; ////////////////////////////////////////////////////////////////////////////// DWORD
WINAPI TimeHandlerEx(DWORD dwControl, DWORD dwEventType, PVOID pvEventData,
PVOID pvContext) { DWORD dwReturn = ERROR_CALL_NOT_IMPLEMENTED; BOOL
fPostControlToServiceThread = FALSE; switch (dwControl) { case SERVICE_CONTROL_STOP:
case SERVICE_CONTROL_SHUTDOWN: g_ssTime.SetUltimateState(SERVICE_STOPPED,
2000); fPostControlToServiceThread = TRUE; break; case SERVICE_CONTROL_PAUSE:
g_ssTime.SetUltimateState(SERVICE_PAUSED, 2000); fPostControlToServiceThread = TRUE;
break; case SERVICE_CONTROL_CONTINUE: g_ssTime.SetUltimateState(SERVICE_RUNNING,
2000); fPostControlToServiceThread = TRUE; break; case SERVICE_CONTROL_INTERROGATE:
g_ssTime.ReportStatus(); break; case SERVICE_CONTROL_PARAMCHANGE: break; case
SERVICE_CONTROL_DEVICEEVENT: case
SERVICE_CONTROL_HARDWAREPROFILECHANGE: case
SERVICE_CONTROL_POWEREVENT: break; case 128: // A user-define code just for testing //
NOTE: Normally, a service shouldn't display UI MessageBox(NULL, TEXT("In HandlerEx
processing user-defined code."), g_szServiceName, MB_OK | MB_SERVICE_NOTIFICATION);
```

```
break; } if (fPostControlToServiceThread) { // The Handler thread is very simple and executes very quickly because // it just passes the control code off to the ServiceMain thread CIOCP* piocp = (CIOCP*) pvContext; piocp->PostStatus(CK_SERVICECONTROL, dwControl); dwReturn = NO_ERROR; } return(dwReturn); } ////////////////////////////////////////////////////////////////////////////// void WINAPI TimeServiceMain(DWORD dwArgc, PTSTR* pszArgv) { ULONG_PTR CompKey = CK_SERVICECONTROL; DWORD dwControl = SERVICE_CONTROL_CONTINUE; CEnsureCloseFile hpipe; OVERLAPPED o, *po; SYSTEMTIME st; DWORD dwNumBytes; // Create the completion port and save its handle in a global // variable so that the Handler function can access it CIOCP iocp(0); g_ssTime.Initialize(g_szServiceName, TimeHandlerEx, (PVOID) &iocp, TRUE); g_ssTime.AcceptControls( SERVICE_ACCEPT_STOP | SERVICE_ACCEPT_PAUSE_CONTINUE); do { switch (CompKey) { case CK_SERVICECONTROL: // We got a new control code switch (dwControl) { case SERVICE_CONTROL_CONTINUE: // While running, create a pipe that clients can connect to hpipe = CreateNamedPipe(TEXT("\\\\.\\pipe\\TimeService"), PIPE_ACCESS_OUTBOUND | FILE_FLAG_OVERLAPPED, PIPE_TYPE_BYTE, 1, sizeof(st), sizeof(st), 1000, NULL); // Associate the pipe with the completion port iocp.AssociateDevice(hpipe, CK_PIPE); // Pend an asynchronous connect against the pipe ZeroMemory(&o, sizeof(o)); ConnectNamedPipe(hpipe, &o); g_ssTime.ReportUltimateState(); break; case SERVICE_CONTROL_PAUSE: case SERVICE_CONTROL_STOP: // When not running, close the pipe so clients can't connect hpipe.Cleanup(); g_ssTime.ReportUltimateState(); break; } break; case CK_PIPE: if (hpipe.IsValid()) { // We got a client request: Send our current time to the client GetSystemTime(&st); WriteFile(hpipe, &st, sizeof(st), &dwNumBytes, NULL); FlushFileBuffers(hpipe); DisconnectNamedPipe(hpipe); // Allow another client to connect ZeroMemory(&o, sizeof(o)); ConnectNamedPipe(hpipe, &o); } else { // We get here when the pipe is closed } } if (g_ssTime != SERVICE_STOPPED) { // Sleep until a control code comes in or a client connects iocp.GetStatus(&CompKey, &dwNumBytes, &po); dwControl = dwNumBytes; } } while (g_ssTime != SERVICE_STOPPED); } ////////////////////////////////////////////////////////////////////////////// void InstallService() { // Open the SCM on this machine CEnsureCloseServiceHandle hSCM = OpenSCManager(NULL, NULL, SC_MANAGER_CREATE_SERVICE); // Get our full pathname TCHAR szModulePathname[_MAX_PATH * 2]; GetModuleFileName(NULL, szModulePathname, chDIMOF(szModulePathname)); // Append the switch that causes the process to run as a service. lstrcat(szModulePathname, TEXT(" /service")); // Add this service to the SCM's database CEnsureCloseServiceHandle hService = CreateService(hSCM, g_szServiceName, g_szServiceName, SERVICE_CHANGE_CONFIG, SERVICE_WIN32_OWN_PROCESS, SERVICE_DEMAND_START, SERVICE_ERROR_IGNORE, szModulePathname, NULL, NULL, NULL, NULL, NULL); SERVICE_DESCRIPTION sd = { TEXT("Sample Time Service from ") TEXT("Programming Server-Side Applications for Microsoft Windows Book") }; ChangeServiceConfig2(hService, SERVICE_CONFIG_DESCRIPTION, &sd); } ////////////////////////////////////////////////////////////////////////////// void RemoveService() { // Open the SCM on this machine CEnsureCloseServiceHandle hSCM = OpenSCManager(NULL, NULL, SC_MANAGER_CONNECT); // Open this service for DELETE access CEnsureCloseServiceHandle hService = OpenService(hSCM, g_szServiceName, DELETE); // Remove this service from the SCM's database DeleteService(hService); } ////////////////////////////////////////////////////////////////////////////// int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) { int nArgc = __argc; #ifdef UNICODE PCTSTR *ppArgv = (PCTSTR*) CommandLineToArgvW(GetCommandLine(), &nArgc); #else PCTSTR *ppArgv = (PCTSTR*) __argv; #endif if (nArgc < 2) { MessageBox(NULL, TEXT("Programming Server-Side Applications for Microsoft Windows: ") TEXT("Time Service Sample\n\n") TEXT("Usage: TimeService.exe [/install] [/remove] [/debug] ") TEXT("[/service]\n") TEXT(" /install\t\tInstalls the service in the SCM's database.\n") TEXT(" /remove\t\tRemoves the service from the SCM's database.\n") TEXT(" /debug\t\tRuns the service as a normal process for ") TEXT("debugging.\n") TEXT(" /service\t\tRuns the process as a service ") TEXT("(should only be set in the SCM's database)."), g_szServiceName, MB_OK); } else { for (int i = 1; i < nArgc; i++) { if ((ppArgv[i][0] == TEXT('-')) || (ppArgv[i][0] == TEXT('/'))) { // Command line switch if (lstrcmpi(&ppArgv[i][1], TEXT("install")) == 0)
```

```
InstallService(); if (lstrcmpi(&ppArgv[i][1], TEXT("remove")) == 0) RemoveService(); if
(lstrcmpi(&ppArgv[i][1], TEXT("debug")) == 0) { // Execute the service code TimeServiceMain(0,
NULL); } if (lstrcmpi(&ppArgv[i][1], TEXT("service")) == 0) { // Connect to the service control
dispatcher SERVICE_TABLE_ENTRY ServiceTable[] = { { g_szServiceName, TimeServiceMain },
{ NULL, NULL } // End of list }; chVERIFY(StartServiceCtrlDispatcher(ServiceTable)); } } } }
#ifdef UNICODE HeapFree(GetProcessHeap(), 0, (PVOID) ppArgv); #endif return(0); }
//////////////////////////////// End Of File ////////////////////////////////
```

ServiceStatus.h

```
/******************************************************************************
Module: ServiceStatus.h Notices: Copyright (c) 2000 Jeffrey Richter Purpose: This class wraps a
SERVICE_STATUS structure ensuring proper use.
******************************************************************************/
#pragma once //////////////////////////////////////////////////////////////////////////////// #include "..\CmnHdr.h" /* See
Appendix A. */ #include "Gate.h" //////////////////////////////////////////////////////////////////////////////// class
CServiceStatus : public SERVICE_STATUS { public: CServiceStatus(); BOOL Initialize(PCTSTR
szServiceName, LPHANDLER_FUNCTION_EX pfnHandler, PVOID pvContext, BOOL
fOwnProcess, BOOL fInteractWithDesktop = FALSE); VOID AcceptControls(DWORD dwFlags,
BOOL fAccept = TRUE); BOOL ReportStatus(); BOOL SetUltimateState(DWORD dwUltimateState,
DWORD dwWaitHint = 0); BOOL AdvanceState(DWORD dwWaitHint, DWORD dwCheckPoint =
0); BOOL ReportUltimateState(); BOOL ReportWin32Error(DWORD dwError); BOOL
ReportServiceSpecificError(DWORD dwError); operator DWORD() const { return(dwCurrentState);
} private: SERVICE_STATUS_HANDLE m_hss; CGate m_gate; };
//////////////////////////////////////////////////////////////////////////////// inline CServiceStatus::CServiceStatus() {
ZeroMemory(this, sizeof(SERVICE_STATUS)); m_hss = NULL; }
//////////////////////////////////////////////////////////////////////////////// inline VOID
CServiceStatus::AcceptControls(DWORD dwFlags, BOOL fAccept) { if (fAccept)
dwControlsAccepted |= dwFlags; else dwControlsAccepted &= ~dwFlags; }
//////////////////////////////////////////////////////////////////////////////// inline BOOL CServiceStatus::ReportStatus() {
BOOL fOk = ::SetServiceStatus(m_hss, this); chASSERT(fOk); return(fOk); }
//////////////////////////////////////////////////////////////////////////////// inline BOOL
CServiceStatus::ReportWin32Error(DWORD dwError) { dwWin32ExitCode = dwError;
dwServiceSpecificExitCode = 0; return(ReportStatus()); } inline BOOL
CServiceStatus::ReportServiceSpecificError(DWORD dwError) { dwWin32ExitCode =
ERROR_SERVICE_SPECIFIC_ERROR; dwServiceSpecificExitCode = dwError;
return(ReportStatus()); } //////////////////////////////////////////////////////////////////////////////// #ifdef
SERVICESTATUS_IMPL //////////////////////////////////////////////////////////////////////////////// BOOL
CServiceStatus::Initialize(PCTSTR szServiceName, LPHANDLER_FUNCTION_EX pfnHandler,
PVOID pvContext, BOOL fOwnProcess, BOOL fInteractWithDesktop) { m_hss =
RegisterServiceCtrlHandlerEx(szServiceName, pfnHandler, pvContext); chASSERT(m_hss !=
NULL); dwServiceType = fOwnProcess ? SERVICE_WIN32_OWN_PROCESS :
SERVICE_WIN32_SHARE_PROCESS; if (fInteractWithDesktop) dwServiceType |=
SERVICE_INTERACTIVE_PROCESS; dwCurrentState = SERVICE_START_PENDING;
dwControlsAccepted = 0; dwWin32ExitCode = NO_ERROR; dwServiceSpecificExitCode = 0;
dwCheckPoint = 0; dwWaitHint = 2000; return(m_hss != NULL); }
//////////////////////////////////////////////////////////////////////////////// BOOL
CServiceStatus::SetUltimateState(DWORD dwUltimateState, DWORD dwWaitHint) { DWORD
dwPendingState = 0; // An invalid state value switch (dwUltimateState) { case SERVICE_STOPPED:
dwPendingState = SERVICE_STOP_PENDING; break; case SERVICE_RUNNING: dwPendingState
= (dwCurrentState == SERVICE_PAUSED) ? SERVICE_CONTINUE_PENDING :
SERVICE_START_PENDING; break; case SERVICE_PAUSED: dwPendingState =
```

SERVICE_PAUSE_PENDING; break; default: chASSERT(dwPendingState != 0); // Invalid parameter break; } // When creating a new ServiceMain thread, the system assumes // dwCurrentState=SERVICE_START_PENDING, dwCheckPoint=0, dwWaitHint=2000 // So, since we must always increment the checkpoint, let's start at 1 dwCheckPoint = 1; this->dwWaitHint = dwWaitHint; // No error to report dwWin32ExitCode = NO_ERROR; dwServiceSpecificExitCode = 0; BOOL fOk = FALSE; // Assume failure if (dwPendingState != 0) { // If another pending operation hasn't completed, wait for it m_gate.WaitToEnterGate(); dwCurrentState = dwPendingState; // Update the state in the structure // If no wait hint, we reached the desired state fOk = (dwWaitHint != 0) ? ReportStatus() : ReportUltimateState(); } return(fOk); } // BOOL CServiceStatus::AdvanceState(DWORD dwWaitHint, DWORD dwCheckPoint) { // A checkpoint of 0 is invalid, so we'll increment the checkpoint by 1 this->dwCheckPoint = (dwCheckPoint == 0) ? this->dwCheckPoint + 1 : dwCheckPoint; this->dwWaitHint = dwWaitHint; // No error to report dwWin32ExitCode = NO_ERROR; dwServiceSpecificExitCode = 0; return(ReportStatus()); } // BOOL CServiceStatus::ReportUltimateState() { DWORD dwUltimateState = 0; // An invalid state value switch (dwCurrentState) { case SERVICE_START_PENDING: case SERVICE_CONTINUE_PENDING: dwUltimateState = SERVICE_RUNNING; break; case SERVICE_STOP_PENDING: dwUltimateState = SERVICE_STOPPED; break; case SERVICE_PAUSE_PENDING: dwUltimateState = SERVICE_PAUSED; break; } dwCheckPoint = dwWaitHint = 0; // We reached the ultimate state // No error to report dwWin32ExitCode = NO_ERROR; dwServiceSpecificExitCode = 0; BOOL fOk = FALSE; // Assume failure if (dwUltimateState != 0) { dwCurrentState = dwUltimateState; // Update the state in the structure fOk = ReportStatus(); // Our state change is complete, allow a new state change m_gate.LiftGate(); } return(fOk); } // #endif // SERVICESTATUS_IMPL ////////////////////////////////// End of File //////////////////////////////////

Gate.h

```
/******************************************************************************
Module:  Gate.h
Notices: Copyright (c) 2000 Jeffrey Richter
Purpose: This class creates a normally open gate that only one thread can
         pass through at a time.
******************************************************************************/


#pragma once   // Include this header file once per compilation unit


//////////////////////////////////////////////////////////////////////////////


#include "..\CmnHdr.h"              /* See Appendix A. */


//////////////////////////////////////////////////////////////////////////////


class CGate {
public:
   CGate(BOOL fInitiallyUp = TRUE, PCTSTR pszName = NULL) {
      m_hevt = ::CreateEvent(NULL, FALSE, fInitiallyUp, pszName);
   }

   ~CGate() {
      ::CloseHandle(m_hevt);
   }

   DWORD WaitToEnterGate(DWORD dwTimeout = INFINITE, BOOL fAlertable = FALSE) {
```

```
      return(::WaitForSingleObjectEx(m_hevt, dwTimeout, fAlertable));
   }

   VOID LiftGate() { ::SetEvent(m_hevt); }

private:
    HANDLE m_hevt;
};


//////////////////////////////// End of File //////////////////////////////////
```

[Previous] [Next]

The TimeClient Sample Application

The TimeClient sample application ("03 TimeClient.exe"), shown in Listing 3-2, tests the TimeService service. The source code and resource files for the application are in the 03-TimeClient directory on the companion CD. When you start the program, the dialog box in Figure 3-8 appears.

Figure 3-8. *Initial dialog box in the TimeClient sample application*

To see the client/server communication work, you must type the server name in the edit control at the top of the dialog box. If you are running the client and the server process on the same machine, type a period as the server name (as shown in Figure 3-8). When you click the Request Server's Time button, the client application calls *CreateFile*, which connects the client to the server, causing the server to wake up and process the client's request. If the server is not running, *CreateFile* fails and a message box similar to the one shown in Figure 3-9 appears.

Figure 3-9. *Message box displayed by TimeClient when the TimeService service isn't running*

If the service is up and running, *CreateFile* returns a valid handle, and the client waits for the time data to come across the pipe by placing a synchronous call to *ReadFile*. After the client has the data, the client's pipe handle is closed, the time from the server (which came across in universal time) is converted to the client's local time, and the initial dialog box is updated to look like Figure 3-10.

Figure 3-10. *The updated dialog box for the TimeClient sample application*

Listing 3-2. *The TimeClient sample application*

TimeClient.cpp

```
/******************************************************************************
Module: TimeClient.cpp Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
#include "..\CmnHdr.h" // See Appendix A #include <WindowsX.h> #include
"..\ClassLib\EnsureCleanup.h" // See Appendix B #include "Resource.h"
////////////////////////////////////////////////////////////////////////////// BOOL Dlg_OnInitDialog(HWND hwnd,
HWND hwndFocus, LPARAM lParam) { chSETDLGICONS(hwnd, IDI_TIMECLIENT); // Assume
that the server is on the same machine as the client SetDlgItemText(hwnd, IDC_SERVER,
TEXT(".")); return(TRUE); } ////////////////////////////////////////////////////////////////////////////// void
Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) { switch (id) { case
IDCANCEL: EndDialog(hwnd, id); break; case IDOK: // Construct the pathname of the pipe TCHAR
sz[500]; sz[0] = sz[1] = TEXT('\\'); GetDlgItemText(hwnd, IDC_SERVER, &sz[2], chDIMOF(sz) -
2); lstrcat(sz, TEXT("\\pipe\\TimeService")); // Attempt to connect to the pipe // Get a handle to use to
talk to the pipe CEnsureCloseFile hpipe = CreateFile(sz, GENERIC_READ, 0, NULL,
OPEN_EXISTING, 0, NULL); if (hpipe.IsValid()) { // Valid handle, read time from pipe
SYSTEMTIME st; DWORD cbRead = 0; ReadFile(hpipe, &st, sizeof(st), &cbRead, NULL); //
Convert UTC time to client machine's local time and display it
SystemTimeToTzSpecificLocalTime(NULL, &st, &st);
GetDateFormat(LOCALE_USER_DEFAULT, DATE_LONGDATE, &st, NULL, sz, chDIMOF(sz));
SetDlgItemText(hwnd, IDC_DATE, sz); GetTimeFormat(LOCALE_USER_DEFAULT,
LOCALE_NOUSEROVERRIDE, &st, NULL, sz, chDIMOF(sz)); SetDlgItemText(hwnd,
IDC_TIME, sz); } else { // Invalid handle, report an error SetDlgItemText(hwnd, IDC_DATE,
TEXT("Error")); SetDlgItemText(hwnd, IDC_TIME, TEXT("Error")); // Get the error code's textual
description HLOCAL hlocal = NULL; // Buffer that gets the error message string FormatMessageA(
FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER, NULL,
GetLastError(), MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US), (PSTR) &hlocal,
0, NULL); if (hlocal != NULL) { chMB((PCSTR) LocalLock(hlocal)); LocalFree(hlocal); } } break; }
} ////////////////////////////////////////////////////////////////////////////// INT_PTR WINAPI Dlg_Proc (HWND hwnd,
UINT uMsg, WPARAM wParam, LPARAM lParam) { switch (uMsg) {
chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
chHANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand); } return(FALSE); }
////////////////////////////////////////////////////////////////////////////// int WINAPI _tWinMain(HINSTANCE
hinstExe, HINSTANCE, PTSTR pszCmdLine, int) { DialogBox(hinstExe,
MAKEINTRESOURCE(IDD_TIMECLIENT), NULL, Dlg_Proc); return(0); } //////////////////////////////// End Of File ////////////////////////////////
```

[Previous] [Next]

Chapter 4

Service Control Programs

As you learned in Chapter 3, a Service Control Program (SCP) is an application that communicates with a Service Control Manager (SCM) running on the local machine or on a remote machine. I usually think of an SCP as an application that controls services by starting, stopping, pausing, or continuing them. However, an SCP can do much more: it can manipulate a SCM's database by adding services, removing services, and enumerating the installed services. The SCP can also change a service's configuration. In this chapter, I'll examine the various ways an SCP can communicate with the SCM. Note that the SCM can also be used to start and stop device drivers. Many of the functions discussed in this chapter apply to both services and device drivers; however, I will concentrate on services and avoid any discussion applying to device drivers.

When you write a service, you typically also create an application that administrators use to control that service. This administrative application should control the service by using the various functions described throughout this chapter. This application should also help the administrator deal with other aspects of the service, such as configuring it (described in Chapter 5) and publishing it in Active Directory. Ideally, this administration application is implemented as a snap-in for the Microsoft Management Console (MMC) or with a Web-based console.

The first step in communicating with the SCM of a machine is to call *OpenSCManager*:

```
SC_HANDLE OpenSCManager(
   PCTSTR pszMachineName,
   PCTSTR pszDatabaseName,
   DWORD  dwDesiredAccess);
```

This function establishes a communication channel with the SCM on the machine identified by the *pszMachineName* parameter; pass NULL to open the SCM on the local machine. The *pszDatabaseName* parameter identifies which database to open; you should always pass either SERVICES_ACTIVE_DATABASE or NULL for this parameter. The *dwDesiredAccess* parameter tells the function what you intend to do with the SCM database. Table 4-1 indicates what access rights are available.

Table 4-1. *Access right values for* OpenSCManager'*s* dwDesiredAccess *parameter that specify access to the SCM*

Access Rights	Description
SC_MANAGER_ALL_ACCESS	Includes STANDARD_RIGHTS_REQUIRED in addition to all the access types listed in this table.
SC_MANAGER_CONNECT	Allows connecting to the SCM. This access is always implied, even if not explicitly specified.
SC_MANAGER_CREATE_SERVICE	Enables calling of *CreateService* to add a service to the SCM database.
SC_MANAGER_ENUMERATE_SERVICE	Enables calling of *EnumServicesStatus* to get the list of services in the SCM database and each service's status.
SC_MANAGER_LOCK	Enables calling of *LockServiceDatabase* to stop the SCM from starting any more services.
SC_MANAGER_QUERY_LOCK_STATUS	Enables calling of *QueryServiceLockStatus* to find out which user (if any) has locked the SCM database.

Windows secures the SCM by providing the following default access:

- Administrators have full access to the SCM.
- LocalSystem and Everyone have SC_MANAGER_CONNECT, SC_MANAGER_ENUMERATE_SERVICE, and SC_MANAGER_QUERY_LOCK_STATUS access to the SCM.

OpenSCManager returns an SC_HANDLE that you pass to other functions so that you can manipulate the SCM's database. When you are finished accessing the SCM database, you must close the handle by passing it to *CloseServiceHandle*:

```
BOOL CloseServiceHandle(SC_HANDLE hSCManager);
```

[Previous] [Next]

Adding a Service to the SCM's Database

One of the most common reasons to manipulate the SCM database is to add a service. To add a service, you must call *OpenSCManager*, specifying the SC_MANAGER_CREATE_SERVICE access, and then call *CreateService*:

```
SC_HANDLE CreateService(
   SC_HANDLE hSCManager,
   PCTSTR    pszServiceName,    // Internal, programmatic string name
   PCTSTR    pszDisplayName,
   DWORD     dwDesiredAccess,
   DWORD     dwServiceType,
   DWORD     dwStartType,
   DWORD     dwErrorControl,
   PCTSTR    pszPathName,
   PCTSTR    pszLoadOrderGroup,
   PDWORD    pdwTagId,          // Always 0 for services
   PCTSTR    pszDependencies,   // Double zero-terminated string
   PCTSTR    pszUserName,
   PCTSTR    pszUserPswd);
```

As you can see, *CreateService* requires quite a few parameters (13 to be exact). The *hSCManager* parameter is the handle returned from *OpenSCManager*. The next two parameters, *pszServiceName* and *pszDisplayName*, indicate the name of the service. Services have an internal name for use by programmers and a display name that is shown to users. The internal name, identified by *pszServiceName*, is used by the SCM to store the service information inside the registry. For example, the Logical Disk Manager service has the internal name "dmserver," and its service information can be found under the following registry key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\dmserver
```

CreateService's *dwDesiredAccess* parameter is useful because it tells the SCM what you intend to do with the newly installed service after *CreateService* returns a handle to it (so you can manipulate the service right away). If you are only installing a service and do not intend to manipulate it after it is installed, simply pass 0 for *dwDesiredAccess*, and then immediately close the handle returned from *CreateService* by calling *CloseServiceHandle*. Table 4-2 shows the access rights you can specify for *dwDesiredAccess* when you use *CreateService*.

Table 4-2. *Access right values for* CreateService's dwDesiredAccess *parameter that specify access to the service added to the SCM database*

Access Rights	Description
SERVICE_START	Enables calling of *StartService*to start the service.
SERVICE_STOP	Enables calling of *ControlService* to stop the service.
SERVICE_PAUSE_CONTINUE	Enables calling of *ControlService* to pause and continue the service. This access also allows changing a service's parameters.
SERVICE_INTERROGATE	Enables calling of *ControlService*to ask the service to report its status immediately.
SERVICE_USER_DEFINED_CONTROL	Enables calling of *ControlService* to specify a user-defined control code.
SERVICE_QUERY_STATUS	Enables calling of *QueryServiceStatus(Ex)* functions to ask the service control manager about the status of the service.
SERVICE_ENUMERATE_DEPENDENTS	Enables calling of *EnumDependentServices* to enumerate all the services dependent on the service.
SERVICE_CHANGE_CONFIG	Enables calling of *ChangeServiceConfig(2)* to change the service configuration.
SERVICE_QUERY_CONFIG	Enables calling of *QueryServiceConfig(2)* to query the service configuration.
DELETE	Enables calling of *DeleteService*to delete the service.

The *CreateService* function does not have a parameter that accepts a pointer to a SECURITY_ATTRIBUTES structure. So when a new service is installed in the SCM's database, the SCM sets default security on the service. You can alter these security settings using the *QueryServiceObjectSecurity* and *SetServiceObjectSecurity* functions. These are the SCM's access settings for the service's default security:

- Administrators and System Operators have SERVICE_CHANGE_CONFIG, SERVICE_ENUMERATE_DEPENDENTS, SERVICE_INTERROGATE, SERVICE_PAUSE_CONTINUE, SERVICE_QUERY_CONFIG, SERVICE_QUERY_STATUS, SERVICE_START, SERVICE_STOP, SERVICE_USER_DEFINED_CONTROL, READ_CONTROL, WRITE_OWNER, WRITE_DAC, and DELETE access to the service.

- LocalSystem has SERVICE_ENUMERATE_DEPENDENTS, SERVICE_INTERROGATE, SERVICE_PAUSE_CONTINUE, SERVICE_QUERY_CONFIG, SERVICE_QUERY_STATUS, SERVICE_START, SERVICE_STOP, SERVICE_USER_DEFINED_CONTROL, and READ_CONTROL access to the service.

- Authenticated users have SERVICE_ENUMERATE_DEPENDENTS, SERVICE_INTERROGATE, SERVICE_QUERY_CONFIG, SERVICE_QUERY_STATUS, SERVICE_USER_DEFINED_CONTROL, and READ_CONTROL access to the service.

- On Windows 2000 Professional and Windows 2000 Server, Power Users have SERVICE_QUERY_CONFIG, SERVICE_QUERY_STATUS, SERVICE_ENUMERATE_DEPENDENTS, SERVICE_INTERROGATE, SERVICE_START, SERVICE_STOP, SERVICE_PAUSE_CONTINUE, SERVICE_USER_DEFINED_CONTROL, and READ_CONTROL access to the service.

The *dwServiceType* parameter tells the system whether the executable file contains one or multiple services. Pass SERVICE_WIN32_OWN_PROCESS when the executable implements a single service, or SERVICE_WIN32_SHARE_PROCESS when the executable implements two or more services. You can also combine the SERVICE_INTERACTIVE_PROCESS flag with either SERVICE_WIN32_OWN_PROCESS or SERVICE_WIN32_SHARE_PROCESS if you want the services in a process to interact with the user's desktop.

NOTE

For SERVICE_WIN32_SHARE_PROCESS service executables, if one service requires the SERVICE_INTERACTIVE_PROCESS flag, all services must use this flag. When the system starts the first service, the setting of that service determines whether the whole process is allowed to interact with the desktop.

The *dwStartType* parameter tells the system when the service should be started. A value of SERVICE_AUTO_START instructs the SCM to start the service when the machine boots. A value of SERVICE_DEMAND_START tells the system that the service should not be started when the machine boots; an administrator can start the service manually. In addition, SERVICE_DEMAND_START specifies that the service is a demand-start service, which tells the SCM to automatically start the service if the administrator attempts to start a service that depends on it. I'll talk more about service dependencies shortly. A value of SERVICE_DISABLED prevents the system from starting the service at all.

A service is a very important part of the system, so the system needs to know what it should do if the service fails to start. This instruction is the job of the *dwErrorControl* parameter. Passing a value of SERVICE_ERROR_IGNORE or SERVICE_ERROR_NORMAL tells the system to log the service's error in the system's event log and continue starting the system. The difference between these two codes is that SERVICE_ERROR_NORMAL has the system display a message box notifying the user that the service failed to start. Services that are demand-started should always specify SERVICE_ERROR_IGNORE.

The values of SERVICE_ERROR_SEVERE and SERVICE_ERROR_CRITICAL tell the system to abort startup when the service fails to start. When a service fails and one of these codes is specified, the system logs the error in the system's event log and then reboots automatically using the last-known good configuration. If the system is booting the last-known good configuration and a service with an error control of SERVICE_ERROR_SEVERE fails to start, the system continues to boot. If a service with an error control of SERVICE_ERROR_CRITICAL fails to start, the system will also abort the booting of the last-known good configuration.

CreateService's *pszPathName* parameter identifies the full pathname of the executable that contains the service or services. Many service files are installed in the \WINNT\System32 directory, but you can place a service executable anywhere in the file system.

Now we get to the issue of service dependencies. Loosely speaking, a service is like part of the operating system, and many services will not work properly unless they know that other parts of the system are up and running first. When the system boots, it follows an algorithm that dictates the order in which services should start. Microsoft has divided the system services into a set of predefined groups, listed here:

- System Reserved
- Boot Bus Extender
- System Bus Extender (PCMCIA support)
- SCSI miniport (SCSI device drivers)
- Port
- Primary disk (floppy/hard drives)
- SCSI class (SCSI drives)
- SCSI CDROM class (CD-ROM drives)
- Filter (CD audio)
- Boot file system (fast FAT drive access)
- Base (system beep)
- Pointer Port (mouse support)
- Keyboard Port (keyboard support)
- Pointer Class (more mouse support)
- Keyboard Class (more keyboard support)
- Video Init (video support)
- Video (video chip support)
- Video Save (more video support)
- File system (CD-ROM and NTFS file system support)
- Event log (event log support)
- Streams Drivers
- NDIS Wrapper
- PNP_TDI (NetBT and TCP/IP support)
- NDIS (network support)
- TDI (AFD networking support and DHCP)
- NetBIOSGroup (NetBIOS support)
- PlugPlay
- SpoolerGroup (print spooling support)
- NetDDEGroup (network DDE support)
- Parallel arbitrator (parallel port support)
- Extended base (modem, serial, and parallel support)
- RemoteValidation (net logon support)
- PCI Configuration
- MS Transactions

You can also find this list in the registry under the following registry subkey:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder
```

As the system boots, it iterates through the list, loading any device drivers and services that are part of each group. For example, the system loads all the device drivers and services that are part of the System Reserved group before loading the device drivers and services that are part of the SCSI miniport group.

When you add a service to the SCM's database, you can assign it to be in one of the predefined groups in the preceding list by passing the name of the group in *CreateService*'s *pszLoadOrderGroup* parameter. Usually, your services don't need to load early in the system's boot cycle and should load after all the grouped devices and services have started to run. To make your service load after all the more system-critical device drivers and services, simply pass NULL for the *pszLoadOrderGroup* parameter.

If you are adding a device driver to the SCM (as opposed to a service), you can get even greater granularity in establishing the driver's start time by specifying a tag ID. Services cannot take advantage of this additional granularity and must always pass NULL to *CreateService*'s *pdwTagId* parameter. If you're interested in device drivers, the Platform SDK documentation and DDK documentation discuss the *pdwTagId* parameter as well as two additional start options (SERVICE_BOOT_START and SERVICE_SYSTEM_START).

In addition to telling the SCM that your service is part of a particular load-order group, you can tell the SCM that your service requires certain other services and groups to be running before your service can run. For example, the Computer Browser service requires that the Workstation and Server services be up and running before it can work properly, and the ClipBook service requires that the Network DDE service be running.

Specifying which services your service depends on is typically much more useful than indicating that your service is part of a group. You use *CreateService*'s *pszDependencies* parameter to tell the SCM's database which services you depend on. If your service has no dependencies, simply pass NULL for this parameter.

The *pszDependencies* parameter is a little unusual because you must pass the address of a double zero-terminated array of zero-separated names. In other words, *pszDependencies* must point to a block of memory that contains a set of zero-terminated strings with an extra zero character at the end of the buffer.

So to create a service that is dependent on the Workstation service (like the Alerter service), you would set *pszDependencies* (as shown in the following code) before passing it to *CreateService*:

```
// The buffer below ends with two zero characters
PCTSTR pszDependencies = TEXT("LanmanWorkstation\0");
CreateService(..., pszDependencies, ...);
```

And to create a service that is dependent on the Workstation and Remote Procedure Call (RPC) services (such as the Messenger service), you would set *pszDependencies* like this:

```
// The buffer below separates the strings with a zero character
// and ends with two zero characters
PCTSTR pszDependencies = TEXT("LanmanWorkstation\0RpcSs\0");
CreateService(..., pszDependencies, ...);
```

A service can also be dependent on a group rather than on a single service, but this is very uncommon. Dependency on a load-order group means that the service can run if at least one member of the group is running after an attempt has been made to start all members of the group. To specify a group in the *pszDependencies* buffer, you must precede the group name with the special SC_GROUP_IDENTIFIER character, defined in WinSvc.h as follows:

```
#define SC_GROUP_IDENTIFIERW           L'+'
#define SC_GROUP_IDENTIFIERA           '+'
```

```
#ifdef UNICODE
#define SC_GROUP_IDENTIFIER            SC_GROUP_IDENTIFIERW
#else
#define SC_GROUP_IDENTIFIER            SC_GROUP_IDENTIFIERA
#endif
```

Therefore, to create a service that is dependent on the Workstation service and the TDI group, you would set *pszDependencies* like this:

```
// The buffer below specifies two dependencies: the Workstation service
// and the TDI group (a group because of the '+' before TDI)
PCTSTR pszDependencies = TEXT("LanmanWorkstation\0+TDI\0");
CreateService(..., pszDependencies, ...);
```

When setting the *pszDependencies* value, you can specify as many services and groups as you like. Just remember to place a zero character between each service or group, to put a plus sign in front of all group names, and to place a terminating zero character just before the closing quote.

This brings us now to *CreateService*'s final two parameters: *pszUserName* and *pszUserPswd*. These parameters allow you to specify under which user account the service is to run. To have the service run under the LocalSystem account (the most common case), pass NULL for these two parameters. If you want the service to run under a specific user account, pass an account name in the form of *DomainName\UserName* for the *pszUserName* parameter, and pass the user account's password for the *pszUserPswd* parameter.

NOTE

Interactive services must be configured to run under the LocalSystem account. *CreateService* fails to add the service to the SCM's database if you attempt to add an interactive service with a non-LocalSystem account.

If *CreateService* is successful in adding the service to the SCM's database, a non-NULL handle is returned. This handle is required by other functions to manipulate the service. Be sure you pass this handle to *CloseServiceHandle* when you're finished using it. If *CreateService* fails, it returns NULL, and a call to *GetLastError* returns a value indicating the reason for failure. Here are the most common reasons *CreateService* can fail:

- The handle returned from *OpenSCManager* doesn't have SC_MANAGER_CREATE_SERVICE access.
- The new service specifies a circular dependency.
- A service with the same display name already exists.
- The specified service name is invalid.
- A parameter is invalid.
- The specified user account does not exist.

I always write my service executables so that they can install themselves. In my *(w)main* or *(w)WinMain* function, I call a function such as *ServiceInstall* (shown in the next code snippet) if "-install" is passed as a command-line argument. The TimeService sample service presented in the last chapter demonstrates this technique.

```
void ServiceInstall(PCTSTR pszInternalName, PCTSTR pszDisplayName,
   DWORD dwServiceType, DWORD dwStartType, DWORD dwErrorControl) {
```

```
   // Open the SCM database to add a service
   SC_HANDLE hSCM = OpenSCManager(NULL, NULL, SC_MANAGER_CREATE_SERVICE);

   // Get the full pathname of our service's executable
   char szModulePathname[_MAX_PATH];
   GetModuleFileName(NULL, szModulePathname, sizeof(szModulePathname));

   // Add this service to the SCM database
   SC_HANDLE hService = CreateService(
      hSCM, pszInternalName, pszDisplayName, 0, dwServiceType,
      dwStartType, dwErrorControl, szModulePathname,
      NULL, NULL, NULL, NULL, NULL);

   // Close the newly created service and the SCM
   CloseServiceHandle(hService);
   CloseServiceHandle(hSCM);
}
```

NOTE

For the sake of clarity, the preceding code does not conduct any error checking. *OpenSCManager* and *CreateService* can fail for many reasons. Please add the proper error checking when adding code like this into your own application.

[Previous] [Next]

Deleting a Service from the SCM's Database

Every good software package supports uninstall just as well as it supports install. So you'll need to know how to remove a service too. To remove a service, you must first open it:

```
SC_HANDLE OpenService(
   SC_HANDLE hSCManager,
   PCTSTR    pszInternalName,
   DWORD     dwDesiredAccess);
```

To the *OpenService* function, you pass the handle returned from *OpenSCManager*, followed by the internal name of the service (the same value that you passed in *CreateService*'s *pszServiceName* parameter), and then DELETE for the desired access. Now that you have the handle to the specific service, you can delete the service by calling *DeleteService*, passing the handle returned from *OpenService*:

```
BOOL DeleteService(SC_HANDLE hService);
```

DeleteService does not actually delete the service right away; it simply marks the service for deletion. The SCM will delete the service only when the service stops running and after all open handles to the service have been closed.

I also write my service executables so that they can delete their services from the SCM's database. If "-remove" is passed as a command-line argument, I call a function such as *ServiceRemove*, shown here:

```
void ServiceRemove(PCTSTR pszInternalName) {

   // Open the SCM database
   SC_HANDLE hSCM = OpenSCManager(NULL, NULL, SC_MANAGER_CONNECT);

   // Open the service for delete access
   SC_HANDLE hService = OpenService(hSCM, pszInternalName, DELETE);

   // Mark the service for deletion
```

```
   // NOTE: The service is not deleted until all handles
   // to it are closed and the service stops running
   DeleteService(hService);

   // Close the service and the SCM
   CloseServiceHandle(hService);
   CloseServiceHandle(hSCM);
}
```

NOTE

For clarity, the preceding code does not have any error checking. *OpenSCManager*, *OpenService*, and *DeleteService* can fail for many reasons. Please add the proper error checking when adding code like this into your own application.

[Previous] [Next]

Starting and Controlling a Service

As I mentioned earlier, many services ship with a client-side application that allows administrators to start, stop, pause, continue, and otherwise control a service. Writing such a service control program is very easy. Here's how one works: The program first opens the SCM on the desired machine by calling *OpenSCManager* using the SC_MANAGER_CONNECT access right. Then the program calls *OpenService* to open the service you want to control by using the desired combination of SERVICE_START, SERVICE_STOP, SERVICE_PAUSE_CONTINUE, SERVICE_USER_DEFINED_CONTROL, and SERVICE_INTERROGATE access rights. After a service is opened, calling *StartService* can start it:

```
BOOL StartService(
   SC_HANDLE hService,
   DWORD     dwArgc,
   PCTSTR*   pszArgv);
```

The *hService* parameter identifies the opened service, and the *dwArgc* and *pszArgv* parameters indicate the set of arguments that you wish to pass to the service's *ServiceMain* function. Most services don't use these parameters, so 0 and NULL are usually passed for the last two arguments. Remember that starting a service can cause several services to start if the service you're starting depends on other services or groups. Here are some of the main reasons *StartService* can fail:

- The handle returned from *OpenService* doesn't have SERVICE_START access.
- The service's executable file is not in the location specified in the directory.
- The service is already running, disabled, or marked for deletion.
- The SCM's database is locked. (I'll talk more about this later in this chapter.)
- The service depends on another service that doesn't exist or failed to start.
- The user account for the service could not be validated.
- The service didn't respond to the start request in a timely fashion.

Note that the *StartService* function returns as soon as the service's primary thread is created, so the service might not be ready to process control codes or handle client requests by the time *StartService* returns. Also, a service must not call *StartService* while it is initializing, or a deadlock (lasting 80 seconds) will occur. The

problem is that the SCM locks the SCM database while starting a service, preventing another service from starting.

Once the service is running, you can call *ControlService* to send controls to it:

```
BOOL ControlService(
   SC_HANDLE       hService,
   DWORD           dwControl,
   SERVICE_STATUS* pss);
```

Again, the *hService* parameter identifies the opened service that you wish to control. The *dwControl* parameter indicates what you wish the service to do and can be any one of the following values:

- SERVICE_CONTROL_STOP
- SERVICE_CONTROL_PAUSE
- SERVICE_CONTROL_CONTINUE
- SERVICE_CONTROL_INTERROGATE
- SERVICE_CONTROL_PARAMCHANGE

Notice that these codes are the same codes that your *HandlerEx* function receives (as discussed in Chapter 3). In addition to these values, you can send a user-defined code ranging from 128 through 255. Note that *ControlService* fails if you pass a value of SERVICE_CONTROL_SHUTDOWN; only the system can send this code to a service's handler function.

ControlService's last parameter, *pss*, must point to a SERVICE_STATUS structure. The function will initialize the members of this structure to report the service's last-reported status information. You can examine this information after *ControlService* returns to see how the service is doing. Here are some of the main reasons *ControlService* can fail:

- The handle returned from *OpenService* doesn't have the proper access.
- The service can't be stopped because other services depend on it. In this case, your application must stop the dependent services first.
- The control code is not valid or is not acceptable to the service. Remember from Chapter 3 that a service sets the SERVICE_STATUS structure's *dwControlsAccepted* member when it calls *SetServiceStatus*.
- The control code can't be sent to the service because the service is reporting SERVICE_STOPPED, SERVICE_START_PENDING, or SERVICE_STOP_PENDING.
- The service is not running.
- The service has not returned from its *HandlerEx* function in a timely fashion (within 30 seconds).

Certainly, if the service's handler function processes the call, you would expect the returned SERVICE_STATUS structure to be initialized properly. But what do you think the contents of the SERVICE_STATUS structure will be if you attempt to send a SERVICE_CONTROL_INTERROGATE control to a stopped service? Well, you'll be pleased to know that Microsoft has enhanced the *ControlService* function so that it returns a valid SERVICE_STATUS structure if the function fails with an error code of ERROR_INVALID_SERVICE_CONTROL, ERROR_SERVICE_CANNOT_ACCEPT_CTRL, or ERROR_SERVICE_NOT_ACTIVE. The following code shows how to stop a service:

```
void StopService(PCTSTR pszInternalName) {

   // Open the SCM and the desired service
   SC_HANDLE hSCM = OpenSCManager(NULL, NULL, SC_MANAGER_CONNECT);
   SC_HANDLE hService = OpenService(hSCM, pszInternalName,
      SERVICE_STOP | SERVICE_QUERY_STATUS);

   // Tell the service to stop
   SERVICE_STATUS ss;
   ControlService(hService, SERVICE_CONTROL_STOP, &ss);

   // Wait up to 15 seconds for the service to stop
   WaitForServiceState(hService, SERVICE_STOPPED, &ss, 15000);

   // Close the service and the SCM
   CloseServiceHandle(hService);
   CloseServiceHandle(hSCM);
}
```

NOTE

For clarity, the preceding code does not have any error checking. *OpenSCManager*, *OpenService*, and *ControlService* can fail for many reasons. Please add the proper error checking when adding code like this into your own application.

You'll notice that *StopService* calls the *WaitForServiceState* function. The *WaitForServiceState* function is not a Windows function but rather a function I wrote, and it demonstrates how to handle the polling of a service's state properly. Consider the following scenario. Using the Services snap-in, you initiate a stop request for a service, causing the SCM to notify the selected service that it should stop. The service should respond by calling *SetServiceStatus*, with the SERVICE_STATUS structure's *dwCurrentState* member set to SERVICE_STOP_PENDING. However, the service has not stopped yet, so the Services snap-in doesn't update its user interface to reflect that the service has stopped. Unfortunately, the system does not provide a way for an application to be notified of service state changes, so an SCP must periodically poll the service to determine when its state has changed. The *WaitForServiceState* function handles this polling.

```
BOOL WaitForServiceState(SC_HANDLE hService, DWORD dwDesiredState,
   SERVICE_STATUS* pss, DWORD dwMilliseconds) {

   BOOL  fServiceOk = TRUE;
   BOOL  fFirstTime = TRUE; // Don't compare state/checkpoint the
                            // first time
   DWORD dwLastState = 0, dwLastCheckPoint = 0;
   DWORD dwTimeout = GetTickCount() + dwMilliseconds;

   // Loop until service reaches desired state, error occurs, or timeout
   for (;;) {
      // Get current state of service
      fServiceOk = ::QueryServiceStatus(hService, pss);

      // If we can't query the service, we're done
      if (!fServiceOk) break;

      // If the service reaches the desired state, we're done
      if (pss->dwCurrentState == dwDesiredState) break;

      // If timeout, we're done
      if ((dwMilliseconds != INFINITE) && (dwTimeout < GetTickCount())){
         fServiceOk = FALSE;
         SetLastError(ERROR_TIMEOUT);
         break;
      }

      // If first time, save service's state/checkpoint
```

```
      if (fFirstTime) {
         dwLastState      = pss->dwCurrentState;
         dwLastCheckPoint = pss->dwCheckPoint;
         fFirstTime       = FALSE;
      } else {
         // If not first time and state changed, save state/checkpoint
         if (dwLastState != pss->dwCurrentState) {
            dwLastState      = pss->dwCurrentState;
            dwLastCheckPoint = pss->dwCheckPoint;
         } else {
            // State hasn't changed; make sure checkpoint isn't
            // decreasing
            if (pss->dwCheckPoint >= dwLastCheckPoint) {
               // Good checkpoint; save it
               dwLastCheckPoint = pss->dwCheckPoint;
            } else {
               // Bad checkpoint, service failed, we're done!
               fServiceOk = FALSE;
               break;
            }
         }
      }
      // We're not done; wait the specified period of time
      // Poll 1/10 of the wait hint
      DWORD dwWaitHint = pss->dwWaitHint / 10;
      // At most once a second
      if (dwWaitHint <  1000) dwWaitHint = 1000;
      // At least every 10 seconds
      if (dwWaitHint > 10000) dwWaitHint = 10000;
      Sleep(dwWaitHint);
   }

   // Note: The last SERVICE_STATUS is returned to the caller so
   // that the caller can check the service state and error codes
   return(fServiceOk);
}
```

We all know that polling is a horrible thing to do because it wastes precious CPU cycles, but we really have no choice in this case. Fortunately, the situation is not as bad as you think because the SERVICE_STATUS structure contains the *dwWaitHint* member. When a service calls *SetServiceStatus*, the *dwWaitHint* member must indicate how many milliseconds the program that is sending the control code should wait before polling the service's status again.

The service control program should also examine the checkpoint returned from the service during the polling process to make sure that it never decreases. If a service returns a smaller checkpoint value, the service control program should assume that the service has failed.

You'll notice that the *WaitForServiceState* function calls *QueryServiceStatus*:

```
BOOL QueryServiceStatus(
   SC_HANDLE       hService,
   SERVICE_STATUS* pss);
```

QueryServiceStatus asks the SCM to return the service's last cached state information (set when the service last called *SetServiceStatus*). Calling *QueryServiceStatus* is similar to calling *ControlService* and passing the SERVICE_CONTROL_INTERROGATE code, but calling *ControlService* with SERVICE_CONTROL_INTERROGATE sends an action request to the service to update the current state information. Another difference between calling *QueryServiceStatus* and calling *ControlService* is that *QueryServiceStatus* always returns in a timely fashion, whereas *ControlService* might return failure if the service has stopped responding. If a service's handler function is busy when you send an interrogate code to it, the service might not be able to respond for a while, which will cause your call to *ControlService* to wait (possibly for 30 seconds). Of course, the downside to using *QueryServiceStatus* is that the SCM's cached data

might not accurately reflect the most up-to-date state of the service. Now that you know the trade-offs, you can query a service's state using whichever method works best in your situation.

In addition to the *QueryServiceStatus* function, Microsoft recently added the new *QueryServiceStatusEx* function:

```
BOOL QueryServiceStatusEx(
   SC_HANDLE      hService,
   SC_STATUS_TYPE InfoLevel,
   PBYTE          pbBuffer,
   DWORD          cbBufSize,
   PDWORD         pdwBytesNeeded);
```

This function queries a service's status and initializes the new SERVICE_STATUS_PROCESS structure:

```
typedef struct _SERVICE_STATUS_PROCESS {
   DWORD  dwServiceType;
   DWORD  dwCurrentState;
   DWORD  dwControlsAccepted;
   DWORD  dwWin32ExitCode;
   DWORD  dwServiceSpecificExitCode;
   DWORD  dwCheckPoint;
   DWORD  dwWaitHint;
   DWORD  dwProcessId;
   DWORD  dwServiceFlags;
} SERVICE_STATUS_PROCESS, *LPSERVICE_STATUS_PROCESS;
```

This structure is identical to the SERVICE_STATUS structure except that it has two additional members at the end: *dwProcessId* and *dwServiceFlags*. The *dwProcessId* member indicates the ID of the process that contains the service, and *dwServiceFlags* indicates some additional information about the service. If *dwServiceFlags* contains SERVICE_RUNS_IN_SYSTEM_PROCESS (the only flag currently defined), the service is running in a system process such as Services.exe or LSASS.exe. You should never attempt to kill services running in a system process since the process itself is an integral component of the operating system.

[Previous] [Next]

Reconfiguring a Service

The *CreateService* function adds the new service's entry to the SCM's database. It's unusual to do so, but occasionally you might want to change the information in the database. For instance, the user account associated with the entry might need its password changed, or you might want to change the service from manual start to automatic start. Well, Windows offers you four functions that help you reconfigure a service. The first function, *QueryServiceConfig*, retrieves the service's entry from the SCM's database:

```
BOOL QueryServiceConfig(
   SC_HANDLE             hService,
   QUERY_SERVICE_CONFIG* pqsc,
   DWORD                 dwBufSize,
   PDWORD                pdwBytesNeeded);
```

When you call this function, the *hService* parameter identifies the service you wish to query. The handle must be opened with SERVICE_QUERY_CONFIG access. You must also allocate a memory buffer large enough to hold a QUERY_SERVICE_CONFIG structure and all the service's string data. A QUERY_SERVICE_CONFIG structure looks like this:

```
typedef struct _QUERY_SERVICE_CONFIG {
   DWORD  dwServiceType;
   DWORD  dwStartType;
```

```
    DWORD  dwErrorControl;
    PTSTR  lpBinaryPathName;
    PTSTR  lpLoadOrderGroup;
    DWORD  dwTagId;
    PTSTR  lpDependencies;
    PTSTR  lpServiceStartName;
    PTSTR  lpDisplayName;
} QUERY_SERVICE_CONFIG, *LPQUERY_SERVICE_CONFIG;
```

The *dwBufSize* parameter of *QueryServiceConfig* tells the function how big your buffer is, and the DWORD pointed to by the *pdwBytesNeeded* parameter is filled in by the function, telling you how big the buffer needs to be. The buffer that you pass to *QueryServiceConfig* will always have to be bigger than the size of a QUERY_SERVICE_CONFIG structure because the function copies all the service's string data into the buffer immediately after copying the fixed-size data structure. The PTSTR members will point to memory addresses inside this buffer.

Once you have the service's current configuration, you can change it by calling the following:

```
BOOL ChangeServiceConfig(
    SC_HANDLE hService,
    DWORD     dwServiceType,
    DWORD     dwStartType,
    DWORD     dwErrorControl,
    PCTSTR    pszPathName,
    PCTSTR    pszLoadOrderGroup,
    PDWORD    pdwTagId,
    PCTSTR    pszDependencies,
    PCTSTR    pszUserName,
    PCTSTR    pszUserPswd,
    PCTSTR    pszDisplayName);
```

As you can see, these parameters are practically identical to those passed to *CreateService*. The differences are that you cannot change the service's internal name and that the display name is the last parameter. When you use *ChangeServiceConfig*, the changes do not take effect until the service stops.

In addition to providing *QueryServiceConfig* and *ChangeServiceConfig*, Windows offers the *QueryServiceConfig2* and *ChangeServiceConfig2* functions, which allow you to get and set a service's description and failure actions. Customers have long desired both of these features, and it's nice to see Microsoft add them to Windows 2000.

A service's description is simply a string (limited to 1024 characters) that describes the service. This string appears in the Services snap-in and really helps administrators understand the purpose of each service installed and running on the system. You are strongly encouraged to add descriptions when you add a service to the SCM's database.

Using failure actions, an administrator can tell the system what action to take when a service fails. (A service fails if its process dies without the service setting its state to SERVICE_STOPPED.) An administrator can have the SCM automatically restart the service, run an application, or reboot the computer. For more information about *QueryServiceConfig2* and *ChangeServiceConfig2*, please see the Platform SDK documentation.

[Previous] [Next]

Locking the SCM's Database

If you find yourself making several changes to services in the SCM's database, you might want to temporarily stop the SCM from starting any services by obtaining the SCM's lock. A good time to obtain the lock is when

you want to query a service's configuration and then change it "atomically." Locking the SCM might also be useful when the service is dependent on other services. To prevent the SCM from starting any more services, call the *LockServiceDatabase* function, passing it the handle returned from a call to *OpenSCManager* by using the SC_MANAGER_LOCK access:

```
SC_LOCK LockServiceDatabase(SC_HANDLE hSCManager);
```

The *LockServiceDatabase* function returns a value that identifies the lock. Hold on to this value because you'll need to pass it to *UnlockServiceDatabase* when you wish to release the lock:

```
BOOL UnlockServiceDatabase(SC_LOCK scLock);
```

Only one process at a time can own the SCM's lock, and of course, you should own the lock for as short a time as possible. If the process that owns the lock terminates, the SCM automatically reclaims the lock so that services can start again.

Note that the lock is not automatically released if you close the handle to the SCM. Consider this example:

```
SC_HANDLE hSCM = OpenSCManager(NULL, NULL, SC_MANAGER_LOCK);
// Lock the SCM's database
SC_LOCK scLock = LockServiceDatabase(hSCM);
CloseServiceHandle(hSCM);
// NOTE: The database is still locked

UnlockServiceDatabase(scLock);
// The database is now unlocked
```

The *QueryServiceLockStatus* function returns information that allows you to see the status of the SCM's lock:

```
BOOL QueryServiceLockStatus(
   SC_HANDLE                   hSCManager,
   QUERY_SERVICE_LOCK_STATUS* pqsls,
   DWORD                       dwBufSize,
   PDWORD                      pdwBytesNeeded);
```

If the SCM is locked, the function also returns which user account owns the lock and how long the lock has been owned. All this information is returned via a QUERY_SERVICE_LOCK_STATUS structure:

```
typedef struct _QUERY_SERVICE_LOCK_STATUS {
   DWORD  fIsLocked;
   PTSTR  lpLockOwner;
   DWORD  dwLockDuration;
} QUERY_SERVICE_LOCK_STATUS, *LPQUERY_SERVICE_LOCK_STATUS;
```

As with the *QueryServiceConfig* function, the buffer that you pass to *QueryServiceLockStatus* must actually be larger than the size of the structure. Again, this is because the structure contains a string value (*lpLockOwner*) that will be copied into the buffer immediately after the fixed-size structure.

[Previous] [Next]

Miscellaneous Service Control Program Functions

Windows offers just a few more service control functions. I'd like to mention them briefly to complete our discussion.

One function looks up a service's display name from its internal name:

```
BOOL GetServiceDisplayName(
   SC_HANDLE hSCManager,
   PCTSTR    pszServiceName,
   PTSTR     pszDisplayName,
   PDWORD    pdwChars);
```

And another function does the reverse:

```
BOOL GetServiceKeyName(
   SC_HANDLE hSCManager,
   PCTSTR    pszDisplayName,
   PTSTR     pszServiceName,
   PDWORD    pdwChars);
```

The parameters passed to these functions should be self-explanatory, so I won't go into them here. See the Platform SDK documentation for more information.

The *EnumServicesStatusEx* function asks the SCM to enumerate all the services (and their states) contained in the database:

```
BOOL EnumServicesStatusEx(
   SC_HANDLE    hSCManager,
   SC_ENUM_TYPE InfoLevel,
   DWORD        dwServiceType,
   DWORD        dwServiceState,
   PBYTE        pbServices,
   DWORD        dwBufSize,
   PDWORD       pdwBytesNeeded,
   PDWORD       pdwServicesReturned,
   PDWORD       pdwResumeHandle,
   PCTSTR       pszGroupName);
```

The Services snap-in calls this function to populate its list of installed services. The first parameter, *hSCManager*, identifies the SCM whose services you wish to enumerate. The second parameter, *InfoLevel*, must be SC_ENUM_PROCESS_INFO, which tells the function that you wish to retrieve the name and service status for each service. Currently no other valid values exist for the *InfoLevel* parameter.

The third parameter, *dwServiceType*, tells the function to enumerate services or device drivers. For services, pass SERVICE_WIN32. The fourth parameter, *dwServiceState*, allows you to fine-tune your request. You can pass SERVICE_ACTIVE, SERVICE_INACTIVE, or SERVICE_STATE_ALL to enumerate running services, stopped services, or both. The last parameter, *pszGroupName*, allows you to fine-tune the set of returned services or device drivers even more. If *pszGroupName* identifies a group, only services that are part of the specified group are enumerated. If *pszGroupName* identifies an empty string (""), services that are not part of any group are enumerated. Finally, if *pszGroupName* is NULL, all services are enumerated.

All the remaining parameters are concerned with the buffer that gets the returned data. When you call *EnumServicesStatusEx*, you pass it a buffer that will be filled with an array of ENUM_SERVICE_STATUS_PROCESS structures that look like this:

```
typedef struct _ENUM_SERVICE_STATUS_PROCESS {
   LPTSTR                  lpServiceName;
   LPTSTR                  lpDisplayName;
   SERVICE_STATUS_PROCESS ServiceStatusProcess;
} ENUM_SERVICE_STATUS_PROCESS, *LPENUM_SERVICE_STATUS_PROCESS;
```

Because each service has string data associated with it, the string data is copied to the end of the buffer. The fixed-size ENUM_SERVICE_ STATUS_PROCESS structures are contiguous at the beginning of the buffer, so you can easily iterate through the returned data structures. When the function returns, the DWORD pointed to by *pdwServicesReturned* contains the number of ENUM_SERVICE_STATUS_PROCESS structures that fit into the buffer.

The first time you call *EnumServicesStatusEx*, make sure that the DWORD pointed to by *pdwResumeHandle* is initialized to 0. This *pdwResumeHandle* is used in cases in which there is more data than your buffer can hold. If the buffer is too small, *EnumServicesStatusEx* fills this DWORD with a special value that it uses the next time you call *EnumServicesStatusEx* so that it knows where to continue the enumeration. The following code shows how to allocate a buffer that is large enough to hold all the service data so that multiple calls to *EnumServicesStatusEx* are not necessary:

```
DWORD dwBytesNeeded, dwServicesReturned, dwResumeHandle = 0;
EnumServicesStatusEx(hSCManager, SC_ENUM_PROCESS_INFO, SERVICE_WIN32,
   SERVICE_STATE_ALL, NULL, 0, &dwBytesNeeded,
   &dwServicesReturned, &dwResumeHandle, NULL);

ENUM_SERVICE_STATUS_PROCESS* pessp =
   (ENUM_SERVICE_STATUS_PROCESS*) _alloca(dwBytesNeeded);
EnumServicesStatusEx(hSCManager, SC_ENUM_PROCESS_INFO, SERVICE_WIN32,
   SERVICE_STATE_ALL, (PBYTE) pessp, dwBytesNeeded, &dwBytesNeeded,
   &dwServicesReturned, &dwResumeHandle, NULL);

for (DWORD dw = 0; dw < dwServicesReturned; dw++) {
   // Refer to the members inside pessp, for example
   _tprintf(TEXT("%s\n"), pessp[dw].lpDisplayName);
}
```

NOTE

For the sake of clarity, the preceding code does not have any error checking. In particular, the call to *_alloca* could raise a stack overflow exception. Structured exception handling (SEH) is required to recover from this exception gracefully and without the service terminating. Using SEH in this code is particularly important because some machines running Windows 2000 Advanced Server and Windows 2000 Data Center might have well over 200 services installed on them.

The next function that I'll discuss allows you to determine which services depend on another service:

```
BOOL EnumDependentServices(
   SC_HANDLE            hService,
   DWORD                dwServiceState,
   ENUM_SERVICE_STATUS* pess,
   DWORD                dwBufSize,
   PDWORD               pdwBytesNeeded,
   PDWORD               pdwServicesReturned);
```

This function is similar to *EnumServicesStatusEx*, so all of its parameters should be self-explanatory. The Services snap-in calls this function if you try to stop a service on which other services depend. For example, if I try to stop the Workstation service, I get the dialog box shown in Figure 4-1—*EnumDependentServices* was used to fill in the list of dependent services.

Many developers call *EnumDependentServices* recursively to get each service's dependent services. Doing so is not necessary because the SCM performs the recursion for you while processing the *EnumDependentServices* function. The set of services returned by this function is the complete set; your SCP application should simply iterate through the set to stop each of the services if desired.

Figure 4-1. *The dialog box displayed when an attempt is made to stop a service that has running dependent services*

Finally we come to the last two service control functions, *QueryServiceObjectSecurity* and *SetServiceObjectSecurity*:

```
BOOL QueryServiceObjectSecurity(
   SC_HANDLE            hService,
   SECURITY_INFORMATION dwSecurityInformation,
   PSECURITY_DESCRIPTOR psd,
   DWORD                dwBufSize,
   PDWORD               pdwBytesNeeded);

BOOL SetServiceObjectSecurity(
   SC_HANDLE            hService,
   SECURITY_INFORMATION dwSecurityInformation,
   PSECURITY_DESCRIPTOR psd);
```

These two functions allow you to query and change a security descriptor associated with a service. These functions are rarely called because the default security placed on the service when *CreateService* is called is sufficient for most needs. See Chapter 10 for more discussion of security descriptors and related topics.

[Previous] [Next]

The SuperSCP Sample Application

The SuperSCP sample application ("04 SuperSCP.exe") is an SCP that does just about anything and everything you can imagine to a service. The source code and resource files for the application are in the 04-SuperSCP directory on the companion CD. When you start the program, the window in Figure 4-2 appears.

Figure 4-2. *The interface for the SuperSCP sample application*

The application initializes by querying the local machine's computer name, and it places this name in the Machine field. Then the application communicates with this machine's SCM, enumerating all the installed services and placing each service's internal name into the Internal Name field's combo box. At any time, you can administer the services on a remote machine by entering the remote machine's name in the Machine field and clicking the Refresh button.

As you select entries from the Internal Name field, all the remaining fields are updated. You can change any combination of settings for a service and then click the Reconfigure button to make the changes permanent. Clicking the Security button allows you to alter the security of the selected service. You can also mark a selected service for deletion from the SCM's database by clicking the Remove button. If you are going to perform a number of tasks with the SCM, you can lock it using the Lock SCM button.

To add a new service to the SCM's database, just type the desired internal name into the Internal Name field. If the internal name field contains a value that does not match an existing service's internal name, SuperSCP assumes that the fields describe a new service that you'd like to add to the SCM's database. After entering the internal name of the new service, configure the service any way you'd like. Use the Browse button to help you locate the service's executable file. Also, if you drag and drop an executable file from a folder into the SuperSCP window, the pathname will appear in the Pathname field. Once you've configured the service's settings, click the Create button to add the new service to the SCM's database.

At the bottom of the window is the execution control section which allows you to alter the execution of the service. All the buttons in this section are self-explanatory. On the right is a list box that receives a new entry every second. Each entry indicates an entry number and the service's current state: Stopped, Start Pending, Stop Pending, Running, Continue Pending, Pause Pending, or Paused. After the state is shown, the most recent checkpoint and wait hint reported by the service are shown. The last two fields, WErr and SErr, show the last Win32 error code and service-specific error code reported by the service.

As far as the code goes, nothing is tricky. It's all just a matter of calling the right SCP function at the right time. However, a number of reusable C++ classes do make the implementation significantly easier. The two main C++ classes are CSCMCtrl and CServiceCtrl. The CSCMCtrl class is a thin wrapper on top of SCP functions that talk directly to the SCM. The CSCMCtrl methods include *Open*, *LockDatabase*,

QueryLockOwner, *GetInternalName*, and *GetDisplayName*. In addition, there are methods for creating a snapshot of the SCM's services and enumerating through them. The C++ class really helps here because it handles all of the memory management issues internally.

The CServiceCtrl class is a thin wrapper on top of SCP functions that talk directly to a service. The CServiceCtrl methods include *InstallAndOpen*, *Open*, *Delete*, *Start*, *Control*, *WaitForState*, *QueryStatus*, *QueryConfig*, *QueryDescription*, *ChangeConfig*, *QueryFailureActions*, *ChangeFailureActions*, and *EditSecurity*. Methods for creating a snapshot of a service's dependencies also exist. Again, the C++ class comes in handy because many of these methods' memory management issues are handled internally.

[Previous] [Next]

Chapter 5

The System Registry

Software has become more complex and also more configurable. In Microsoft Windows, that configuration can be achieved with the registry. The registry is a mechanism that makes it much easier for an application or service to maintain persistent configuration settings. The registry is a centrally located hierarchical database that offers the following capabilities:

- Although the registry is composed of multiple physical files, programmers treat it as a single database for storing and retrieving information.
- Designed in a hierarchical fashion, the registry allows an application to impose its own organization on its configuration settings.
- The registry provides support for multiple users and is essentially separated into two parts: a part for local machine settings (HKEY_LOCAL_MACHINE) and a part for user settings (HKEY_USERS).
- The registry provides security capabilities with which permissions and auditing settings can be applied to specific keys.
- The registry allows for multiple data types, including binary, DWORD, string, and multistring.

Before we jump into the details of the registry, I should discuss the two utilities available in Microsoft Windows 2000 for browsing and modifying the registry, RegEdit.exe and RegEdt32.exe. RegEdit.exe is basically the same registry utility found in Microsoft Windows 98, but it lacks features that support the security aspects of the Windows 2000 registry. RegEdt32.exe is a little less intuitive to use, but it has full support for the securing of registry keys and also provides more capabilities for manipulating registry values. For many purposes you can use either utility to browse and modify the registry, but for certain cases you will be restricted to RegEdt32.exe or some other utility.

[Previous] [Next]

The System Registry Structure

The registry is a limited, shared system resource. With it come some rules of etiquette (or convention) as well as some rules that are strictly enforced by the system. Before we go on, I want to briefly cover the structure of the registry in more detail.

As I mentioned earlier, the system registry is laid out in a hierarchy consisting of keys and values. Keys can contain an arbitrary number of keys (or subkeys) and values. Subkeys are equal citizens and can contain an arbitrary number of subkeys and values of their own. Key names must be unique among their siblings and cannot contain a backslash. Figure 5-1 shows the registry's structure.

The registry has a logical structure and a physical structure. Programmers usually concern themselves only with the logical structure. Logically, the registry contains a number of root keys similar to how drive A: and drive C: are considered root directories. A registry's root keys identify the tops of registry trees.

Physically, however, the registry is maintained in multiple files, called hives, on the user's hard drive. Programmers need only to think about the path to a key in the registry. Internally, the operating system determines which hive contains this key, and the system accesses the proper file.

Figure 5-1. *The Windows 2000 registry structure viewed with RegEdit.exe*

The registry for Windows 2000 provides five predefined root keys and a predefined performance data key (HKEY_PERFORMANCE_DATA), which are shown in Table 5-1. The HKEY_LOCAL_MACHINE and HKEY_USERS keys are root keys under which all subkeys in the registry exist. This logical division of the registry under HKEY_LOCAL_MACHINE and HKEY_USERS refines the registry's purpose by addressing two distinct needs: the storage of machine-specific configuration information, and the storage of user-specific configuration information. The next three predefined registry keys are virtual bookmarks for sections of the hierarchy falling under the HKEY_LOCAL_MACHINE and HKEY_USERS keys. For example, the subkeys found under HKEY_LOCAL_MACHINE\Software\Classes can also be accessed under the system-defined key HKEY_CLASSES_ROOT.

Table 5-1. *The predefined system registry keys for Windows 2000*

Predefined Key Name	Description
HKEY_LOCAL_MACHINE	The set of registry data that applies to the local machine, regardless of which user is logged on.
HKEY_USERS	The set of registry data that applies to specific users. HKEY_USERS includes trees for the default user and any currently loaded user profiles.
HKEY_CURRENT_USER	A system-defined bookmark or alias for a subtree under

	HKEY_USERS that dynamically points to the registry information for the user associated with the calling thread. There are special rules about the calling thread's user that affect services using impersonation. For more information, see Chapter 11.
HKEY_CLASSES_ROOT	A system-defined bookmark or alias for HKEY_LOCAL_MACHINE\Software\Classes. This tree houses configuration information for registered COM components and shell-related associations in the system.
HKEY_CURRENT_CONFIG	A system-defined bookmark or alias referring to HKEY_LOCAL_MACHINE\System\CurrentControlSet\Hardware Profiles\Current. This tree of the registry houses hardware configuration information.
HKEY_PERFORMANCE_DATA	This predefined key refers to realtime performance data supplied by the system, services, and applications. It is not backed by physical values within the registry. Rather, the registry functions provide a uniform method to dynamically retrieve system performance data.

Like subkey names, values under a single subkey must all have unique names with the exception of the *default* value, which is unnamed. All values, including the default value, are optional. That is to say, a key can contain one value, many values, or no values at all.

NOTE

Like the filenames of system components, key names, subkey names, and value names are not typically localized for international versions of Windows. Naturally, however, any human-readable data stored in values commonly uses local translations.

Each value has data associated with it. A value's data is formatted as one of the system-defined data types, shown in Table 5-2. I will describe the uses and ramifications of some of these data types later in this chapter.

Table 5-2. *Registry value data types*

Registry Value Data Types	Description
REG_BINARY	A stream of bytes.
REG_DWORD	A 32-bit number.
REG_DWORD_LITTLE_ENDIAN	A 32-bit number in little-endian format. All Windows systems store numbers in little-endian format.
REG_DWORD_BIG_ENDIAN	A 32-bit number in big-endian format. Some non-Windows systems, such as some hardware that runs UNIX and a variety of Motorola CPUs, use big-endian format to store integers.
REG_QWORD	A 64-bit number.
REG_QWORD_LITTLE_ENDIAN	A 64-bit number in little-endian format. This is equivalent to REG_QWORD.
REG_EXPAND_SZ	A zero-terminated string that contains unexpanded references to environment variables in the format "%VARIABLE_NAME%". This type is often used to store file paths, because variables such as "%SystemRoot%" allow your path to stay correct even when the user adjusts the paths for his system. (I'll discuss how to use

	values of this type later in "Storing Data in the System Registry.")
REG_LINK	A Unicode symbolic link used by the system. Your application or server code should neither query nor store values of type REG_LINK.
REG_MULTI_SZ	A series of zero-terminated strings, with two zero characters following the final string in the series.
REG_NONE	No defined value type. It is functionally the same as REG_BINARY.
REG_RESOURCE_LIST	A device-driver resource list. Not used in user-mode applications.
REG_SZ	A zero-terminated string.

Microsoft defines conventions that programmers should use when accessing the registry. However, the operating system is unable to enforce these conventions. For example, a common mistake developers make is to place machine-specific settings under the user-specific part of the registry, or vice versa.

[Previous] [Next]

Registry Conventions

This section discusses conventions for registry usage, focusing on the portions specifically applicable to server software designed for Windows.

The integrity of the registry is maintained through the marriage of system-enforced rules and convention. Convention keeps the registry from becoming a general-purpose database engine for a piece of software, and also helps avoid undue clutter and traffic in this shared system resource.

The first decision you should make regarding storing configuration information for your software is whether the information applies to the machine (or all users) or to a specific user of the system. Said another way, if the information maps to a feature of your software, do you want the feature's configuration to be the same regardless of the user involved, or do you want the information to be configured differently for each user?

In the typical case, information stored in the registry when software is installed applies to all users of the system. This process is much like hardware being installed on the system. In contrast, user settings and configuration changes made at software run time are commonly stored for a specific user.

Machine-Specific Registry Settings

When an application (or service) needs to store data specific to the configuration of the machine it is running on, convention mandates that it store its keys and values in this hierarchy:

```
HKEY_LOCAL_MACHINE
   Software
      Your Company Name
         Your Product Name
            Your Product Version (optional)
               Key1
                  Value1
```

```
                Value2

            Key2
```

Following this structure helps avoid registry clutter and makes your software-specific information easier to find. Typically, server software doesn't store configuration information for specific users. Its configuration is machine-specific; it has no need or opportunity to adjust its behavior based on a specific user.

User-Specific Registry Settings

The subkeys under HKEY_USERS represent user configuration information. The system automatically maps HKEY_CURRENT_USER to the subkey for the current user—that is, the subkey for the user associated with the current process. If a thread is impersonating another user (impersonation is covered in detail in Chapter 11), all references to HKEY_CURRENT_USER by that thread will refer to the impersonated user.

If your software will be accessing user-specific registry information, it should follow a convention similar to that used with HKEY_LOCAL_MACHINE:

```
HKEY_CURRENT_USER
   Software
      Your Company Name
         Your Product Name
            Your Product Version (optional)
               Key1
                  Value1
                  Value2

               Key2
```

Of course, you can access this hierarchy directly via the HKEY_USERS root key, but this is not recommended. If you really feel the need to do this, you can use the *RegOpenCurrentUser* function. Understanding the way in which HKEY_CURRENT_USER relates to the data found under HKEY_USERS involves the understanding of tokens and user context, which are discussed in detail in Chapter 11.

[Previous] [Next]

Working with Registry Keys

This section describes functions implemented by the system that can be used to access the system registry. Before discussing specific functions, however, I would like to make a distinction between two sets of functions: the registry functions, and the shell registry functions.

Registry and Shell Registry Functions

Registry functions are a basic set of functions, implemented by the system, for accessing the system registry. These functions have existed since the creation of the Win32 programming interface and reside in a DLL named AdvAPI32.dll. Anything that can be done with the registry can be implemented using the registry functions. They are characterized by the three-character prefix *Reg*—for example, *RegOpenKeyEx*.

Shell registry functions are built on these "regular" registry functions and provide extended or simplified functionality. Shell registry functions are intended to ease access to the registry, as well as to enforce a more

uniform usage of the registry by applications. These functions are available only when Microsoft Internet Explorer 4.0 or later is installed on the system.

To use the shell functions, you have to include the ShlWAPI.h header file in your source code as well as the ShlWAPI.lib file with your linker options. The regular registry functions do not require any special inclusions.

The shell functions are characterized by simpler parameter lists as well as the two-character prefix *SH*—for example, *SHCopyKey*.

Because the regular registry functions provide the base interface to the registry, I will be focusing on them in this chapter. However, I will make occasional mention of shell functions where appropriate.

Opening Registry Keys

Before you can begin storing and retrieving information in registry values, you must obtain a handle to a registry key. You can obtain a handle to a registry key in two basic ways: open an existing key or create a new key. The *RegOpenKeyEx* function should be used to open existing registry keys. It is prototyped as follows:

```
LONG RegOpenKeyEx(
   HKEY   hkeyRoot,
   PCTSTR pszSubKey,
   DWORD  ulOptions,
   REGSAM samDesired,
   PHKEY  phkResult);
```

NOTE

The *RegOpenKey* function can also be used to obtain a handle to a registry key. However, you should always use the *Ex* version of a registry function, if it exists.

To open a subkey, you pass the handle of the root key for the *hkeyRoot* parameter. The handle should be either the parent of the subkey you want to open or some grandparent of the desired subkey. The system also allows you to pass any of the predefined root key values as listed in Table 5-1 to *hkeyRoot*. Typically, an application will have to make at least one call to *RegOpenKeyEx*, passing HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER as the *hkeyRoot* parameter.

The *pszSubKey* parameter is the textual name of the key that you want to open. This name must include all keys between the desired key and the root, separated with backslash characters. For example, if you wanted to obtain a handle with read access to the HKEY_LOCAL_MACHINE\Software\Microsoft\Windows subkey, your call to *RegOpenKeyEx* would look something like the following:

```
HKEY hkeyWindows;
LONG lRet = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
   TEXT("Software\\Microsoft\\Windows"), NULL, KEY_READ, &hkeyWindows);
```

If you are opening a key that is an immediate subkey of the specified root key, you don't need to include parent key names or backslashes in the *pszSubKey* parameter; include only the name of the desired key. If you pass an empty string or a NULL value as the *pszSubKey* parameter, the system returns a new handle to the specified root key. Pass NULL to *pszSubKey* when you need a handle with security access that is different from that of the original root key handle.

The *ulOptions* parameter is currently unused and reserved for future use. For now, you should always pass zero for this parameter.

The *samDesired* parameter describes what your application intends to do with the requested registry key, and can be any combination of the flags described in Table 5-3. Avoid the temptation to pass KEY_ALL_ACCESS, because your application might have limited security access to a given registry key. Requesting only the access your application intends to use increases the odds of a successful call to *RegOpenKeyEx*.

Table 5-3. *Flags that can be passed to* RegOpenKeyEx'*s* samDesired *parameter*

Registry Access Flags	Description
KEY_ALL_ACCESS	Combination of KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, KEY_CREATE_SUB_KEY, KEY_CREATE_LINK, and KEY_SET_VALUE access
KEY_CREATE_LINK	Permission to create a symbolic link
KEY_CREATE_SUB_KEY	Permission to create subkeys
KEY_ENUMERATE_SUB_KEYS	Permission to enumerate subkeys
KEY_EXECUTE	Permission for read access
KEY_NOTIFY	Permission for change notifications
KEY_QUERY_VALUE	Permission to query subkey data
KEY_READ	Combination of KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY access
KEY_SET_VALUE	Permission to set subkey data
KEY_WRITE	Combination of KEY_SET_VALUE and KEY_CREATE_SUB_KEY access

You should pass the address of an HKEY variable as the value of the *phkResult* parameter. The system fills this variable with the handle of the open key, if *RegOpenKeyEx* succeeds. The *RegOpenKeyEx* function returns ERROR_SUCCESS if it is successful and ERROR_FILE_NOT_FOUND if the key does not exist.

NOTE

Unlike most other Windows functions, the registry functions return error codes. You should not call *GetLastError* to get more specific error information for a registry function.

Creating Registry Keys

The *RegCreateKeyEx* function allows you to create a key. You can also use the *RegCreateKeyEx* function to retrieve a handle to a key. Unlike *RegOpenKeyEx*, however, *RegCreateKeyEx* will not automatically fail if the key does not exist. Instead, it will attempt to create a new key with the name you provide. *RegCreateKeyEx* is prototyped as follows:

```
LONG RegCreateKeyEx(
   HKEY                hkeyRoot,
   PCTSTR              pszSubKey,
   DWORD               Reserved,
   PTSTR               pszClass,
   DWORD               dwOptions,
```

```
    REGSAM                samDesired,
    PSECURITY_ATTRIBUTES  psa,
    PHKEY                 phkResult,
    PDWORD                pdwDisposition);
```

You will recognize the *hkeyRoot*, *pszSubKey*, *samDesired*, and *phkResult* parameters from the *RegOpenKeyEx* function. These parameters are used identically to those for *RegCreateKeyEx*, but with one noteworthy exception: if the key specified in the *pszSubKey* parameter does not already exist, the system attempts to create a new subkey to complete the request.

In addition to the familiar parameters, *RegCreateKeyEx* has a *pszClass* parameter, which points to a zero-terminated string with a class name for the key. The class name is reserved, and you should always pass NULL for this parameter.

The *dwOptions* parameter can be any of the values in Table 5-4.

The *psa* parameter points to a SECURITY_ATTRIBUTES structure, which allows you to define the access rights for the newly created subkey. The security descriptor information is ignored if the subkey already exists. (For more information on security and the registry, see Chapter 10.)

Table 5-4. *Values that can be passed for* RegCreateKeyEx'*s* dwOptions *parameter*

Option Values	Description
REG_OPTION_NON_VOLATILE	The key is created in the registry and is persistent on the system's hard disk.
REG_OPTION_VOLATILE	The key is stored in memory and does not persist when the system is restarted.
REG_OPTION_BACKUP_RESTORE	The key is used by backup software to back up and restore the registry.

NOTE

The typical registry key is nonvolatile. Volatile data is always deleted when the computer reboots. Volatile keys are convenient for avoiding clutter while using keys that are only temporarily necessary. Volatile keys can be used to efficiently exchange data between processes—the communication even works well for processes operating on different machines because the registry is remoteable. (See the *RegConnectRegistry* function discussed later in "Accessing the Registry Remotely.")

The *pdwDisposition* parameter is a pointer to a DWORD variable that the system fills with one of two values: REG_CREATED_NEW_KEY or REG_OPENED_EXISTING_KEY. Examining the variable when the function returns reveals whether the function created the subkey or simply opened the existing subkey. Most of the time, your software does not need to know whether it actually created the subkey and, therefore, *RegCreateKeyEx* allows you to pass NULL for this last parameter.

Software commonly uses *RegCreateKeyEx* to create and open its keys for configuration data, understanding that the system might have created a new key for the software on the spot. *RegCreateKeyEx* provides a convenient way for an application to rebuild necessary portions of its configuration information with default data in the event that a registry key is deleted, which the curious administrator has been known to do from time to time.

Whether you received your handle to an open registry key by making a call to *RegCreateKeyEx* or by making a call to *RegOpenKeyEx*, you should always pass your open registry handles to *RegCloseKey* when you are finished using them:

```
LONG RegCloseKey(HKEY hkey);
```

Enumerating Registry Keys

When software is being installed or configured, it typically knows the required registry key names. However, in some cases, the subkey names within an open key may not be known. The *RegEnumKeyEx* function can be used to enumerate subkeys. *RegEnumKeyEx* has the following prototype:

```
LONG RegEnumKeyEx(
   HKEY      hkey,
   DWORD     dwIndex,
   PTSTR     pszName,
   PDWORD    pcbName,
   PDWORD    pdwReserved,
   PTSTR     pszClass,
   PDWORD    pcbClass,
   PFILETIME pftLastWriteTime);
```

Because your code must provide the buffers for the returned class name and key name values, it is your responsibility to pass buffers large enough to hold any subkey and class name; otherwise, the function will fail. To do this, you should make an initial call to *RegQueryInfoKey* to retrieve the length of the longest subkey and class names under the desired key. Your code must still be capable of handling an even longer subkey name, because a new subkey might be added between the times that your code calls *RegQueryInfoKey* and *RegEnumKeyEx*. The *RegQueryInfoKey* function is prototyped as follows:

```
LONG RegQueryInfoKey(
   HKEY      hkey,
   PTSTR     pszClass,
   PDWORD    pcbClass,
   PDWORD    pdwReserved,
   PDWORD    pcSubKeys,
   PDWORD    pcbMaxSubKeyLen,
   PDWORD    pcbMaxClassLen,
   PDWORD    pcValues,
   PDWORD    pcbMaxValueNameLen,
   PDWORD    pcbMaxValueLen,
   PDWORD    pcbSecurityDescriptor,
   PFILETIME pftLastWriteTime);
```

The following code fragment illustrates how to enumerate subkey names under an arbitrary key:

```
VOID PrintSubKeyNames(HKEY hkey) {

   // Get length of longest key name in characters
   DWORD dwMaxSubkeyLen;
   RegQueryInfoKey(hkey, NULL, NULL, NULL, NULL, &dwMaxSubkeyLen,
      NULL, NULL, NULL, NULL, NULL, NULL);

   // Add one for the NULL terminator
   dwMaxSubkeyLen++;

   // Allocate buffer for subkey names
   PTSTR pszKeyName = (PTSTR) _alloca(dwMaxSubkeyLen * sizeof(TCHAR));

   // Store buffer length
   DWORD dwKeyNameLen = dwMaxSubkeyLen;

   // Start from the beginning
   DWORD dwIndex = 0;
   FILETIME ftLastWritten;
```

```
    // Loop until failure
    while (ERROR_SUCCESS == RegEnumKeyEx(hkey, dwIndex++, pszKeyName,
       &dwKeyNameLen, NULL, NULL, NULL, &ftLastWritten)) {

       // Print key name
       _tprintf(TEXT("%s\n"), pszKeyName);

       // Restore buffer length with each iteration
       dwKeyNameLen = dwMaxSubkeyLen;
    }
}
```

NOTE

The shell registry functions include two functions that parallel the functionality of *RegQueryInfoKey* and *RegEnumKeyEx*. They are defined as follows:

```
DWORD SHQueryInfoKey(
   HKEY   hkey,
   PDWORD pcSubKeys,
   PDWORD pcchMaxSubKeyLen,
   PDWORD pcValues,
   PDWORD pcchMaxValueNameLen);
```

and

```
DWORD SHEnumKeyEx(
   HKEY   hkey,
   DWORD  dwIndex,
   PTSTR  pszName,
   PDWORD pcchName);
```

Notice that the shell registry functions are simpler to use because infrequently needed parameters are not required. [Previous] [Next]

Working with Registry Values

Up to this point, I have mostly discussed registry keys, without much mention of how to work with registry values. The registry value is the actual meat and potatoes of the registry. Registry values store the actual configuration data for your applications. Once you have a valid handle to an open registry key, you are ready to retrieve data contained within that key.

Three functions retrieve values from the system registry: *RegQueryValueEx*, *RegEnumValue*, and *RegQueryMultipleValues*. I will focus on *RegQueryValueEx* and *RegEnumValue* now. I'll discuss *RegQueryMultipleValues* shortly, in the section "Accessing the Registry Remotely."

You should use the *RegEnumValue* function when you do not know the name of the value that contains the data you want to retrieve, or when you want to systematically retrieve the data for all values contained within a given key. Conversely, you should use *RegQueryValueEx* to obtain data from a single value within a key, whose name (or lack of a name, in the case of the default value) is known ahead of time. The *RegQueryValueEx* function is prototyped as follows:

```
LONG RegQueryValueEx(
   HKEY   hkey,
   PTSTR  pszValueName,
   PDWORD pdwReserved,
   PDWORD pdwType,
```

```
    PBYTE  pbData,
    PDWORD pcbData);
```

The *hkey* parameter is the handle to an open key with KEY_QUERY_VALUE access, and the *pszValueName* is the name of the value you wish to retrieve. If you pass NULL or a pointer to an empty string for the *pszValueName* parameter, the system will retrieve data from the default value for the key.

The *pdwType* parameter is a pointer to a DWORD variable that the system will fill with one of the data type values (such as REG_SZ or REG_BINARY) listed in Table 5-2, indicating the registry value's data type.

The *pbData* parameter is a pointer to a buffer that the function fills with the registry value's data, and the *pcbData* parameter points to a DWORD that contains the size of the passed buffer. You can also pass NULL for the *pbData* parameter, in which case the function fills the DWORD pointed to by *pcbData* with the size, in bytes, of the data contained in the registry value.

NOTE

RegQueryValueEx returns the error ERROR_MORE_DATA if your buffer size is not sufficient to retrieve all the data from the value. Under all circumstances, your application should be able to deal gracefully with this error. Usually the response is to allocate a larger buffer and execute the call to the function again. Checking the size of a value's data before calling *RegQueryValueEx* is not a guarantee of success, because some other process might make changes to the registry value after your application checked its data size.

The CAutoBuf class included on this book's companion CD offers an elegant (if I do say so myself) way to handle functions that require data buffers of varying sizes. This C++ class is used frequently in the sample applications in this book.

Typically if an application is going to retrieve only a single registry value, it will make two calls to *RegQueryValueEx*. The initial call is used to retrieve the required size of the buffer. After allocating a buffer of the proper size, a second call to *RegQueryValueEx* is made to actually retrieve the data.

There is, however, a second approach that might be more efficient depending largely on the number of registry values that you will be reading: make a single call to *RegQueryInfoKey* that returns the size of the largest value contained in the key. Then your application can allocate a single buffer to be used in multiple calls to *RegQueryValueEx*.

NOTE

The shell registry functions implement a function for retrieving data from registry values, named *SHGetValue*. It is defined as follows:

```
DWORD SHGetValue(
   HKEY   hkey,
   PCTSTR pszSubKey,
   PCTSTR pszValue,
   PDWORD pdwType,
   PVOID  pvData,
   PDWORD pcbData);
```

Notice that this function is very similar to *RegQueryValueEx*, but it also includes a *pszSubKey* parameter that allows you to specify the name of the key containing the value. This frees your application from the responsibility of calling *RegOpenKeyEx* and *RegCloseKey*.

If you do not know the name of the registry value in question, or if you wish to retrieve the data for every value contained within a single key, you should use *RegEnumValue*, which is prototyped as follows:

```
LONG RegEnumValue(
   HKEY   hkey,
   DWORD  dwIndex,
   PTSTR  pszValueName,
   PDWORD pcbValueName,
   PDWORD pdwReserved,
   PDWORD pdwType,
   PBYTE  pbData,
   PDWORD pcbData);
```

As you can see, *RegEnumValue* is very similar to *RegQueryValueEx*, except that rather than passing *RegEnumValue* a value name, you pass it an index value that correlates to the sequential location of the value within the key referenced in *hkey*. *RegEnumValue* returns the value name in a buffer pointed to by the *pszValueName* parameter.

To find the proper size of the buffer so that you can retrieve the value's name, your application can make an initial call to *RegQueryValueEx*, specifying NULL for *pbData*. Doing so retrieves the length of the longest value name contained within the key.

The RegScan Sample Application

The RegScan sample application ("05 RegScan.exe"), shown in Listing 5-1, demonstrates how to enumerate keys and values under a specified key, as well as how to retrieve data from the enumerated values. The source code and resource files for the application are in the 05-RegScan directory on the companion CD. The sample application will recursively traverse the registry as it searches for a specified text string either in a key name or in a value.

When you launch RegScan, you can enter the machine name whose registry you wish to search. Then you select a root key and a starting subkey and enter a search string in the String edit box. Figure 5-2 shows the results of my search.

Later in this chapter (in the section "Accessing the Registry Remotely"), I'll discuss how RegScan can search a remote machine's registry.

Figure 5-2. *A search using the RegScan sample application*

Listing 5-1. *The RegScan sample application*

RegScan.cpp

```
/******************************************************************************
Module:  RegScan.cpp Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
#include "..\CmnHdr.h"                // See Appendix A. #include <WindowsX.h> #include <stdio.h>
#include "Resource.h" #define UILAYOUT_IMPL
#include "..\ClassLib\UILayout.h"      // See Appendix B. #define PRINTBUF_IMPL
#include "..\ClassLib\PrintBuf.h"      // See Appendix B. #define AUTOBUF_IMPL
#include "..\ClassLib\AutoBuf.h"       // See Appendix B. #define REGWALK_IMPL
#include "RegWalk.h" ////////////////////////////////////////////////////////////////////////////
class CRegScan : private CRegWalk { public:    CRegScan() : m_pb(256 * 1024) { }
  BOOL Go(PCTSTR pszMachine, HKEY hkeyRoot, PCTSTR pszSubkey,
    PCTSTR pszString, BOOL fSearchKeyNames, BOOL fSearchValueNames,
    BOOL fSearchValueData, BOOL fCaseSensitive);    PCTSTR Result() { return(m_pb); }
  void ForceSearchStop() { m_fStopSearch = TRUE; }
  BOOL WasSearchStopped() { return(m_fStopSearch); }     private:
  PCTSTR m_pszString;          // String to search for
  BOOL   m_fSearchKeyNames;    // Search key names?
  BOOL   m_fSearchValueNames;  // Search values names?
  BOOL   m_fSearchValueData;   // Search value string data?
  BOOL   m_fShownThisSubkey;   // Any matches from the current subkey?
  BOOL   m_fStopSearch;        // Prematurely stop the search?
  CPrintBuf m_pb;              // Growable results buffer
  typedef PTSTR (WINAPI* PFNSTRCMP)(PCTSTR pszFirst, PCTSTR pszSearch);
  PFNSTRCMP m_pfnStrCmp;       // String comparison function protected:
  REGWALKSTATUS onSubkey(PCTSTR pszSubkey, int nDepth,       BOOL fRecurseRequested);
  REGWALKSTATUS onValue(HKEY hkey, PCTSTR pszValue, int nDepth);    void ProcessUI(); };
////////////////////////////////////////////////////////////////////////////
BOOL CRegScan::Go(PCTSTR pszMachine, HKEY hkeyRoot, PCTSTR pszSubkey,
  PCTSTR pszString, BOOL fSearchKeyNames, BOOL fSearchValueNames,
  BOOL fSearchValueData, BOOL fCaseSensitive) {    m_pszString        = pszString;
  m_fSearchKeyNames   = fSearchKeyNames;    m_fSearchValueNames = fSearchValueNames;
  m_fSearchValueData  = fSearchValueData;    m_pfnStrCmp        = fCaseSensitive ? StrStr : StrStrI;
  m_fShownThisSubkey  = FALSE;    m_fStopSearch      = FALSE;    m_pb.Clear();
  BOOL fOk = TRUE;
  if (!m_fSearchKeyNames && !m_fSearchValueNames && !m_fSearchValueData) {
    chMB("You must at least select one field to search.");
  } else fOk = CRegWalk::Go(pszMachine, hkeyRoot, pszSubkey, TRUE);    return(fOk); }
//////////////////////////////////////////////////////////////////////////// void CRegScan::ProcessUI() {    MSG msg;
  while (PeekMessage(&msg, 0, 0, 0, PM_REMOVE)) {
    // There are UI messages, process them.
    if (!IsDialogMessage(GetActiveWindow(), &msg)) {          TranslateMessage(&msg);
      DispatchMessage(&msg);        }       } } ////////////////////////////////////////////////////////////////////////////////
CRegWalk::REGWALKSTATUS CRegScan::onSubkey(PCTSTR pszSubkey, int nDepth,
  BOOL fRecurseRequested) {    REGWALKSTATUS rws = RWS_FULLSTOP;
  if (fRecurseRequested || (nDepth == 0)) rws = RWS_RECURSE;
  // Get this subkey's name without the full path
  PCTSTR pszSubkeyName = PathFindFileName(pszSubkey);    if (m_fSearchKeyNames)
    m_fShownThisSubkey = (m_pfnStrCmp(pszSubkeyName, m_pszString) != NULL);
  else m_fShownThisSubkey = FALSE;       if (m_fShownThisSubkey) {
    m_pb.Print(TEXT("%s\r\n"), pszSubkey);    }    ProcessUI();
  return(WasSearchStopped() ? RWS_FULLSTOP : rws); }
////////////////////////////////////////////////////////////////////////////
```

```
CRegWalk::REGWALKSTATUS CRegScan::onValue(
   HKEY hkey, PCTSTR pszValue, int nDepth) {
   if (m_fSearchValueNames && (m_pfnStrCmp(pszValue, m_pszString) != NULL)) {
      if (!m_fShownThisSubkey) {          m_pb.Print(TEXT("%s\r\n"), m_szSubkeyPath);
         m_fShownThisSubkey = TRUE;       }       m_pb.Print(TEXT("\t%s\r\n"), pszValue);    }
   if (m_fSearchValueData) {       // Check the value's data       DWORD dwType;
      RegQueryValueEx(hkey, pszValue, NULL, &dwType, NULL, NULL);
      if ((dwType == REG_EXPAND_SZ) || (dwType == REG_SZ)) {
         CAutoBuf<TCHAR, sizeof(TCHAR)> szData;
         // Give buffer a size > 0 so that RegQueryValueEx returns
         // ERROR_MORE_DATA instead of ERROR_SUCCESS.          szData = 1;
         while (RegQueryValueEx(hkey, pszValue, NULL, NULL, szData, szData)
            == ERROR_MORE_DATA) ;          // szData is NULL is there is no value data
         if (((PCTSTR) szData != NULL) &&              (m_pfnStrCmp(szData, m_pszString) != NULL)) {
                  if (!m_fShownThisSubkey) {
            m_pb.Print(TEXT("%s\r\n"), m_szSubkeyPath);              m_fShownThisSubkey = TRUE;
           }            m_pb.Print(TEXT("\t%s (%s)\r\n"),
            ((pszValue[0] == 0) ? TEXT("(default)") : pszValue),              (PCTSTR) szData);          }
      }    }    ProcessUI();    return(WasSearchStopped() ? RWS_FULLSTOP : RWS_CONTINUE); }
//////////////////////////////////////////////////////////////////////////////
CUILayout g_UILayout;   // Repositions controls when dialog box size changes.
//////////////////////////////////////////////////////////////////////////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
   chSETDLGICONS(hwnd, IDI_REGSCAN);
   HWND hwndRootKey = GetDlgItem(hwnd, IDC_ROOTKEY);    int n = 0;
   n = ComboBox_AddString(hwndRootKey, TEXT("HKEY_LOCAL_MACHINE"));
   ComboBox_SetItemData(hwndRootKey, n, HKEY_LOCAL_MACHINE);
   ComboBox_SetCurSel(hwndRootKey, n);   // HKLM is default
   n = ComboBox_AddString(hwndRootKey, TEXT("HKEY_CURRENT_CONFIG"));
   ComboBox_SetItemData(hwndRootKey, n, HKEY_CURRENT_CONFIG);
   n = ComboBox_AddString(hwndRootKey, TEXT("HKEY_CLASSES_ROOT"));
   ComboBox_SetItemData(hwndRootKey, n, HKEY_CLASSES_ROOT);
   n = ComboBox_AddString(hwndRootKey, TEXT("HKEY_USERS"));
   ComboBox_SetItemData(hwndRootKey, n, HKEY_USERS);
   n = ComboBox_AddString(hwndRootKey, TEXT("HKEY_CURRENT_USER"));
   ComboBox_SetItemData(hwndRootKey, n, HKEY_CURRENT_USER);
   // Set up the resizeing of the controls    g_UILayout.Initialize(hwnd);
   g_UILayout.AnchorControl(CUILayout::AP_TOPLEFT, CUILayout::AP_TOPRIGHT,
      IDC_MACHINE, FALSE);
   g_UILayout.AnchorControl(CUILayout::AP_TOPLEFT, CUILayout::AP_TOPRIGHT,
      IDC_ROOTKEY, FALSE);
   g_UILayout.AnchorControl(CUILayout::AP_TOPLEFT, CUILayout::AP_TOPRIGHT,
      IDC_SUBKEY, FALSE);
   g_UILayout.AnchorControl(CUILayout::AP_TOPLEFT, CUILayout::AP_TOPRIGHT,
      IDC_STRING, FALSE);
   g_UILayout.AnchorControl(CUILayout::AP_TOPRIGHT, CUILayout::AP_TOPRIGHT,
      IDC_SEARCHKEYNAMES, FALSE);
   g_UILayout.AnchorControl(CUILayout::AP_TOPRIGHT, CUILayout::AP_TOPRIGHT,
      IDC_SEARCHVALUENAMES, FALSE);
   g_UILayout.AnchorControl(CUILayout::AP_TOPRIGHT, CUILayout::AP_TOPRIGHT,
      IDC_SEARCHVALUEDATA, FALSE);
   g_UILayout.AnchorControl(CUILayout::AP_TOPRIGHT, CUILayout::AP_TOPRIGHT,
      IDC_CASESENSITIVE, FALSE);
   g_UILayout.AnchorControl(CUILayout::AP_TOPRIGHT, CUILayout::AP_TOPRIGHT,
```

```
      IDC_SEARCHSTART, FALSE);
   g_UILayout.AnchorControl(CUILayout::AP_TOPRIGHT, CUILayout::AP_TOPRIGHT,
      IDC_SEARCHSTOP, FALSE);
   g_UILayout.AnchorControl(CUILayout::AP_TOPLEFT, CUILayout::AP_BOTTOMRIGHT,
      IDC_SEARCHRESULTS, FALSE);
   CheckDlgButton(hwnd, IDC_SEARCHKEYNAMES,   TRUE);
   CheckDlgButton(hwnd, IDC_SEARCHVALUENAMES, TRUE);
   CheckDlgButton(hwnd, IDC_SEARCHVALUEDATA,  TRUE);    return(TRUE); }
////////////////////////////////////////////////////////////////////////////
void EnableControls(HWND hwnd, BOOL fEnable) {
   EnableWindow(GetDlgItem(hwnd, IDC_MACHINE),          fEnable);
   EnableWindow(GetDlgItem(hwnd, IDC_ROOTKEY),          fEnable);
   EnableWindow(GetDlgItem(hwnd, IDC_SUBKEY),           fEnable);
   EnableWindow(GetDlgItem(hwnd, IDC_STRING),           fEnable);
   EnableWindow(GetDlgItem(hwnd, IDC_SEARCHKEYNAMES),   fEnable);
   EnableWindow(GetDlgItem(hwnd, IDC_SEARCHVALUENAMES), fEnable);
   EnableWindow(GetDlgItem(hwnd, IDC_SEARCHVALUEDATA),  fEnable);
   EnableWindow(GetDlgItem(hwnd, IDC_CASESENSITIVE),    fEnable);
   ShowWindow(GetDlgItem(hwnd, IDC_SEARCHSTART), fEnable ? SW_SHOW : SW_HIDE);
   ShowWindow(GetDlgItem(hwnd, IDC_SEARCHSTOP),  fEnable ? SW_HIDE : SW_SHOW); }
////////////////////////////////////////////////////////////////////////////
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
   static CRegScan x;        switch (id) {    case IDCANCEL:       EndDialog(hwnd, id);       break;
   case IDC_SEARCHSTOP:       x.ForceSearchStop();       break;    case IDC_SEARCHSTART:
      SetDlgItemText(hwnd, IDC_SEARCHRESULTS, TEXT("Scanning Registry..."));
      EnableControls(hwnd, FALSE);       TCHAR szString[1000];
      GetDlgItemText(hwnd, IDC_STRING, szString, chDIMOF(szString));
      TCHAR szMachine[100], szSubkey[1000];
      GetDlgItemText(hwnd, IDC_MACHINE, szMachine, chDIMOF(szMachine));
      GetDlgItemText(hwnd, IDC_SUBKEY, szSubkey, chDIMOF(szSubkey));
      HWND hwndRootKey = GetDlgItem(hwnd, IDC_ROOTKEY);
      int nIndex = ComboBox_GetCurSel(hwndRootKey);
      HKEY hkeyRoot = (HKEY) ComboBox_GetItemData(hwndRootKey, nIndex);       if (!x.Go(
         (szMachine[0] == 0) ? NULL : szMachine, hkeyRoot, szSubkey,           szString,
         IsDlgButtonChecked(hwnd, IDC_SEARCHKEYNAMES),
         IsDlgButtonChecked(hwnd, IDC_SEARCHVALUENAMES),
         IsDlgButtonChecked(hwnd, IDC_SEARCHVALUEDATA),
         IsDlgButtonChecked(hwnd, IDC_CASESENSITIVE))) {
         chMB("Couldn't access the registry");       }
      SetDlgItemText(hwnd, IDC_SEARCHRESULTS,
         x.WasSearchStopped() ? TEXT("Scan Canceled") :
            ((x.Result()[0] == 0) ? TEXT("No entries found") : x.Result()));
      EnableControls(hwnd, TRUE);                break;    } }
////////////////////////////////////////////////////////////////////////////
void Dlg_OnSize(HWND hwnd, UINT state, int cx, int cy) {    // Reposition the child controls
   g_UILayout.AdjustControls(cx, cy);     } ////////////////////////////////////////////////////////////////////////////
void Dlg_OnGetMinMaxInfo(HWND hwnd, PMINMAXINFO pMinMaxInfo) {
   // Return minimum size of dialog box    g_UILayout.HandleMinMax(pMinMaxInfo); }
////////////////////////////////////////////////////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
   switch (uMsg) {    chHANDLE_DLGMSG(hwnd, WM_INITDIALOG,    Dlg_OnInitDialog);
   chHANDLE_DLGMSG(hwnd, WM_COMMAND,       Dlg_OnCommand);
   chHANDLE_DLGMSG(hwnd, WM_SIZE,          Dlg_OnSize);
   chHANDLE_DLGMSG(hwnd, WM_GETMINMAXINFO, Dlg_OnGetMinMaxInfo);    }
```

```
  return(FALSE); } ////////////////////////////////////////////////////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {
  DialogBox(hinstExe, MAKEINTRESOURCE(IDD_REGSCAN), NULL, Dlg_Proc);    return(0); }
////////////////////////////////// End of File /////////////////////////////////
```

RegWalk.h

```
/******************************************************************************
Module: RegWalk.h Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
#pragma once   // Include this header file once per compilation unit
//////////////////////////////////////////////////////////////////////////// #include <ShlWapi.h>
//////////////////////////////////////////////////////////////////////////// class CRegWalk { public:    CRegWalk() {}
  virtual ~CRegWalk() {}
  BOOL Go(PCTSTR pszMachine, HKEY hkeyRoot, PCTSTR pszSubkey, BOOL fRecurse);
  enum REGWALKSTATUS { RWS_FULLSTOP, RWS_CONTINUE, RWS_RECURSE };
protected:    virtual REGWALKSTATUS onSubkey(PCTSTR pszSubkey, int nDepth,
    BOOL fRecurseRequested);
  virtual REGWALKSTATUS onValue(HKEY hkey, PCTSTR pszValue, int nDepth); protected:
  HKEY   m_hkeyRootMachine;       // Root key on machine
  BOOL   m_fRecurse;              // Recurse into subkeys?    int    m_nDepth;               // Recurse depth
  TCHAR  m_szSubkeyPath[MAX_PATH]; // Subkey path private:
  REGWALKSTATUS RegWalkRecurse();    REGWALKSTATUS EnumValuesInSubkey(); };
//////////////////////////////////////////////////////////////////////////// #ifdef REGWALK_IMPL
//////////////////////////////////////////////////////////////////////////// #pragma comment(lib, "shlwapi")
////////////////////////////////////////////////////////////////////////////
CRegWalk::REGWALKSTATUS CRegWalk::onSubkey(PCTSTR pszSubkey, int nDepth,
  BOOL fRecurseRequested) {
  return(fRecurseRequested ? RWS_RECURSE : RWS_CONTINUE); }
CRegWalk::REGWALKSTATUS CRegWalk::onValue(HKEY hkey, PCTSTR pszValue,
  int nDepth) {    return(RWS_CONTINUE); } ////////////////////////////////////////////////////////////////////////////////
CRegWalk::REGWALKSTATUS CRegWalk::EnumValuesInSubkey() {    HKEY hkey = NULL;
  REGWALKSTATUS rws = RWS_CONTINUE;
  if (ERROR_SUCCESS == RegOpenKeyEx(m_hkeyRootMachine, m_szSubkeyPath, 0,
    KEY_QUERY_VALUE, &hkey)) {
    for (int nIndex = 0; rws != RWS_FULLSTOP; nIndex++) {
      TCHAR szValueName[256]; // No value name exceeds 255 characters
      DWORD cbValueName = chDIMOF(szValueName);
      if (ERROR_SUCCESS != RegEnumValue(hkey, nIndex,
        szValueName, &cbValueName, NULL, NULL, NULL, NULL))          break;
      rws = onValue(hkey, szValueName, m_nDepth);      }
    chVERIFY(RegCloseKey(hkey) == ERROR_SUCCESS);    }    return(rws); }
////////////////////////////////////////////////////////////////////////////
CRegWalk::REGWALKSTATUS CRegWalk::RegWalkRecurse() {    // Report this Subkey
  REGWALKSTATUS rws = onSubkey(m_szSubkeyPath, ++m_nDepth, m_fRecurse);
  // Enumerate the values in this subkey?
  if (rws == RWS_RECURSE) rws = EnumValuesInSubkey();    // Continue enumerating subkeys?
  if (rws != RWS_FULLSTOP) {      HKEY hkey = NULL;
    if (ERROR_SUCCESS == RegOpenKeyEx(m_hkeyRootMachine, m_szSubkeyPath, 0,
      KEY_ENUMERATE_SUB_KEYS, &hkey)) {
      for (int nIndex = 0; rws != RWS_FULLSTOP; nIndex++) {
        TCHAR szSubkeyName[256]; // No subkey name exceeds 255 characters
```

```
      DWORD cbSubkeyName = chDIMOF(szSubkeyName);
      if (ERROR_SUCCESS != RegEnumKeyEx(hkey, nIndex,
        szSubkeyName, &cbSubkeyName, NULL, NULL, NULL, NULL))          break;
      // Append the subkey to the path
      if (m_szSubkeyPath[0] != 0) StrCat(m_szSubkeyPath, TEXT("\\"));
      StrCat(m_szSubkeyPath, szSubkeyName);          rws = RegWalkRecurse();
      // Truncate the last subkey from the path
      PTSTR p = StrRChr(m_szSubkeyPath, NULL, TEXT('\\'));          if (p != NULL) *p = 0;
      else m_szSubkeyPath[0] = 0;        }
    chVERIFY(RegCloseKey(hkey) == ERROR_SUCCESS);      }    }    m_nDepth--;
  return(rws); } ////////////////////////////////////////////////////////////////////////////
BOOL CRegWalk::Go(PCTSTR pszMachine, HKEY hkeyRoot, PCTSTR pszSubkey,
  BOOL fRecurse) {    // nDepth indicates how many levels from the top we are.    m_nDepth = -1;
  m_fRecurse = fRecurse;    m_hkeyRootMachine = NULL;
  REGWALKSTATUS rws = RWS_FULLSTOP;    __try {      if (ERROR_SUCCESS !=
      RegConnectRegistry(pszMachine, hkeyRoot, &m_hkeyRootMachine))        __leave;
    lstrcpy(m_szSubkeyPath, pszSubkey);      // Call the recursive function to walk the subkeys
    rws = RegWalkRecurse();    }    __finally {      if (m_hkeyRootMachine != NULL)
      RegCloseKey(m_hkeyRootMachine);    }    return(rws != RWS_FULLSTOP); }
//////////////////////////////////////////////////////////////////////////// #endif   // REGWALK_IMPL
//////////////////////////////// End of File /////////////////////////////////
```

[Previous] [Next]

Storing Data in the System Registry

Before I discuss how to store data in the system registry, I will revisit the topic of proper registry usage. As you know, the system registry is a shared resource. As such, it is limited and access to it is serialized. Your application can store many thousands of kilobytes of data in the registry in various data types, but it is rarely in your application's best interest to take these capabilities to the extreme.

You should ask yourself two questions when considering registry storage:

- Is the data I am storing part of my application's configuration information, or is it general application data? (For example, it would be inappropriate for a word processor to store documents in the registry.)

- Is the amount of data I am storing large enough (and will I be accessing the data frequently enough) to make storage in a file more efficient than storage in the registry? (For example, a spell-checking dictionary used by a word processing application would be better stored in a file.)

In making these decisions, you end up weighing the pros and cons of file system clutter vs. the inefficiencies of storing large data in the system registry. I will be further discussing efficient use of the system registry in the section "Using the System Registry Efficiently" later in this chapter.

You can use the *RegSetValueEx* function to store data to the registry. Here is the prototype for *RegSetValueEx*:

```
LONG RegSetValueEx(
   HKEY    hkey,
   PCTSTR pszValueName,
   DWORD   dwReserved,
   DWORD   dwType,
   CONST   BYTE *pbData,
   DWORD   cbData);
```

The *hkey* parameter refers to the key under which the value is stored. The *pszValueName* is the name you want to use for the value. If you pass NULL or a pointer to an empty string for *pszValueName*, the system stores your data in the default value for the key. The *dwType* parameter should be set to one of the data type values listed in Table 5-2, depending on what type of data you are storing in this value. The *pbData* parameter points to the actual data to be stored in the registry. You pass the size of the data, in bytes, in the *cbData* parameter.

If *RegSetValueEx* succeeds, it returns ERROR_SUCCESS. The following code illustrates the use of *RegSetValueEx* by wrapping it in a function named *RegSetStringValue*.

```
LONG RegSetStringValue(HKEY hkey, PCTSTR pszValueName,
   PCTSTR pszString) {

   // Get the length of the string
   int nDataSize = lstrlen(pszString);

   // Add one for the zero terminator
   nDataSize++;

   // Multiply by the character size
   nDataSize = nDataSize * sizeof(TCHAR);

   // Attempt to store the data to the registry
   return(RegSetValueEx(hkey, pszValueName, 0, REG_SZ,
      (PBYTE) pszString, nDataSize));
}
```

NOTE

The shell registry functions include a function called *SHSetValue*, which, like *SHGetValue*, does not require your application to first obtain a handle to a key. The *SHSetValue* function is defined as follows:

```
DWORD SHSetValue(
   HKEY   hkey,
   PCTSTR pszSubKey,
   PCTSTR pszValue,
   PDWORD pdwType,
   PVOID  pvData,
   PDWORD pcbData);
```

As Table 5-2 shows, the registry supports many data types. You choose your type according to the needs of your application. You will likely find yourself using the REG_DWORD, REG_SZ, and REG_BINARY types most often. An application would use the REG_DWORD type to store numerical settings, REG_SZ (or REG_MULTI_SZ) to store textual settings, and REG_BINARY for other data that does not have a matching registry type. When using the REG_BINARY type, generic registry tools such as RegEdt32 will not know how to present the data other than as a block of byte values.

Before moving on, I would like to clarify two potentially confusing registry types: REG_EXPAND_SZ and REG_MULTI_SZ. The REG_EXPAND_SZ registry type is very similar to the REG_SZ type in that it is a zero-terminated Unicode or ANSI string stored in the registry. They differ in that, by convention, data stored in values of REG_EXPAND_SZ is typically passed to the *ExpandEnvironmentStrings* function after being extracted from the registry.

NOTE

The registry functions do not automatically expand the data in values of type REG_EXPAND_SZ. Your application is responsible for passing the string to

ExpandEnvironmentStrings.

Here is the prototype for *ExpandEnvironmentStrings*:

```
DWORD ExpandEnvironmentStrings(
   PCTSTR pszSrc,
   PTSTR  pszDst,
   DWORD  nSize);
```

REG_MULTI_SZ is largely misunderstood because of its description in the documentation. The documentation states that a REG_MULTI_SZ value is "an array of zero-terminated strings terminated by two zero characters." I think the documentation should read as follows: "A REG_MULTI_SZ value is stored as a series of zero-terminated strings, with two zero characters terminating the final string in the series." Remember that when you are storing or retrieving data of this type, your buffer size must also include all terminating zero characters as well as the double zero at the end. Figure 5-3 shows an example of a multizero-terminated string registry value consisting of three strings.

Figure 5-3. *A multizero-terminated string registry value consisting of three strings*

The data in the example represents the words "Jason's", "Test", and "MULTI_SZ" as separate individual strings. The buffer required to hold this data would be 23 bytes in length for ANSI and 46 bytes in length for Unicode.

[Previous] [Next]

Accessing the Registry Remotely

Windows allows you to access the registry on a remote machine in much the same way you access the registry on the local machine. This greatly eases the task of developing software that can be configured from a remote location on the network.

NOTE

To give remote registry access under Windows its fair consideration, I would need to delve into security. I'll defer a more in-depth discussion on security to Chapter 10.

The designers of the Windows registry functions really simplified the task of remotely accessing the registry by allowing application developers to use the same functions they use to access the local registry. For example, you can use *RegOpenKeyEx* to open keys on remote machines and on the local machine. Likewise, *RegSetValueEx*, *RegQueryValueEx*, *RegEnumKey*, and so on can be used to manipulate the registry on a remote machine. The only difference is in obtaining a handle to one of the system's predefined registry keys on the remote machine. This is done via a call to *RegConnectRegistry*, which is prototyped as follows:

```
LONG RegConnectRegistry(
   PTSTR pszMachineName,
   HKEY  hkey,
   PHKEY phkResult);
```

As you can see, *RegConnectRegistry* is fairly straightforward. You pass the name of the machine to which you want to connect in the *pszMachineName* parameter. You also pass one of the predefined registry keys (such as HKEY_LOCAL_MACHINE or HKEY_USERS) to indicate which portion of the remote machine's registry you wish to access. Finally, you provide *RegConnectRegistry* with a pointer to an HKEY variable in which

the function will store an open handle to the remote registry's root key.

NOTE

Your application can pass NULL as the *pszMachineName* parameter to indicate a connection to the local machine. This is a convenient mechanism for creating registry tools that work remotely and locally with minimal extra logic.

To retrieve values from a remote registry, you can use the *RegQueryMultipleValues* function. The *RegQueryMultipleValues* function is similar to *RegQueryValueEx*, but it can request multiple values. The *RegQueryMultipleValues* function allows your application to minimize network traffic by requesting multiple values in a single network operation. *RegQueryMultipleValues* is not exclusively available for remote registry manipulation, but it is in this capacity that it really shines. In an industry where network bandwidth is at a premium, each hit to the network comes at a cost. Here is the prototype for *RegQueryMultipleValues*:

```
LONG RegQueryMultipleValues(
   HKEY    hkey,
   PVALENT val_list,
   DWORD   num_vals,
   PTSTR   pszValueBuf,
   PDWORD  dwTotsize);
```

Your application must pass an array of VALENT structures to the function via the *val_list* parameter. The system then uses this array to find the values to be retrieved, and populates the structure with the information for each value. The VALENT structure is declared as follows:

```
typedef struct value_ent {
   PTSTR ve_valuename;
   DWORD ve_valuelen;
   DWORD ve_valueptr;
   DWORD ve_type;
} VALENT;
```

It is odd that the designers of the registry functions didn't also implement a function to set multiple values in the registry as a single operation. I assume that this was not done because the registry is typically read from much more often than it is written to. Of course, you could implement your own "*RegSetMultipleValues*" by making multiple calls to *RegSetValueEx*. Network traffic, however, would not be reduced by this implementation of the function.

NOTE

The *RegQueryMultipleValues* function is limited to reading no more than 1 MB of data per call. The function will succeed only if it can return all of the requested values. It will fail if collectively more than 1 MB of data is requested.

[Previous] [Next]

Using the System Registry Efficiently

Now that you know about the features that enable you to manipulate data in the system registry, you can learn how to make these features work in your server application. The registry should serve as the repository for your server's configuration information, but in some respects, it will also serve as a mechanism that an administrator can use to communicate configuration needs to your server.

Most servers are implemented as *services*, and services do not have a visible user interface. In fact, regardless of the process used to develop a server, there are compelling reasons to exclude a user interface from your server. When you exclude the user interface, you typically create a second, simple administering application to configure your service. This application can store configuration information in the registry, which your service can read. Administering applications should be able to run remotely as well as locally, and can also run on Microsoft Windows 95 or Windows 98.

When you design your service and administering application, you should follow some important rules to make the most efficient use of the system registry.

- **Your server should touch the registry as infrequently as possible.** The system registry is a shared resource and access to it must be serialized. This serialization is handled by the system. Each time you call a registry function, the calling thread risks waiting while another thread is accessing the registry. This can dramatically affect the performance of an application that makes frequent registry calls. Your server should read its configuration from the registry upon startup, and optionally reread configuration information upon request by the software that is administering the server. Most services require the service to be stopped and then restarted to operate using the new configuration parameters, but this not necessary.

- **Your server should not poll the registry for changes.** The system provides a means by which your service can be notified of changes made to the registry (discussed in the section "Registry Change Notifications"). Your service should use this feature as a mechanism for detecting changes made to its configuration rather than conduct a periodic polling of the system registry.

- **Avoid the temptation to report events or performance data via the registry.** Your administering application and your service should avoid reporting events or performance data to nonvolatile registry keys. Instead the application and service should use the performance monitoring functionality implemented via the volatile HKEY_PERFORMANCE_DATA registry key. Please see Chapter 7 for more information about how to report performance data.

[Previous] [Next]

Registry Change Notifications

Another way to more effectively use the registry is to take advantage of registry change notifications. An application can be efficiently notified of changes made within a registry key by using the *RegNotifyChangeKeyValue* function. This function allows you to tell the system what type of changes you want to be notified of and whether or not you wish to be notified of changes made within subkeys. Here is the function prototype:

```
LONG RegNotifyChangeKeyValue(
   HKEY   hkey,
   BOOL   fWatchSubtree,
   DWORD  dwNotifyFilter,
   HANDLE hEvent,
   BOOL   fAsynchronous);
```

The *hkey* parameter identifies the key for which you want to receive change notifications. The *fWatchSubtree* parameter indicates whether you want to be notified of changes within the entire tree below the specified key. The *dwNotifyFilter* parameter indicates which changes you want to be notified of. The *dwNotifyFilter* parameter can take any combination of the flags listed in Table 5-5.

Table 5-5. *Flags that can be passed for* RegNotifyChangeKeyValue*'s* dwNotifyFilter *parameter*

Notification Flags	Description
REG_NOTIFY_CHANGE_NAME	Notification occurs when a subkey is added or deleted.
REG_NOTIFY_CHANGE_ATTRIBUTES	Notification occurs when the attributes of the key, including the security descriptor information, changes.
REG_NOTIFY_CHANGE_LAST_SET	Notification occurs when the value of the key, including the addition or deletion of values, changes.
REG_NOTIFY_CHANGE_SECURITY	Notification occurs when the security descriptor of the key changes.

The *RegNotifyChangeKeyValue* function can be used in one of two ways: synchronously or asynchronously. To use the function synchronously, pass FALSE as the value of *fAsynchronous*. In a synchronous call to *RegNotifyChangeKeyValue*, the calling thread is suspended until a registry notification occurs. If you want to use the function asynchronously, you must pass a valid event handle in the *hEvent* parameter and TRUE for the *fAsynchronous* parameter. Calling the function asynchronously causes the function to return immediately. The system will automatically signal the event kernel object when the next registry change (matching the specified filter) occurs.

I prefer the asynchronous feature of *RegNotifyChangeKeyValue* because it does not waste threads in your process. Remember that whether you are using the function asynchronously or not, each call to *RegNotifyChangeKeyValue* is good for only a single notification. The function must be called repeatedly for you to receive future notifications.

NOTE

If the thread that called *RegNotifyChangeKeyValue* terminates, the system cancels the request for notification. This is typically not a problem, because commonly the thread waiting on the event is the same thread that called *RegNotifyChangeKeyValue*.

If the thread that called *RegNotifyChangeKeyValue* does terminate before a change notification occurs, the system signals the event, releasing any threads waiting on that event.

NOTE

To cause a thread to return from a synchronous call to *RegNotifyChangeKeyValue* without making a change to the key, your application must close the handle to the registry key.

The RegNotify Sample Application

The RegNotify sample application ("05 RegNotify.exe"), shown in Listing 52, demonstrates the asynchronous use of the *RegNotifyChangeKeyValue* function. The source code and resource files for the application are in the 05-RegNotify directory on the companion CD. The program simply monitors notifications for a registry key specified by the user.

When RegNotify is executed, the user enters a registry key for which he wants to receive automatic change notifications. The user then clicks the Watch button to begin the notification process for the specified registry key. Figure 5-4 shows the RegNotify sample application after new values have been added to the key being watched.

Figure 5-4. *The dialog box for the RegNotify sample application after three new values have been added to HKLM\SOFTWARE\RegNotify*

The notification process works like this: The RegNotify application creates a second thread, which enters into a loop that calls *RegNotifyChangeKeyValue* repeatedly for the selected key, enabling display of the value information in the read-only edit control. Each time a notification occurs, the event is signaled, causing *WaitForSingleObjectEx* to return WAIT_OBJECT_0. A return value of WAIT_OBJECT_0 causes the function to loop again for another notification.

When the user closes the application, the primary thread posts a user-mode asynchronous procedure call (APC) to the notification thread, which causes *WaitForSingleObjectEx* to exit with the value WAIT_IO_COMPLETION. The loop exits, causing the notification thread to exit.

> **NOTE**
>
> Using an APC as a form of interthread communication is sometimes desirable because it can be more efficient. It does not require the creation of a kernel object such as a semaphore or an event. When you use an APC for interthread communication, you frequently must create an APC callback function that is used as nothing other than a placeholder function for an APC.
>
> If you have not seen this technique in the past, the empty placeholder function can be initially confusing. For more information on this technique and other interthread communication mechanisms, see Chapter 2.

Listing 5-2. *The RegNotify sample application*

RegNotify.cpp

```
/******************************************************************************
Module:  RegNotify.cpp Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
#include "..\CmnHdr.h"              // See Appendix A. #include <WindowsX.h>
#include <Process.h>                // For _beginthreadex #include "Resource.h"
#define UILAYOUT_IMPL #include "..\ClassLib\UILayout.h"     // See Appendix B.
//////////////////////////////////////////////////////////////////////////////
CUILayout g_UILayout;   // Repositions controls when dialog box size changes.
//////////////////////////////////////////////////////////////////////////////
// We are using this function in place of an event object to let the thread
// know that it is time to exit its primary loop.
void WINAPI DoNothingAPC(ULONG_PTR dwParam) { }
//////////////////////////////////////////////////////////////////////////////
DWORD WINAPI RegSubkeyWatcher(PVOID pv) {    HWND hwnd = (HWND) pv;
```

```
   // Create our event for notification
   HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);    TCHAR szSubkey[200];
   // Get the Reg Key that we are going to create or open for watching
   GetDlgItemText(hwnd, IDC_REGKEY, szSubkey, chDIMOF(szSubkey));
   // Get our registry key to watch.    HKEY hkey = NULL;
   RegCreateKeyEx(HKEY_LOCAL_MACHINE, szSubkey, 0, NULL,
     REG_OPTION_NON_VOLATILE, KEY_NOTIFY | KEY_QUERY_VALUE,
     NULL, &hkey, NULL);    do {       DWORD dwIndex = 0, cbValName;       BYTE bData[1024];
     TCHAR szValName[100];       cbValName = chDIMOF(szValName);
     DWORD dwType, cbData = chDIMOF(bData);       TCHAR szRegVals[20 * 1024] = { 0 };
     // RegEnumValue to enumerate the values in the key
     while (ERROR_SUCCESS == RegEnumValue(hkey,
       dwIndex++, szValName, (cbValName = chDIMOF(szValName), &cbValName),
       NULL, &dwType, bData, (cbData = chDIMOF(bData), &cbData))) {
       PTSTR p = szRegVals + lstrlen(szRegVals);          wsprintf(p, TEXT("\r\n%s\t"), szValName);
       p = szRegVals + lstrlen(szRegVals);          // Handle the different types          switch (dwType) {
       case REG_DWORD:             wsprintf(p, TEXT("0x%08x"), * (PDWORD) bData);
         break;          case REG_EXPAND_SZ:          case REG_LINK:          case REG_MULTI_SZ:
       case REG_RESOURCE_LIST:          case REG_SZ:             wsprintf(p, TEXT("%s"), bData);
         break;          case REG_NONE:          case REG_DWORD_BIG_ENDIAN:          default:
         wsprintf(p, TEXT("Unknown type"));             break;          case REG_BINARY:
         for (DWORD x = 0; x < cbData; x++)
           wsprintf(p + lstrlen(p), TEXT("%02X "), bData[x]);             break;          }       }
     // Set the display       SetDlgItemText(hwnd, IDC_REGVALUES, &szRegVals[2]);   // skip "\r\n"
     // Set up the notification... notice we have to do this each time.
     // It is also important that the thread that makes the call to
     // RegNotifyChangeKeyValue be the one that waits on the event.
     RegNotifyChangeKeyValue(hkey, FALSE,
       REG_NOTIFY_CHANGE_NAME    | REG_NOTIFY_CHANGE_ATTRIBUTES |
       REG_NOTIFY_CHANGE_LAST_SET | REG_NOTIFY_CHANGE_SECURITY,
       hEvent, TRUE);       // Wait forever for the event or the APC
   } while (WaitForSingleObjectEx(hEvent, INFINITE, TRUE) == WAIT_OBJECT_0);
   // We are done with the event and key    CloseHandle(hEvent);    RegCloseKey(hkey);    return(0); }
////////////////////////////////////////////////////////////////////////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
   chSETDLGICONS(hwnd, IDI_REGNOTIFY);    // Set up the resizeing of the controls
   g_UILayout.Initialize(hwnd);
   g_UILayout.AnchorControl(CUILayout::AP_TOPLEFT, CUILayout::AP_BOTTOMRIGHT,
     IDC_REGVALUES, FALSE);
   g_UILayout.AnchorControl(CUILayout::AP_TOPLEFT, CUILayout::AP_TOPRIGHT,
     IDC_REGKEY, FALSE);
   SetDlgItemText(hwnd, IDC_REGKEY, TEXT("SOFTWARE\\RegNotify"));    return(TRUE); }
////////////////////////////////////////////////////////////////////////////
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
   static HANDLE s_hThread = NULL;    switch (id) {    case IDCANCEL:
     if (s_hThread != NULL) {          // Queue the useless APC function to signal the exit of our
       // other thread          QueueUserAPC(DoNothingAPC, s_hThread, 0);
       // Wait for the thread to exit          WaitForSingleObject(s_hThread, INFINITE);
       CloseHandle(s_hThread);          s_hThread = NULL;       }       EndDialog(hwnd, id);       break;
      case IDOK:       // Disable our button       EnableWindow(hwndCtl, FALSE);
     // Start the notification thread
     s_hThread = chBEGINTHREADEX(NULL, 0, RegSubkeyWatcher, hwnd, 0, NULL);       break;
   } } ////////////////////////////////////////////////////////////////////////////
void Dlg_OnSize(HWND hwnd, UINT state, int cx, int cy) {    // Reposition the child controls
```

```
   g_UILayout.AdjustControls(cx, cy); } ////////////////////////////////////////////////////////////////////////////
 void Dlg_OnGetMinMaxInfo(HWND hwnd, PMINMAXINFO pMinMaxInfo) {
   // Return minimum size of dialog box     g_UILayout.HandleMinMax(pMinMaxInfo); }
 ////////////////////////////////////////////////////////////////////////
 INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
   switch (uMsg) {    chHANDLE_DLGMSG(hwnd, WM_INITDIALOG,    Dlg_OnInitDialog);
   chHANDLE_DLGMSG(hwnd, WM_COMMAND,       Dlg_OnCommand);
   chHANDLE_DLGMSG(hwnd, WM_SIZE,          Dlg_OnSize);
   chHANDLE_DLGMSG(hwnd, WM_GETMINMAXINFO, Dlg_OnGetMinMaxInfo);    }
   return(FALSE); } ////////////////////////////////////////////////////////////////////////////
 int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {
   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_REGNOTIFY), NULL, Dlg_Proc);    return(0);
 } ////////////////////////////////// End of File /////////////////////////////////
```

[Previous] [Next]

Maintaining a Clean Registry

As I have pointed out several times in this chapter, the responsibility of avoiding registry clutter rests on the application developer. Your server (or any other application you write) should not only attempt to make efficient use of the system registry but also clean up after itself when it is finished.

The system provides two functions for deleting data from the registry: *RegDeleteKey* and *RegDeleteValue*. These functions are prototyped as follows:

```
LONG RegDeleteValue(
   HKEY   hkey,
   PCTSTR pszValueName);

LONG RegDeleteKey(
   HKEY   hkey,
   PCTSTR pszSubKey);
```

You might be surprised to find out that deleting data is not a completely trivial topic. The *RegDeleteValue* function is as simple as it looks, assuming your application has security permission to delete a particular key. However, the possibility of your application *not* having permission to delete a key forced the developers to implement *RegDeleteKey* with the following major restriction: *RegDeleteKey* will delete only a key that contains no subkeys. The reason for this restriction becomes clear with a little thought on the matter—your process might not have permission to delete a subkey in the tree below the key that you wish to delete. The system cannot simply traverse the tree and check security permissions before beginning the delete procedure, because doing so could be a lengthy operation, and the permissions might actually change on a subkey after the security check has passed. This, in turn, could cause the deletion of subkeys to fail in the middle of the delete operation, leaving a partially destroyed registry tree.

Although *RegDeleteKey* will not delete a key with subkeys, the developers of the Windows shell implemented a function similar to *RegDeleteKey* that will. This function is named *SHDeleteKey* and is defined as follows:

```
DWORD SHDeleteKey(
   HKEY   hkey,
   PCTSTR pszSubKey);
```

Your process should have permission to delete all underlying keys before calling *SHDeleteKey*. If your process does not have permission to delete all underlying keys when it calls *SHDeleteKey*, the function detects this and returns "Access denied." Note that *SHDeleteKey* suffers from the same previously mentioned race condition in which, if permissions for a key change during a call to *SHDeleteKey*, *SHDeleteKey* can leave a partially deleted tree.

[Previous] [Next]

More Registry Functions

The designers of Windows 2000 added a registry function known as *RegOverridePredefKey* that allows your code to map one of the predefined registry keys listed in Table 5-1 to another registry key. The intended purpose of *RegOverridePredefKey* is to redirect registry changes made by a software component (such as an ActiveX control or a DLL) to a temporary key. This allows your application to easily and safely determine what keys and values must be added to the registry to accommodate the component. If necessary, your application can make changes or adjustments to these keys and values before copying them to their intended location in the registry. *RegOverridePredefKey* has this prototype:

```
LONG RegOverridePredefKey(
   HKEY hkey
   HKEY hNewKey);
```

You should pass one of the predefined registry keys listed in Table 5-1 for the *hkey* parameter, and you should pass an existing registry key as the *hNewKey* parameter. Passing NULL as *hNewKey* causes the predefined key to revert to its default mapping.

Finally, this chapter would not be complete without mention of *RegSaveKey* and *RegRestoreKey*. *RegSaveKey* allows you to copy a registry key and its underlying tree to a file. *RegRestoreKey* allows you to restore the registry key from a file. The prototypes of these functions are as follows:

```
LONG RegSaveKey(
   HKEY                  hkey,
   PCTSTR                pszFile,
   PSECURITY_ATTRIBUTES psa);

LONG RegRestoreKey(
   HKEY   hkey,
   PCTSTR pszFile,
   DWORD  dwFlags);
```

Although these functions are somewhat useful in their own right, they are closely related to *RegLoadKey* and *RegUnloadKey* and deal with registry hives, which are covered in the Platform SDK documentation.

[Previous] [Next]

Chapter 6

Event Logging

When a typical software application needs to make the user aware of some special situation, it commonly uses visual or audio feedback. Most software is given the luxury of this type of event reporting, because the software can make an important assumption: when it is running, a human is sitting in front of the computer. Most server software, however, can't make such an assumption.

Therefore, server developers use files or some other similar persistent storage to maintain a log of events reported by their software. The system administrator can then routinely view the log file and keep tabs on important events and errors. This solves the problem of not having a person sitting at the machine at all times, but it introduces a new problem: manageability. What system administrator is going to enjoy using a dozen different server applications that report events in a dozen different formats in files stored in a dozen different places on the system? And this scenario doesn't even consider events reported by the operating system itself.

Microsoft Windows addresses this manageability issue by introducing a standard event reporting mechanism: the Event Log service. The Event Log service enforces a standard logging format and makes it easy to view logs by using a single event viewer application that provides one-stop shopping for the system administrator. Windows uses the Event Log service to report system events such as low hard disk space and failed logon attempts. Of course, your server software is not required to take advantage of event logging, but your users will appreciate consistency with the other server software they use as well as consistency with the actual system.

In this chapter, we'll learn about the event log and how to write software that reports events. This includes learning how to compile and utilize message files associated with events. Event reporting is all that most application developers care about, and the Event Viewer snap-in is usually all that any administrator will need to read events, but we'll also cover how to design an application to read an event log.

So let's take a moment to explore event logging, first from the administrator's perspective, and then from the operating system's perspective.

[Previous] [Next]

What Is the Event Log?

From the system administrator's point of view, the event log is little more than a list of messages posted by the system or by application software. This list of messages is organized into logical groups called *log files* (or *logs*). The collection of logs is referred to generally as the *event log*. The system administrator's window to the event log is the Event Viewer snap-in of the Microsoft Management Console (MMC), which is installed with Windows 2000. You can open the Event Viewer by clicking Start, pointing to Programs, pointing to Administrative Tools, and then choosing the Event Viewer option. You can also access the Event Viewer by choosing Computer Management in Administrative Tools. Figure 61 shows the Event Viewer snap-in in Computer Management.

Figure 6-1. *The Event Viewer snap-in in Computer Management*

In the Event Viewer node in the MMC, you see a set of logs. When you select a log, the right-hand pane displays the list of that log's events. Double-clicking on an event entry gives you detailed information about the event. I will discuss the information found in each event in a moment, but first we need to discuss the purpose of the different logs.

By default, your system's event log contains three logs: Application, System, and Security. Applications can add their own custom logs to the system; however, this is not typically necessary or common. If you decide to report to your own custom log file, you tell the Event Viewer snap-in about it by selecting the Event Viewer node in the left-hand pane and choosing Open Log File from the Action menu. This produces an Open dialog box that allows you to open a log file. You must report at least one event to a custom log before viewing the

log with the Event Viewer snap-in.

Table 6-1 describes the three standard event logs. Since the subject of this book is writing server applications, the Application log will be of most interest to us.

Table 6-1. *Standard event logs in Event Viewer*

Log Name	Description
Application	Contains events generated by application software and services
System	Contains events generated by device drivers and other operating system components
Security	Contains events generated by security audits

Now let's take a moment to dissect a logged event entry. An event is a single entry in an event log and consists of the following information fields: event type, date and time generated, date and time written, event source, event category, event ID, user, and system. In addition to this information, each event can contain a detailed textual description and have binary data associated with it. The Event Viewer snap-in is capable of displaying most of this information. Table 6-2 offers a brief description of each field.

Most of the fields are self-explanatory, but the event source, event ID, event category, and event type deserve more explanation.

The event source represents the application, service, or system component that reported the event. Typically a one-to-one relationship exists between the reporting agent and event source. However, the code that is reporting the event decides which source it is reporting as, so a single application can report as multiple sources. Likewise, multiple applications can report as a single source. Windows does not restrict this reporting flexibility in any way.

The event ID is a source-defined value that identifies a certain type of event. Any event can be identified via a composite of the event's source and ID. For example, the Browser service defines event ID 8021 as "The browser was unable to retrieve a list of servers from the browser master <*servername*> on the network …" and event ID 8033 as "The browser has forced an election …."

The event category is an optional source-defined category for the event. It is helpful for applications and system components that report a large number of different types of events to be further broken down into logical categories.

Table 6-2. *Fields for an entry in an event log*

Field	Description
Event type	Identifies the type of event. The system defines five different event types, listed in Table 6-3.
Date and time generated	Identifies the time that the source wanted the event added to a log.
Date and time written	Identifies the time that the system recorded the entry in a log.
Event source	Identifies the component responsible for adding the event to a log. Usually the source is an application or a service.
Event category	Identifies a source-defined category for the event.
Event ID	Identifies a source-defined number that uniquely indicates the nature of the incident that caused an entry to be added to a log.

User	Identifies the user account context that generated the event entry. This value is a user's security identifier (SID). See Chapter 9 for more discussion on SIDs.
System	Identifies the machine on which the incident occurred.

You must decide whether the event category is necessary or useful for your software. If an event source chooses to ignore categories, the Event Viewer snap-in will report no categories for events from that source.

The event type can be one of five system-defined event types shown in Table 6-3. The software that is reporting the event selects the event type.

Table 6-3. *Event types*

Event Type	Description
EVENTLOG_INFORMATION_TYPE	Information events indicate a situation or an operation that occurred that is not problematic to the application or system—for example, the starting or stopping of a service application.
EVENTLOG_WARNING_TYPE	Warning events signify potential or future problem situations—for example, relatively low memory or disk space, which might become problematic if resources continue to be consumed.
EVENTLOG_ERROR_TYPE	Error events are logged when an application or a system component actually failed some part of its functionality—for example, an inability to write data to a disk, which resulted in data loss.
EVENTLOG_AUDIT_SUCCESS	Success audit events are logged by Windows security when an audited action is performed successfully.
EVENTLOG_AUDIT_FAILURE	Failure audit events are logged by Windows security when an audited action is attempted and fails.

[Previous] [Next]

Reporting Events

Before we submerse ourselves in event reporting, we'll get our feet wet with some discussion about what events your software should report to the event log. Then we'll take the plunge into learning how to write software that reports events.

What Events Should Be Reported?

If you are developing a server application, you will likely want to do some event reporting. Before you do, however, you must really understand the intended use of the event log. Remember that the event log is the system administrator's source of feedback from your service. A service that reports nonsensical events or too many events will be as aggravating to the system administrator as an application that displays too many message boxes to a user.

Most developers find it easy to decide which situations warrant the logging of an error event. And likewise, it is reasonably easy to decide when your software needs to issue a warning event. However, you enter into a gray area with information events. What software activity is significant enough to be worthy of logging an information event? You can usually answer this question if you consider it from the system administrator's point of view.

No administrator wants to wade through hundreds or even dozens of information events at the end of the day, just to find the one or two items that are relevant to her job. So one aspect of the "significance" question you should consider is how common a particular situation is. A Web server most likely won't want to log an event for every connection it receives. It might, on the other hand, want to report an event for every connection that was denied because it was too busy to process the request. But even logging this event might be unnecessary. A compromise might be to log an event once per hour if connections were denied. The log could include in the detailed description the total number of connections denied. One possibility is to make this kind of reporting optional so that the administrator can pick and choose what is reported.

Another use of information events would be to report infrequent changes in the state of your software. For example, you might want to generate an event each time your service is paused or continued. Or suppose your software goes into a standby mode, freeing up certain resources when connection load has been low for a certain period of time—such an occurrence might warrant an information event. State-change information events can be useful because they give the administrator a sense of what activities your software performed before a warning or error event occurred.

However, you need to avoid getting into the habit of thinking about the event log as a repository of trace information for your application. This type of debugging information will overwhelm an administrator and most likely cause her to ignore every event your application makes. Each event takes up disk space on your system as well, so efficient use of storage is also a concern. If your server will be reporting many events, or will be reporting events from multiple event sources, you might want to consider logging your events to a custom log. Ultimately common sense has to rule here; however, you can offer your user the best of all worlds by making the reporting mechanism of your application as user-configurable as possible.

Well, that is enough on what events should be reported. Now let's jump into the how.

How to Report Events

Reporting software events is a simple process, but you need to understand how the puzzle is put together before you can really take advantage of event logging. The easiest way to fit the pieces together is to look at the simplest piece first, which is the reporting process.

If your process wants to begin reporting events to the event log, it needs to register an event source using the following function:

```
HANDLE RegisterEventSource(
   PCTSTR pszMachineName,
   PCTSTR pszSourceName);
```

The *pszMachineName* parameter identifies the system that contains the log files that you want to append event entries to. Passing NULL for this parameter opens the log file on the local machine. Very rarely are events reported to a remote machine's log files.

The *pszSourceName* parameter is the name of the event source. This name is the text that is shown in the source column in the Event Viewer snap-in; it is not necessarily the name of your executable program (but it usually is).

NOTE

I need to offer a word about security: the Security and System logs are secured. To have your application append events to either of these log files, your process must be running under a privileged security context when it calls *RegisterEventSource*. Here are the rules: the Security log can be written to only by the LocalSystem account, and the System log can be written to only by the LocalSystem and Administrator accounts.

If *RegisterEventSource* is successful, it returns a valid handle. Your application now uses this handle to report events. Remember that like all handles, when you are finished with it you should let the system know. You do this with a call to the *DeregisterEventSource* function:

```
BOOL DeregisterEventSource(HANDLE hEventLog);
```

So far so good, right? Well, reporting an event is also fairly trivial. You do it with a call to the following function:

```
BOOL ReportEvent(
   HANDLE  hEventLog,
   WORD    wType,
   WORD    wCategory,
   DWORD   dwEventID,
   PSID    psidUser,
   WORD    wNumStrings,
   DWORD   dwDataSize,
   PCTSTR* ppszStrings,
   PVOID   pvRawData);
```

The *hEventLog* parameter is the handle returned by *RegisterEventSource*. The *wType* parameter is one of the five predefined event types in Table 6-3. The *wCategory* and *dwEventID* parameters are values chosen arbitrarily by you. The *dwEventID* parameter will uniquely identify the event for your source. Event IDs can be any arbitrary value and arranged in any order you like. However, the category of an event, which indicates a source-defined grouping of events, must have values that start with 1 and progress from there. The value 0 is reserved to indicate no category. The *psidUser* parameter identifies a user; it does not imply security restrictions on the event. Frequently, NULL is passed to indicate an event not related to any user.

The *ppszStrings* parameter points to an array of text strings to associate with the event, where *wNumStrings* is the number of strings in the array. (I will discuss the *ppszStrings* parameter later in the section "Building Message DLLs and EXEs.") Finally, the *pvRawData* parameter points to a block of binary data to associate with the event. The *dwDataSize* parameter indicates the size of the data block, in bytes. The data is defined by the reporting application, and can be any data that you feel will be useful to the user or administrator. For example, network drivers place raw packet data in certain events reported by the driver.

According to the documentation, the functions just described are all you need to know to log events to the event log. Technically this is the truth; however, some key points are left out. If you compare the information that we are giving the Event Log service using *ReportEvent* with the information that the Event Viewer snap-in displays for a typical event, you will find that some data is missing. Specifically, we pass a category number, whereas the Event Viewer snap-in displays a text string. Also, neither *RegisterEventSource* nor *ReportEvent* indicate which log we wish to report under (that is, Application, System, or Security). And finally, you might have noticed that *ReportEvent* lacks a parameter for the detailed description of the event.

By default the approach I explained places your event in the Application log. The log includes generic text for the detailed description of your event and then appends your string data to the end of the text. We could stop here, but your users will thank you if you go the extra mile and implement your own detailed descriptions and categories. To do this, it helps to understand the design goals of event logging.

Message Files

The designers of the Windows event log wanted to create an efficient mechanism that promoted language independence. As a result of this goal, the portions of an event that are human-readable—that is, the category and the detailed description—are abstracted from your application into separate message files. These message files are implemented as DLLs or EXEs, and they contain a custom binary resource containing the text for your messages. I will explain the details surrounding the message DLL later in this chapter; however, at this time there are a couple of things you should know.

Three types of message files can be associated with a given event source: event message files, category message files, and parameter message files. A single message DLL (or EXE) can represent any combination of these, so your application can store your event message file, category message file, or parameter message file in one message DLL. The converse is also true. For example, it is possible to use more than one message DLL to represent the event message file for a single event source.

> **NOTE**
>
> The DLL and EXE files are generally referred to as message DLLs and message EXEs, respectively. For ease of reading, I'll primarily use "message DLL" in the remainder of this chapter, but all the information applies equally to a message EXE.

You associate your message DLL with your event source by creating a handful of registry values under a subkey with the same name as the *pszSourceName* parameter that you passed to *RegisterEventSource*. The event log kills two birds with one stone with this new registry key by also using it to designate into which log your source's events will be logged. Typically the software that installs your application adds these entries to the registry, but no rule says that your service can't add them. When the entries are added to the registry, the layout of the registry ends up following this hierarchy:

```
HKEY_LOCAL_MACHINE
   SYSTEM
      CurrentControlSet
         Services
            EventLog
               Application
                  Event Source
                     ...
               Security
                  Event Source
                     ...
               System
                  Event Source
                     ...
               CustomLog
                     ...
```

Notice that the CustomLog key is under the EventLog key. By default, the only keys under EventLog are Application, Security, and System, but your code can add another key as a peer to the standard log keys to introduce a custom log to the event log. When you call *RegisterEventSource*, the system searches the keys under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\ Services\EventLog*Log* for a key that matches the *pszSourceName* parameter. Each log is searched in alphabetic order until a matching source subkey is found. So as you can see, the namespace for event sources is shared for all logs, standard or custom. If an event source under System matches an event source under Application, it will be ignored.

To associate your message files with your event source, you create the appropriate registry values under your event source key. The supported registry values are described in Table 6-4.

Table 6-4. *Supported registry values for event source subkeys*

Registry Value	Type	Description
TypesSupported	REG_DWORD	Identifies a set of flags, shown in Table 6-3, indicating the types of events supported by the message DLL.
EventMessageFile	REG_EXPAND_SZ	Identifies the pathnames (separated by semicolons) to the event message files. The Event Viewer searches this set of files when trying to convert an event ID to a human-readable string.
CategoryMessageFile	REG_EXPAND_SZ	Identifies the pathnames (separated by semicolons) to category message files. The Event Viewer searches this set of files when trying to convert a category ID to a human-readable string.
ParameterMessageFile	REG_EXPAND_SZ	Identifies the pathnames (separated by semicolons) to parameter message files. The Event Viewer searches this set of files when trying to convert a replaceable string parameter ID to a human-readable string.
CategoryCount	REG_DWORD	Indicates the number of categories supported by the event source.

Notice that all the message file values in Table 6-4 are of the REG_EXPAND_SZ data type. This means that the pathname can include system environment variables that will be expanded at run time. The following example pathname set is a legal value for any of the message file registry values:

```
"%SystemRoot%\System32\msg.dll;c:\messages\msg2.dll"
```

NOTE

Listing the path to a message file in UNC (Universal Naming Convention) format (even if it is on the local machine) is often preferable to using a drive letter and path, because your users might be viewing your logged events from a remote machine. When an event viewing utility (such as the Event Viewer snap-in) looks up the path of a message file DLL and finds it on a network share, the Event Viewer will be able to load the DLL and read the messages. If the Event Viewer snap-in cannot locate the specified message DLL, it cannot convert IDs to human-readable strings. Here is an example of what the Event Viewer snap-in shows when IDs are not converted to strings:

Notice that the Event ID and Category fields are displayed as numbers instead of strings. Also notice that the Description field displays the best information it can under the circumstances.

By the way, containing all the event, category, and parameter messages in a single EXE or DLL file is common. When this is this case, the three message-file registry values are set identically.

Each individual log, such as Application or a custom log, is maintained in its own log file with the extension .evt. Normally the event viewer grabs the registry subkey name, appends the .evt extension, and attempts to open a log file with the resulting filename. However, you can override this behavior and specify a custom pathname for the log file by adding a File registry value under the Log name key. Changing the pathname this way requires rebooting the system for the changes to take affect. Note that the previous log information for a changed log file will not display in the new log.

Two more registry values that affect a log are worth mentioning. The first is MaxSize, which is of type REG_DWORD. This value is the maximum size, in bytes, that the system will allow the log file to grow to. The second value is Retention, which is also a REG_DWORD. This value specifies how old an event should be, in seconds, before it is automatically deleted from the log file. Your code is free to modify these values. If the MaxSize and Retention values do not exist in the registry for a given log, the system defaults to a maximum log file size of 512 KB and a 7-day retention time.

You will unlikely change these values manually in the registry. Rather, you will probably change them using the Event Viewer snap-in by selecting a log and displaying its properties dialog box. Figure 6-2 shows the properties dialog box for a log.

Figure 6-2. *The properties dialog box for the Application log*

Now that we have discussed logging events and associating message DLLs with your event source, it is time to tie the two topics together with a detailed discussion of the message files.

[Previous] [Next]

Building Message DLLs and EXEs

The questions we have yet to answer are as follows:

- How is the text in the message file associated with an event?
- How does the *ppszStrings* parameter of *ReportEvent* get utilized?
- How do we create a message DLL?

In this section, I'll answer all of these questions. Figure 6-3 shows the whole event logging architecture including the steps necessary for building a message DLL. You'll want to refer to this figure as the discussion continues.

Your first step in creating a message DLL is to create a text file for your message source. This message source is generally referred to as an "MC" file and can have any name you choose as long as it has an .mc extension. An example is MyMsgs.mc.

Your .mc file is structured as a series of message entries, arranged in any order. See the Platform SDK documentation for a complete description of the message file syntax. Here is an example of a message file:

```
;/************************************************************
;Module name: MyMsgs.mc
;************************************************************/

;//********************** MESSAGE SECTION ***********************
MessageIdTypedef=DWORD

MessageId=0x1
SymbolicName=MSG_DATE
Language=English
Today's date is %1.  %%536871912
.

MessageId=0x2
SymbolicName=MSG_TIME
Language=English
The current time is %1. %%536871912
.

MessageId=
SymbolicName=MSG_SEC
Language=English
The seconds are %1. %%536871912
.


;//***************** STRING PARAMETERS SECTION ******************
MessageIdTypedef=DWORD

MessageId=0x100
Language=English
I hope you enjoy today.
.

;//************************ END OF FILE ************************
```

Figure 6-3. *The event logging architecture, including steps for building a message DLL*

The *MessageId* is the ID associated with the text. Notice that you can omit an explicit value for the *MessageId* field in the message file. This will cause the compiler to generate an ID that is one more than the previous message ID. The *SymbolicNam*e field refers to the name that will be given to the macro defined in the generated header file. You can include the generated header file with your event reporting code and use the macro defined by the *SymbolicName* field to identify a message. The *Language* line defines the language for the message. In this way you can create a single message resource that has multiple language support for a single message ID. Since a message can be several lines long, you use a line with a single period to denote the end of the message. Note that the trailing period is required, and the last entry in your .mc file should have a carriage return after the trailing period.

When you log an event using *ReportEvent*, the *dwEventID* parameter designates the ID of the event. Later the Event Viewer snap-in searches the event message file for a human-readable string by using the value of the event ID. Likewise, the *wCategory* parameter of *ReportEvent* correlates to a message with the same ID in the category message file.

As you can see, the message file syntax is quite simple, but a few aspects require some additional discussion. As I mentioned earlier, one of the main goals of event logging is language independence. The message resource allows you to easily include messages for multiple languages. To do this, you just append an additional *Language* line followed by the message in the desired language to each message's entry. It is also common to compile different languages into parallel message DLLs. For example, your project might include a MsgEnglish.dll and a MsgFrench.dll. As long as both of these DLLs are included in the registry entry for your message file, the Event Viewer snap-in can find the message for the appropriate language.

With the knowledge you have so far, you might think that for any given event ID, the Event Viewer snap-in would report an unchanging detailed message. If you think that an event logging mechanism made up of static strings defined at compile time is not very useful, you are right. We need the ability to report multiple events of the same ID but with different text, depending on the situation. For example, if your application is reporting that it is unable to open a file, you want it to dynamically include the name of the culprit file in the description text of the event. You might have guessed that event logging's solution to this lies in *ReportEvent*'s *ppszString* parameter.

When you create a message, the message can contain special character sequences that indicate where the event-specific strings should be placed in the text. For example, the following event string indicates two replaceable strings:

```
"The file %1 was replaced with the file %2"
```

If you pass an array of two strings using *ReportEvent*'s *ppszStrings* parameter, the Event Viewer snap-in substitutes "%1" with the first string and "%2" with the second string.

As a rule, you should only pass string values that are not language dependent, such as numbers, filenames, and names of other resources. Although you could pass any text and the Event Viewer snap-in will substitute it, passing an English phrase, for instance, breaks the language independence of event logging.

Parameter replacement can be used in conjunction with string expansion to include additional detailed strings from the parameter message file. When expanding an event string, a "%%" character sequence followed by a number indicates a replaceable parameter. For example, the following event string indicates one replaceable parameter string:

```
"This is an example %%237"
```

When expanding this string, the Event Viewer snap-in searches the parameter message file's message table resource for a string with an ID of 237 and replaces the event string's "%%237" with the parameter string.

Additionally, parameter expansion is performed after string expansion, which allows for complex expansions. Assume the message string "Replace with a dynamic parameter %%%1" and a string of value "237" were passed to *ReportEvent*. The Event Viewer snap-in first replaces the "%1" with "237", and then continues to replace "%%237" with the parameter in the parameter message file that has the ID of 237.

If you set your event source's ParameterMessageFile registry value to Kernel32.dll, you can dynamically insert error message text using values returned from *GetLastError*. For example, assume Kernel32.dll is your parameter message file, and your message string is "GetLastError() returned the following error: %%%1". You could call *ReportEvent* with a single string value of "5", indicating access denied. The resulting event message would be as follows:

```
"GetLastError() returned the following error: Access is denied".
```

Being able to insert the last error message text in your event description is a useful feature, indeed.

Compiling Your Messages

After you create your .mc file, you need to compile it using the Message Compiler (MC.exe) that ships with Microsoft Visual Studio. The following text shows the message compiler's usage:

```
C:\>mc.exe
Microsoft (R) Message Compiler  Version 1.00.5239
Copyright (c) Microsoft Corp 1992-1995. All rights reserved.

usage: MC [-?vcdwso] [-m maxmsglen] [-h dirspec] [-e extension]
          [-r dirspec] [-x dbgFileSpec] [-u] [-U] filename.mc
   -? - displays this message
   -v - gives verbose output.
   -c - sets the Customer bit in all the message Ids.
   -d - FACILTY and SEVERITY values in header file in decimal.
        Sets message values in header to decimal initially.
   -w - warns if message text contains non-OS/2 compatible inserts.
   -s - insert symbolic name as first line of each message.
   -o - generate OLE2 header file (use HRESULT definition instead of
        status code definition)
   -m maxmsglen - generate a warning if the size of any message exceeds
                  maxmsglen characters.
   -h pathspec - gives the path of where to create the C include file
                 Default is .\
   -e extension - Specify the extension for the header file.
                  From 1 - 3 chars.
   -r pathspec - gives the path of where to create the RC include file
                 and the binary message resource files it includes.
                 Default is .\
   -x pathspec - gives the path of where to create the .dbg C include
                    file that maps message Ids to their symbolic name.
   -u - input file is Unicode.
   -U - messages in .BIN file should be Unicode.
   filename.mc - gives the names of a message text file
                 to compile.
   Generated files have the Archive bit cleared.
```

The message compiler parses your .mc file and produces three files:

- **MSG00001.bin** This file contains all the message strings in binary format. It also contains information to map a message ID number to its string.

- ***MyMsgs*.rc** This resource script file contains just a reference to the binary information contained in the MSG00001.bin file.

- ***MyMsgs*.h** This file is a C/C++ header file that contains a *#define* for every symbol name appearing in the .mc file. You should include this file in any of your source code modules that call the *ReportEvent* function.

The Resource File

After running MyMsgs.mc through the message compiler, I end up with a very simple resource script file (MyMsgs.rc) that looks like Figure 6-4:

Figure 6-4. *A resource script file generated by the message compiler*

You are probably already familiar with icon, bitmap, and dialog box template resources. Well, a message table is just another type of resource. Like icons and bitmaps, message table resources are binary files that are referred to in the resource script, unlike dialog box templates and menu templates, which are embedded in the script.

If you open the WinUser.h header file in the Platform SDK, you will find the following set of defined symbols:

```
#define RT_CURSOR           MAKEINTRESOURCE(1)
#define RT_BITMAP           MAKEINTRESOURCE(2)
#define RT_ICON             MAKEINTRESOURCE(3)
#define RT_MENU             MAKEINTRESOURCE(4)
#define RT_DIALOG           MAKEINTRESOURCE(5)
#define RT_STRING           MAKEINTRESOURCE(6)
#define RT_FONTDIR          MAKEINTRESOURCE(7)
#define RT_FONT             MAKEINTRESOURCE(8)
#define RT_ACCELERATOR      MAKEINTRESOURCE(9)
#define RT_RCDATA           MAKEINTRESOURCE(10)
#define RT_MESSAGETABLE     MAKEINTRESOURCE(11)   // Look Here!
#define RT_GROUP_CURSOR     MAKEINTRESOURCE(12)
#define RT_GROUP_ICON       MAKEINTRESOURCE(14)
#define RT_VERSION          MAKEINTRESOURCE(16)
#define RT_DLGINCLUDE       MAKEINTRESOURCE(17)
#define RT_PLUGPLAY         MAKEINTRESOURCE(19)
#define RT_VXD              MAKEINTRESOURCE(20)
#define RT_ANICURSOR        MAKEINTRESOURCE(21)
#define RT_ANIICON          MAKEINTRESOURCE(22)
#define RT_HTML             MAKEINTRESOURCE(23)
```

Message table resources have been assigned the number 11. When you add resources to a resource script (.rc) file, you must assign each type of resource a unique number. For example, I can add an icon to my resource script with an ID of 53, and I can add another icon to a resource script file with an ID of 172. The numbers don't actually matter as long as they're unique.

Message table resources work a little differently than other resources. A resource script file can have only one message table resource, and that resource must be assigned an ID of 1. If you assign a different ID to a message table resource, tools such as the Event Viewer snap-in will not be able to convert the event and

category IDs to human-readable strings.

NOTE

If your message DLL or EXE contains additional resources, just copy the contents of the message compiler_generated .rc file into your own .rc file. Then you can discard the .rc file generated by the message compiler.

Using Visual Studio to Create a Message File Project

Although you can run the message compiler utility manually each time you change your messages, doing so can become tedious and cumbersome. I strongly recommend incorporating the message-compiling step into your Visual Studio project for your message DLL or EXE. For some unknown reason, the Visual Studio environment is not aware of the message compiler and .mc files. So you need to add custom build steps to your project. Here are the steps:

1. Add your .mc file to your EXE or DLL project.

2. Display the Project Settings dialog box.

3. Select the .mc file and click on the Custom Build tab.

4. Set the Description text box to whatever you like. I always use "Message Compiler".

5. Set the Commands text box to "mc -s -U -h $(ProjDir) -r $(ProjDir) $(InputName)" and "del $(ProjDir)\$(InputName).rc".

6. In the Outputs section, add two entries: "$(InputName).h" and "Msg00001.bin". The dialog box should look like Figure 6-5.

   Click OK.

Figure 6-5. *The Project Settings dialog box, after adding message compiler commands for the .mc file*

7. Include the generated header file in the source files that call the *ReportEvent* function.

8. In your project's .rc file, add the lines to include resource number 1 and resource type 11, as well as a filename of MSG00001.bin. For example,

```
1 11 MSG00001.bin
```

After performing these steps, Visual Studio will know how to compile your message text file and include the resource when producing the final EXE or DLL file. Note that if you are creating a DLL module that contains only resources and no code, you should use the /NOENTRY linker switch to prevent linking an entry point and thus reducing the size of the resulting module.

The only code you have left to write is that which configures the registry so that the Event Viewer snap-in is able to locate the message EXE or message DLL.

The MsgTableDump Sample Application

The MsgTableDump sample application ("06 MsgTableDump.exe") is a simple utility that opens the message table resource contained inside an EXE or DLL and dumps all the strings in the table. The source code and resource files for the application are in the 06-MsgTableDump directory on the companion CD. The code is very simple and straightforward. It locates the message table resource in the specified module and walks the resource placing each string in a read-only edit control. Figure 6-6 shows what the output looks like when MsgTableDump performs its magic on Kernel32.dll.

Figure 6-6. *The MsgTableDump sample application showing the message strings for Kernel32.dll*

The AppLog Sample Application

The AppLog sample application ("06 AppLog.exe"), shown in Listing 6-1, demonstrates event reporting from an application. The source code and resource files for the application are in the 06-AppLog directory on the companion CD. The AppLog sample application is implemented as a standard application, but code that reports events would be identical if the code were residing in a service.

When AppLog starts, it opens the local system's Application log and adds an entry indicating that the application has started executing. AppLog also adds an entry just before it terminates. Normally, you would not report these types of informational events to the event log in order to improve performance and not waste space in the event log's database.

Once AppLog is running, you can enter a Win32 error code in the edit box and click the Simulate Error button. This button causes AppLog to append an event to the system's Application log. You can simulate as many Win32 errors as you like while AppLog is running. To see the errors, click the Open Event Viewer

button, which causes the Event Viewer snap-in to display in the MMC.

If you look at the entries using the Event Viewer snap-in, you'll see the category and message ID numbers as well as the replaceable string values (your simulated error code numbers). Event Viewer can't map the category and message IDs to their English strings until the registry has been configured properly. To install the message file module information into the registry, click the Install Event Message File In Registry button. Then go back to the Event Viewer snap-in and look at the events added by AppLog. This time, you should see the ID numbers converted to the appropriate strings. AppLog also allows you to delete the registry information by clicking the Remove Event Message File From Registry button. Figure 6-7 shows AppLog simulating an error.

Figure 6-7. *The AppLog sample application simulating the "Access is denied" (5) Win32 error*

Listing 6-1. *The AppLog sample application*

AppLog.cpp

```
/******************************************************************************
Module: AppLog.cpp Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
#include "..\CmnHdr.h" // See Appendix A. #include <WindowsX.h> #define EVENTLOG_IMPL
#include "EventLog.h" #include "Resource.h" #include "AppLogMsgs.h" // Generated by MC.exe
////////////////////////////////////////////////////////////////////////////// CEventLog g_EventLog(TEXT("AppLog"));
////////////////////////////////////////////////////////////////////////////// BOOL Dlg_OnInitDialog(HWND hwnd,
HWND hwndFocus, LPARAM lParam) { chSETDLGICONS(hwnd, IDI_APPLOG); return(TRUE); }
////////////////////////////////////////////////////////////////////////////// void Dlg_OnCommand(HWND hwnd, int id,
HWND hwndCtl, UINT codeNotify) { switch (id) { case IDCANCEL: EndDialog(hwnd, id); break;
case IDC_SPAWNEVENTVIEWER: // Spawn the Event Viewer Snap-In ShellExecute(hwnd,
TEXT("Open"), TEXT("eventvwr.msc"), TEXT("/s"), NULL, SW_SHOWDEFAULT); break; case
IDC_INSTALL: // Install the event log parameter file info into the registry
chVERIFY(g_EventLog.Install( EVENTLOG_INFORMATION_TYPE |
EVENTLOG_ERROR_TYPE, NULL, TEXT("Kernel32.dll"), 2, NULL)); break; case
```

```
IDC_REMOVE: // Uninstall the event log parameter file info from the registry
chVERIFY(g_EventLog.Uninstall()); break; case IDC_SIMULATEERROR: // Report a Win32 error
code event TCHAR szErrorCode[10]; PCTSTR pszErrorCode = szErrorCode; GetDlgItemText(hwnd,
IDC_ERRORCODE, szErrorCode, chDIMOF(szErrorCode));
chVERIFY(g_EventLog.ReportEvent(EVENTLOG_ERROR_TYPE, CAT_APPEVENT,
MSG_ERROR, CEventLog::REUSER_NOTAPPLICABLE, 1, (PCTSTR*) &pszErrorCode)); break;
} } ////////////////////////////////////////////////////////////////////////// INT_PTR WINAPI Dlg_Proc(HWND
hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) { switch (uMsg) {
chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
chHANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand); } return(FALSE); }
////////////////////////////////////////////////////////////////////////// int WINAPI _tWinMain(HINSTANCE
hinstExe, HINSTANCE, PTSTR pszCmdLine, int) { // Report an event that this application is starting
g_EventLog.ReportEvent(EVENTLOG_INFORMATION_TYPE, CAT_APPEXECSTATUS,
MSG_APPSTART); DialogBox(hinstExe, MAKEINTRESOURCE(IDD_APPLOG), NULL,
Dlg_Proc); // Report an event that this application is stopping
g_EventLog.ReportEvent(EVENTLOG_INFORMATION_TYPE, CAT_APPEXECSTATUS,
MSG_APPSTOP); return(0); } //////////////////////////////// End of File ////////////////////////////////
```

EventLog.h

```
/******************************************************************************
Module: EventLog.h Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
#pragma once // Include this header file once per compilation unit
////////////////////////////////////////////////////////////////////////// #include "..\CmnHdr.h" /* See Appendix A. */
/****************************************************************************** To
add a message compiler file to Visual Studio's Build Environment, perform the following steps: 1)
Insert the *.mc file into the project 2) Select the MC file in the Project Settings dialog box 3) Change
the description to "Message Compiler" 4) Add the following 2 Build Command(s): "mc -s -U -h
$(ProjDir) -r $(ProjDir) $(InputName)" "del $(ProjDir)\$(InputName).rc" 5) Add the following 2
Output file(s): "$(InputName).h" "Msg00001.bin" 6) Include the generated header file in the source
file(s) that call the ReportEvent function 7) Since I delete the MC generated .rc file, you must
manually import the MSG0001.bin file into your project's .rc file using a resource type of 11 and a
resource ID of 1.
******************************************************************************/
class CEventLog { public: CEventLog(PCTSTR pszAppName); ~CEventLog(); BOOL
Install(DWORD dwTypesSupported, PCTSTR pszEventMsgFilePaths = NULL, PCTSTR
pszParameterMsgFilePaths = NULL, DWORD dwCategoryCount = 0, PCTSTR
pszCategoryMsgFilePaths = NULL); BOOL Uninstall(); // Records an event into the event log enum
REPORTEVENTUSER { REUSER_NOTAPPLICABLE, REUSER_SERVICE, REUSER_CLIENT
}; BOOL ReportEvent(WORD wType, WORD wCategory, DWORD dwEventID,
REPORTEVENTUSER reu = REUSER_NOTAPPLICABLE, WORD wNumStrings = 0, PCTSTR
*pStrings = NULL, DWORD dwDataSize = 0, PVOID pvRawData = NULL); private: PCTSTR
m_pszAppName; HANDLE m_hEventLog; }; //////////////////////////////////////////////////////////////////////////
#ifdef EVENTLOG_IMPL //////////////////////////////////////////////////////////////////////////
CEventLog::CEventLog(PCTSTR pszAppName) { m_pszAppName = pszAppName; m_hEventLog =
NULL; } CEventLog::~CEventLog() { if (m_hEventLog != NULL) {
::DeregisterEventSource(m_hEventLog); } } //////////////////////////////////////////////////////////////////////////
BOOL CEventLog::Install(DWORD dwTypesSupported, PCTSTR pszEventMsgFilePaths, PCTSTR
pszParameterMsgFilePaths, DWORD dwCategoryCount, PCTSTR pszCategoryMsgFilePaths) { //
Make sure at least one valid Support Type is specified chASSERT(0 != (dwTypesSupported &
(EVENTLOG_INFORMATION_TYPE | EVENTLOG_WARNING_TYPE |
```

```
EVENTLOG_ERROR_TYPE))); BOOL fOk = TRUE; TCHAR szSubKey[_MAX_PATH];
wsprintf(szSubKey, TEXT("System\\CurrentControlSet\\Services\\EventLog\\Application\\%s"),
m_pszAppName); // If the application doesn't support any types (the default), // don't install an event
log for this service HKEY hkey = NULL; __try { LONG l; l =
RegCreateKeyEx(HKEY_LOCAL_MACHINE, szSubKey, 0, NULL,
REG_OPTION_NON_VOLATILE, KEY_SET_VALUE, NULL, &hkey, NULL); if (l !=
NO_ERROR) __leave; l = RegSetValueEx(hkey, TEXT("TypesSupported"), 0, REG_DWORD,
(PBYTE) &dwTypesSupported, sizeof(dwTypesSupported)); if (l != NO_ERROR) __leave; TCHAR
szModulePathname[MAX_PATH]; GetModuleFileName(NULL, szModulePathname,
chDIMOF(szModulePathname)); if (pszEventMsgFilePaths == NULL) pszEventMsgFilePaths =
szModulePathname; l = RegSetValueEx(hkey, TEXT("EventMessageFile"), 0, REG_EXPAND_SZ,
(PBYTE) pszEventMsgFilePaths, chSIZEOFSTRING(pszEventMsgFilePaths)); if (l != NO_ERROR)
__leave; if (pszParameterMsgFilePaths == NULL) pszParameterMsgFilePaths = szModulePathname; l
= RegSetValueEx(hkey, TEXT("ParameterMessageFile"), 0, REG_EXPAND_SZ, (PBYTE)
pszParameterMsgFilePaths, chSIZEOFSTRING(pszParameterMsgFilePaths)); if (l != NO_ERROR)
__leave; if (dwCategoryCount > 0) { if (pszCategoryMsgFilePaths == NULL)
pszCategoryMsgFilePaths = szModulePathname; l = RegSetValueEx(hkey,
TEXT("CategoryMessageFile"), 0, REG_EXPAND_SZ, (PBYTE) pszCategoryMsgFilePaths,
chSIZEOFSTRING(pszCategoryMsgFilePaths)); if (l != NO_ERROR) __leave; l =
RegSetValueEx(hkey, TEXT("CategoryCount"), 0, REG_DWORD, (PBYTE) &dwCategoryCount,
sizeof(dwCategoryCount)); if (l != NO_ERROR) __leave; } fOk = TRUE; } __finally { if (hkey !=
NULL) RegCloseKey(hkey); } return(fOk); } ////////////////////////////////////////////////////////////////////////////
BOOL CEventLog::Uninstall() { // Install each service's event log TCHAR szSubKey[_MAX_PATH];
wsprintf(szSubKey, TEXT("System\\CurrentControlSet\\Services\\EventLog\\Application\\%s"),
m_pszAppName); return(NO_ERROR == RegDeleteKey(HKEY_LOCAL_MACHINE, szSubKey));
} ////////////////////////////////////////////////////////////////////////////// BOOL CEventLog::ReportEvent(WORD
wType, WORD wCategory, DWORD dwEventID, REPORTEVENTUSER reu, WORD
wNumStrings, PCTSTR* pStrings, DWORD dwDataSize, PVOID pvRawData) { BOOL fOk =
TRUE; // Assume success if (m_hEventLog == NULL) { // This is the first time that ReportEvent is
being // called, open the log first m_hEventLog = ::RegisterEventSource(NULL, m_pszAppName); }
if (m_hEventLog != NULL) { PSID psidUser = NULL; if (reu != REUSER_NOTAPPLICABLE) {
HANDLE hToken; if (REUSER_SERVICE == reu) fOk = OpenProcessToken(GetCurrentProcess(),
TOKEN_QUERY, &hToken); else fOk = OpenThreadToken(GetCurrentThread(), TOKEN_QUERY,
TRUE, &hToken); if (fOk) { BYTE bTokenUser[1024]; PTOKEN_USER ptuUserSID =
(PTOKEN_USER) bTokenUser; DWORD dwReturnLength; GetTokenInformation(hToken,
TokenUser, ptuUserSID, sizeof(bTokenUser), &dwReturnLength); CloseHandle(hToken); psidUser =
ptuUserSID->User.Sid; } } fOk = fOk && ::ReportEvent(m_hEventLog, wType, wCategory,
dwEventID, psidUser, wNumStrings, dwDataSize, pStrings, pvRawData); } return(fOk); }
////////////////////////////////////////////////////////////////////////////// #endif // SERVICECTRL_IMPL
////////////////////////////////// End of File //////////////////////////////////
```

AppLogMsgs.h

```
/****************************************************************** Module:
AppLogMsgs.mc Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************/
//******************** CATEGORY SECTION ********************** // Category IDs are
16-bit values // // Values are 32 bit values layed out as follows: // // 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1
1 1 1 1 // 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 //
+---+-+-+----------------------+------------------------------+ // |Sev|C|R| Facility | Code | //
+---+-+-+----------------------+------------------------------+ // // where // // Sev - is the severity code // //
00 - Success // 01 - Informational // 10 - Warning // 11 - Error // // C - is the Customer code flag // // R
```

- is a reserved bit // // Facility - is the facility code // // Code - is the facility's status code // // // Define the facility codes // // // Define the severity codes // // // MessageId: CAT_APPEXECSTATUS // // MessageText: // // Application execution status // #define CAT_APPEXECSTATUS ((WORD)0x20000001L) // // MessageId: CAT_APPEVENT // // MessageText: // // Application event // #define CAT_APPEVENT ((WORD)0x20000002L) //********************* MESSAGE SECTION ********************** // Event IDs are 32-bit values // // MessageId: MSG_APPSTART // // MessageText: // // Application started. // #define MSG_APPSTART ((DWORD)0x20000064L) // // MessageId: MSG_APPSTOP // // MessageText: // // Application stopped. // #define MSG_APPSTOP ((DWORD)0x20000065L) // // MessageId: MSG_ERROR // // MessageText: // // Application generated error code %1: "%%%1" // #define MSG_ERROR ((DWORD)0x20000066L) //************************ END OF FILE ************************

AppLogMsgs.mc

;/** ;Module: AppLogMsgs.mc ;Notices: Copyright (c) 2000 Jeffrey Richter ;**/ ;//********************* CATEGORY SECTION ********************** ;// Category IDs are 16-bit values MessageIdTypedef=WORD MessageId=1 SymbolicName=CAT_APPEXECSTATUS Language=English Application execution status . MessageId=2 SymbolicName=CAT_APPEVENT Language=English Application event . ;//********************* MESSAGE SECTION ********************** ;// Event IDs are 32-bit values MessageIdTypedef=DWORD MessageId=100 SymbolicName=MSG_APPSTART Language=English Application started. . MessageId= SymbolicName=MSG_APPSTOP Language=English Application stopped. . MessageId= SymbolicName=MSG_ERROR Language=English Application generated error code %1: "%%%1" . ;//************************ END OF FILE ************************

AppLog.rc

//Microsoft Developer Studio generated resource script. // #include "Resource.h" #define APSTUDIO_READONLY_SYMBOLS // // // Generated from the TEXTINCLUDE 2 resource. // #include "afxres.h" // #undef APSTUDIO_READONLY_SYMBOLS // // English (U.S.) resources #if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU) #ifdef _WIN32 LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US #pragma code_page(1252) #endif //_WIN32 // // // Icon // // Icon with lowest ID value placed first to ensure application icon // remains consistent on all systems. IDI_APPLOG ICON DISCARDABLE "AppLog.ico" // // // Dialog // IDD_APPLOG DIALOG DISCARDABLE 15, 24, 280, 41 STYLE DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU CAPTION "Application Log" FONT 8, "MS Sans Serif" BEGIN LTEXT "Win32 &error code:",IDC_STATIC,4,6,58,8 EDITTEXT IDC_ERRORCODE,68,4,40,12,ES_AUTOHSCROLL | ES_NUMBER DEFPUSHBUTTON "&Simulate error",IDC_SIMULATEERROR,112,4,53,14 PUSHBUTTON "Open Event &Viewer",IDC_SPAWNEVENTVIEWER,196,4,80,14 PUSHBUTTON "&Install event message file in registry",IDC_INSTALL,4, 24,132,14 PUSHBUTTON "&Remove event message file from registry",IDC_REMOVE, 144,24,132,14 END #ifdef APSTUDIO_INVOKED // // // TEXTINCLUDE // 1 TEXTINCLUDE DISCARDABLE BEGIN "Resource.h\0" END 2 TEXTINCLUDE DISCARDABLE BEGIN "#include ""afxres.h""\r\n" "\0" END 3 TEXTINCLUDE DISCARDABLE BEGIN "\r\n" "\0" END

#endif // APSTUDIO_INVOKED // // // DESIGNINFO // #ifdef APSTUDIO_INVOKED GUIDELINES DESIGNINFO DISCARDABLE BEGIN IDD_APPLOG, DIALOG BEGIN RIGHTMARGIN, 276 BOTTOMMARGIN, 37 END END #endif // APSTUDIO_INVOKED // // // 11 // LANGUAGE 0x9,0x1 1 11 MSG00001.bin #endif // English (U.S.) resources // #ifndef APSTUDIO_INVOKED // // // Generated from the TEXTINCLUDE 3 resource. // // #endif // not APSTUDIO_INVOKED

[Previous] [Next]

Reading the Event Log

The Event Viewer snap-in that ships with Windows is usually sufficient for most event-reading needs. However, Windows does provide functions that allow your own applications to access event logs. There are lots of possibilities for a feature like this—for example, you could write an application that sends an e-mail when it detects that an event entry of a certain ID is being added to an event log.

To get started reading the event log, your application must first retrieve a handle to a log by calling *OpenEventLog*:

```
HANDLE OpenEventLog(
   PCTSTR pszUNCServerName,
   PCTSTR pszLogName);
```

The *pszUNCServerName* parameter identifies the machine containing the event log you wish to access. The *pszLogName* parameter identifies the specific log on the server machine. Once you have a valid handle to the log file, you are prepared to read events. As always, you should close the handle when you are finished accessing the log file by calling *CloseEventLog*:

```
BOOL CloseEventLog(HANDLE hEventLog);
```

Reading events requires a call to *ReadEventLog*:

```
BOOL ReadEventLog(
   HANDLE hEventLog,
   DWORD  dwReadFlags,
   DWORD  dwRecordOffset,
   PVOID  pvBuffer,
   DWORD  nNumberOfBytesToRead,
   PDWORD pnBytesRead,
   PDWORD pnMinNumberOfBytesNeeded);
```

The *hEventLog* parameter is the handle returned from *OpenEventLog*. The *dwReadFlags* parameter identifies whether you will read the log sequentially or start reading from a specific record. Table 6-5 lists the possible flags for *dwReadFlags*. A common value to pass is EVENTLOG_FORWARDS_READ | EVENTLOG_SEQUENTIAL_READ.

Table 6-5. *Flags that can be passed for* ReadEventLog's dwReadFlags *parameter*

Flag	Description
EVENTLOG_SEEK_READ	Allows you to specify a zero-based index in the log file for the event that you want to begin reading from. You specify the index with the *dwRecordOffset* parameter. You must

	select either seek or sequential reading.
EVENTLOG_SEQUENTIAL_READ	Indicates that you will be reading the event log sequentially, starting with the event record that follows the most recently read record. This is the most commonly selected read type for the event log.
EVENTLOG_FORWARDS_READ	Indicates that you will be reading forward through the event log file. This flag can be used for either sequential or seek reads.
EVENTLOG_BACKWARDS_READ	Indicates that you will be reading backward through the event log file. This flag can be used with either sequential or seek reads.

To receive event log data, you supply a buffer pointer (*pvBuffer*) and a buffer size in bytes (*nNumberOfBytesToRead*) to *ReadEventLog*. If your buffer is not large enough to read the next record in the log, the function will fail, and *GetLastError* will report ERROR_INSUFFICIENT_BUFFER. The variable pointed to by the *pnMinNumberOfBytesNeeded* parameter will contain the number of bytes needed to read a single record. However, if your buffer is large enough for one or more records, *ReadEventLog* will fill your buffer with the data for as many records as will completely fit in your buffer. I suggest your application make a single call to *ReadEventLog* to find out the necessary buffer size for reading a single event, allocate the buffer, and then call *ReadEventLog* again to read the log. Although you can read multiple events at a time, doing so is not significantly more efficient, and it complicates your parsing code because the data returned is of variable length.

If no more records are left to read, *ReadEventLog* will return FALSE, and *GetLastError* will report ERROR_HANDLE_EOF.

After you successfully call *ReadEventLog*, your buffer will contain one or more EVENTLOGRECORD structures. This structure is of variable length and is declared as follows:

```
typedef struct _EVENTLOGRECORD {
   DWORD Length;
   DWORD Reserved;
   DWORD RecordNumber;
   DWORD TimeGenerated;
   DWORD TimeWritten;
   DWORD EventID;
   WORD  EventType;
   WORD  NumStrings;
   WORD  EventCategory;
   WORD  ReservedFlags;
   DWORD ClosingRecordNumber;
   DWORD StringOffset;
   DWORD UserSidLength;
   DWORD UserSidOffset;
   DWORD DataLength;
   DWORD DataOffset;
   //
   // Then follow:
   //
   // TCHAR SourceName[]
   // TCHAR Computername[]
   // SID UserSid
   // TCHAR Strings[]
   // BYTE Data[]
   // CHAR Pad[]
   // DWORD Length;
   //
} EVENTLOGRECORD;
```

A precious few members of this structure are straightforward, and you should recognize them immediately as the *EventID*, *EventType*, and *EventCategory* fields. But the other members of this structure get more complex from here.

Let's start with the *TimeGenerated* and *TimeWritten* values. As you might have guessed, they correspond to the date and time the event was generated and written to the log, respectively. However, the format of the time value is not one commonly used with Win32 API functions, so it may seem awkward at first. Here is what the documentation has to say about this time format: "This time is measured in the number of seconds elapsed since 00:00:00 January 1, 1970, Universal Coordinated Time." This format is similar to that for the C runtime's *time_t* type, except that the time values for events or "event times" are unsigned. What does this mean? It means that you will have to jump through a couple of hoops to get event time values into a useful time format such as the SYSTEMTIME structure.

If you are familiar with the different time structures supported by Windows, you might recognize the event time format as being similar to the FILETIME format. The FILETIME format, however, is defined as a 64-bit value representing the number of 100-nanosecond intervals since 00:00:00 January 1, 1601. Since Windows provides useful functions for converting from FILETIME to SYSTEMTIME, our best move is to convert our event time to a FILETIME value. Somewhere in the process we should adjust for local time. (Remember that event times are Universal Coordinated Time.) The following code wraps all of this logic in a simple function:

```
void EventTimeToLocalSystemTime(DWORD dwEventTime,
   SYSTEMTIME* pstTime) {

   SYSTEMTIME st1970;
   // Create a FILETIME for 00:00:00 January 1, 1970
   st1970.wYear         = 1970;
   st1970.wMonth        = 1;
   st1970.wDay          = 1;
   st1970.wHour         = 0;
   st1970.wMinute       = 0;
   st1970.wSecond       = 0;
   st1970.wMilliseconds = 0;

   union {
      FILETIME ft;
      LONGLONG ll;
   } u1970;
   SystemTimeToFileTime(&st1970, &u1970.ft);

   union {
      FILETIME ft;
      LONGLONG ll;
   } uUCT;
   // Scale from seconds to 100-nanoseconds
   uUCT.ll = 0;
   uUCT.ft.dwLowDateTime = dwEventTime;
   uUCT.ll *= 10000000;
   uUCT.ll += u1970.ll;

   FILETIME   ftLocal;
   FileTimeToLocalFileTime(&uUCT.ft, &ftLocal);
   FileTimeToSystemTime(&ftLocal, pstTime);
}
```

Well, now that the mystery of the time values is solved, let's move on to the *SourceName* and *Computername* members. These are both string values that are somewhat awkward to retrieve. Although the EVENTLOGRECORD structure provides offsets to some of the variably positioned items in the structure, the designers of the system apparently didn't feel the need to be consistent. As a result, *SourceName* is simply defined as the zero-terminated string immediately following the *DataOffset* member of the structure. Similarly, the *Computername* string starts on the first character following the *SourceName* string. I have thrown together a couple of useful macros (included in EventMonitor.cpp on the companion CD) to ease the

pain of extracting these values from the EVENTLOGRECORD structure. These macros will work with source modules built using either Unicode or ANSI strings.

Fortunately the EVENTLOGRECORD structure includes the *UserSidOffset* and *StringOffset* values for accessing the *UserSid* and *Strings* members gracefully. These offsets start from the beginning of the structure and are measured in bytes. The *UserSid* is a SID structure identifying the user for which the event was logged. In Chapter 9, I discuss how to convert a SID structure into a human-readable user name. The *Strings* array is an array of pointers to the expansion strings, which were passed to the *ppszStrings* parameter of *ReportEvent*.

Converting a Message ID to a Human-Readable String

You are probably wondering how you are supposed to get the text for the event category and the detailed description of the event. Knowing what you know about event reporting, you could infer that it would be possible to look up the event source in the registry, load the appropriate message DLLs, and extract the detailed category and message from the resource manually—but that would be unnecessarily messy, wouldn't it? Well, it would, but the method I described is the only way to retrieve the event text. Fortunately the system does implement a handy function named *FormatMessage* that extracts the resource text, but the rest of the task is up to us. The EventMonitor sample application, discussed at the end of this chapter, shows how to use *FormatMessage* for event reading. The function is defined as follows:

```
DWORD FormatMessage(
   DWORD  dwFlags,
   PCVOID pSource,
   DWORD  dwMessageId,
   DWORD  dwLanguageId,
   PTSTR  pBuffer,
   DWORD  nSize,
   va_list *Arguments);
```

I would have liked to see a pair of functions defined as follows:

```
PTSTR GetEventCategory(
   PTSTR           pszLog,
   PEVENTLOGRECORD pelr);

PTSTR GetEventMessage(
   PTSTR           pszLog,
   PEVENTLOGRECORD pelr);
```

These functions would take a pointer to a log file and an event record from that log file, and then return a buffer containing the requested text. The returned buffer could be freed using *LocalFree* in the tradition of the *FormatMessage* function. However, the system doesn't provide us with these lovely functions. So I took it upon myself to implement them as a macro and a function that wrap my own *FormatEventMessage* function. The complete code for this function is available in EventMonitor.cpp on the companion CD. Let me point out some highlights.

You probably remember from our earlier discussion of reporting events that the message DLLs are stored in registry values named *EventMessageFile*, *ParameterMessageFile*, or *CategoryMessageFile*, depending on which message you are attempting to find. If the event source in question is named MySource and is logged to the Application log, the registry looks like this:

```
HKEY_LOCAL_MACHINE
   SYSTEM
      CurrentControlSet
         Services
            EventLog
               Application
```

```
MySource
   CategoryMessageFile
   EventMessageFile
   ParameterMessageFile
```

It is your application's job, when reading events, to locate the appropriate registry value and parse the string for the one or many message DLLs that contain the desired message string. Remember, you must pass the semicolon-delimited string of message files retrieved from the registry to *ExpandEnvironmentStrings* before using the string to load the modules into your address space, because the registry value type for these values is REG_EXPAND_SZ.

After you have retrieved your list of message DLLs and EXEs, you must load each one in turn (from left to right) into your process's address space using *LoadLibraryEx*. Use *LoadLibraryEx* rather than *LoadLibrary*, because *LoadLibraryEx* allows you to load a module as a resource-only module by passing the LOAD_LIBRARY_AS_DATAFILE flag. After you have retrieved an instance handle from *LoadLibraryEx*, you pass it, along with a message ID and expansion strings, from the EVENTLOGRECORD structure to *FormatMessage*. If the call to *FormatMessage* succeeds, your message has been located, and you can unload the library and return the message text. If it does not succeed, you must unload the message DLL, and then load and try the next message DLL. If you will be extracting message text for more than a single event, you might find it advantageous to optimize your code to avoid repeated loading and unloading of message DLLs.

Although *FormatMessage* automatically expands your strings into the message, it does not automatically expand messages from the *ParameterMessageFile* DLLs into your message. Your code must manually scan for instances of "%%" in the string returned from *FormatMessage* and, using the same algorithm, replace them with the text extracted from the *ParameterMessageFile* DLLs. The EventMonitor sample application in this chapter shows how to do all of this properly.

One final point about *FormatMessage* before we move on: If the message text calls for more strings than the number of strings you pass to *FormatMessage*, the function will blindly attempt to access the nonexistent strings. Most likely, this will cause an access violation, meaning that you either need to parse the string and precount the number of strings expected, which ensures that the correct number of strings is passed, or—and this is the easier approach to take—wrap the call to *FormatMessage* in a structured exception-handling frame that handles the access violation gracefully. Remember that the system makes no promises about the correctness of the events reported by other applications.

While we are on the topic of robust event viewing, I should point out that it is also possible for an event source to have an invalid or nonexistent event message file. Your code should deal gracefully with situations like these. For example, the Event Viewer snap-in shows an improperly logged event, as shown earlier and in Figure 6-8.

Figure 6-8. *An event in Event Viewer where message file information could not be found*

As you can see, event reading is not a trivial problem. Because of the complexity of this task, you might find it helpful to adapt your event reading code from the sample code for this chapter.

Before I move on to the next section, I feel compelled to mention the *BackupEventLog*, *OpenBackupEventLog*, and *ClearEventLog* functions, not because they directly relate to event reading, but because many event-reading tools use these functions to offer additional functionality. These functions are prototyped as follows:

```
BOOL BackupEventLog(
   HANDLE hEventLog,
   PCTSTR pszBackupFileName);

HANDLE OpenBackupEventLog(
   PCTSTR pszUNCServerName,
   PCTSTR pszFileName);

BOOL ClearEventLog(
   HANDLE hEventLog,
   PCTSTR pszBackupFileName);
```

BackupEventLog creates a file using the provided filename and then copies the contents of event log (identified by the *hEventLog* parameter) to this new file. This function allows an administrator to save the history of events.

To open the event history and read its events, an application calls *OpenBackupEventLog*, which returns an event log handle (just like the *OpenEventLog* function discussed earlier). Using this handle, you can call the other familiar event log functions to retrieve the stored event entries. When you are finished retrieving the entries, you close the event log handle by calling *CloseEventLog*. Backing up an event log can be very helpful for archiving and later examining the event log, and it eases the customer's task of wrapping up a log and shipping it back to you for troubleshooting.

The last function, *ClearEventLog*, simply erases all the event entries from a log file opened by *OpenEventLog* or *OpenBackupEventLog*. For convenience, this function allows you to back up the log file before erasing the entries. You can pass NULL for the *pszBackupFileName* parameter to clear the log file without producing a backup file.

Event Notification

You can have the system notify you as events are added to an event log. For example, if your favorite chat server logs an event for every client connection as well as every client disconnection, you could write a utility to wait for notification of these events and maintain a running log of connections to your chat server. To receive event log notifications, you must call the *NotifyChangeEventLog* function:

```
BOOL NotifyChangeEventLog(
   HANDLE hEventLog,
   HANDLE hEvent);
```

The *hEventLog* parameter is the handle returned from a call to *OpenEventLog*, and the *hEvent* parameter is the handle of a previously created event kernel object. When the system detects a change to the event log, the system automatically signals the event kernel object. A thread in your application will detect the signaled event kernel object and then do whatever processing it desires to the event log.

You should know several facts about *NotifyChangeEventLog*. First, once you have associated an event object with an event log, there is no way to turn off notification to that event object short of calling *CloseEventLog*. This is typically not a problem, however, since you can simply choose to pass your event object handle to *CloseHandle* and create a new event kernel object if needed.

Second, the system signals the event by calling *PulseEvent* when a change is made to the event log. This means that you do not have to reset the event and that you must have a thread waiting on the event consistently; otherwise, you're likely to miss some notifications.

Third, the system does not promise *PulseEvent* for every event record added to the event log. Rather, it pulses your event roughly every 5 seconds if one or more changes were made to the event log during that 5-second period. So if you are waiting on the event object, and then read the log as a result of a pulse, you should not assume that only one event has been added, and you should read until you have reached the end of the log.

Last, the system pulses the event when any event record is added to any log file, regardless of which log you specified when calling *NotifyChangeEventLog*. So your thread might wake up when an event is added to the System log even though you wanted only notifications from the Application log. Your application must be able to gracefully handle receipt of a notification even when nothing has changed in the log.

The EventMonitor Sample Application

The EventMonitor sample application ("06 EventMonitor.exe") demonstrates how to read event records from an event log. In addition, the sample application calls the *NotifyChangeEventLog* function and updates its display as new entries are added to the event log. The source code and resource files for the application are in the 06-EventMonitor directory on the companion CD. When the user executes the EventMonitor sample application, the dialog box shown in Figure 6-9 appears.

By default, EventMonitor shows the contents of the local machine's Application event log. But you can easily select a different machine or log and then click the Monitor button to dump and monitor the newly selected machine's log. The sample only allows you to view the System, Security, and Application logs, but the source code can easily be modified so that the application monitors any custom logs you may create.

Figure 6-9. *The dialog box for the EventMonitor sample application*

When you start monitoring an event log, EventMonitor creates an entry in its list box for every entry in the selected log. Once the list box is full, EventMonitor calls *NotifyChangeEventLog* and has a thread that waits for new event log entries to appear. As new entries appear in the system's event log, EventMonitor retrieves the new entries and appends them to the list box as well. The last thing that EventMonitor demonstrates is how to convert category and message IDs into the proper string text. As you select entries in the list box, EventMonitor loads in the proper message file or files, grabs the appropriate string text, and displays the string text in the read-only edit box at the bottom of the dialog box.

[Previous] [Next]

Chapter 7

Performance Monitoring

Administrators, end users, and developers know that monitoring the health of the computer system is vitally important. Microsoft, well aware of this fact, built performance monitoring into Windows 2000. Unfortunately, very few applications take advantage of performance monitoring. Here are some of the reasons why:

- Microsoft has not made it easy to expose performance information.
- Developers must take the time to expose performance information in their applications.
- Microsoft has not emphasized or promoted the importance of performance monitoring.

I first looked into adding performance information to my own applications several years ago; flabbergasted by the complexity of the task, I postponed the job as long as I could. My eventual solution was to create a C++ class that encapsulates the process of exposing performance data to the operating system, allowing me to add performance data easily to any application. This C++ class appears at the end of the chapter.

[Previous] [Next]

Performance-Monitoring Perspectives

Before we learn about exposing performance information in our applications, I want to explain the basic performance-monitoring facilities that Windows offers. You can examine performance monitoring from several perspectives, and I'd like to discuss a few of them.

Let's begin by examining performance monitoring from a user perspective. I'll explain how the system organizes performance information and how administrators, users, and developers can use the System Monitor ActiveX control to gauge system health.

I'll then discuss some of the more common reasons developers incorporate performance monitoring into their applications (or more likely, into a Windows service). With this introductory material out of the way, I'll get more technical and discuss the Windows performance-monitoring architecture from a system and programmer perspective.

Performance Monitoring from a User Perspective

Included with Windows is an ActiveX control, named the System Monitor control, that allows administrators to view performance information. Since I'm amazed daily by the number of people who are unfamiliar with this control, I'll start out with a brief introduction.

To experiment with this ActiveX control, you can choose Performance from the Administrative Tools menu, or you can add the ActiveX control to a Microsoft Management Console (MMC) by doing the following:

1. Run the Microsoft Management Console application (MMC.exe).
2. From the Console menu, select Add/Remove Snap-In, and then click the Add button.
3. In the Add Standalone Snap-In dialog box, select ActiveX Control, and then click the Add button.
4. In the start screen of the Insert ActiveX Control wizard, click the Next button.
5. Select the System Monitor Control and click the Next button, followed by the Finish button.
6. In the Add Standalone Snap-In dialog box, click the Close button.
7. In the Add/Remove Snap-In dialog box, click OK.
8. Select the System Monitor Control node in the left-hand pane.

Figure 7-1 shows how the MMC window should look if you have done everything correctly.

Initially, the System Monitor control has no idea what performance information you would like to monitor, so its chart is empty. To add information to the chart, you must display the Add Counters dialog box (shown in Figure 7-2) by clicking the Add (+) button on the toolbar. As you can see, many options are available for the user to select. I'm going to explain these options now so that you'll understand how all this information fits together when I discuss performance counters later in this chapter.

Figure 7-1. *The System Monitor control in MMC*

Figure 7-2. *The Add Counters dialog box for the System Monitor control*

Your first decision is which computer to collect performance information from. One of the System Monitor control's best features is its ability to collect performance information from the local machine as well as a remote machine. In fact, information from both machines can be collected simultaneously and displayed in a single graph. This makes it very easy for an administrator to compare the performance of two or more computers.

Once you have chosen a computer, you then select a performance object. A performance object is a component in the system offering performance information. Out of the box, Windows exposes many objects, most of which are system related. Here are a few examples of system objects: Processor (the CPU itself), Process (the applications currently running), PhysicalDisk (the hard disks), System (the operating system itself), Thread (the threads running in processes), and Memory (RAM).

Bear in mind that performance monitoring is not limited to operating system components; device drivers are also able to expose their own performance objects. Telephony and TCP are some examples of device driver objects. All these objects are of particular use when configuring Windows on a computer.

Microsoft was not shortsighted in its design of the performance-monitoring system: the system also allows services and applications to expose their own performance objects. Some service examples include the Indexing Service and the Distributed Transaction Coordinator (DTC). How to expose performance objects from a service or application constitutes the bulk of this chapter.

The designer of a performance object also defines what counters that object supports. For example, in the Add Counters dialog box, the Process object offers a choice of several counters (visible in the Select Counters From List list box). Each entry in the list box indicates one item you can monitor about a process. For example, the % Processor Time counter will show you the percentage of time that threads within a process are actually running on a processor. The Handle Count counter shows how many kernel objects a process has open. The ID Process counter shows the system-wide unique ID that was assigned to a process when it was created. These counters represent just a sampling of the counters available for a Process object.

Having selected a performance object, you can now turn your attention to the Select Instances From List portion of the dialog box. An *instance* is a name given to an instance of an object type. For example, there are many instances of Process objects running in the system; each Process object instance is identified by its .exe file name. Most objects support instances, but some do not. For example, the System object shows no entries in the Select Instances From List portion of the dialog box because only one operating system is running on the computer.

Figure 7-3 shows the relationships between objects, instances, and counters. On the left, you see an object that supports instances; it might have zero or more instances currently associated with it. Each of these instances has the same number of counters, but the values of the counters will differ from instance to instance. Keep in mind that if an object that supports instances has no instances currently associated with it, no counter information can be obtained.

The object on the right does not support instances; it will always have one set of counters associated with it.

Figure 7-3. *The relationships between objects, instances, and counters*

As you navigate the Add Counters dialog box, you'll see that many objects are available, each with its own set of counters with cryptic names. If you click the Explain button in the dialog box, a separate window appears (shown in Figure 7-4) that offers a description of the currently selected counter.

Figure 7-4. *The Explain Text window, which provides additional information about a performance counter*

Windows actually allows explanatory text to be associated with a performance object itself in addition to its counters. However, at present the System Monitor control doesn't offer any way to display an object's Help text. I'm hoping that Microsoft will enhance the control in the future.

After selecting a computer, object, counter(s), and instance(s) (if applicable), click the Add button to have the System Monitor control start charting the specified information. The Add Counters dialog box will not close when you click the Add button, so you can easily add multiple counters to the chart. Keep in mind that when you are adding counter information to the chart, you can select multiple instances as well as multiple counters by using the Shift and Ctrl keys. The number of lines that will be charted is the product of the number of instances times the number of counters. Once you're finished adding counters, click the Close button.

By the way, you'll notice that some multi-instance objects support a pseudo-instance named _Total. This pseudo-instance is not actually an instance of the object, but a value that allows you to conveniently see the counter total for all instances of the object. For example, selecting the _Total instance for the Page Faults/sec counter of the Process object causes the chart to show the Page Faults/sec for all processes.

Be aware that displaying a total makes no sense for some counters. For example, if you were to chart the process ID for all process instances, you would see that the counter simply shows a value of 0, which never changes.

Performance Monitoring from a Designer Perspective

The reasons to consider adding performance counter information to your own applications and servers are manifold. First and foremost, you want to make it easy for administrators and help desk people to check on the health of a computer system. Further, the end users of most applications do not want to be bogged down by statistical information that is meaningless to them. It's far more expedient to use performance counters to expose information; that way, the people who can do something with the information can easily gain access to it. Also, Windows makes the performance information available across the network. As I mentioned in the previous section, a person using the System Monitor control can select which computer the information is to be collected from. Using Windows built-in performance monitoring to expose your application's performance information allows the information to be remotely accessible without any additional work from you.

One very important issue that turns a lot of people off to performance monitoring is, ironically, that of performance. Performance monitoring is not free. Somewhere inside your application, you need to allocate a block of memory to store current counter data, and your code must periodically update these values. Updating these values means that your code is going to be larger and will execute more slowly, something we always try to avoid. However, if performance monitoring affected the system drastically, no one would use it, and Microsoft certainly took this into account when designing performance monitoring into the system.

When designing performance information for your own applications, decide carefully what it is you want to monitor. If, for example, you have a tight calculation loop, you should avoid updating counters while inside this loop. On the other hand, you might be writing the server-side of a client/server application. In this case, you might want to have a performance counter that tracks how many clients have connected, how many

clients are currently connected, the number of bytes received per client, the number of bytes sent per client, what types of requests your clients are making, and so on. In fact, Microsoft Internet Information Services (IIS) keeps track of exactly this type of information.

Performance monitoring can be a powerful tool for software developers. I've used performance counters to monitor how much dynamic memory my applications allocate and free up. If I'm forgetting to free memory, the System Monitor control shows my memory usage line as always increasing. Developers and testers alike can use performance counters to easily monitor whether an application is behaving correctly.

To demonstrate performance monitoring, I wanted to present a sample program that created a Stock object. This object would have one instance in it for each stock symbol, and the counters would be information about the stock—for example, last price, high price, low price, and P/E ratio. Using the System Monitor control, you could select the stocks and counters that you're interested in, and the application would connect over the Internet to gather this information and make it available using the System Monitor control's charting. Unfortunately, I could not find an Internet site that would allow me to legally produce a derivative work from its stock ticker information. So, although I'm presenting a different sample application in this chapter, the Stock object concept should give you a good idea of the possibilities inherent in Windows performance monitoring.

[Previous] [Next]

The Architecture of Performance Objects and Counters

To expose your own performance information, you must make several modifications to the registry. You can think of these modifications as falling into two groups. The first set of modifications tells the system that your counters exist; the second set of modifications tells the system how to get the performance information for your counters. As shown in Figure 7-5, the initial modifications are performed under the following registry subkey:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib
```

Figure 7-5. *Settings in the registry that indicate your counters' existence*

Under the Perflib subkey is another subkey, 009, which corresponds with the LANG_ENGLISH symbol defined in WinNT.h. Under the 009 subkey are two values: Counter and Help. Both of these values are of the REG_MULTI_SZ type. You can examine or edit these two registry values with the RegEdt32.exe registry editor utility that comes with Windows. Unfortunately, you cannot use the RegEdit.exe tool (which also ships with Windows) because this tool has an internal limitation that prevents it from displaying or editing values larger than 32 KB.

The Counter value contains a set of strings defining the objects and counters that the system is to be made aware of. The following list shows a small sampling of the strings in this value:

```
2
System
4
Memory
6
% Processor Time
10
File Read Operations/sec
12
File Write Operations/sec
14
File Control Operations/sec
16
File Read Bytes/sec
18
File Write Bytes/sec
```

The Help value contains a set of strings that explain the meaning of a performance object or counter. Here is a small sampling of the Help value:

```
3
The System performance object consists of counters that apply to more
than one instance of a component processors on the computer.1
5
The Memory performance object consists of counters that describe the
behavior of physical and virtual memory on the computer. Physical memory
is the amount of random access memory on the computer. Virtual memory
consists of the space in physical memory and on disk. Many of the memory
counters monitor paging, which is the movement of pages of code and data
between disk and physical memory. Excessive paging, a symptom of a memory
shortage, can cause delays which interfere with all system processes.
7
% Processor Time is the percentage of time that the processor is
executing a non-Idle thread. This counter was designed as a primary
indicator of processor activity. It is calculated by measuring the time
that the processor spends executing the thread of the Idle process in
each sample interval, and subtracting that value from 100%. (Each
processor has an Idle thread which consumes cycles when no other threads
are ready to run). It can be viewed as the percentage of the sample
interval spent doing useful work. This counter displays the average
percentage of busy time observed during the sample interval. It is
calculated by monitoring the time the service was inactive, and then
subtracting that value from 100%.
9
% Total DPC Time is the average percentage of time that all processors
spent receiving and servicing deferred procedure calls (DPCs). (DPCs are
interrupts that run at a lower priority than the standard interrupts). It
is the sum of Processor: % DPC Time for all processors on the computer,
divided by the number of processors. System: % Total DPC Time is a
component of System: % Total Privileged Time because DPCs are executed in
privileged mode. DPCs are counted separately and are not a component of
the interrupt count. This counter displays the average busy time as a
percentage of the sample time.
11
File Read Operations/sec is the combined rate of file system read
requests to all devices on the computer, including requests to read from
the file system cache. It is measured in numbers of reads. This counter
displays the difference between the values observed in the last two
samples, divided by the duration of the sample interval.
13
File Write Operations/sec is the combined rate of the file system write
requests to all devices on the computer, including requests to write to
```

```
data in the file system cache. It is measured in numbers of writes. This
counter displays the difference between the values observed in the last
two samples, divided by the duration of the sample interval.
15
File Control Operations/sec is the combined rate of file system
operations that are neither reads nor writes, such as file system control
requests and requests for information about device characteristics or
status. This is the inverse of System: File Data Operations/sec and is
measured in number of operations perf second. This counter displays the
difference between the values observed in the last two samples, divided
by the duration of the sample interval.
17
File Read Bytes/sec is the overall rate at which bytes are read to
satisfy file system read requests to all devices on the computer,
including reads from the file system cache. It is measured in number of
bytes per second. This counter displays the difference between the values
observed in the last two samples, divided by the duration of the sample
interval.
19
File Write Bytes/sec is the overall rate at which bytes are written to
satisfy file system write requests to all devices on the computer,
including writes to the file system cache. It is measured in number of
bytes per second. This counter displays the difference between the values
observed in the last two samples, divided by the duration of the sample
interval.
```

1. Unfortunately, this Help text doesn't make any sense. I pointed out this "bug" to Microsoft, but they decided not to correct it because Windows 2000 was so close to shipping.

You'll notice that the Counter value strings are in pairs. Each pair consists of an even number and a short string. The Help value strings are also in pairs, but the numbers are odd. The system uses the Counter value to determine which performance objects and counters are available on the system. The Help value identifies each object's or counter's explanation text; the Help text's numerical value is always one greater than the object's or counter's numerical value. As I mentioned earlier, the System Monitor control currently has no way of showing the Help text for performance objects. This means that you'll never see the Help text associated with the numbers 3 and 5, for example.

To add your own performance objects and counters to the system, you must append your objects, counters, and Help string pairs to these two registry values. Referring back to Figure 7-5, two of the values you see under the Perflib subkey are Last Counter and Last Help. These two values tell you the highest number used in the Counter and Help values, respectively. Since, in my registry, Last Counter has a value of 2276, any new objects or counters added to the system start at 2278. Similarly, any new Help text would start at 2279. Microsoft reserves a certain set of numbers for the system itself. The Base Index value (1847) indicates the highest number that Microsoft has reserved for the system's own objects and counters. Objects and counters that you add must be above this number.

Now we'll turn our attention to the other part of the registry. To expose your own performance objects, instances, and counters, you must create a DLL that is responsible for returning your performance information. Once the DLL is created, you must tell the system about it by making some more modifications to the registry. As shown in Figure 7-6, however, these modifications are made in a different part of the registry:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\HWInputMon\
Performance
```

The HWInputMon portion of this subkey uniquely identifies my performance data. You will, of course, replace this portion with a name that uniquely identifies your performance data.

Figure 7-6. *Settings in the registry that indicate how the system can query performance information from a DLL*

Within this subkey are several data values. The most important value, Library, specifies the pathname of the DLL that knows how to return your performance information. This DLL must export three functions: an open function, a collect function, and a close function. You can choose any names for these functions that you desire, but the names must be specified using the Open, Collect, and Close values in the registry.

When performance information is being requested, the system will load your DLL and immediately call its *Open* function. Your *Open* function should look like this:

```
DWORD __declspec(dllexport) WINAPI Open(PWSTR) {
   // Note: The parameter is always ignored
   // Initialize the DLL
   return(ERROR_SUCCESS);
}
```

This gives your DLL the opportunity to initialize itself. Once initialized, the system will periodically make calls to your *Collect* function. This function is responsible for initializing a memory buffer that contains all the performance information you want to return. I'll discuss the prototype of the *Collect* function later in "Collecting Performance Data," and provide details about its implementation.

When the system decides to unload your DLL, it calls the *Close* function, giving you the opportunity to perform any necessary cleanup. The *Close* function should look like this:

```
DWORD __declspec(dllexport) WINAPI Close() {
   // Clean up the DLL
   return(ERROR_SUCCESS);
}
```

The four registry values discussed so far are the only ones required by the system. However, it is frequently useful to have some additional registry values, as shown in Figure 7-6. As mentioned earlier, when you add new performance objects and counters to the system, you must append your performance object and counter strings to the registry. These new strings have to begin with unique numbers. It is absolutely essential that your performance DLL remember what object and counter numbers it has been assigned. The easiest place to store this information is in the Performance registry subkey. Figure 7-6 shows that I have added the First Counter, Last Counter, First Help, and Last Help values to the registry for just this purpose. When my DLL loads, I open the registry, extract these values, and save them for reference by my *Collect* function. It is necessary to know your object and counter numbers because when the system calls the *Collect* function, it passes these numbers to identify which performance information should be returned.

Now you're probably wondering into which process your DLL gets loaded. There are two scenarios, depending on whether your performance information is being queried locally or remotely. When the System Monitor control is used to query performance information from the local machine, the system loads your DLL into the address space of the process that created the System Monitor control. (See Figure 7-7.) Because the DLL is running in another process's address space, it is critically important that you thoroughly test your code

and make sure that it is very robust. If your code contains any infinite loops, calls *ExitProcess*, or deadlocks waiting for some thread synchronization object, you'll be adversely affecting the process.

Figure 7-7. *Local performance monitoring*

Microsoft designed a robust System Monitor control to withstand a lot of "passive-aggressive" behavior. For example, the System Monitor control spawns a separate thread when calling into a DLL's *Collect* function; if that thread doesn't return in a fixed amount of time, the System Monitor control kills it and continues running. The System Monitor control also wraps the call to your DLL functions inside a structured exception-handling frame to catch any access violations, divisions by 0, or other hard errors that your code might generate.

A slightly different scenario exists when your performance information is being queried by a remote computer, as shown in Figure 7-8. In this scenario, the remote computer is actually talking to an RPC server contained inside WinLogon.exe. When WinLogon detects a request for performance information, it loads your DLL into WinLogon's address space. The information that you return from the *Collect* function is then marshaled to the machine making the request. For the System Monitor control, this remote communication occurs transparently.

Figure 7-8. *Remote performance monitoring*

Be aware that the ramifications of loading a performance DLL into WinLogon's address space are much more critical than loading a DLL into an application's address space. If the DLL calls the *ExitProcess* function now, WinLogon will terminate, which will, in turn, crash the entire operating system. Again, I cannot stress enough

how important it is to test the code residing in your performance DLL.

[Previous] [Next]

Collecting Performance Data

The System Monitor control requests performance information by making registry calls. To collect performance information, the requesting application must first call *RegQueryValueEx* as follows:

```
// Special registry key used to query performance data
HKEY hkeyPerf = HKEY_PERFORMANCE_DATA;

// Size of buffer to get performance data
DWORD cb = 10240;
// Allocate buffer to get performance data
PBYTE pbPerfData = (PBYTE) malloc(cb);

// Request all ("Global") performance data offered on the
// local machine
while (RegQueryValueEx(hkeyPerf, TEXT("Global"), 0, NULL,
   pbPerfData, &cb) == ERROR_MORE_DATA) {

   // Buffer was too small; make it bigger and try again
   cb += 1024;
   pbPerfData = (PBYTE) realloc(pbPerfData, cb);
}

// Buffer was big enough
// pbPerfData contains the performance counter information
```

NOTE

For the sake of clarity, the preceding code does not conduct any error checking. Please add the proper error checking when adding code like this into your own application.

To repeatedly request performance information, just call *RegQueryValueEx* again. The "Global" in the call to *RegQueryValueEx* tells the system that you want performance information returned for all components in the system. This can be a lot of information; if you want a subset of the available information, simply replace "Global" with a set of object numbers (separated by spaces). For example, if you want the performance information for the System and Memory objects, you would pass a string of "2 4" to *RegQueryValuEx*. You always pass object numbers to *RegQueryValueEx*, and the corresponding DLL always returns all the counter and instance information associated with the specified object.

If an application wants to request performance information from a remote computer, all it has to do is call *RegConnectRegistry* before entering the *while* loop shown in the preceding code:

```
RegConnectRegistry(TEXT("\\\\RemoteMachineName"),
   HKEY_PERFORMANCE_DATA, &hkeyPerf);
```

NOTE

By default, Administrators and the System have full access to a system's performance counter information, while Interactive users are allowed only read access. The permissions applied to the following registry subkey determine who can access the performance counter information:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft Windows NT\CurrentVersion\Perflib
```

You can use the RegEdt32.exe tool to alter the permissions.

If you're not going to make any more requests of performance information, you should close the registry by calling

```
RegCloseKey (hkeyPerf) ;
```

Notice that you should make this call even though you never explicitly call *RegOpenKeyEx* to open this key.

NOTE

Do not close the key after every call to *RegQueryValueEx*—this action will severely undermine your application's performance. When a component such as the System Monitor control calls *RegQueryValueEx* to request performance information, the system looks in the target machine's HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services subkey and loads the specified DLL for each entry that contains a Performance subkey. If this is the first time the DLL is loaded, the DLL's *Open* function is called. When *RegCloseKey* is called, the system calls each DLL's *Close* function and unloads the DLL from the process's address space. Calling *RegCloseKey* only when your application is terminating greatly improves performance.

Once loaded and initialized, the system calls each DLL's *Collect* function. This gives the DLL the chance to load the memory buffer with its performance information. The *Collect* function should appear as follows:

```
DWORD __declspec(dllexport) WINAPI Collect(PWSTR pszValueName,
   PVOID* ppvData, PDWORD pcbTotalBytes, PDWORD pdwNumObjectTypes) {

   // Collect the performance data
   return(ERROR_SUCCESS);   // or ERROR_MORE_DATA if buffer too small
}
```

The first parameter, *pszValueName*, is a Unicode string that is the same value that was passed to *RegQueryValueEx*. If this string is "Global", the *Collect* function must return all the performance information that it is responsible for. If the string contains a set of space-separated numbers, the DLL must determine whether it offers information about the requested objects and, if so, return only this information.

The *ppvData* parameter is a pointer to a memory address. On input to the *Collect* function, **ppvData* points to the memory buffer where the DLL should write its information. Before returning from *Collect*, **ppvData* should be updated so that it points to the memory address immediately following the new data that was placed into the buffer. The next performance DLL will append its data starting at this new address.

The *pcbTotalBytes* parameter is a pointer to a DWORD that indicates the size of the buffer. On input to the *Collect* function, **pcbTotalBytes* indicates the number of bytes that are available in the buffer for your DLL to add its information. After the DLL appends its data to the buffer, *Collect* must set **pcbTotalBytes* to the total number of bytes added to the buffer. When the *Collect* function returns, the system will subtract this value from the remaining buffer size and pass this new number to the next performance DLL.

The *pdwNumObjectTypes* parameter also points to a DWORD, but this DWORD means nothing on input to the *Collect* function. A single *Collect* function can return performance information for several objects. Before *Collect* returns, **pdwNumObjectTypes* should be set to the number of objects whose performance information was added to the data buffer.

If, for example, the *Collect* function is called and the *pszValueName* parameter doesn't indicate any of the objects the DLL is responsible for, the *Collect* function should leave **ppvData* unchanged, set **pcbTotalBytes* and **pdwNumObjectTypes* to 0, and return ERROR_SUCCESS.

If the *Collect* function determines that the data buffer is too small for the amount of data you need to return, you should leave **ppvData* unchanged, set both **pcbTotalBytes* and **pdwNumObjectTypes* to 0, and return ERROR_MORE_DATA.

If the *Collect* function does successfully append information to the buffer, it adds the number of bytes appended to **ppvData*, sets **pcbTotalBytes* to the number of bytes appended, sets **pdwNumObjectTypes* to the number of objects added, and returns ERROR_SUCCESS.

[Previous] [Next]

Performance Information Data Structures

At this point, the only item we have not covered is the format of the performance information that the *Collect* function must place into the caller's data block. Unfortunately, the data block contains a set of different data structures, some of which are variable in length, which makes building the contents of this data buffer difficult at best. Figure 7-9 shows how the performance information is laid out for an object.

Although this particular object doesn't support instances, it does support two counters. This means that three string entries exist in the registry for this object: one string identifies the object name and was assigned a value of 2938; two other strings identify the counter names and were assigned the values 2940 and 2942. The Help strings for these three items are 2939, 2941, and 2943, respectively.

Figure 7-9. *Data buffer describing a performance object that does not support instances*

To report performance information for this object, the data block passed to our *Collect* function must first have a PERF_OBJECT_TYPE structure placed into it. Since the Platform SDK documentation provides details of this structure's members, I'll describe them just briefly in Table 7-1.

Table 7-1. *Members of the PERF_OBJECT_TYPE structure*

Member	Description
TotalByteLength *DefinitionLength* *HeaderLength*	Indicate byte sizes. These values must be initialized correctly so that a tool such as the System Monitor control can walk the data block properly.

ObjectNameTitleIndex *ObjectHelpTitleIndex*	Indicate the numbers that were assigned to the object text and its Help when they were added to the registry.
ObjectNameTitle *ObjectHelpTitle*	Should always be set to NULL because these members are used only by the application requesting the data.
DetailLevel	Indicates how "hard to understand" this object is to most users. Most objects are understandable to novice users. But you can say that your object is best understood by novice, advanced, expert, or wizard users.
NumCounters	Indicates how many counters the object offers. In Figure 7-9, the object offers two counters and therefore has two PERF_COUNTER_DEFINITION structures immediately following this PERF_OBJECT_TYPE structure. (I'll discuss the PERF_COUNTER_DEFINITION structures shortly.)
DefaultCounter	When an object is selected, this member indicates which counter should be selected by default in the Counters list box. The System Monitor control's Add Counter dialog box uses this member.
NumInstances	Indicates how many instances the object currently has. In Figure 7-9, the object doesn't support instances, so PERF_NO_INSTANCES (-1) is returned.
CodePage	If the object supports instances, each instance is returned as a string name. It's best to return the instance name as a Unicode string, so set this member to 0. However, if you prefer to return non-Unicode strings, set this member to the CodePage used for the instance string name.
PerfTime *PerfFreq*	Should always be set to 0 because these members are only used by the application requesting the data.

Immediately following the PERF_OBJECT_TYPE structure is one or more PERF_COUNTER_DEFINITION structures, one structure for each counter offered by the object. Table 7-2 offers a brief description of this structure's members.

Table 7-2. *Members of the PERF_COUNTER_DEFINITION structure*

Member	Description
ByteLength	Length in bytes of this structure. Used for walking the memory block.
CounterNameTitleIndex *CounterHelpTitleIndex*	Indicate the numbers that were assigned to the counter text and its Help when they were added to the registry.
CounterNameTitle *CounterHelpTitle*	Should always be set to NULL because these members are used only by the application requesting the data.
DefaultScale	Power of 10 by which to scale chart line, assuming vertical axis is 100 (0 specifies a scale value of 1, 1 specifies a scale value of 10, -1 specifies a scale value of 1/10, and so on).
DetailLevel	Indicates how "hard to understand" this counter is to most users. Most counters are understandable to novice users. But you can say that your counter is best understood by novice, advanced, expert, or wizard users.
CounterType	Type of counter. (See the WinPerf.h header file for a description.)
CounterSize	Number of bytes used for counter information (usually 4 or 8).
CounterOffset	Offset from the start of the PERF_COUNTER_BLOCK to the first byte of

this counter.

Each PERF_COUNTER_DEFINITION structure defines characteristics of a single counter, but the actual value of the counter is not returned inside one of these structures. Following the PERF_OBJECT_TYPE structure and all the PERF_COUNTER_DEFINITION structures comes a single PERF_COUNTER_BLOCK structure. PERF_COUNTER_BLOCK is a variablelength structure. The first member, *ByteLength*, indicates how many bytes are in the variable-length structure. (This number must include the 4 bytes for the *ByteLength* member itself.) Immediately following the *ByteLength* member come the counter values. It is critically important that each new value is aligned on a 32-bit boundary, or data misalignment exceptions might occur.

Inside the PERF_COUNTER_DEFINITION structure, the *CounterSize* member indicates how many bytes are used by the counter value, and the *CounterOffset* member indicates the offset of where the counter value is inside the PERF_COUNTER_BLOCK structure.

Figure 7-10 shows how the performance information is laid out for an object that supports instances. This particular object offers one counter for each instance and currently has two instances defined. Because just one counter is defined, only two string entries exist in the registry for this object: one string identifies the object name and is assigned a value of 2944; the other string identifies the counter name and is assigned a value of 2946. The Help strings for the object and counter are 2945 and 2947, respectively.

Figure 7-10. *Data buffer describing a performance object with two instances. All counter values for all instances are in the buffer.*

The beginning of this memory block is laid out just like the previous example: it begins with a PERF_OBJECT_TYPE structure followed by one PERF_COUNTER_DEFINITION for every counter offered by the object. However, this is where the similarity ends. Since this object supports instances, the PERF_COUNTER_DEFINITION structures are followed by one PERF_INSTANCE_DEFINITION structure for each instance currently existing for the object. Table 7-3 gives a brief description of the members of the PERF_INSTANCE_DEFINITION structure.

Table 7-3. *Members of the PERF_INSTANCE_DEFINITION structure*

Member	Description
ByteLength	Length in bytes of this structure. Used for walking the memory block.

ParentObjectTitleIndex	An object instance can be the child of another object or instance; for example, threads are children of a process. If the instance has a parent, this number identifies the parent object's assigned registry number (0 = no parent).
ParentObjectInstance	If this instance has a parent, this number identifies the parent instance's index relative to the parent object.
UniqueID	An instance can be identified by a number or by a string name. If you want to use a number, set this member to the number and set *NameOffset* and *NameLength* to 0. If you want to use a string name, set this member to -1 and set the *NameOffset* and *NameLength* members appropriately.
NameOffset	Number of bytes from the beginning of the structure to this instance's Unicode string name.
NameLength	Number of bytes (not characters) of this instance's Unicode string name. (Add 2 bytes for the terminating zero character.)

Immediately following the fixed-length portion of the PERF_INSTANCE_DEFINITION structure is the variable-length portion. This portion contains the Unicode string name for the instance, terminated with a zero character. If the name contains an odd number of characters (including the zero character), you must add another 2 bytes of padding so that the next PERF_INSTANCE_DEFINITION or PERF_COUNTER_BLOCK structure begins on a 32-bit boundary. If the instance name is an even number of characters, no padding is required.

[Previous] [Next]

Debugging Your Performance Counter DLL

As you can see, the data block produced by your DLL's *Collect* function is complex, and it usually takes most developers quite some time to completely debug it. To help you out, Microsoft has added some support in the operating system to analyze *Collect* function behavior. If the system detects any problem, it will generate an entry into the system's Application event log. Figure 7-11 provides an example.

Figure 7-11. *A performance DLL error reported to the Application Log*

By default, the system does not perform this additional checking because it adversely affects the speed of retrieving performance counter data. To turn on this checking, go to the subkey in the registry that is listed here.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib
```

Under this key, add a REG_DWORD value named ExtCounterTestLevel and set it to one of the values listed in Table 7-4.

Table 7-4. *ExtCounterTestLevel values that specify levels of performance data testing*

Value	Description
1	The system should check all object and counter lengths for consistency.
2	The system should check for buffer overflows and guard pages.
3	The system should not perform any checking.
4	Don't even allocate a guard page. This is the default if this registry value does not exist.

Whenever the system detects a problem with performance data, it writes an entry to the system's Application Log. You can control the nature of this entry: under the same registry subkey, create a REG_DWORD value named EventLogLevel and set it to one of the values listed in Table 7-5.

Table 7-5. *EventLogLevel values that specify what events should be recorded in the event log*

Value	Description	Event ID Range

0	The system should never report anything to the event log.	(not applicable)
1	The system should report user-related events in the event log.	1000-1015, 1017-1020
2	The system should report warnings and errors in the event log.	1000-1015, 1017-1020, 1016, 2000-2003
3	The system should report information events as well as warnings and errors in the event log.	1000-1015, 1017-1020, 1016, 2000-2002, 3000-3001

Table 7-6 shows the types of events that can be placed in the event log.

Table 7-6. *Events that indicate performance data problems*

Event ID	Description
1000	Access to performance data was denied to <username> as attempted from <calling module name>
1001	The buffer size returned by a collect procedure in Extensible Counter DLL "<dll name>" for the "<service name>" service was larger than the space available. Performance data returned by counter DLL will not be returned in Perf Data Block. Overflow size is data DWORD 0.
1002	A Guard Page was modified by a collect procedure in Extensible Counter DLL "<dll name>" for the "<service name>" service. Performance data returned by counter DLL will not be returned in Perf Data Block.
1003	The object length of an object returned by Extensible Counter DLL "<dll name>" for the "<service name>" service was not correct. The sum of the object lengths returned did not match the size of the buffer returned. Performance data returned by counter DLL will not be returned in Perf Data Block. Count of objects returned is data DWORD 0.
1004	The instance length of an object returned by Extensible Counter DLL "<dll name>" for the "<service name>" service was incorrect. The sum of the instance lengths plus the object definition structures did not match the size of the object. Performance data returned by counter DLL will not be returned in Perf Data Block. The object title index of the bad object is data DWORD 0.
1005	Unable to locate the open procedure "<open proc name>" in DLL "<dll name>" for the "<service name>" service. Performance data for this service will not be available. Error Status is data DWORD 0.
1006	Unable to locate the collect procedure "<collect proc name>" in DLL "<dll name>" for the "<service name>" service. Performance data for this service will not be available. Error Status is data DWORD 0.
1007	Unable to locate the close procedure "<close proc name>" in DLL "<dll name>" for the "<service name>" service. Performance data for this service will not be available. Error Status is data DWORD 0.

1008	The Open Procedure for service "<service name>" in DLL "<dll name>" failed. Performance data for this service will not be available. Status code returned is DWORD 0.
1009	The Open Procedure for service "<service name>" in DLL "<dll name>" generated an exception. Performance data for this service will not be available. Exception code returned is DWORD 0.
1010	The Collect Procedure for the "<service name>" service in DLL "<dll name>" generated an exception or returned an invalid status. Performance data returned by counter DLL will not be returned in Perf Data Block. Exception or status code returned is DWORD 0.
1011	The library file "<dll name>" specified for the "<service name>" service could not be opened. Performance data for this service will not be available. Status code is data DWORD 0.
1012	The system reported an idle process time that was less than the last time reported. The data shows the current time and the last reported time for the system's idle process.
1013	The collect procedure in Extensible Counter DLL "<dll name>" for the "<service name>" service returned a buffer that was larger than the space allocated and may have corrupted the application's heap. This DLL should be disabled or removed from the system until the problem has been corrected to prevent further corruption. The application accessing this performance data should be restarted. The Performance data returned by counter DLL will not be returned in Perf Data Block. Overflow size is data DWORD 0.
1014	An error occurred while trying to collect data from the Server Object. The Error code returned by the function is DWORD 0. The Status returned in the IO Status Block is DWORD 1. The Information field of the IO Status Block is DWORD 2.
1015	The timeout waiting for the performance data collection function "<function name>" in the "<dll name>" Library to finish has expired. There may be a problem with this extensible counter or the service it is collecting data from or the system may have been very busy when this call was attempted.
1016	The data buffer created for the "<service name>" service in the "<dll name>" library is not aligned on an 8-byte boundary. This may cause problems for applications that are trying to read the performance data buffer. Contact the manufacturer of this library or service to have this problem corrected or to get a newer version of this library.
1017	Performance counter data collection from the "<service name>" service has been disabled due to one or more errors generated by the performance counter library for that service. The error(s) that forced this action have been written to the application event log. The error(s) should be corrected before the performance counters for this service are enabled again.
1018	Performance counter data collection from the "<service name>" service has been disabled for this session due to one or more errors generated by the performance counter library for that service. The error(s) that forced this action have been written to the application event log.
1019	A definition field in an object returned by Extensible Counter DLL "<dll name>" for the "<service name>" service was incorrect. The sum of the definitions block lengths in the object definition structures did not match the size specified in the object definition header. Performance data returned by this counter DLL will be not be returned in Perf Data Block. The object title index of the bad object is data DWORD 0.

1020	The size of the buffer used is greater than that passed to the collect function of the "<dll name>" Extensible Counter DLL for the "<service name>" service. The size of the buffer passed in is data DWORD 0 and the size returned is data DWORD 1.
2000	The pointer returned did not match the buffer length returned by the Collect procedure for the "<service name>" service in Extensible Counter DLL "<dll name>". The Length will be adjusted to match and the performance data will appear in the Perf Data Block. The returned length is data DWORD 0, the new length is data DWORD 1.
2001	The "<service name>" service does not have a Performance subkey or the key could not be opened. No performance counters will be collected for this service. The Win32 error code is returned in the data.
2002	The open procedure for service "<service name>" in DLL "<dll name>" has taken longer than the established wait time to complete. There may be a problem with this extensible counter or the service it is collecting data from or the system may have been very busy when this call was attempted.
2003	The configuration information of the performance library "<dll name>" for the "<service name>" service does not match the trusted performance library information stored in the registry. The functions in this library will not be treated as trusted.
3000	Open procedure for service "<service name>" in DLL "<dll name>" was called and returned successfully.
3001	An updated performance counter library file has been detected. The performance counters for the "<dll name>" have been enabled.

You'll notice that some of the events listed in Table 7-6 have to do with timing. For example, if the DLL's *Open* function takes too long to return, the system assumes that something is wrong with the performance-counter DLL and generates an event with ID 2002. Set the OpenProcedureWaitTime value in the registry to let the system know how long it should wait before reporting this event. This value is listed under the same registry subkey as the other two values (ExtCounterTestLevel and EventLogLevel). Furthermore, this value is also a REG_DWORD value that you set to the number of milliseconds that will be good enough for all *Open* functions.

NOTE

This value affects only how long the system should wait to generate the event log entry. Setting a timeout does not force the system to abandon the *Open* function and do something else. If the *Open* function does not return, the application requesting the performance data stops executing.

Whereas a single registry value is used for all *Open* functions, each performance-counter DLL can set a unique *Collect* function timeout value. To set a time limit (in milliseconds) for your DLL's *Collect* function, create a REG_DWORD registry value named Collect Timeout. (The space between the words is mandatory.) This value must be placed in the following subkey:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\ServiceName\Performance
```

Again, the *ServiceName* portion of this subkey is the part that uniquely identifies your service or application performance data. If your DLL's *Collect* function doesn't return in the specified number of milliseconds, an event ID 1015 is added to the system's Application Log.

[Previous] [Next]

The HWInputMon Sample Application

The HWInputMon sample application ("07 HWInputMon.exe"), shown in Listing 7-1 at the end of this section, exposes keystroke and mouse move performance information. It also includes a sample DLL ("07 HWInputMonPerfInfo.dll"), shown in Listing 7-2, that collects and returns the performance information. The source code for the sample application and DLL are in the 07-HWInputMon and the 07-HWInputMonPerfInfo directories on the companion CD.

HWInputMon creates a performance object named Hardware Input and exposes four counters: Keystrokes, Keystrokes/sec, Mouse moves, and Mouse moves/sec, as shown in Figure 7-12. I strongly suggest that you examine this sample application.

Figure 7-12. *The Add Counters dialog box for the HWInputMon sample application*

The CPerfData Class

To simplify adding performance objects and counters to HWInputMon, I've created a C++ class named CPerfData. The CPerfData file (PerfData.h) and its supporting files (Optex.h and RegKey.h) are available on the companion CD. CPerfData does all of the really tedious work, such as shared memory management, data structure initialization, memory block construction, addition and removal of object instances, and registry handling. If you use the CPerfData class for your own applications or services, the most difficult part of adding performance information to your code will be deciding which objects and counters to expose.

By referencing the commented source code, you should have no trouble understanding the gritty details of the CPerfData's member functions. However, in order to use this class to add performance counters to your application, you don't need to understand how it works internally; you only need to know how to set up the class.

Using the CPerfData Class

To create a service (or application) that exposes performance information, you will need two projects: a Win32 Application project that is an executable service or application and a Win32 DLL project that collects and returns the performance information.

Once you have these two projects set up, you create a header file that defines programmatic symbols for the specific objects and counters you want the application to expose. For the HWInputMon sample, this header file is named HWInputPerfDataMap.h and is shown in Listing 7-3.

You define an object's symbol by using the *PERFDATA_DEFINE_OBJECT* macro (defined in PerfData.h). This macro takes two parameters: a symbolic name that you can use in your application to refer to this object, and an ID that you also define. You must never repeat an ID number, and IDs must never be 0. You define a counter in exactly the same way, this time using the *PERFDATA_DEFINE_COUNTER* macro. Any source code modules that change or update a performance counter must include this header file.

Now you create a table indicating which objects and counters your service or application supports. For convenience, I also include this information in HWInputPerfDataMap.h. The table is easily created by using macros that are declared in the PerfData.h header file. These macros create what I call a performance data map. This map is similar to the message maps used by MFC programmers.

To create a performance data map, start with the *PERFDATA_MAP_BEGIN* macro, which instantiates an array of structures that defines your objects and counters. Follow this macro with one or more *PERFDATA_MAP_OBJ* or *PERFDATA_MAP_CTR* macros. Note that the order of the entries in the map is important. Declare the first object, followed by its counters, and then declare the next object, followed by its counters, and so forth.

To declare an object, use the *PERFDATA_MAP_OBJ* macro. This macro requires seven parameters:

1. Programmatic symbol that identifies the object.
2. Unicode string name for the object. (This will be added to the registry.)
3. Unicode Help text for the object. (This will also be added to the registry.)
4. Object's detail level.
5. Programmatic symbol name of a counter that should be selected by default when the object is selected in the System Monitor control.
6. Maximum number of instances that the object supports. (Pass PERF_NO_INSTANCES if the object doesn't support instances.)
7. Maximum number of characters that can appear in an instance's string name. (Pass 0 if the object doesn't support instances.)

After you've added an object to the map, use the *PERFDATA_MAP_CTR* macro to add one or more counters for the object. This macro requires six parameters:

1. Programmatic symbol that identifies the counter.
2. Unicode string name for the counter. (This name will be added to the registry.)
3. Unicode Help text for the counter. (This will also be added to the registry.)

4. Counter's detail level.

5. Default scale for the counter.

6. Type of counter.

After you've added all the objects and counters to the map, finish the map by using the *PERFDATA_MAP_END* macro. This macro takes just one parameter—the name of your service or application—and is used to create counter information in the following registry key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\ServiceName\
Performance
```

This macro terminates the map and creates a global instance of the CPerfData class. This global instance is named g_PerfData, and you'll need to refer to this variable in your code when you want to manipulate the performance objects, instances, or counters.

Notice that the performance data map is placed in a source code file all by itself. This is necessary because both the application and the DLL will have to include this file in their projects.

Let's turn our attention now to some code that demonstrates exactly how to use the CPerfData class's public member functions. Specifically, let's look at the *_tWinMain* function inside HWInputMon.cpp. (See Listing 7-1.)

In order for the application to expose performance information, the registry must be configured as described earlier in this chapter. CPerfData has a static member function named *Install*:

```
void CPerfData::Install(PCWSTR pszDllPathname);
```

This function must be passed the full pathname of the DLL that exposes your performance information. In the HWInputMon sample application, *_tWinMain* calls this function if the user clicks Yes in the Install Performance Counter Data Into Registry message box. To determine the DLL's full pathname, I get the full pathname of the executable file and replace the executable's filename with the DLL's filename—assuming, of course, that the executable and DLL files are both in the same directory.

To remove the performance counter information from the system's registry, call CPerfData's *Uninstall* function:

```
void CPerfData::Uninstall();
```

In *_tWinMain*, I call this function if the user clicks Yes in the Remove Performance Counter Data From The Registry message box. When you are debugging the counters, you might want to have your application install the registry information on startup and uninstall it during shutdown. That way, if you decide to add, delete, or move any of the entries in the performance data map, the registry won't get out of sync.

Once the registry is completely configured, call the CPerfData class's *Activate* method to tell the CPerfData object to start keeping track of the performance counter information:

```
DWORD CPerfData::Activate();
```

This function allocates the shared memory block and initializes it with the information contained in the performance data map. An application should call this function only if the performance information has already been installed.

NOTE

Internally, the CPerfData class implements this shared memory block by using a memory-mapped file kernel object. This memory-mapped file is going to be mapped into the service's address space and will also be mapped into the requesting application's address space (MMC.exe or WinLogon, for example). Because these two processes might very well be executing under different security contexts, steps must be taken so that they can communicate with each other using this kernel object.

So that the two processes can communicate, CPerfData's internal kernel objects are created using a security descriptor that allows Everyone GENERIC_ALL access. Depending on your needs, you might want to alter this descriptor, but I suspect that most developers will find it sufficient.

Once the performance data has been activated, *_tWinMain* adds instances to one of its objects. Instances are added by simply calling the *AddInstance* function:

```
INSTID CPerfData::AddInstance(BOOL fIgnoreIfExists, OBJID ObjId,
   PCTSTR pszInstName, OBJID ObjIdParent = 0, INSTID InstIdParent = 0);
```

The *fIgnoreIfExists* parameter tells the function whether or not to add this instance if an instance with the specified name already exists. The *ObjId* parameter is the programmatic symbol identifying the object that is to get the new instance. The *pszInstName* parameter is the string name of the instance. The last two parameters allow you to indicate whether this instance is the child of some other object's instance. Most instances do not have parent instances, so you will usually pass just the first three parameters to this function. If the function is successful, it returns an INSTID. This is my own data type and is simply a handle to the newly created instance. If the function fails, -1 is returned.

The CPerfData class has another version of *AddInstance*:

```
INSTID CPerfData::AddInstance(BOOL fIgnoreIfExists, OBJID ObjId,
   LONG lUniqueId, OBJID ObjIdParent = 0, INSTID InstIdParent = 0);
```

This version is identical to the first with the exception that it allows you to identify the instance by using a unique ID instead of a string.

Because instances can come and go as your application executes, you should feel free to add new instances at any time. You can also remove instances by calling *RemoveInstance*:

```
void CPerfData::RemoveInstance(OBJID ObjId, INSTID InstId);
```

Now we get to the fun stuff—changing a counter's value. Two functions exist that allow you to alter a counter's value:

```
LONG&    CPerfData::GetCtr32(CTRID CtrId, int nInstId = 0) const;
__int64& CPerfData::GetCtr64(CTRID CtrId, int nInstId = 0) const;
```

If the counter value you want to change is 32 bits wide, call *GetCtr32*; if the counter value is 64 bits wide, call *GetCtr64*. In debug builds, the source code will raise an assertion if you accidentally call a function that doesn't match the counter value's width. To both of these functions, you pass the programmatic symbol for a counter that you've defined. If this counter is inside an object that doesn't support instances, omit the second parameter. If this counter is contained in an object that does support instances, you must pass the INSTID (returned from *AddInstance*) as the second parameter.

These functions return either a LONG reference or an __int64 reference that identifies the counter's value inside the shared memory block. With this reference, changing the value of a counter is a trivial undertaking. Here is an example:

```
LONG& lCounterValue = g_PerfData.GetCtr32(SOME_COUNTER_SYMBOL);
```

```
lCounterValue = 5;   // Make the counter's value 5
lCounterValue++;     // Add 1 to the counter's value
lCounterValue *= 13; // Multiply the counter's current value by 13
```

What could be simpler! Just sprinkle lines like these through the application's source code whenever you want to change the value of a counter. These lines of code execute very quickly and should not significantly hurt the performance of your application.

Listing 7-1. *The HWInputMon sample application*

HWInputMon.cpp

```
/******************************************************************************
Module: HWInputMon.cpp Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
#include "..\CmnHdr.h" // See Appendix A. #include <WindowsX.h> #define
HWINPUTPERFDATAMAP_IMPL #include "HWInputPerfDataMap.h"
////////////////////////////////////////////////////////////////////////////// LRESULT CALLBACK
LowLevelKeyboardProc(int nCode, WPARAM wParam, LPARAM lParam) { if (nCode ==
HC_ACTION) { switch (wParam) { case WM_KEYDOWN: case WM_SYSKEYDOWN: case
WM_KEYUP: case WM_SYSKEYUP: g_PerfData.GetCtr32(HWINPUT_KEYS)++;
g_PerfData.GetCtr32(HWINPUT_KEYSPERSEC)++; break; } } return(CallNextHookEx(NULL,
nCode, wParam, lParam)); } ////////////////////////////////////////////////////////////////////////////// typedef enum {
mciFirst = 0, mciTotal = mciFirst, mciLeft, mciMiddle, mciRight, mciLast = mciRight }
MOUSECLCKINST; CPerfData::INSTID g_MouseClckInstToPrfInstId[mciLast + 1];
////////////////////////////////////////////////////////////////////////////// LRESULT CALLBACK
LowLevelMouseProc(int nCode, WPARAM wParam, LPARAM lParam) { if (nCode ==
HC_ACTION) { if (wParam == WM_MOUSEMOVE) {
g_PerfData.GetCtr32(HWINPUT_MOUSEMOVES)++;
g_PerfData.GetCtr32(HWINPUT_MOUSEMOVESPERSEC)++; } BOOL fDown = ((wParam ==
WM_LBUTTONDOWN) || (wParam == WM_MBUTTONDOWN) || (wParam ==
WM_RBUTTONDOWN)); if (fDown) { MOUSECLCKINST mci = mciLeft; if ((wParam ==
WM_LBUTTONDOWN) || (wParam == WM_LBUTTONUP)) mci = mciLeft; if ((wParam ==
WM_MBUTTONDOWN) || (wParam == WM_MBUTTONUP)) mci = mciMiddle; if ((wParam ==
WM_RBUTTONDOWN) || (wParam == WM_RBUTTONUP)) mci = mciRight;
g_PerfData.GetCtr32(MOUSECLCKS_CLICKS, g_MouseClckInstToPrfInstId[mciTotal])++;
g_PerfData.GetCtr32(MOUSECLCKS_CLICKSPERSEC,
g_MouseClckInstToPrfInstId[mciTotal])++; g_PerfData.GetCtr32(MOUSECLCKS_CLICKS,
g_MouseClckInstToPrfInstId[mci])++; g_PerfData.GetCtr32(MOUSECLCKS_CLICKSPERSEC,
g_MouseClckInstToPrfInstId[mci])++; } } return(CallNextHookEx(NULL, nCode, wParam, lParam));
} ////////////////////////////////////////////////////////////////////////////// int WINAPI _tWinMain(HINSTANCE
hinstExe, HINSTANCE, PTSTR pszCmdLine, int) { static TCHAR szAppName[] =
TEXT("Hardware Input Monitor"); if (MessageBox(NULL, TEXT("Install Performance Counter Data
into Registry?"), szAppName, MB_YESNO) == IDYES) { TCHAR szPath[_MAX_PATH];
GetModuleFileName(hinstExe, szPath, chDIMOF(szPath)); lstrcpy(_tcsrchr(szPath, TEXT(`\\')) + 1,
TEXT("07 HWInputMonPerfInfo.dll")); g_PerfData.Install(szPath); } if (MessageBox(NULL,
TEXT("Collect Performance Counter Data?"), szAppName, MB_YESNO) == IDYES) {
g_PerfData.Activate(); // Add the four Mouse Click Object Instances
g_MouseClckInstToPrfInstId[mciTotal] = g_PerfData.AddInstance(TRUE,
PERFOBJ_MOUSECLCKS, TEXT("_Total")); g_MouseClckInstToPrfInstId[mciLeft] =
g_PerfData.AddInstance(TRUE, PERFOBJ_MOUSECLCKS, TEXT("Left"));
g_MouseClckInstToPrfInstId[mciMiddle] = g_PerfData.AddInstance(TRUE,
PERFOBJ_MOUSECLCKS, TEXT("Middle")); g_MouseClckInstToPrfInstId[mciRight] =
g_PerfData.AddInstance(TRUE, PERFOBJ_MOUSECLCKS, TEXT("Right")); // Install the low-level
```

```
keyboard & mouse hooks HHOOK hhkLowLevelKybd =
SetWindowsHookEx(WH_KEYBOARD_LL, LowLevelKeyboardProc, hinstExe, 0); HHOOK
hhkLowLevelMouse = SetWindowsHookEx(WH_MOUSE_LL, LowLevelMouseProc, hinstExe, 0); //
Keep this app running until we're told to stop int x = IDRETRY; while (x == IDRETRY) { if (x ==
IDRETRY) { // Reset all of the counters to zero g_PerfData.LockCtrs();
g_PerfData.GetCtr32(HWINPUT_KEYS) = 0; g_PerfData.GetCtr32(HWINPUT_KEYSPERSEC) =
0; g_PerfData.GetCtr32(HWINPUT_MOUSEMOVES) = 0;
g_PerfData.GetCtr32(HWINPUT_MOUSEMOVESPERSEC) = 0; MOUSECLCKINST mci =
mciFirst; while (mci <= mciLast) { g_PerfData.GetCtr32(MOUSECLCKS_CLICKS,
g_MouseClckInstToPrfInstId[mci]) = 0; g_PerfData.GetCtr32(MOUSECLCKS_CLICKSPERSEC,
g_MouseClckInstToPrfInstId[mci]) = 0; mci = (MOUSECLCKINST) ((int) mci + 1); }
g_PerfData.UnlockCtrs(); } x = MessageBox(NULL, TEXT("Click \"Retry\" to reset the counters.\n")
TEXT("Click \"Cancel\" to terminate the application."), szAppName, MB_RETRYCANCEL); }
UnhookWindowsHookEx(hhkLowLevelKybd); UnhookWindowsHookEx(hhkLowLevelMouse); //
Remove the four Mouse Click Object Instances
g_PerfData.RemoveInstance(PERFOBJ_MOUSECLCKS, g_MouseClckInstToPrfInstId[mciTotal]);
g_PerfData.RemoveInstance(PERFOBJ_MOUSECLCKS, g_MouseClckInstToPrfInstId[mciLeft]);
g_PerfData.RemoveInstance(PERFOBJ_MOUSECLCKS, g_MouseClckInstToPrfInstId[mciMiddle]);
g_PerfData.RemoveInstance(PERFOBJ_MOUSECLCKS, g_MouseClckInstToPrfInstId[mciRight]); }
if (MessageBox(NULL, TEXT("Remove Performance Counter Data from the Registry?"),
szAppName, MB_YESNO) == IDYES) { g_PerfData.Uninstall(); } return(0); }
////////////////////////////////// End Of File //////////////////////////////////
```

Listing 7-2. *The HWInputMonPerfInfo DLL*

HWInputMonPerfInfo.cpp

```
/******************************************************************************
Module: HWInputMonPerfInfo.cpp Notices: Copyright (c) 2000 Jeffrey Richter Description: DLL that
exposes HWInputMon's Performance Information
******************************************************************************/
#include "..\CmnHdr.h" // See Appendix A. #define PERFDATA_COLLECT_SUPPORTED //
NOTE: PERFDATA_COLLECT_SUPPORTED must be defined for this project #if
!defined(PERFDATA_COLLECT_SUPPORTED) #error PERFDATA_COLLECT_SUPPORTED
must be defined for this project #endif #define HWINPUTPERFDATAMAP_IMPL #include
"..\07-HWInputMon\HWInputPerfDataMap.h" ////////////////////////////////// End Of File
//////////////////////////////////
```

Listing 7-3. *The HWInput data map*

HWInputPerfDataMap.h

```
/******************************************************************************
Module: HWInputPerfDataMap.h Notices: Copyright (c) 2000 Jeffrey Richter Description: Definition
of performance objects and counters
******************************************************************************/
#ifdef HWINPUTPERFDATAMAP_IMPL #define PERFDATA_IMPL #endif #include "PerfData.h"
//////////////////////////////////////////////////////////////////////////
PERFDATA_DEFINE_OBJECT(PERFOBJ_HWINPUT, 100);
PERFDATA_DEFINE_COUNTER(HWINPUT_KEYS, 101);
PERFDATA_DEFINE_COUNTER(HWINPUT_KEYSPERSEC, 102);
PERFDATA_DEFINE_COUNTER(HWINPUT_MOUSEMOVES, 103);
PERFDATA_DEFINE_COUNTER(HWINPUT_MOUSEMOVESPERSEC, 104);
```

```
PERFDATA_DEFINE_OBJECT(PERFOBJ_MOUSECLCKS, 200);
PERFDATA_DEFINE_COUNTER(MOUSECLCKS_CLICKS, 201);
PERFDATA_DEFINE_COUNTER(MOUSECLCKS_CLICKSPERSEC, 202);
////////////////////////////////////////////////////////////////////////////// #ifdef HWINPUTPERFDATAMAP_IMPL
////////////////////////////////////////////////////////////////////////////// PERFDATA_MAP_BEGIN()
PERFDATA_MAP_OBJ(PERFOBJ_HWINPUT, TEXT("Hardware Input"), TEXT("The Hardware
Input object type includes those counters ") TEXT("that apply to keystrokes and mouse moves."),
PERF_DETAIL_NOVICE, HWINPUT_KEYS, PERF_NO_INSTANCES, 0)
PERFDATA_MAP_CTR(HWINPUT_KEYS, TEXT("Keystrokes"), TEXT("The number of down and
up keystrokes"), PERF_DETAIL_NOVICE, 0, PERF_COUNTER_RAWCOUNT)
PERFDATA_MAP_CTR(HWINPUT_KEYSPERSEC, TEXT("Keystrokes/sec"), TEXT("The number
of down and up keystrokes per second"), PERF_DETAIL_NOVICE, 0,
PERF_COUNTER_COUNTER) PERFDATA_MAP_CTR(HWINPUT_MOUSEMOVES,
TEXT("Mouse moves"), TEXT("The number of mouse moves"), PERF_DETAIL_NOVICE, 0,
PERF_COUNTER_RAWCOUNT) PERFDATA_MAP_CTR(HWINPUT_MOUSEMOVESPERSEC,
TEXT("Mouse moves/sec"), TEXT("The number of mouse moves per second"),
PERF_DETAIL_NOVICE, 0, PERF_COUNTER_COUNTER)
PERFDATA_MAP_OBJ(PERFOBJ_MOUSECLCKS, TEXT("Mouse Clicks"), TEXT("The Mouse
Clicks object type includes those counters ") TEXT("that apply to mouse button clicks."),
PERF_DETAIL_NOVICE, MOUSECLCKS_CLICKS, 4, 10)
PERFDATA_MAP_CTR(MOUSECLCKS_CLICKS, TEXT("Clicks"), TEXT("The number of down
clicks"), PERF_DETAIL_NOVICE, 0, PERF_COUNTER_RAWCOUNT)
PERFDATA_MAP_CTR(MOUSECLCKS_CLICKSPERSEC, TEXT("Clicks/sec"), TEXT("The
number of down clicks per second"), PERF_DETAIL_NOVICE, 0, PERF_COUNTER_COUNTER)
PERFDATA_MAP_END("HWInputMon") ////////////////////////////////////////////////////////////////////////////
#endif // HWINPUTPERFDATAMAP_IMPL ////////////////////////////////// End Of File
//////////////////////////////////
```

[Previous] [Next]

Synchronizing Access to the Counter Values

Synchronizing access to the counter values is an issue that every programmer needs to take seriously. To implement counters "correctly," you should wrap each modification of a counter value inside a critical section or something similar. However, the CPU time required to enter and leave a critical section is usually much higher than the CPU time required to change a simple 32-bit or 64-bit value.

Because access synchronization could adversely affect your application's performance quite significantly, I spoke with a developer at Microsoft in charge of Windows performance. He told me that most of the system counters do not synchronize access to the counter value. This, of course, reduces the overhead of properly synchronizing the counter value but means that the value could potentially become corrupted. However, corruption of the value is such an unlikely scenario that the development team decided the speed benefit far outweighed the possibility of inaccurate information. Yes, the System Monitor control could show an incorrect value that would throw off the statistics, but this is a remote possibility.

From my own experience creating and testing performance objects and counters, I concur with the Windows team. That is, properly synchronizing access to a counter value to avoid the potential of an incorrect value is not worth the overhead. However, when designing my C++ class I felt it was necessary to allow you to make this choice for yourself, so my C++ class offers three public functions that allow you to lock and unlock the counters as you see fit:

```
void CPerfData::LockCtrs()    const;
BOOL CPerfData::TryLockCtrs() const;
```

```
void CPerfData::UnlockCtrs()  const;
```

Most applications will not use these functions; however, I do call these functions inside the implementation of the CPerfData class itself. For example, my implementation of the *Collect* function always locks and unlocks the counter information when called. I felt that this was necessary because the *Collect* function has a lot of work to do, and the additional overhead is insignificant compared to the other instructions that will execute when collecting the data.

In addition, I also lock the shared memory block when adding or removing object instances. This prevents the data structures from becoming corrupt and the code from crashing. The HWInputMon.cpp module also demonstrates the use of these functions from inside the *_tWinMain* function.

One important note: since threads running in different processes access the shared memory block, the easy way to synchronize these threads would be to use a mutex kernel object. However, waiting on a kernel object comes with a significant performance hit. A critical section would afford better performance, but, unfortunately, can be used only to synchronize threads that are running in a single process. Since I need very high-speed mutual-exclusive access to the shared memory buffer from threads in multiple processes, I decided to synchronize access to the shared memory buffer by using my COptex class as described in *Programming Applications for Microsoft Windows, Fourth Edition* (Jeffrey Richter, Microsoft Press, 1999).

Please examine the accompanying source code for more information about using the CPerfData class and to see how the workspace and its projects are configured.

[Previous] [Next]

Chapter 8

Windows Management Instrumentation

Windows Management Instrumentation (WMI) is a uniform way to manage a computer system. WMI is Microsoft's implementation of a standard known as Web-Based Enterprise Management (WBEM), which is support by the Distributed Management Task Force (DMTF).

WMI can provide you with capabilities to manage your services. With WMI, you can start a service, stop it, and change its various parameters such as start mode, start name, path, and service type. WMI also has comprehensive event handling features that allow you to provide simple scripts along with your service that will alert users if, for example, the service is not running when it should be. All of WMI's management features are remoteable, so given suitable authorization, you can access them from anywhere on the network.

[Previous] [Next]

WMI Architecture

WMI is an extensible data model of a computer system. The WMI architecture is illustrated in Figure 8-1. In this section, I discuss the components of the WMI architecture.

Windows Management Service

The Windows Management Service (WinMgmt.exe) is the primary component of WMI. WinMgmt brokers the communication between WMI consumers (management applications) and WMI providers. WMI data is

stored in an object-oriented schema. This schema is designed by the DMTF and offers a single data description mechanism for all components exposing WMI data. By offering a standard schema that supports inheritance, WMI data providers can offer standard data classes and properties while also allowing others to derive vendor-specific extensions to differentiate specific products. Much of this chapter is dedicated to showing exactly how this is done.

Figure 8-1. *WMI architecture*

CIM Object Manager

The Common Information Model (CIM) Object Manager provides client interfaces that support a variety of access techniques such as COM, scripting, XML, ODBC, and ADO. The object manager supports APIs for creating, deleting, modifying, and retrieving classes and instances. The class instances are either supplied by the CIM repository, in which case they are static instances, or by a provider of some sort (implemented as a COM server), in which case they are dynamic instances.

CIM Repository

The CIM repository is typically used only for storing information about the classes supported by WMI. It is essentially a symbol table and should not be used for large volumes of data. The CIM repository is populated either through object manager APIs that allow the declaration of classes and instances or through a textual form known as MOF (Managed Object Format).

WMI Providers

Providers are COM components that complete the connection between the CIM Object Manager and the managed object. Providers can be built-in or application-specific.

Providers come in various forms, the most common of which is the instance provider, which returns specific information for a managed object. The class provider is a more complex type of provider that is able to return definitions of classes as well as instances. If, for example, you were trying to return information from a

database in which someone might be adding new tables, it would be possible to return a class per table, with each class being defined at the time the provider opened the database. This type of provider is advanced and is usually not implemented by services. Active Directory is a rare example of a service that does implement a dynamic class provider. Property providers are a simpler type of provider that allows dynamic properties to be added to static instances.

The general intent of the WMI architecture is to make writing providers as simple as possible—for example, writing an instance provider might be just a matter of generating a few lines of code (that is, with the help of the provider-generating tools and samples in the WMI SDK). The construction of providers also takes advantage of inheritance. If you are adding a new subclass to a class that already has a provider, you need to supply only property values for your new properties or for the ones you are overriding.

Managed Objects

A managed object can be any enterprise component—for example, a Win32 object or service. A service that allows clients to connect using a named pipe could allow itself to be managed by exposing the name of the pipe in a WMI object instance.

Management Applications

Management applications are applications that communicate with a local or remote WinMgmt service. These applications can query the WinMgmt service to determine all the logical and physical components residing on a machine. This information can then be displayed and modified, allowing the user to really understand and change a machine's configuration. In addition, management applications can instruct the WinMgmt service to send notifications when special events occur. For example, a management application can receive a notification when the machine's hard drive has less than 20 percent free space, or when a new process begins running on the managed machine.

Schema

WMI hides all the ugly complexities of the management environment. It takes the Win32 APIs, registry settings, Simple Network Management Protocol (SNMP) Management Information Bases (MIBs) and traps, and all other miscellaneous management interfaces, and it wraps them in a carefully defined schema.

The schema isn't really a separate component of WMI, but it is an important part of WMI. The schema is a collection of classes that describe managed objects. All the components in the WMI architecture support the schema.

The schema, which is provided by the object manager, handles a whole range of information, including systems, networks, applications, devices, and physical components, that affects the installation, configuration, and management of services. The schema consists of instances arranged in classes that have properties, associations, and methods.

WMI Name Evolution

The Common Information Model (CIM) is a specification defined by the Distributed Management Task Force (DMTF) that describes an object-oriented approach to the management of systems and networks. Microsoft originally named its implementation of the CIM "Web-Based Enterprise Management (WBEM)" and named the kernel-mode portion "Windows Management Information (WMI)." The DMTF started a marketing initiative and

used "WBEM" as the name to describe a set of management and Internet technologies based on CIM. Microsoft then changed the name of its implementation of the CIM to "WMI" and changed the name of the kernel-mode portion to "WMI extensions for the Windows Driver Model (WDM)." Because of this name evolution, you find "WMI," "WBEM," and "CIM" used throughout the Windows management architecture for functions, interfaces, and classes, as well as other elements.

[Previous] [Next]

WMI Tools

Various tools are available when you're working with WMI. The two tools you will probably use most often are CIM Studio and the MOF compiler. Later in this chapter in the section titled "The TimeServiceProvider Sample WMI Provider," we'll discuss another tool called the WMI Provider Code Generator Wizard.

WMI CIM Studio

WMI CIM Studio is a tool for viewing and editing the class and instance information for a CIM repository. Classes and instances are grouped into units called namespaces. WMI provides a namespace called root\CIMV2, which contains all the classes and instances that represent a system. To experiment with WMI, install the WMI SDK (specifically, WMI Tools, which is included as part of the Platform SDK on the companion CD). After you install the SDK, you can use the CIM Studio tool to look at the contents of the root\CIMV2 namespace. I will be using CIM Studio in this chapter to show you many of WMI's capabilities.

To open CIM Studio, select WMI CIM Studio from the Programs menu. It will be listed under the Microsoft Platform SDK group (or the WMI SDK group). To see a list of the services installed on your machine, navigate to the Win32_Service class by opening CIM_ManagedSystemElement, CIM_ LogicalElement, CIM_Service, Win32_BaseService, and Win32_Service. Click the Instances toolbar button above the right pane, and you should see the screen shown in Figure 8-2.

Figure 8-2. *WMI CIM Studio showing installed services*

MOF Compiler

A MOF compiler is provided as a part of WMI and can be used to introduce new classes and instances into the system. The following MOF example illustrates a number of features of WMI classes:

```
[static] class Msft_Joke {
   [key] string JokeName;
         string JokeText;
};

instance of Msft_Joke {
   JokeName = "KnockKnock1";
   JokeText = "Knock, knock. :Who's there? Dwayne: Dwayne, who?: "
      "Dwayne the bathtub, I'm dwowning!";
};
```

Classes are identified by the keyword *class* and can be preceded by qualifiers in square brackets (*[ ]*). Qualifiers are used to supply information relevant to the handling of the class, such as how it's implemented or localization information. Class properties are defined as a type followed by a name and optionally preceded by qualifiers in brackets.

In the Msft_Joke example, the class is defined as *[static]*, meaning that the instances of the class are stored in the WMI repository. The JokeName property is of type string and is a key for the class, which means that every instance of the class must have a value for the property, and the value must be different from the value present in all other instances.

[Previous] [Next]

WMI Data Organization

WMI offers a basic set of classes that organize how data should be presented to the user. By using these classes, you avoid much of the burden of schema design—all you have to do is extend the existing design. But you need to understand what WMI provides for free before you extend WMI to represent the special characteristics of your service.

WMI divides management information into categories, which are represented by some of the top-level classes in the WMI schema. These categories are operational data, setting data, statistical data, and historical data.

Operational Data

Operational data is concerned with current state and deals with questions such as "How much memory does the service require?" It represents what the system is actually doing and what it is capable of doing. For example, in the context of a service, the system might indicate whether the service is capable of interacting with the desktop, what the service's display name is, and so on.

Setting Data

Setting data represents desired state and configuration information and deals with questions such as "How much memory should the service use?" When you work with setting data, ask yourself questions such as these: "Is there more than one set of possible values for the property, and do certain values apply to different users or at different times?" These kinds of questions help you organize the data. For example, regarding the

preceding questions, if more than one set of values exists for the property, you should separate the data into a separate setting class that provides this configuration information. In many cases you might find similar looking values in both setting and operational data.

Statistical Data

Statistical data answers questions such as "How much memory is the service using?" It is similar to operational data except that it represents infrequently requested statistical data that is useful for understanding performance but not essential for understanding the state of the managed object. Separating statistical data from operational data is essential if for no other reason than to avoid having to evaluate the statistics when accessing regular operational data.

Historical Data

Historical data answers questions such as "How much memory was the service using?" and represents past state. WMI does not provide a log that is intended for capturing large volumes of data, but it does provide a mechanism for delivering the data and a form that is appropriate for representing historical state, as you'll see when you look at events later in this chapter.

When considering whether to extend WMI, first determine whether WMI already supports the data you want to introduce. If it's already there, you don't do anything. If it is not, you figure out whether you are dealing with operational, setting, statistical, or historical data; find a suitable class; and then see whether the data is already populated with content that represents the type of data you are interested in. For example, if you are adding a new service, you will find that the service already turns up as an instance of Win32_Service. If you need to add new operational data, you should derive a new class by using Win32_Service as a base and then populate the derived class in an appropriate way (more about this in a moment).

[Previous] [Next]

Core Service-Related Classes Provided by WMI

Let's take a brief look at the core service-related classes provided by WMI. These classes are shown in Figure 8-3. The CIM_ManagedSystemElement, CIM_Setting, and CIM_StatisticalInformation classes provide root classes for operational, setting, and statistical data, respectively. As you can see from looking at the Win32_BaseService class, a wide range of information is provided by WMI about services, as well as about basic methods for defining, modifying, starting, stopping, and deleting services.

NOTE

Class names prefixed with "CIM_" indicate classes defined by the DMTF, whereas classes prefixed with "Win32_" are defined by Microsoft. If you intend to add new classes to the CIMV2 namespace, you should decide on a name for your schema that is familiar to you, such as a trademark, to ensure that no one else uses the same name.

Figure 8-3. *Core service-related classes provided by WMI*

CIM_ManagedSystemElement

The CIM_ManagedSystemElement class is the base class for all the operational data held in the CIMV2 namespace. It represents all the physical and logical components of the system. CIM_ManagedSystemElement is a base class for exactly two derived classes: CIM_LogicalElement and CIM_PhysicalElement.

> **NOTE**
>
> You should never derive a class using CIM_ManagedSystemElement as a base. If you do, you are basically saying that the data you are defining is neither physical nor logical; however, the CIM schema claims that every element of a system is either physical *or* logical and cannot be both.

CIM_ManagedSystemElement defines the following fundamental set of properties for which every component of a system is expected to supply values:

- **Caption** A short textual description (a one-line string) for the object.
- **Description** A textual description of the object of unlimited size.

- **InstallDate** The date and time value indicating when the object was installed. A lack of a value does not indicate that the object is not installed.
- **Name** The label by which the object is known. Derived classes can override the name to make the property a key, but they do not have to.
- **Status** An enumeration indicating the current state of the object

NOTE

These properties and those defined for the other classes described in this section are described in the WMI SDK and CIM Studio. To take a look at a property description in CIM Studio, select the class that contains the property, and click the Help For Class button.

Win32_BaseService

The Win32_BaseService class represents executables that are installed in the Service Control Manager's registry database. Instances of this class identify services or device drivers. Your service should derive a class using Win32_Service (discussed in the next section) as its base class. A device driver would derive a class using Win32_SystemDriver as its base class.

NOTE

You should never derive a class from Win32_BaseService. Doing so tells the system that you have changed the operating system architecture.

The properties supported by the Win32_BaseService class include the following:

- AcceptPause
- AcceptStop
- DesktopInteract
- DisplayName
- ErrorControl
- ExitCode
- PathName
- ServiceSpecificExitCode
- ServiceType
- StartName
- State
- TagId

NOTE

In CIM Studio, some properties have a yellow arrow icon. This yellow arrow indicates that the property is inherited from a base class.

Win32_Service

The Win32_Service class represents a service on a Microsoft Win32 computer system. A service application conforms to the interface rules of the Service Control Manager as discussed in Chapters 3 and 4 of this book.

In WMI, the service class should be used for managing the service, not for providing access to the service's tasks. For example, it would be inappropriate to declare a DHCPService class derived from Win32_Service and implement a method in the class to obtain a new DHCP address. However, it would be entirely appropriate to implement a method in a derived class that would allow you to restrict the range of addresses that could be returned by the service.

The Win32_Service class supports the following properties:

- CheckPoint
- ProcessId
- WaitHint

CIM_ServiceAccessPoint

The CIM_ServiceAccessPoint class represents access points for your service, such as the command-line parameters accepted by an application and port 80 in the context of a protocol service. Use this class to manage the access point. Provide instances of this class if you need to control, for example, which machine hosts the access point.

CIM_Setting

You know that setting data is represented separately from operational data. The CIM_Setting class provides a base class for all setting data. Any time you want to add setting-related data for your service, you should make it accessible by deriving from the CIM_Setting class.

Notice the CIM_ElementSetting association between CIM_Setting and CIM_ManagedSystemElement that defines which CIM_ManagedSystemElement a particular setting applies to. You should always derive a class from an association class to establish the particular association between your CIM_ManagedSystemElement class (typically a Win32_Service or CIM_ServiceAccessPoint class) and the CIM_Setting-derived class that configures it. You can store the setting data in the CIM repository (as you saw in the joke class example earlier in this chapter), but I don't advise it because the CIM repository is not meant as a general-purpose store. Rather, save the information elsewhere&"142">

CIM_StatisticalInformation

The CIM_StatisticalInformation class provides a location for statistical data in the CIMV2 namespace. You can select any of the classes under this class and look at an instance of it to see its current statistics. For example, look at Win32_PerfRawData_PerfDisk_PhysicalDisk under CIM_StatisticalInformation\Win32_Perf\Win32_PerfRawData.

NOTE

> Some of the class names include multiple underscores. Strictly speaking, this is not allowed—these are cases in which Microsoft has not conformed to the rules. Be careful to include only one underscore in each of your class names; this underscore should separate the trademark-derived name from the class name.

Association Classes

An important feature I have only touched on is association classes, which link classes together. One example of an association class that you have seen is CIM_ElementSetting, which links CIM_ManagedSystemElement classes to the settings that represent their configuration parameters.

You must understand the more important association classes to use WMI properly. The next few sections describe the important associations you should be aware of.

CIM_Dependency

Most of the associations that link two CIM_ManagedSystemElement classes are one of two types—a CIM_Dependency association or a CIM_Component association. CIM_Dependency associations represent functional dependency between objects. Dependencies show which components another component depends upon. For example, the Win32_DependentService association class (derived from CIM_ServiceServiceDependency, which is derived from CIM_Dependency) is used to indicate a dependency relationship between services. If a service fails to start due to a dependent service, it is possible to follow the dependency chain until you locate the service that is causing all the trouble.

If you open CIM Studio and expand the tree under CIM_Dependency, you will see well over 40 association classes, each representing a different type of dependency that can be expressed in the WMI model. Some of them are specific to services and their associated objects. CIM_Dependency is frequently used as a base class.

CIM_Component

The CIM_Component association class is used to identify that a component is part of another component. Classes derived from CIM_Component represent the association between a system component and the various CIM_ManagedSystemElement classes that make up the system. CIM_Component is frequently used as a base class.

CIM_ElementSetting

CIM_ElementSetting associates a setting with the CIM_ManagedSystemElement class for which it provides configuration parameters. You must derive a class from the CIM_ElementSetting association class to associate any settings you add to the CIMV2 namespace with the CIM_ManagedSystemElement-derived class that the settings apply to.

CIM_Statistics

The statistics association class links a CIM_ManagedSystemElement-derived class to the various statistics that apply to it. You must derive a class from the CIM_Statistics association class to provide a way for users to get at a CIM_ManagedSystemElement-derived class object's statistics.

CIM_ServiceServiceDependency

This class is derived from the CIM_Dependency association class. Use it to define any dependency relationships that exist between services. It is actually quite unusual to see a class such as CIM_ServiceServiceDependency as part of the CIMV2 namespace, because service dependencies are typically defined as properties of the service itself.

CIM_ServiceAccessBySAP

This class is also derived from the CIM_Dependency association class and should be used to define associations between a service and the access points that can be used to access the service.

Win32 Service-Related Classes

The next few sections describe some of the Win32 service-related classes you should be aware of. These include classes related to the system, classes related to user accounts, and classes that specify execution dependencies for groups of services. Some of the classes are shown in Figure 8-4.

Win32_ComputerSystem

The Win32_ComputerSystem class represents a computer in the Win32 environment. It includes multiple properties related to booting, power, hardware, owners, and other information. The Win32_ComputerSystem class ultimately derives from the CIM_System class. Typically you will never add anything to the CIM_System class or any of its descendants. Although system manufacturers might derive a class from Win32_ComputerSystem to define specific features of their Win32 systems, it is highly unlikely.

Win32_Process

The Win32_Process class is strictly limited to the processes known to the Win32 system. You should not add instances or properties to this class, but you can establish associations to process instances. For example, you could use associations to expose the details of how your service maps to a set of processes and threads.

Figure 8-4. *Some Win32 service-related classes*

Win32_Account

The Win32_Account class contains information about user accounts and group accounts known to the Win32 system. User or group names recognized by a Windows domain are descendants (or members) of this class.

Win32_SystemAccount

The Win32_SystemAccount class represents the LocalSystem account.

Win32_LoadOrderGroup

The Win32_LoadOrderGroup class identifies which load order group a service can be dependent upon. The two important associations for the Win32_LoadOrderGroup class are the Win32_LoadOrderGroupServiceMembers class, which represents a component (or membership) association between a base service and a load order group, and the Win32_LoadOrderGroupServiceDependencies class, which represents a dependency association between a base service and a load order group that the service depends on to start running. The difference between the two can be represented this way: the component association says "this service or driver is a member of this group," whereas the dependency association says "this service or driver is dependent on this group."

Software Installation Classes

Figure 8-5 shows the main classes involved in software installation. These classes directly reflect the information used to control the Microsoft Windows Installer technology. If you use Windows Installer to install your service, these installation classes will be populated for you automatically, and you will be able to install, uninstall, and repair your installation by using WMI.

Figure 8-5. *Classes involved in software installation*

Win32_Product is a concrete class that is a collection of physical elements, software features, and/or other products, acquired by a consumer as a unit. Instances of this class represent products installed by the Windows Installer. A product generally correlates to a single installation package.

The CIM_SoftwareElement class breaks up a CIM_SoftwareFeature object into a set of individually manageable or deployable elements for a particular platform. A software element's platform is uniquely identified by its underlying hardware architecture and operating system (for example, Microsoft Windows 98 or Windows 2000).

[Previous] [Next]

Events

Any practical management application must be largely driven by events. This is true both for a reactive management application (in which components are fixed after they break) and a preventive management application (in which components are fixed before they break). WMI has an extensive set of event-related features that require minimum effort on your part to use. In this section, I will look at what WMI provides and then describe how you can turn your service into an enterprise class application capable of being managed in a large and complex environment, or suitable for use in situations where remote service and support are the norm.

Any event architecture has to solve at least three problems:

- **Publication** How do I find out what events there are?
- **Subscription** How do I tell the system that I am interested in an event?
- **Delivery** How does the event get delivered to me?

Event Publication

In WMI an event is published by declaring an event class. The occurrence of an event is represented by an instance of the class. All event classes are derived from the __Event system class.

> **NOTE**
>
> You can find the __Event class in CIM Studio under __SystemClass__IndicationRelated. The leading double underscore for system classes is a naming convention used to keep the system class names separate from other class names. WMI does not allow you to declare a class with a leading underscore in the name.

The most important event classes defined by WMI are __InstanceOperationEvent and its derived classes. They are actually very simple, as you can see from the following MOF code:

```
class __InstanceOperationEvent : __Event {
   object TargetInstance;
};
class __InstanceCreationEvent : __InstanceOperationEvent {
};
class __InstanceDeletionEvent : __InstanceOperationEvent {
};
class __InstanceModificationEvent : __InstanceOperationEvent {
   object PreviousInstance;
};
```

The two *object* declarations define embedded objects. These objects trigger the event. For example, when a new process comes into existence, as represented by a new instance of the Win32_Process class, a new

instance of the __InstanceCreationEvent class is produced with its TargetInstance property containing the new process as an embedded object.

In this way, WMI provides an event representing every possible creation, modification, or deletion of any object in the rest of the namespace. Of course, these events are received only when requested—in the simplest computer systems, if you asked for all possible __InstanceOperationEvents, you would receive literally hundreds of events a second. You would essentially be monitoring every instance of a thread, process, file, service, or other component of the system changing state.

Event Subscription

In WMI, a client application subscribes to an event by creating a query. Queries are a natural way to interact with a class. You tell WMI which event you are interested in by calling one of several APIs, passing in a query that defines the event. For example, the following Microsoft Visual Basic code detects when a new process has been started and displays the name of the process:

```
Dim wbemService As SWbemServices
Dim events As SWbemEventSource
Dim singleEvent As SWbemObject
Dim sQuery As String

Set wbemServices = GetObject("winmgmts:{impersonationLevel=impersonate}")
sQuery = "select * from __instancecreationevent within 10 " & _
   "where targetinstance isa `Win32_Process'"
Set events = wbemServices.ExecNotificationQuery(sQuery)
Do
   Set singleEvent = events.NextEvent
   MsgBox singleEvent.TargetInstance.Name
Loop
```

The first statement of the code opens WMI by getting a handle on the WinMgmt service. The second statement provides the subscription query. The third statement invokes the *ExecNotificationQuery* function provided by the service to execute the query. The *events* object returned by the *ExecNotificationQuery* call has a *NextEvent* method that allows you to wait for the events as they occur. As you will see when we discuss event delivery, this example is the simplest possible case of using a query to request an event. WMI supports very sophisticated event subscription and delivery mechanisms suitable to a wide range of management applications.

NOTE

You should note some other features about the preceding code example as well. It takes advantage of the WMI scripting convention that assumes root\CIMV2 is the default namespace. Also, it does not include any error handling or termination code.

Event Delivery

Now let's turn to event delivery. You have seen one basic event delivery mechanism supported by the scripting interface that allows you to write simple scripts that can capture and handle events. Realistically, if you are going to use events as a part of a management application, you will need to deal with them in an asynchronous manner. You will also need to filter the events so that you see only the events you are interested in.

WMI allows you to control the way an event is delivered both in terms of which event is delivered and which data gets delivered with the event. This control is afforded by the fact that subscriptions are expressed using

queries. We have already seen one very simple subscription query that gets an event each time a process is started.

Let's consider a slightly more complex example. Assume you want to get only the name of the service each time a manually started service is stopped. Your first step is to find a property that defines the way a service is started. Take a look at the Win32_Service class definition in CIM Studio. You should see a property named StartMode. Right-click on the property, and select Property Qualifiers to see its qualifiers. The ValueMap qualifier contains information about the start mode and can be one of the following values: Boot, System, Auto, Manual, and Disabled. We can check the StartMode property to see if it is "Manual". The name of the service is given by the Name property, so your subscription query would be as follows:

```
Select TargetInstance.Name
From __InstanceModificationEvent
Within 10
Where TargetInstance isa `Win32_Service' and
   TargetInstance.StartMode = `Manual' and
   TargetInstance.Started = FALSE and
   PreviousInstance.Started = TRUE
```

Notice that the query uses the PreviousInstance value to determine whether the service's start status has changed from true to false. Many aspects of a service can change, but in this case we are interested only in whether the service has stopped. The query uses the *Within* clause to define a polling interval to be used by the CIM Object Manager in detecting the occurrence of the event. Basically the object manager re-evaluates the event every 10 seconds.

You can write a permanent event subscriber that is registered with WMI in such a way that any time the event occurs, the subscriber is invoked and can take some action. In the example we just looked at, you could make sure that an operator is notified every time your service is stopped.

WMI comes with a variety of standard subscribers that you can use to build sophisticated event handlers. For example, WMI can make use of Microsoft Message Queue (MSMQ) to guarantee delivery of events. WMI has standard subscribers that can issue command-line calls that, for example, allow you to send an e-mail message when an event occurs.

[Previous] [Next]

The TimeServiceProvider Sample WMI Provider

The TimeServiceProvider sample WMI provider ("08 TimeServiceProvider.dll"), shown in Listing 8-1, demonstrates how to create a WMI provider DLL for the TimeService service presented in Chapter 3. TimeServiceProvider exposes a property of the TimeService service to WMI. By using WMI, a client could retrieve this property. Unlike the other samples in this book, the WMI SDK tools create the source code for this sample. In this section, you'll learn how to use these tools to create a dynamic extension to WMI's Win32_Service class. The source code files for the provider are in the 08-TimeServiceProvider directory on the companion CD. Now I will step through the creation of a simple dynamic provider.

Selecting Information to Provide

The first step in creating a provider is deciding what information about the service to make available. The TimeService service uses a named pipe for the client/server connections—the name of this named pipe is one piece of information you can make available. In the TimeService.cpp file (from Chapter 3), we see that the pipe name is "\\.\pipe\TimeService". We'll create a dynamic provider that allows an administrator or client to query the name of the named pipe by using WMI. You might consider revising the WMI provider so that it

also allows an administrator to change the name of the named pipe by using WMI.

Deriving a Class Using MOF

The next step is to derive a class by using Win32_Service as the base class and add a named pipe property to the derived class. The following code shows the MOF code used to create this class and is in the TimeServiceStart.mof file on the companion CD:

```
#pragma namespace("\\\\.\\root\\cimv2")
class Richter_TimeServiceProvider : Win32_Service {
   string PipeName;
};
```

Using the MOF Compiler

Once you have created the .mof file, you need to compile it using the MOF compiler (MOFComp.exe), which is a part of WMI. The MOF compiler will add the new Richter_TimeServiceProvider class to the system. The following command shows how you can compile:

```
C:\WINNT\system32\wbem\mofcomp.exe _class:forceupdate
   "<filepath>\TimeServiceStart.mof"
```

Notice the use of the "-class:forceupdate" switch and the use of quotation marks around the MOF file path. The "-class:forceupdate" switch forces the update of classes when there might be conflicting classes, and the quotes allow you to use names containing spaces. For more information on the switches available for the MOF compiler, see the "MOF Compiler" topic in the WMI SDK and Platform SDK documentation.

Integrating the MOF Compiler in Visual Studio

If you use the MOF compiler frequently, you'll find it convenient to add it to the Microsoft Visual Studio Tools menu. To do this, open Visual Studio and select Customize from the Tools menu. In the Customize dialog box, click on the Tools tab, scroll to the bottom of the list, and add a new tool named MOF Compiler, as shown in Figure 8-6.

Because you've established the MOF compiler as a Visual Studio tool, you can open any MOF file and compile it. Since the Use Output Window check box is checked, any compilation errors will appear as errors in the Output window. You can double-click the errors in the Output window and have Visual Studio automatically position itself on the offending line in the source file.

Figure 8-6. *The Tools tab of the Customize dialog box, where you can add the MOF Compiler to the Visual Studio Tools menu*

After you compile the TimeServiceStart.mof file, run CIM Studio and open the CIMV2 namespace. You should see the new class under the following node. If CIM Studio is already open you will need to refresh the display.

```
CIM_ManagedSystemElement\CIM_LogicalElement\CIM_Service\
   Win32_BaseService\Win32_Service\Richter_TimeServiceProvider
```

Figure 8-7 shows what CIM Studio should look like. The new PipeName property is highlighted so that you can easily see that it is part of this class instance.

To populate the new Richter_TimeServiceProvider class, a provider must be defined and associated with the class definition. This is how the Windows Management Service (WinMgmt.exe) knows to call the provider to get the instance's data. The easiest way to create the provider is to have the CIM Studio tool generate the DLL's source code for you.

Using the WMI Provider Code Generator Wizard

To generate a provider DLL for the Richter_TimeServiceProvider class, you must first select the class by checking the box to the left of the class's name in the left pane (as shown in Figure 8-7). Then double-click the Provider Code Generator button at the top of CIM Studio to invoke the WMI Provider Code Generator Wizard.

Figure 8-7. *The Richter_TimeServiceProvider class in CIM Studio, after compiling TimeServiceStart.mof*

In the welcome dialog box, click Next to begin the wizard. In the next dialog box, add the appropriate text in the Description text box. Then check the Override Inherited Properties check box, and check the Name property in the list of properties that the provider is to override from the base class. You must override the Name property because it is used by the object manager to determine whether the instance properties you supply correspond to the instance properties supplied by the derived-class provider. The dialog box should look like Figure 8-8. (Note that you can't see that I checked the Name property.)

Figure 8-8. *The WMI Provider Code Generator Wizard dialog box used to specify names and properties*

Click the Next button to display the dialog box shown in Figure 8-9. You use this dialog box to give the provider DLL a filename and description, and to specify the directory where you'd like the generated code to be output.

Click the Finish button to generate the provider source code. The wizard will generate six files: MAINDLL.cpp, Richter_TimeServiceProvider.cpp, Richter_TimeServiceProvider.h, TimeServiceProvider.def, TimeServiceProvider.mak, and TimeServiceProvider.mof.

Modifying the Wizard-Generated Code

You still need to make a few changes to the generated code before it will work correctly. By default, the TimeServiceProvider class will be placed in the root\default namespace. To make the class part of the root\CIMV2 namespace, you must add the following pragma to the top of TimeServiceProvider.mof:

```
#pragma namespace("\\\\.\\root\\cimv2")
```

Figure 8-9. *The WMI Provider Code Generator Wizard dialog box used to specify the provider's filename, description, and output directory*

Deploying a provider is more convenient if the contents of the TimeServiceProvider.mof and TimeServiceStart.mof files are combined. I appended the contents of TimeServiceStart.mof to the bottom of the TimeServiceProvider.mof file produced by the wizard. I also added the following line so that the WinMgmt service is aware that the values of this class's properties are generated dynamically. Listing 8-2 shows the TimeServiceProvider.mof file after the modifications have been made.

```
[provider("TimeServiceProvider"), dynamic]
```

Now let's focus on the Richter_TimeServiceProvider.cpp file, which requires a few modifications before it is useful. First, find this line:

```
CRichter_TimeServiceProvider MyRichter_TimeServiceProviderSet
   (PROVIDER_NAME_RICHTER_TIMESERVICEPROVIDER, L"NameSpace") ;
```

Replace *NameSpace* with the appropriate namespace:

```
CRichter_TimeServiceProvider MyRichter_TimeServiceProviderSet
   (PROVIDER_NAME_RICHTER_TIMESERVICEPROVIDER, L"root\\cimv2") ;
```

Second, modify the *EnumerateInstances* function so that it returns the correct property values. For the TimeService service, this is very easy because there is only one property, the name of the named pipe. The following code shows how the *EnumerateInstances* function should look after its modifications.

```
HRESULT CRichter_TimeServiceProvider::EnumerateInstances (
   MethodContext* pMethodContext, long lFlags ) {
   HRESULT hRes = WBEM_S_NO_ERROR;
```

```
   // Create a new instance based on the passed-in MethodContext.
   // Note that CreateNewInstance may throw but will never return NULL.
   CInstance* pInstance = CreateNewInstance(pMethodContext);

   // Class name must match programmatic name of service
   pInstance->SetCHString(pName,
      "Programming Server-Side Applications Time");

   pInstance->SetCHString(pPipeName, "TimeService");
   hRes = pInstance->Commit();
   pInstance->Release();
   return(hRes);
}
```

Third, you must modify the *GetObject* function so that it returns the data for a given instance. For a service, modifying the function is very simple because there is only one instance of a service on a machine at any given time. The following code shows how the *GetObject* function should look after its modifications. Listing 8-1 shows the resulting Richter_TimeServiceProvider.cpp file.

```
HRESULT CRichter_TimeServiceProvider::GetObject (CInstance* pInstance,
   long lFlags)
{
   HRESULT hr = WBEM_E_NOT_FOUND;

   // Class name must match programmatic name of service
   pInstance->SetCHString(pName,
      "Programming Server-Side Applications Time");
   pInstance->SetCHString(pPipeName, "TimeService");
   hr = WBEM_S_NO_ERROR;
   return(hr);
}
```

Building the TimeServiceProvider Sample

To build the provider DLL, I created an empty Win32 project using Visual Studio. Then I added the MAINDLL.cpp, Richter_TimeServiceProvider.cpp, Richter_TimeServiceProvider.h, and TimeServiceProvider.def files to the project. You must also remember to link with the FrameDyn.lib library (supplied with the WMI SDK). I ensure this by placing the following line in the Richter_ TimeServiceProvider.cpp file:

```
#pragma comment(lib, "FrameDyn.lib")
```

Finally, I added the USE_POLARITY symbol for both the Debug and Release builds of the project. I did this on the C/C++ tab of the Project Settings dialog box, shown in Figure 8-10. You'll want to compile the files using warning level 3, because the framework header files generate many warnings when compiled at warning level 4.

Figure 8-10. *The C/C++ tab, where you add symbols and set warning levels for your files*

Deploying the TimeServiceProvider Sample

To deploy the provider, you must run the MOF compiler, passing it the TimeServiceProvider.mof file. This makes the WinMgmt service aware of the WMI class provider information. Next you must run RegSvr32, passing it the "08 TimeServiceProvider.dll" file. Finally, you must make sure that the TimeService service (which has a display name of "Programming Server-Side Applications Time") is installed on the machine. Note that the service does not have to be running, but it must be installed. You can install the service by using the install switch ("03 TimeService.exe" _install).

To see the results of all this hard work, open CIM Studio and navigate to the following node:

```
CIM_ManagedSystemElement\CIM_LogicalElement\CIM_Service\
   Win32_BaseService\Win32_Service\Richter_TimeServiceProvider
```

Click the Instances button above the right pane. You should see information being returned from the TimeServiceProvider DLL, specifically the normal properties of the Win32_Service class and the new PipeName property, as shown in Figure 8-11.

Figure 8-11. *The TimeServiceProvider DLL exposing the PipeName property of the TimeService service in CIM Studio*

NOTE

> The provider DLL is loaded into WinMgmt.exe's address space when it is needed. If you want to make changes to your provider's source code and rebuild it, you must stop the WinMgmt service first and then restart it when you're ready to have it load your newly built DLL.

If your provider does not work, take a look at the C:\WINNT\System32\WBEM\Logs directory, which contains a number of text log files. These files track various errors such as the failure to locate or load a provider when needed. Also, to help you debug a provider, you can run WinMgmt.exe as a normal process instead of a service. To do this, you must first stop the WinMgmt service and then run WinMgmt.exe from the command line using the /exe switch.

Listing 8-1. *The TimeServiceProvider sample*

Richter_TimeServiceProvider.cpp

```
/*****************************************************************
Richter_TimeServiceProvider.CPP -- WMI provider class implementation Generated by Microsoft WMI Code Generation Engine TO DO: - See individual function headers - When linking, make sure you link to framedyd.lib & msvcrtd.lib (debug) or framedyn.lib & msvcrt.lib (retail). Description: Programming Server-Side Applications for Microsoft Windows TimeService WMI Provider
*****************************************************************/ #pragma message("This project requires that the Platform SDK's WMI components") #pragma message("be installed. You may install these componenets from the ") #pragma message("book's CD-ROM or from msdn.microsoft.com") // identifier was truncated to `255' characters in the debug information #pragma warning(disable: 4786)
//***************************************************************************
#pragma warning(push, 3) #include <fwcommon.h> // This must be the first include. #pragma comment(lib, "FrameDyn.lib") #include "Richter_TimeServiceProvider.h" // TO DO: Replace "NameSpace" with the appropriate namespace for your // provider instance. For instance: "root\\default or "root\\cimv2".
//=================================================================
CRichter_TimeServiceProvider MyRichter_TimeServiceProviderSet ( PROVIDER_NAME_RICHTER_TIMESERVICEPROVIDER, L"root\\cimv2") ; // Property names
//=============== const static WCHAR* pName = L"Name" ; const static WCHAR* pPipeName = L"PipeName" ;
/***************************************************************************** * * FUNCTION : CRichter_TimeServiceProvider::CRichter_TimeServiceProvider * * DESCRIPTION : Constructor * * INPUTS : none * * RETURNS : nothing * * COMMENTS : Calls the Provider constructor. *
*****************************************************************************/
CRichter_TimeServiceProvider::CRichter_TimeServiceProvider (LPCWSTR lpwszName, LPCWSTR lpwszNameSpace ) : Provider(lpwszName, lpwszNameSpace) { }
/***************************************************************************** * * FUNCTION : CRichter_TimeServiceProvider::~CRichter_TimeServiceProvider * * DESCRIPTION : Destructor * * INPUTS : none * * RETURNS : nothing * * COMMENTS : *
*****************************************************************************/
CRichter_TimeServiceProvider::~CRichter_TimeServiceProvider () { }
/***************************************************************************** * * FUNCTION : CRichter_TimeServiceProvider::EnumerateInstances * * DESCRIPTION : Returns all the instances of this class. * * INPUTS : A pointer to MethodContext for communication with WinMgmt. * A long that contains the flags described in * IWbemServices::CreateInstanceEnumAsync. Note that the following * flags are handled by (and filtered out by) WinMgmt: * WBEM_FLAG_DEEP * WBEM_FLAG_SHALLOW * WBEM_FLAG_RETURN_IMMEDIATELY * WBEM_FLAG_FORWARD_ONLY *
```

```
WBEM_FLAG_BIDIRECTIONAL * * RETURNS : WBEM_S_NO_ERROR if successful * *
COMMENTS : TO DO: All instances on machine should be returned here and * all properties this
class knows how to populate must * be filled in. If there are no instances, return *
WBEM_S_NO_ERROR. Not an error to have no instances. * If you are implementing a `method only'
provider, you * should remove this method. *
*****************************************************************************/
HRESULT CRichter_TimeServiceProvider::EnumerateInstances ( MethodContext*pMethodContext,
long lFlags) { HRESULT hRes = WBEM_S_NO_ERROR; // TO DO: The following commented lines
contain the `set' methods for the // properties entered for this class. They are commented because they
// will NOT compile in current form. Each <Property Value> should be // replaced with the appropriate
value. Also, consider creating a new // method and moving these set statements and the ones from
GetObject // into that routine. See the framework sample (ReindeerProv.cpp) for // an example of how
this might be done. // // If the expectation is there is more than one instance on the machine //
EnumerateInstances should loop through instances & fill accordingly. // // Note that you must
ALWAYS set ALL the key properties. See docs for // further details.
////////////////////////////////////////////////////////////////////////////// // Create a new instance based on the passed-in
MethodContext. // Note that CreateNewInstance may throw, but will never return NULL. CInstance*
pInstance = CreateNewInstance(pMethodContext); // Class name must match programmatic name of
service pInstance->SetCHString(pName, "Programming Server-Side Applications Time");
pInstance->SetCHString(pPipeName, "TimeService"); hRes = pInstance->Commit();
pInstance->Release(); return(hRes); }
/***************************************************************************** * *
FUNCTION : CRichter_TimeServiceProvider::GetObject * * DESCRIPTION : Find a single instance
based on the key properties for the * class. * * INPUTS : A pointer to a CInstance object containing
key properties. * A long that contains the flags described in * IWbemServices::GetObjectAsync. * *
RETURNS : WBEM_S_NO_ERROR if the instance can be found * WBEM_E_NOT_FOUND if the
instance described by the key * properties could not be found * WBEM_E_FAILED if the instance
could be found but another * error occurred. * * COMMENTS : If you are implementing a `method
only' provider, you * should remove this method. *
*****************************************************************************/
HRESULT CRichter_TimeServiceProvider::GetObject (CInstance* pInstance, long lFlags) { // TO
DO: The GetObject function is used to search for an instance of this // class on the machine based on
the key properties. Unlike // EnumerateInstances which finds all instances on the machine, //
GetObject uses the key properties to find the matching single // instance and returns that instance. // //
Use the CInstance Get functions (for example, call // GetCHString(L"Name", sTemp)) against
pInstance to see the key // values the client requested. HRESULT hr = WBEM_E_NOT_FOUND; //
Class name must match programmatic name of service pInstance->SetCHString(pName,
"Programming Server-Side Applications Time"); pInstance->SetCHString(pPipeName,
"TimeService"); hr = WBEM_S_NO_ERROR; return(hr); }
/***************************************************************************** * *
FUNCTION : CRichter_TimeServiceProvider::ExecQuery * * DESCRIPTION : You are passed a
method context to use in the creation of * instances that satisfy the query, and a CFrameworkQuery *
which describes the query. Create and populate all * instances which satisfy the query. You may return
more * instances or more properties than are requested and * WinMgmt will post filter out any that do
not apply. * * INPUTS : A pointer to MethodContext for communication with WinMgmt. * A query
object describing the query to satisfy. * A long that contains the flags described in *
IWbemServices::CreateInstanceEnumAsync. Note the fol- * lowing flags are handled by (and filtered
out by) WinMgmt: * WBEM_FLAG_FORWARD_ONLY * WBEM_FLAG_BIDIRECTIONAL *
WBEM_FLAG_ENSURE_LOCATABLE * * RETURNS :
WBEM_E_PROVIDER_NOT_CAPABLE if queries not supported for * this class or if query is too
complex for this class * to interpret. The framework will call the * EnumerateInstances function
instead and let Winmgmt post filter. * WBEM_E_FAILED if the query failed *
WBEM_S_NO_ERROR if query was successful * * COMMENTS : TO DO: Most providers will not
need to implement this method. If you don't, WinMgmt will call your enumerate * function to get all
```

```
the instances and perform the * filtering for you. Unless you expect SIGNIFICANT * savings from implementing queries, you should remove * this method. You should also remove this method * if you are implementing a `method only' provider. *
*****/
HRESULT CRichter_TimeServiceProvider::ExecQuery (MethodContext *pMethodContext, CFrameworkQuery& Query, long lFlags) { return (WBEM_E_PROVIDER_NOT_CAPABLE); }
/***************************************************************************** * *
FUNCTION : CRichter_TimeServiceProvider::PutInstance * * DESCRIPTION : PutInstance should be used in provider classes that can * write instance information back to the hardware or * software. For example: Win32_Environment will allow a * PutInstance to create or update an environment variable. * However, a class like MotherboardDevice will not allow * editing of the number of slots, since it is difficult for * a provider to affect that number. * * INPUTS : A pointer to a CInstance object containing key properties. * A long that contains the flags described in * IWbemServices::PutInstanceAsync. * * RETURNS : WBEM_E_PROVIDER_NOT_CAPABLE if PutInstance is not * available * WBEM_E_FAILED if there is an error delivering the instance * WBEM_E_INVALID_PARAMETER if any of the instance properties * are incorrect * WBEM_S_NO_ERROR if instance is properly delivered * * COMMENTS : TO DO: If you don't intend to support writing to your * provider, or are creating a `method only' provider, * remove this method. *
*****/
HRESULT CRichter_TimeServiceProvider::PutInstance ( const CInstance &Instance, long lFlags) { // Use the CInstance Get functions (for example, call // GetCHString(L"Name", sTemp)) against Instance to see the key values // the client requested. return (WBEM_E_PROVIDER_NOT_CAPABLE); }
/***************************************************************************** * *
FUNCTION : CRichter_TimeServiceProvider::DeleteInstance * * DESCRIPTION : DeleteInstance, like PutInstance, actually writes * information to the software or hardware. * For most hardware devices, DeleteInstance should not be * implemented, but for software configuration, * DeleteInstance implementation is plausible. * * INPUTS : A pointer to a CInstance object containing key properties. * A long that contains the flags described in * IWbemServices::DeleteInstanceAsync. * * RETURNS : WBEM_E_PROVIDER_NOT_CAPABLE if DeleteInstance is not * available. * WBEM_E_FAILED if there is an error deleting the instance. * WBEM_E_INVALID_PARAMETER if any of the instance properties * are incorrect. * WBEM_S_NO_ERROR if instance is properly deleted. * * COMMENTS : TO DO: If don't intend to support deleting instances or are * creating a `method only' provider, remove this method. *
*****/
HRESULT CRichter_TimeServiceProvider::DeleteInstance ( const CInstance &Instance, long lFlags ) { // Use the CInstance Get functions (for example, call // GetCHString(L"Name", sTemp)) against Instance to see the key values // the client requested. return (WBEM_E_PROVIDER_NOT_CAPABLE); }
/***************************************************************************** * *
FUNCTION : CRichter_TimeServiceProvider::ExecMethod * * DESCRIPTION : Override this function to provide support for methods. * A method is an entry point for the user of your provider * to request your class perform some function above and * beyond a change of state. (A change of state should be * handled by PutInstance() ) * * INPUTS : A pointer to a CInstance containing the instance the * method was executed against. * A string containing the method name * A pointer to CInstance which contains the IN parameters. * A pointer to CInstance to contain the OUT parameters. * A set of specialized method flags * * RETURNS : WBEM_E_PROVIDER_NOT_CAPABLE if not implemented for this * class * WBEM_S_NO_ERROR if method executes successfully * WBEM_E_FAILED if error occurs executing method * * COMMENTS : TO DO: If you don't intend to support Methods, remove this * method. *
*****/
HRESULT CRichter_TimeServiceProvider::ExecMethod ( const CInstance& Instance, const BSTR bstrMethodName, CInstance *pInParams, CInstance *pOutParams, long lFlags) { // For non-static
```

methods, use the CInstance Get functions (for example, // call GetCHString(L"Name", sTemp)) against Instance to see the key // values the client requested. return (WBEM_E_PROVIDER_NOT_CAPABLE); }

Listing 8-2. *The MOF file for the TimeServiceProvider sample WMI provider*

TimeServiceProvider.mof

// TimeServiceProvider.MOF // // Generated by Microsoft WBEM Code Generation Engine // // TO DO: If this class is intended to be created in a namespace // other than the default (root\default), you should add // the #pragma namespace command here. If these classes // are going into your own namespace, consider creating // the namespace here as well. See CIMWIN32.MOF for an // example of how to create a namespace. Also, consider // combining this mof with the mof the defines the class // that this provider provides. // //========== // NOTE: The line below was added manually: #pragma namespace("\\\\.\\root\\cimv2") //********** //*** Registers Framework Provider *** //********** instance of __Win32Provider as $P { Name = "TimeServiceProvider"; ClsId = "{ccb338cc-f3af-4d80-8bba-d8ba0f8c325d}"; }; instance of __InstanceProviderRegistration { Provider = $P; SupportsGet = TRUE; SupportsPut = TRUE; SupportsDelete = TRUE; SupportsEnumeration = TRUE; QuerySupportLevels = {"WQL:UnarySelect"}; }; instance of __MethodProviderRegistration { Provider = $P; }; // NOTE: The following lines (from TimeServiceStart.mof) were added manually: [provider("TimeServiceProvider"), dynamic] class Richter_TimeServiceProvider : Win32_Service { string PipeName; };

[Previous] [Next]

Permanent Configuration Settings

Many services require keeping permanent configuration settings. (We discussed how to use the registry for this purpose in Chapter 5.) As an alternative, you can create properties in a Win32_Service-derived class. A third option is to use a standard provider that ships with WMI. To use this provider, you just have to write a suitable MOF file.

WMI keeps all of its configuration information, such as the directory location of auto-recover MOF files and start-up heap allocation size, under the following registry key:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\WBEM
```

The following MOF file defines a class derived from the CIM_Setting class. The derived class allows you to use WMI to see these WBEM registry settings.

```
#pragma namespace("\\\\.\\root\\cimv2")
// Instance provider
instance of __Win32Provider as $InstProv
{
    Name    = "RegProv" ;
    ClsId   = "{fe9af5c0-d3b6-11ce-a5b6-00aa00680c3f}" ;
};

instance of __InstanceProviderRegistration
{
```

```
    Provider = $InstProv;
    SupportsPut = TRUE;
    SupportsGet = TRUE;
    SupportsDelete = FALSE;
    SupportsEnumeration = TRUE;
};

[dynamic, provider("RegProv"),
ClassContext("local|hkey_local_Machine\\software\\microsoft\\wbem")]
class Richter_MySettings : CIM_Setting
{
  [key,PropertyContext("CIMOM")] string  SettingId;
  [PropertyContext("Autorecover MOFs")] string AutoRecoverMOFs[];
  [PropertyContext("Startup Heap Preallocation Size")]
    uint32 HeapPreAllocationSize;
};
```

NOTE

These values are already exposed via the Win32_WMISetting class. This MOF file is for demonstration purposes only.

Note two important features of this MOF file. First, the standard registry provider is declared and registered using the "instance of Win32Provider" and "instance of InstanceProviderRegistration" syntax. This declaration and registration might already have been done if a developer used the registry provider in the CIMV2 namespace. Doing it a second time has no ill effect.

Second, the declaration of the settings subclass is handled using the "class Richter_MySettings" statement. Critical to note here is the association of the class with the registry provider by using the provider qualifier, and the association of the property with the registry fields using the PropertyContext qualifiers. The PropertyContext qualifiers indicate which registry values the property values are to be mapped to.

[Previous] [Next]

Chapter 9

Managing Trustees

The developers at Microsoft have done a truly excellent job in designing the security features of Microsoft Windows 2000. These security features are far more sophisticated and flexible than the environments to which most people are accustomed. In fact, with proper administration and software development, Windows 2000 is one of the most secure and advanced operating environments today. With this sophistication, however, come myriad details and intricacies that will daunt most developers new to security under Windows. Although the topic of security represents a challenge for the readers of this book and me, the rewards of rising to the challenge will be secure server software and a real sense of accomplishment.

Windows 2000 has a wealth of features that fall comfortably under the heading of "security," most of which will be covered in this part of the book. In this book, I am dividing the security-related features of Windows into one or more of the following topics:

- **Identity and user context** Much of Windows security is user-based and relies heavily on its ability to identify and maintain a "context" for a user. This ability allows the system to decide whether or not software running on behalf of a user is allowed to perform certain activities.

- **Access control** Access control refers to the securing of private data as well as system objects such as files and registry keys. The access-control features of Windows 2000 rely heavily on identity and user

context. The extent of this reliance will become clearer as our discussion unfolds over the next couple of chapters.

- **Secure connectivity** Communication security is an important facet of Windows 2000. It includes data encryption as well as authentication, both of which assure that you are communicating with a trusted entity.

Because security relies so heavily on establishing and managing user identity, we'll begin our discussion with the use of trustees in Windows 2000.

[Previous] [Next]

What Is a Trustee?

When you log on to a machine running Windows, you typically type in your username and a password. This username identifies an account that represents you to the system. Accounts such as this are generally referred to as *user accounts*.

User accounts maintain a set of information about a user, such as name, password (or credentials), comments, and address. Two critical items stored with a user account are a list of the user's privileges on the system and a digital identity—referred to as a security identifier, or SID (pronounced "sid")—that the system uses to identify the account when securing objects and private data. A SID is a variable-length binary structure that identifies a trustee account. (Privileges and SIDs will be discussed in detail later in this chapter.) A third and similarly important piece of information associated with a user account is a list of groups of which the user is a member.

The developers of Windows implemented group accounts in an effort to manage complexity. A *group account* contains a set of members that can be user accounts or global group accounts. A group account maintains similar information to a user account, such as system privileges and a SID. By simply assigning the user accounts to a group account, the security for a set of users can be managed in one place without having to make adjustments for each user.

User accounts and group accounts are trustees of a system. A *trustee* is an entity to which access can be denied or allowed in Windows. Some trustees exist implicitly in the system (such as the built-in group Everyone), others are created dynamically by the system (such as the logon SID discussed in more detail in Chapters 10 and 11), and still other trustees exist because accounts have been created for them. Users, groups, and computers are all trustees of Windows that have associated *trustee accounts*. All trustees share common functionality throughout the system, especially when dealing with access control (which is covered in the next chapter).

NOTE

User accounts and group accounts are members of a broader category called *trustees*. Because we are dealing primarily with user and group accounts in this chapter, unless otherwise specified, the terms *trustee* and *trustee account* are used to represent user and group accounts.

When access or ownership to a file or some other system object is granted by the system, it is granted via a trustee account. When a privileged function is performed by the system, it is allowed because the privilege was granted to a trustee account. When a user is identified to the system, the user is identified by his or her trustee account and by the trustee accounts of the groups of which the user is a member. So as you can see, the trustee account is a key part of Windows security.

Before we jump into our discussion on programmatically administering user and group accounts, let's take a moment to look at the user interface for administering trustees of the system.

[Previous] [Next]

Security Tools in Windows 2000

To effectively write software that is security aware, it is necessary to have an understanding of how security is administered in Windows 2000 and familiarity with some useful tools.

First, you will need administrative rights (or, in the case of a domain system, account operators' rights) to a Windows 2000 system. If necessary, create a clean installation of Windows 2000 for which you are the administrator.

Now you should become familiar with the Microsoft Management Console (MMC) if you aren't already. This application is installed with the system and resides in MMC.exe. The MMC allows you to perform administrative tasks, from configuring the hard drives on your system to monitoring the event log, including the security administration. Each function of the MMC is implemented as a *snap-in* that must be loaded. For more details on using the MMC, see the topic "Using the Microsoft Management Console" in Windows 2000 Help.

The Local Users and Groups snap-in for the MMC allows you to create users and groups, as well as to add members to groups. This snap-in also allows you to change the password of a user and to set some of the rules that apply to a trustee. Ideally the samples from this book should be run by a user who is a member of the Administrators group (although you will find it educational to create less powerful users and groups so that you can experience life as one of the less privileged).

The Group Policy snap-in allows you to adjust the privileges assigned to a trustee. This snap-in is somewhat more complex than the Local Users and Groups snap-in because of the vast functionality of Group Policy on Windows. Go directly to the privileges functionality of the Group Policy snap-in by expanding the following path: Local Computer Policy\Computer Configuration\ Windows Settings\Security Settings\Local Policies\User Rights Assignment. Figure 9-1 shows how the resultant screen should look.

Figure 9-1. *Privileges shown in the Group Policy snap-in*

NOTE

It is important to understand that the MMC snap-in interface to privileges in Windows 2000 is realized through a logical layer called Group Policy. Group Policy is a complex topic and could fill several chapters on its own. This approach to assigning privileges is a step away from the approach taken by the User Manager tool in Microsoft Windows NT 4.0, which assigns privileges directly to trustees. Direct privilege manipulation is still possible through calls to system functions; however, the user interface provided with Windows 2000 uses only Group Policy.

Using the MMC snap-ins, you might experience latency for the assignment of user rights. In addition, the assignment of user rights might be overridden completely by domain group policies. You can work around these potential problems by using a tool that manages privileges directly.

Finally, before moving on to programmatically administering trustees, I should mention that the TrusteeMan sample application described later in this chapter can also be used as a tool to create and delete user and group accounts, as well as assign privileges to trustee accounts. Although this sample is included as an instructional tool, it also provides a simple alternative to the MMC snap-ins included with Windows 2000.

[Previous] [Next]

Administering Trustee Accounts

In the typical enterprise environment, a system administrator or an account operator creates user and group accounts on the system. Server software running in such environments can operate with active awareness of security and user identification without ever directly creating or managing trustee accounts. The software simply assumes that these accounts have already been created by another entity. As such, it would seem that many service developers would never need to programmatically create trustee accounts.

Although this may be true, you will find that Windows 2000 security is extremely flexible. You can make creative use of trustees that aren't directly associated with a specific user nor would ever be used by a user to log on to a system. For example, Windows 2000 allows you to limit the access of a user account to its original rights combined with the rights of a trustee determined by your software, a technique known as creating a *restricted token.* (Restricted tokens are discussed in detail in Chapter 11.) As such, your software can create accounts that your system administrator would not have the ability (or the desire) to create manually.

Furthermore, if you are writing server software that directly supports creation of an account such as an Internet gaming community or an online banking front end, your service will have to be able to create and manage trustee accounts programmatically. Windows 2000 provides two sets of functions that allow you to do this:

- **Net API** The "Net" functions allow you to create and manage user accounts on any Windows 2000 system, including workstation and server systems, whether or not the system is a domain controller.

- **Active Directory Services Interface (ADSI)** ADSI is a set of COM objects that allows you to administer Active Directory on Windows 2000. Active Directory is the repository for all user and group accounts (among many other objects) on a Windows 2000 domain and is a very important addition to Windows. Your software can use the ADSI objects to create, delete, and otherwise administer user and group objects on a domain controller's Active Directory.

Which set of APIs you use depends on your needs. Active Directory is an enterprise management tool, and as such, the ADSI interface to Active Directory should be used if your software is creating user accounts in an enterprise environment. However, if your service software is creating trustee accounts that are only local to a single server that may not be a domain controller, you might find it much simpler and more direct to use the

Net functions.

NOTE

ADSI objects will allow your software to administer users and groups on systems that are not domain controllers, but because such systems do not maintain an Active Directory, the ADSI objects simply pass through to the Net functions. Similarly, if the Net functions are used to add, remove, or otherwise administer trustees on a domain controller, they will automatically use the ADSI objects to modify Active Directory on that machine.

This chapter focuses on the Net functions, since they are the most commonly used. However, corresponding ADSI components that can be used to perform similar tasks are listed as notes so that you can look up the particular topic in the Platform SDK documentation.

Understanding the Net Functions

The Net API is feature rich and contains many functions and structures that are defined to facilitate the management of trustee accounts. The Net functions allow you to manage user accounts, local group accounts (which only have relevance on a single machine), and domain or global group accounts, which exist on domain controllers and function for the entire domain. These functions are logically and consistently designed, and we can learn a lot by knowing a few rules.

NOTE

If you are writing a source module that will be calling a Net function, you need to take a couple of preliminary steps with your source module and projects. Unlike most API functions in Win32, the Net functions are not declared in a header file that is included in Windows.h. Therefore, you must include the Lm.h header file at the beginning of your module. Similarly, you must add the NetApi32.lib file to the list of library files linked with your project.

One positive feature of the Net functions is that they are laid out in a consistent manner. For this reason, I'm able to provide some general rules for understanding these functions without covering every single function in the API set:

- Net functions are named consistently, using the pattern Net+[Trustee Type]+[Action to Perform]. Examples: NetUserAdd, NetLocalGroupGetInfo, and NetUserSetGroups.
- Net functions support only Unicode strings. If your project is built for ANSI, you will have to make a Unicode copy of each string buffer before passing it to a Net function. (This is just another reason to always use Unicode when writing server software for Windows 2000.)
- Net functions identify trustee accounts by system name and account name. You can use these functions to manage trustees on a remote system, assuming you have sufficient rights on the remote machine.
- Net functions implement sets of structures that are used to report trustee information to your software or that can be used by your software to set trustee information. Because of the many structures involved, this is a potentially confusing feature of the Net functions, though very logical and useful once understood.

Here's an example: If you were using the *NetLocalGroupAdd* function (discussed in detail shortly) to create a local group, you have the option of filling in and passing an instance of the LOCALGROUP_INFO_0 structure, which contains only a group name, or the LOCALGROUP_INFO_1 structure, which allows you to

specify a group name and a comment string. The choice is yours, and each Net function allows you to select the structure to use by passing a parameter called *level*. In our example, LOCALGROUP_INFO_0 is a level 0 structure and LOCALGROUP_INFO_1 is a level 1 structure.

NOTE

Because a single function can take or return more than one structure type defined by the system, the Net functions define the type as PBYTE and rely on you to pass the proper information, using the proper casting. If you are passing a structure to a function, you should pass a pointer to the specific structure and cast it to a PBYTE. If a Net function is returning a structure to you, the function will expect a pointer to a PBYTE, which it fills in, and you can later cast to the appropriate structure type.

In this chapter, I will be referring to sets of structures where I replace the level indicator with an asterisk character (*) so that I can generalize information without detailing each of the many structures available for use. For example, the two structures mentioned in the last paragraph are both in the set of LOCALGROUP_INFO_* structures. I will be defining some of the more common structures that are used, as well as some of the larger ones that include members of the shorter structures. Detailed explanations of all the structures can be found in the Platform SDK documentation.

Table 9-1 lists the structure sets defined for trustee management and the functions with which they can be used.

Table 9-1. *Structure sets for managing trustees*

Structure Set	Use and Functions
GROUP_INFO_*	These structures are used to retrieve information for global or domain groups, as well as to set information for global groups. They are used when creating groups as well as when working with existing groups. **Functions:** *NetGroupAdd*, *NetGroupEnum*, *NetGroupGetInfo*, *NetGroupSetInfo*
GROUP_USERS_INFO_*	These structures are used when managing user membership in global or domain groups. They can be used for setting or getting user information for a group. **Functions:** *NetGroupGetUsers*, *NetGroupSetUsers*, *NetUserGetGroups*, *NetUserSetGroups*
LOCALGROUP_INFO_*	These structures are used to retrieve and set information for local groups. They are used with existing groups as well as when creating new local groups. **Functions:** *NetLocalGroupAdd*, *NetLocalGroupEnum*, *NetLocalGroupGetInfo*, *NetLocalGroupSetInfo*
LOCALGROUP_MEMBERS_INFO_*	These structures are used to set and retrieve member lists for local groups. **Functions:** *NetLocalGroupAddMembers*, *NetLocalGroupDelMembers*, *NetLocalGroupGetMembers*, *NetLocalGroupSetMembers*

LOCALGROUP_USERS_INFO_*	This set of structures contains only a single structure called LOCALGROUP_USERS_INFO_0, which is used in calls to *NetUserGetLocalGroups* to get a list of all the local groups of which a user account is a member. **Functions:** *NetUserGetLocalGroups*
USER_INFO_*	This set of structures contains, by far, the largest number of structures, due to the wealth of information you can associate with a user account in Windows 2000. These structures are used when creating users as well as when getting and setting information for existing user accounts. **Functions:** *NetUserAdd*, *NetUserEnum*, *NetUserSetInfo*, *NetUserGetInfo*

When a Net function is returning information about a trustee to your software, it will allocate a buffer for you. As I indicated before, the Net functions require that you pass the address of a pointer to the appropriate structure type, cast to a PBYTE*. When you are finished with the buffer returned by the system, you should always pass the buffer to *NetApiBufferFree*, which will free the buffer. This function is defined as follows:

```
NET_API_STATUS NetApiBufferFree(PVOID pvBuffer);
```

When setting information for a trustee, a Net function might determine that your software did not properly set one of the members of the structure. In this case, the "Set" function (e.g., *NetUserSetInfo*) will return ERROR_INVALID_PARAMETER. To determine the cause of the error, you can use the function's error parameter that will be filled in with a predefined value indicating the first structure member that caused the error. If you are not interested in this information, you can pass NULL to the "Set" function.

With this general knowledge of the Net functions, you'll find that learning the details is a breeze. So let's start with creating trustee accounts.

Creating Trustee Accounts

User and group accounts are similar in terms of their roles in controlling access to securable objects in the system and privileged functions of the system. The system allows you to assign and deny rights to user and group accounts interchangeably. However, a human user can utilize a user account to log on to a system, whereas a group account cannot be used in this way. This association with a flesh and bone user makes it necessary for the system to maintain a great deal of information for a user account that is not stored for group accounts.

> **NOTE**
>
> Domain or global group accounts have much in common with local group accounts, in concept and in software. Although I'll be discussing the management details of local groups from this point forward, many of the concepts also apply when programmatically administering global groups. For a complete discussion of the global group functionality provided by the Net API, see the Platform SDK documentation for the set of functions with the "NetGroup" prefix.

To create a local group account, you can use the *NetLocalGroupAdd* function, prototyped as follows:

```
NET_API_STATUS NetLocalGroupAdd(
   PCWSTR pstrServername,
```

```
    DWORD  dwLevel,
    PBYTE  pbBuf,
    PDWORD pdwParmErr);
```

The first parameter is the name of the system on which you wish to create the group account. Passing NULL for *pstrServername* indicates that you wish to create a group on the local system. The *dwLevel* parameter indicates the type of structure you will be passing as a reference for the *pbBuf* parameter. *NetLocalGroupAdd* uses the LOCALGROUP_INFO_* set of structures discussed in Table 9-1. You can pass either a 0 or a 1 as the level value for this function. The definitions of both of these structures are as follows:

```
typedef struct _LOCALGROUP_INFO_0 {
   PWSTR   lgrpi0_name;
} LOCALGROUP_INFO_0;

typedef struct _LOCALGROUP_INFO_1 {
   PWSTR   lgrpi0_name;
   PWSTR   lgrpi0_comment;
} LOCALGROUP_INFO_1;
```

Both structures contain a pointer to a string that is used as the group's name, and the LOCALGROUP_INFO_1 structure contains an additional pointer to a comment string. You can use either structure type when creating groups.

The final parameter to pass to *NetLocalGroupAdd* is *pdwParmErr*, which should be the pointer to a DWORD. In the case of a bad parameter, the system returns a value indicating which parameter was bad in the variable referenced in the *pdwParmErr* parameter. The value returned in the *pdwParmErr* is valid only if the return value of *NetLocalGroupAdd* is equal to ERROR_INVALID_PARAMETER. Table 9-2 lists the possible values. If you do not wish to receive this information, you can pass NULL for *pdwParmErr*.

Table 9-2. *Possible values that can be returned in the* pdwParamErr *parameter*

Value	Description
LOCALGROUP_NAME_PARMNUM	This value indicates that you have specified an invalid group name for your new group.
LOCALGROUP_COMMENT_PARMNUM	This value indicates that the comments value was invalid in your call to *NetLocalGroupAdd*.

You should always check the return value of *NetLocalGroupAdd* to insure that the system has returned *NERR_Success*. Otherwise, the system has failed to create a new local group.

> **NOTE**
>
> The ADSI set of components can also be used to create group accounts. Active Directory is organized as a hierarchy of objects, and a group object can be created in any container object. You can create a group object by using the *Create* method of the *IADsContainer* interface. Once you have created the object, you can use the *QueryInterface* method to obtain a pointer to the *IADsGroup* interface, which can be used to further manage your new group object. See the Platform SDK documentation for more details.

Creating a user account is similar to creating a group account; however, you can provide the system with significantly more information for user accounts. You can use the *NetUserAdd* function to create new users on a Windows 2000 system.

```
NET_API_STATUS NetUserAdd(
   PCWSTR pstrServername,
   DWORD  dwLevel,
```

```
    PBYTE  pbBuf,
    PDWORD pdwParmErr);
```

Notice that the parameter list for *NetUserAdd* is exactly the same as that for *NetLocalGroupAdd*. The differences lie in the structure types passed as the *pbBuf* parameter. You can pass a pointer to a level 1, 2, or 3 USER_INFO_* structure to create your new user. Here are the definitions of USER_INFO_1 (the simplest structure that you can use with *NetUserAdd*) and USER_INFO_3 (the most comprehensive structure you can use).

```
typedef struct _USER_INFO_1 {
    PWSTR   usri1_name;
    PWSTR   usri1_password;
    DWORD   usri1_password_age;
    DWORD   usri1_priv;
    PWSTR   usri1_home_dir;
    PWSTR   usri1_comment;
    DWORD   usri1_flags;
    PWSTR   usri1_script_path;
} USER_INFO_1 ;

typedef struct _USER_INFO_3 {
    PWSTR   usri3_name;
    PWSTR   usri3_password;
    DWORD   usri3_password_age;
    DWORD   usri3_priv;
    PWSTR   usri3_home_dir;
    PWSTR   usri3_comment;
    DWORD   usri3_flags;
    PWSTR   usri3_script_path;
    DWORD   usri3_auth_flags;
    PWSTR   usri3_full_name;
    PWSTR   usri3_usr_comment;
    PWSTR   usri3_parms;
    PWSTR   usri3_workstations;
    DWORD   usri3_last_logon;
    DWORD   usri3_last_logoff;
    DWORD   usri3_acct_expires;
    DWORD   usri3_max_storage;
    DWORD   usri3_units_per_week;
    PBYTE   usri3_logon_hours;
    DWORD   usri3_bad_pw_count;
    DWORD   usri3_num_logons;
    PWSTR   usri3_logon_server;
    DWORD   usri3_country_code;
    DWORD   usri3_code_page;
    DWORD   usri3_user_id;
    DWORD   usri3_primary_group_id;
    PWSTR   usri3_profile;
    PWSTR   usri3_home_dir_drive;
    DWORD   usri3_password_expired;
} USER_INFO_3 ;
```

As you can see, you could choose to pass a great deal of information to the system when creating a user. For a detailed description of each member of the USER_INFO_3 structure, as well as the USER_INFO_2 structure, see the Platform SDK documentation. For the purposes of this discussion, it is only necessary to describe the members found in the USER_INFO_1 structure. They are listed in Table 9-3.

Table 9-3. *USER_INFO_1 members*

Member	Description	Value Returned in *pdwParmErr* if ERROR_INVALID_PARAMETER
usri1_name		USER_NAME_PARMNUM

	Points to a buffer containing the Unicode user name for the user account to be created. This member must contain a valid pointer or *NetUserAdd* will return ERROR_INVALID_PARAMETER.	
usri1_password	Points to a buffer containing the Unicode password for the user account to be created. Passwords are limited to 14 characters and must also conform to the rules set forth by the system to which you are adding a user account.	USER_PASSWORD_PARMNUM
usri1_password_age	*NetUserAdd* ignores this member.	USER_PASSWORD_AGE_PARMNUM
usri1_priv	When you create a user, this member must be set to the value USER_PRIV_USER. Any other value will cause *NetUserAdd* to return ERROR_INVALID_PARAMETER.	USER_PRIV_PARMNUM
usri1_home_dir	A user can have a home directory defined for his account. If you wish to associate a home directory with a new user, this member should point to a buffer containing a Unicode string indicating the path to the directory. You can assign NULL to this member.	USER_HOME_DIR_PARMNUM
usri1_comment	This member points to a buffer containing the Unicode comment for the new user. If you do not wish to associate a comment with your new user account, you can assign NULL to this member.	USER_COMMENT_PARMNUM
usri1_flags	This member can take a combination of flags indicating the functionality of the new user account. (See Table 9-4.)	USER_FLAGS_PARMNUM
usri1_script_path	The system will execute the .exe, .cmd, or .bat file associated with your user when she logs on to a system. You can use this member to set the path of this log-on script.	USER_SCRIPT_PATH_PARMNUM

Any combination of the flags listed in Table 9-4 can be assigned to the *usri1_flags* member of USER_INFO_1 when calling *NetUserAdd*. Other flags are available to specify additional features. See the Platform SDK documentation for information on these other flags.

Table 9-4. *USER_INFO_1 flags of interest*

Flag	Description
UF_ACCOUNTDISABLE	Creates an account that is disabled.
UF_PASSWD_NOTREQD	Creates an account that does not require a password. Note that there might be a policy in place for the system or domain that requires all accounts to have passwords.
UF_PASSWD_CANT_CHANGE	Creates an account for which the user is unable to change his password. An administrator of the system can still change the password.
UF_DONT_EXPIRE_PASSWD	The password for this user will never expire.
UF_NOT_DELEGATED	This account cannot be delegated. (See Chapters 11 and 12 for discussions on delegation.)
UF_SMARTCARD_REQUIRED	Requires the user to use a smart card to log on with this new account.
UF_TRUSTED_FOR_DELEGATION	This account can be delegated. (See Chapters 11 and 12 for discussions on delegation.)

Despite what it may seem, the minimum coding requirement to create a new user account using *NetUserAdd* is actually quite reasonable. The following code demonstrates the simplest possible case:

```
BOOL CreateUser(PWSTR pszSystem, PWSTR pszName, PWSTR pszPassword) {
   USER_INFO_1 userInfo = { 0 };

   userInfo.usri1_name = pszName;
   userInfo.usri1_password = pszPassword;
   userInfo.usri1_priv = USER_PRIV_USER;

   NET_API_STATUS netStatus =
      NetUserAdd(pszSystem, 1, (PBYTE) &userInfo, NULL);
   return(netStatus == NERR_Success);
}
```

As you can see, this is fairly trivial code to create a user on a given system with a given name and password. Calling this sample function as follows would create a user named "MrMan" with the password of "HowDoYouDo" on the local system:

```
CreateUser(NULL, L"MrMan", L"HowDoYouDo");
```

NOTE

The ADSI set of components can also be used to create user accounts. Like group objects, user objects can be created in any container object. You can create a user object using the *Create* member of the *IADsContainer* interface. Once you have created the object, you can use the *QueryInterface* method to obtain a pointer to the *IADsUser* interface, which can be used to further manage your new user object. See the Platform SDK documentation for more details on this topic.

Setting User and Group Information

The system also provides Net functions for getting and setting information for a user or group account once it has been created. You will see that the calling style for these functions is similar to that of *NetGroupAdd* and *NetUserAdd*.

To get and set information for a user account, you can use the *NetUserGetInfo* and *NetUserSetInfo* functions prototyped as follows:

```
NET_API_STATUS NetUserGetInfo(
   PCWSTR pszServerName,
   PCWSTR pzUsername,
   DWORD  dwLevel,
   PBYTE  *ppbBuf);

NET_API_STATUS NetUserSetInfo(
   PCWSTR pszServerName,
   PCWSTR pszUsername,
   DWORD  dwLevel,
   PBYTE  pbBuf,
   PDWORD pdwParmErr);
```

As you can see, the *NetUserGetInfo* and *NetUserSetInfo* functions have nearly the exact same parameter list as *NetUserAdd*. In fact, the only additional parameter is *pstrUsername*, which indicates the name of the user for whom to get or set information.

Note also that for *NetUserGetInfo*, the familiar *pbBuf* parameter has been modified to become the address of a pointer to a buffer. As I pointed out earlier, this is because the system actually allocates the requested structure for you and returns a pointer to the new buffer in the variable pointed to by the *ppbBuf* parameter.

The *dwLevel* parameter indicates the level of the USER_INFO_* structure that you will use in your call to *NetUserGetInfo* or *NetUserSetInfo*. These functions accept USER_INFO_1 or USER_INFO_3, as defined above, as well as several other USER_INFO_* structures that *NetUserAdd* does not support. For a full description of these structures, refer to the Platform SDK documentation.

NOTE

When setting user information, you should use a structure level that includes only the information you want to set or accepts NULL as a member value. For example, to set just the password for a user account, use the USER_INFO_1003 structure type because it includes a member only for the user's new password.

The following two example functions demonstrate how to use *NetUserSetInfo* and *NetUserGetInfo* to set and get information for a user account. The first function shows how to set the password for a user account. The second function shows how to retrieve the comment field for a user. The principles used in these functions can be extrapolated to get or set any valid information for a user account.

```
BOOL SetUserPassword(PWSTR pszSystem, PWSTR pszName, PWSTR pszPassword) {
   USER_INFO_1003 userInfo = { 0 };
   userInfo.usri1003_password = pszPassword;
   NET_API_STATUS netStatus =
      NetUserSetInfo(pszSystem, pszName, 1003, (PBYTE) &userInfo, NULL);
   return (netStatus == NERR_Success);
}

BOOL GetUserComment(PWSTR pszSystem, PWSTR pszName, PWSTR pszComment,
   int nBufLen) {
   USER_INFO_10 *puserInfo;
   BOOL fSuccess = FALSE;
   NET_API_STATUS netStatus =
      NetUserGetInfo(pszSystem, pszName, 10, (PBYTE*) &puserInfo);
   if (netStatus == NERR_Success) {
      if (nBufLen > lstrlen(puserInfo->usri10_comment)) {
         lstrcpy(pszComment, puserInfo->usri10_comment);
         fSuccess = TRUE;
      }
```

```
        NetApiBufferFree(puserInfo);
    }
    return(fSuccess);
}
```

NOTE

All successful calls to *NetUserGetInfo* must have a matching call to *NetApiBufferFree* to free the buffer returned by *NetUserGetInfo*. The *NetApiBufferFree* function is defined to take a single parameter, which is the pointer to the buffer to be freed.

As you may already have guessed, the system also provides similar functions for getting and setting information for group trustee accounts. To get information for a trustee, you can call *NetLocalGroupGetInfo*, and to set information for a group, call *NetLocalGroupSetInfo*. These functions are defined as follows:

```
NET_API_STATUS NetLocalGroupGetInfo(
    PCWSTR pservername,
    PCWSTR groupname,
    DWORD  dwLevel,
    PBYTE  *ppbBuf);

NET_API_STATUS NetLocalGroupSetInfo(
    PCWSTR servername,
    PCWSTR groupname,
    DWORD  dwLevel,
    PBYTE  pbBuf,
    PDWORD pdwParmErr);
```

As you can see, calling these functions will be almost identical to calling the functions to get and set information for user accounts. The difference is that you'll be dealing with group information and the LOCALGROUP_INFO_* structures discussed previously.

Although a name and a comment might seem like a minimal amount of information to associate with group accounts, remember that group member information is also stored with group accounts. However, I will defer this topic to a later part of this chapter because it closely relates to the topic of SIDs, which I will be discussing later in the section "Understanding SIDs."

Enumerating Users and Groups

It is often desirable to obtain a list of existing users or groups on a given system. Once again, the Net API provides two similar functions to perform these tasks. You can use *NetLocalGroupEnum* to obtain a list of the groups on a system. Similarly you can use *NetUserEnum* to get a list of user accounts on a system. These functions are defined as follows:

```
NET_API_STATUS NetLocalGroupEnum(
    PCWSTR     pszServerName,
    DWORD      dwLevel,
    PBYTE*     ppbBuf,
    DWORD      dwPrefMaxLen,
    PDWORD     pdwEntriesRead,
    PDWORD     pdwTotalEntries,
    PDWORD_PTR pdwResumeHandle);

NET_API_STATUS NetUserEnum(
    PCWSTR     pszServerName,
    DWORD      dwLevel,
    DWORD      dwFilter,
    PBYTE*     ppbBuf,
    DWORD      dwPrefMaxLen,
```

```
    PDWORD     pdwEntriesRead,
    PDWORD     pdwTotalEntries,
    PDWORD_PTR pdwResumeHandle);
```

These functions are similar, but with a few notable differences. The *NetUserEnum* function will be returning structures from the USER_INFO_* set of structures, while *NetLocalGroupEnum* will be returning structures from the LOCALGROUP_INFO_* set of structures. Additionally, *NetUserEnum* allows you to specify a filter to narrow the scope of the list of user accounts returned. Passing 0 for the filter parameter of *NetUserEnum* indicates no filtering, so any account types can be returned. You will typically want to pass FILTER_NORMAL_ACCOUNT for the filter parameter; however, you could also pass any of the following values listed in Table 9-5.

Table 9-5. *Filter values that can be passed for* NetUserEnum*'s filter parameter*

Value	Meaning
FILTER_NORMAL_ACCOUNT	Returns global user account data on a system
FILTER_TEMP_DUPLICATE_ACCOUNT	Returns local user account data on a domain controller
FILTER_INTERDOMAIN_TRUST_ACCOUNT	Returns domain trust account data on a domain controller
FILTER_WORKSTATION_TRUST_ACCOUNT	Returns member server or workstation account data on a domain controller
FILTER_SERVER_TRUST_ACCOUNT	Returns domain controller account data on a domain controller

Apart from the filter parameter and the structures returned, the use of *NetUserEnum* and *NetLocalGroupEnum* is identical. Like the functions we've looked at earlier, the *pszServerName* parameter indicates the system for which we wish to enumerate trustee accounts. The *dwLevel* parameter indicates the version of the LOCALGROUP_INFO_* or USER_INFO_* structure that we will be associating with the *ppbBuf* parameter.

You should pass the address of a pointer variable for the type of structure requested as the *ppbBuf* parameter. Depending on whether *NetUserEnum* and *NetLocalGroupEnum* are used, the system will allocate a buffer to hold an array of USER_INFO_* or LOCALGROUP_INFO_* structures, where *ppbBuf* is a pointer to the address of the buffer. The buffer will be the accounts enumerated by the function. Remember that like *NetUserGetInfo*, since the system allocates a buffer for you, that buffer must be freed by calling *NetApiBufferFree*.

You should indicate the preferred maximum size of the buffer returned by the function you are calling by passing the size in bytes as the *dwPrefMaxLen* parameter. If you would like the system to allocate as large a buffer as possible, you can pass MAX_PREFERRED_LENGTH. In this case, the system will commonly complete the enumeration in a single call to the function.

You might ask why you would ever choose to limit the size of the buffer returned by a "Net*Enum" function and thereby necessitate multiple calls to the function. There are a few reasons: A small number of accounts is not guaranteed, and a large number of accounts could overflow the largest possible buffer allocated by the system. Additionally, you might want your application to regain control periodically if the enumeration is a lengthy process.

NOTE

Because you can't be sure how much memory the system will be able to allocate, your code should be able to handle a case where multiple calls to the enumeration function are required,

whether or not you selected the MAX_PREFERRED_LENGTH value for the buffer length.

The system returns the number of entries read to the variable pointed to by the *pdwEntriesRead* parameter. The total number of remaining available entries (including those returned) is stored in the variable pointed to by the *pdwTotalEntries* parameter.

The final parameter is an opaque value returned by the system that can be passed back to the system in a subsequent call to an enumeration function to continue receiving account information. You should set the variable pointed to by the *pdwResumeHandle* parameter to 0 on your initial call to the enumeration function, and then you should not modify the returned value.

The enumeration functions will return ERROR_MORE_DATA if they have successfully returned data but there are more accounts to be enumerated. When the last accounts available have been returned, the enumeration functions will return NERR_Success.

The following code demonstrates how to use *NetLocalGroupEnum* to enumerate the groups on the local system and print them to a console window:

```
void PrintLocalGroups() {
   ULONG_PTR lResume = 0;
   ULONG  lTotal = 0;
   ULONG  lReturned = 0;
   ULONG  lIndex = 0;
   NET_API_STATUS netStatus;
   LOCALGROUP_INFO_0* pinfoGroup;

   do {
      netStatus = NetLocalGroupEnum(NULL, 0, (PBYTE*) &pinfoGroup,
         MAX_PREFERRED_LENGTH, &lReturned, &lTotal, &lResume);
      if ((netStatus == ERROR_MORE_DATA) ||
          (netStatus == NERR_Success)) {

         for (lIndex = 0; lIndex < lReturned; lIndex++) {
            wprintf(L"%s\n", pinfoGroup[lIndex].lgrpi0_name);
         }
         NetApiBufferFree(pinfoGroup);
      }
   } while (netStatus == ERROR_MORE_DATA);
}
```

This function would require only minor modifications to make it enumerate the users on a system using *NetUserEnum.*

Destroying Users and Groups

Before switching gears and discussing the system structure for identifying trustee accounts, SIDs, I'd like to wrap up the discussion of user and group management by talking about how to destroy user and group accounts.

The reason I bring up the SID at this point is because it does have some relevance to the destruction of a trustee account. Although the Net functions allow you to deal with trustee accounts in terms of their names, the rest of the system largely ignores the name associated with a trustee account. Instead, the system uses the SID binary value associated with an account to identify an account. What does this have to do with destroying trustee accounts?

If you create a trustee with the name "JClark" and then I log on using this account and create a file, the system maintains that I am the owner of this object. However, if you destroy my user account and then create a new

user account named "JClark," the system will assign a different SID value and therefore not recognize the new account as the owner of the old file object.

That said, you can use the following function to delete a user account:

```
NET_API_STATUS NetUserDel(
   PCWSTR pszServerName,
   PCWSTR pszUsername);
```

As you can see, this simple function takes only a system name and a name for a user account as parameters. The *pszServerName* parameter can be NULL, indicating the local system. This function returns NERR_Success if the function succeeds.

The Net API implements a similar function for deleting groups, which is defined as follows:

```
NET_API_STATUS NetLocalGroupDel(
   PCWSTR pszServerName,
   PCWSTR pszGroupname);
```

NOTE

You can also use the ADSI objects to enumerate and delete user and group accounts, as well as to get and set user and group information. See the Platform SDK documentation for discussion of the *IADsUser* and *IADsGroup* interfaces, as well as the *IDirectorySearch* interface used for searching and enumerating.

Managing Group Membership

Group membership information can be modified in several ways, but first we need to learn a couple of ways to retrieve it. The first method uses the *NetUserGetLocalGroups* function to retrieve a list of groups of which a given user is a member. The second method uses the *NetLocalGroupGetMembers* function to retrieve the set of members who are associated with a single group.

Using the *NetUserGetLocalGroups* Function

The *NetUserGetLocalGroups* function has the following prototype:

```
NET_API_STATUS NetUserGetLocalGroups(
   PCWSTR pszServerName,
   PCWSTR pszUsername,
   DWORD  dwLevel,
   DWORD  dwFlags,
   PBYTE* ppbBuf,
   DWORD  dwPrefMaxLen,
   PDWORD pdwEntriesRead,
   PDWORD pdwTotalEntries);
```

As you can see, this function looks similar to the *NetLocalGroupEnum* and *NetUserEnum* functions, which we have already discussed. In fact, the primary difference is that this function uses the LOCALGROUP_USERS_INFO_* set of structures (of which LOCALGROUP_USERS_INFO_0 exists so far).

The only valid values for *dwFlags* are LG_INCLUDE_INDIRECT, which indicates that *NetUserGetLocalGroups* should also return groups for which the user (indicated by *pszUsername*) is indirectly a member, or 0, which indicates that the function should return only groups of which the user is

directly a member. Indirect membership can happen when *pszUsername* is a member of a global or domain group that is in turn a member of a local group.

You can pass MAX_PREFERRED_LENGTH for the *dwPrefMaxLen* parameter, but either way you must pay attention to the values returned via the *pdwEntriesRead* parameter and the *pdwTotalEntries* parameter to be sure that all possible entries were returned.

Unlike with the previous enumeration functions, there is no way to call *NetUserGetLocalGroups* again to continue enumeration. It is unlikely that you will face a situation in which you will be enumerating more than a couple of dozen groups, whereas enumerating users could feasibly return thousands of account entries.

Using the *NetLocalGroupGetMembers* function

The second way to enumerate membership information is to retrieve the set of members associated with a single group. You should use the *NetLocalGroupGetMembers* function to do this:

```
NET_API_STATUS NetLocalGroupGetMembers(
   PCWSTR     pszServerName,
   PCWSTR     pszLocalGroupName,
   DWORD      dwLevel,
   PBYTE*     ppbBuf,
   DWORD      dwPrefMaxLen,
   PDWORD     pdwEntriesRead,
   PDWORD     pdwTotalEntries,
   PDWORD_PTR pdwResumeHandle);
```

This function takes a server name and the name of a local group as its first parameters. The familiar *dwLevel* parameter indicates which level of the LOCALGROUP_MEMBERS_INFO_* set of structures you want returned via the *ppbBuf* parameter.

The LOCALGROUP_MEMBERS_INFO_* structures allow you to deal with your trustees in terms of their textual names and domain names, or in terms of their SIDs, which are covered in detail in the next section. Here are the definitions for the LOCALGROUP_MEMBERS_INFO_0 and LOCALGROUP_MEMBERS_INFO_3 structures:

```
typedef struct _LOCALGROUP_MEMBERS_INFO_0 {
   PSID   lgrmi0_sid;
} LOCALGROUP_MEMBERS_INFO_0;

typedef struct _LOCALGROUP_MEMBERS_INFO_3 {
   PWSTR   lgrmi3_domainandname;
} LOCALGROUP_MEMBERS_INFO_3;
```

NOTE

It is important to know that many security functions require SID values when dealing with trustees. Also, the local group member manipulation functions allow you to retrieve trustee entries in terms of SIDs.

The PSID type used in LOCALGROUP_MEMBERS_INFO_0 indicates a pointer to a SID structure. Remember that either of these structures can be used with *NetLocalGroupGetMembers* and should depend on the needs of your software.

The *dwPrefMaxLen*, *pdwEntriesRead*, *pdwTotalEntries*, and *pdwResumeHandle* parameters are the values for preferred buffer size, entries read, remaining entries, and resume enumeration. These parameters work in exactly the same way as the parameters of the same name for the *NetUserEnum* and *NetLocalGroupEnum* functions already discussed. In fact, the sample *PrintLocalGroups* function from the section on

NetLocalGroupEnum can easily be modified as follows for use with *NetLocalGroupGetMembers*:

```
void PrintLocalGroupMembers(WCHAR *pszGroup) {
   ULONG_PTR lResume = 0;
   ULONG  lTotal = 0;
   ULONG  lReturned = 0;
   ULONG  lIndex = 0;
   NET_API_STATUS netStatus;
   LOCALGROUP_MEMBERS_INFO_3* pinfoMembers;

   do {
      netStatus = NetLocalGroupGetMembers(NULL, pszGroup, 3,
         (PBYTE*) &pinfoMembers, MAX_PREFERRED_LENGTH,
         &lReturned, &lTotal, &lResume);
      if ((netStatus == ERROR_MORE_DATA) ||
          (netStatus == NERR_Success)) {

         for (lIndex = 0; lIndex < lReturned; lIndex++) {
            wprintf(L"%s\n",
               pinfoMembers[lIndex].lgrmi3_domainandname);
         }
         NetApiBufferFree(pinfoMembers);
      }
   } while (netStatus == ERROR_MORE_DATA);
}
```

To set the members of a local group, use the *NetLocalGroupSetMembers* function:

```
NET_API_STATUS NetLocalGroupSetMembers(
   PCWSTR pszServerName,
   PCWSTR pszGroupName,
   DWORD  dwLevel,
   PBYTE  pbBuf,
   DWORD  dwTotalEntries);
```

The *pszServerName* and *pszGroupName* parameters indicate the system name and local group for which you are setting the members. The *dwLevel* parameter indicates which level of the LOCALGROUP_MEMBERS_INFO_* set of structures you wish to use. You can pass a 0 or a 3, indicating LOCALGROUP_MEMBERS_INFO_0 or LOCALGROUP_MEMBERS_INFO_3, respectively, both of which are defined above. This means that you can choose to set membership in terms of trustee account name strings or SIDs.

You should pass a pointer to an array of the selected structure type as the *pbBuf* parameter, and the number of entries in the array as the *dwTotalEntries* parameter. If the function succeeds, the list of trustees represented by the array passed as *pbBuf* will be the new member list for the group. Remember that this list of trustees will replace any current members of the group. If the function succeeds, it will return NERR_Success.

Other Useful Functions

Now you know how to retrieve and set all the member trustees of a local group. I'd like to point out two useful functions that allow you to add only a specific list of trustees to and delete only a specific list of trustees from the current membership of a group. These functions are *NetLocalGroupAddMembers* and *NetLocalGroupDelMembers*:

```
NET_API_STATUS NetLocalGroupAddMembers(
   PCWSTR pszServerName,
   PCWSTR pszGroupName,
   DWORD  dwLevel,
   PBYTE  pbBuf,
   DWORD  dwTotalEntries);
```

```
NET_API_STATUS NetLocalGroupDelMembers(
   PCWSTR pszServerName,
   PCWSTR pszGroupName,
   DWORD  dwLevel,
   PBYTE  pbBuf,
   DWORD  dwTotalEntries);
```

Use these functions in exactly the same way as *NetLocalGroupSetMembers*. For examples, see the TrusteeMan sample application described later in this chapter.

[Previous] [Next]

Understanding SIDs

SIDs are integral to Windows 2000 security. SIDs have already been touched on in this chapter and will appear again in subsequent chapters, but now is the time to look at them in detail. Here is what you know so far: SIDs are security identifiers, which identify trustees to the system. A SID is a unique binary representation of a trustee.

Here are a few common tasks you'll become familiar with:

- Converting a user's or group's textual name to a SID.
- Converting a SID to the textual name for the trustee.
- Building a well-known SID from scratch. These SIDs exist on all systems and indicate accounts such as Guest or Everybody.
- Copying a SID structure.
- And finally, converting to and from binary and string representations of a SID.

A SID is a security identifier, a structure that identifies a trustee to the system. A SID is composed of a 48-bit value followed by a variable number of 32-bit components. In documentation, a SID is typically represented in the SRI-S-S… format. (See the sidebar.)

SID Format

A SID is typically formatted as: S-R-I-S-S…

where:

S is written literally as "S" and indicates that this series of numbers is a SID.

R is the SID's revision level, indicated by a number (currently 1).

I is a 48I-bit number indicating the authority.

S is a 32-bit number indicating a subauthority, also known as a relative identifier, or RID.

S is another subauthority. There can be any number of subauthorities in a SID.

Example : S-1-5-11

The revision number of the SID structure has been 1 since Windows NT 3.1; however, it is possible that the structure will change and we will one day see a revision level of 2. The authority "number" indicates under whose authority the SID was created—that is, what "entity" owns the SID and manages its account. Table 9-6 provides a list of currently supported authorities.

Table 9-6. *SID authorities*

Authority	Definition
SECURITY_NULL_SID_AUTHORITY	Defined as 0, this authority is used in building a SID indicating a null group or nobody.
SECURITY_WORLD_SID_AUTHORITY	Defined as 1, this authority is used in building the well-known group account Everybody.
SECURITY_LOCAL_SID_AUTHORITY	Defined as 2, this authority is used in building the well-known group account LOCAL.
SECURITY_CREATOR_SID_AUTHORITY	Defined as 3, this authority is used with the well-known Creator Owner SID.
SECURITY_NON_UNIQUE_AUTHORITY	Defined as 4, this authority is used in creating SIDs that are not unique. The first subauthority is always set to SECURITY_NT_NON_UNIQUE, which is defined as 0x15.
SECURITY_NT_AUTHORITY	Defined as 5, this authority is used by user and group accounts created by Windows 2000 systems.

The subauthority is a 32-bit value that is unique relative to the authority of the SID. The next subauthority is unique relative to the previous subauthorities and finally the authority of the SID. Subauthorities are also known as RIDs.

Table 9-7 shows some well-known RIDs. (For a complete list, see WinNT.h in the Platform SDK documentation.) Remember that unlike the authorities listed previously, these RIDs are not all unique. This is because they are only unique relative to an authority and, as such, have no meaning independent of the authority.

Table 9-7. *Some well-known RIDs*

RID	Relevant Authority	Definition
SECURITY_NULL_RID	NULL	Defined as 0, this is the single subauthority for the well-known group NULL.
SECURITY_WORLD_RID	WORLD	Defined as 0, this is the single subauthority for the well-known group Everyone.
SECURITY_CREATOR_OWNER_RID	CREATOR	Defined as 0, this is the single subauthority for the well-known user Creator Owner.
SECURITY_CREATOR_GROUP_RID	CREATOR	

		Defined as 1, this is the single subauthority for the well-known user Creator Group.
SECURITY_CREATOR_OWNER_SERVER_RID	CREATOR	Defined as 2, this is the single subauthority for the well-known user Creator Owner Server.
SECURITY_CREATOR_GROUP_SERVER_RID	CREATOR	Defined as 3, this is the single subauthority for the well-known user Creator Group Server.

By combining a well-known RID with its respective authority, you can create a well-known SID. Well-known SIDs identify users and groups that are defined by the system and are known by every installation of Windows 2000 on every network. Table 9-8 lists some well-known SIDs and their uses.

Table 9-8. *Well-known SIDs and their uses*

SID	Common Name	Use
S-1-0-0	Null SID	Indicates an empty or NULL group. It is defined as a group with no users and is typically used to indicate nobody.
S-1-1-0	Everyone	Indicates a group of which all trustees are implicitly members. This is a very important SID and can be useful in creating access lists for securable objects in the system. It is also known as the World SID or World group.
S-1-2-0	Local SID	Indicates a group that includes all users who log on to a system locally or physically.
S-1-3-0	Creator Owner SID	Acts as a placeholder for the creator of an object. It is used with inheritable access-control lists. You will find more information on this SID in Chapter 10.
S-1-3-1	Creator Group SID	Acts as a placeholder for the primary group of the creator of an object. It is used with inheritable access-control lists. You will find more information on this SID in Chapter 10.
S-1-5-1	Dialup	Indicates a group of which all user accounts are automatically made members while they are logged on to a Windows 2000 system via dialup.
S-1-5-2	Network	Indicates a group of which all user accounts are automatically made members while they are logged on to a Windows 2000 system via the network.
S-1-5-3	Batch	Indicates a group of which all user accounts are automatically made members while they are logged on to a Windows 2000 system via a batch logon.
S-1-5-4	Interactive	Indicates a group of which all user accounts are automatically made members while they are logged on to a Windows 2000 system via an interactive logon.
S-1-5-6	Service	Indicates a group of which all user accounts are automatically made members while they are logged on to a Windows 2000 system as a service.

S-1-5-7	AnonymousLogon	Associated with a null session logon.
S-1-5-9	ServerLogon	Associated with a domain controller account.
S-1-5-10	Self (or Principal Self)	Acts as a placeholder and applies only to the access lists of group or user accounts. When present, it indicates the trustee account for which the access list applies.
S-1-5-11	Authenticated User	Indicates a well-known group of which all currently authenticated user accounts are implicitly members.
S-1-5-13	Terminal Server	Associated with a user logged on to a terminal server.
S-1-5-18	LocalSystem	This special account, which many service processes run under, exists on all Windows 2000 systems. For more information, see Chapter 11.

You will see some of these well-known SIDs only occasionally, and others crop up quite often. It is good to be familiar with these built-in trustees of the system.

The SID data structure itself is defined as follows:

```
typedef struct _SID {
   BYTE  Revision;
   BYTE  SubAuthorityCount;
   SID_IDENTIFIER_AUTHORITY IdentifierAuthority;
   DWORD SubAuthority[ANYSIZE_ARRAY];
} SID;
```

Each member of this structure should now be familiar to you. ANYSIZE_ARRAY is defined as 1, primarily to indicate that the structure does not necessarily end with only a single DWORD representing a subauthority. Although the SID structure itself is very clear, it is an "opaque" data structure and should be manipulated only by using the system functions provided. This opacity affords the developers at Microsoft the freedom to change the internal structure of the SID in the future. Your software should follow this rule to a fault.

Building SIDs

As a rule, you will be building SIDs for well-known trustees such as the Everyone group. You build a SID by looking up the authority and trustee values for the trustee in question and passing them to the *AllocateAndInitializeSid* function:

```
BOOL AllocateAndInitializeSid(
   PSID_IDENTIFIER_AUTHORITY pIdentifierAuthority,
   BYTE  nSubAuthorityCount,
   DWORD dwSubAuthority0,
   DWORD dwSubAuthority1,
   DWORD dwSubAuthority2,
   DWORD dwSubAuthority3,
   DWORD dwSubAuthority4,
   DWORD dwSubAuthority5,
   DWORD dwSubAuthority6,
   DWORD dwSubAuthority7,
   PSID* ppSid);
```

You should recognize the purpose of each of these parameters, but some points are worth mentioning. The *pIdentifierAuthority* parameter identifies the authority value for the SID you are building. The *pIdentifierAuthority* parameter is of type SID_IDENTIFIER_AUTHORITY, which is defined as an array of 6 bytes. This might seem a bit awkward, but thankfully the Platform SDK documentation defines the useful SID authorities. You should pass one of the values shown in Table 9-6, such as SECURITY_NT_AUTHORITY.

The second parameter, *nSubAuthorityCount*, indicates the number of subauthorities that you require for the SID. Although the system has defined the value SID_MAX_SUB_AUTHORITIES as 15, *AllocateAndInitializeSid* will only build SIDs with 8 or fewer subauthorities.

> **NOTE**
>
> Regardless of the limitations of *AllocateAndInitializeSid*, you should continue to treat SIDs as though they can be any size now and in the future (that is, avoid static buffers).

The next eight parameters indicate the subauthorities for your new SID. I've often wondered why the developers at Microsoft didn't choose to consolidate these parameters into a single pointer to an array of DWORDs. Regardless, you should pass relevant values for as many subauthorities as you need and pass 0 for the remaining parameters.

The final parameter, *ppSid*, is worthy of extra notice. It is defined as PSID*, which is a pointer to a pointer to a SID. Unlike most functions provided by Windows, *AllocateAndInitializeSid* allocates a buffer for you, rather than expecting you to provide a buffer of sufficient length. The address of the allocated memory is returned via this parameter.

If *AllocateAndInitializeSid* returns FALSE, most likely you have passed a subauthority count larger than 8. Because the function allocates memory for you, another cause of failure can be low memory.

If *AllocateAndInitilizeSid* succeeds, the PSID variable will contain a pointer to your new SID. When you are finished with the SID, you should pass it to *FreeSid* so that the system can free the memory used by the SID:

```
PVOID FreeSid(PSID pSid);
```

Using *AllocateAndInitializeSid*, you can build well-known SIDs as well as dynamic SIDs for which your software knows the authority and subauthority values. The following example shows how to use this function to build a SID representing the well-known group Everyone:

```
PSID BuildEveryoneSid() {
   SID_IDENTIFIER_AUTHORITY auth = SECURITY_WORLD_SID_AUTHORITY;
   PSID pSID = NULL;
   BOOL fSuccess = AllocateAndInitializeSid(&auth, 1,
      SECURITY_WORLD_RID, 0, 0, 0, 0, 0, 0, 0, &pSID);
   return(fSuccess ? pSID : NULL);
}
```

Once you are finished with the SID returned by this function you would naturally pass its pointer to the *FreeSid* function. Notice that you can easily modify this function to build another well-known SID by changing the initialization value of the *auth* variable and passing a different RID define to *AllocateAndInitializeSid*.

Trustee Name and Binary SID Conversion

Two useful tasks you will commonly need to perform are conversions to and from trustee account names and binary SIDs. The system provides two functions to perform these tasks. These functions have potentially confusing names, so here is the best way to remember which is which:

- If you have the account name of a trustee but you want a binary SID, call *LookupAccountName*.
- Conversely, if you have a binary SID but you want the trustee account name, call *LookupAccountSid*.

As you can see, the functions have the name of the information you already have.

Retrieving a Trustee's Binary SID

LookupAccountName is prototyped as follows:

```
BOOL LookupAccountName(
   PCTSTR        pszSystemName,
   PCTSTR        pszAccountName,
   PSID          pSid,
   PDWORD        pcbSid,
   PTSTR         pszReferencedDomainName,
   PDWORD        pcbReferencedDomainName,
   PSID_NAME_USE peUse);
```

The *pszSystemName* parameter should be the name of the system on which you wish to look up the name specified in the *pszAccountName* parameter. You can pass NULL for the *pszSystemName* parameter to indicate the local machine.

You should pass a pointer to a buffer in memory large enough to hold the requested SID as the *pSid* parameter, as well as the address of a DWORD containing the size of your buffer as the *pcbSid* parameter. If your buffer is too small, the system will return failure and the *GetLastError* function will return ERROR_INSUFFICIENT_BUFFER. *LookupAccountName* will also fill the DWORD variable you supplied via the *pcbSid* parameter with the size the buffer should be.

The *pszReferencedDomainName* and *pcbReferencedDomainName* work similarly for a buffer to receive the domain name associated with the account and for the pointer to a DWORD variable containing the size, respectively. You cannot pass NULL for either of these values, even if the trustee's associated domain name is not required.

You should pass the address of a SID_NAME_USE variable as the *peUse* parameter. *LookupAccountName* will return a value, in the variable pointed to by *peUse*, indicating the type of SID it has returned to you. Table 9-9 lists the possible values.

Table 9-9. *SID_NAME_USE enumeration type values*

Value	Use
SidTypeUser	Indicates a SID for a user trustee account
SidTypeGroup	Indicates a SID for a group trustee account
SidTypeDomain	Indicates a SID representing a domain object
SidTypeAlias	Indicates a SID representing a built-in group such as the Administrators group
SidTypeWellKnownGroup	Indicates a SID for a well-known group such as the Everyone group
SidTypeDeletedAccount	Indicates a SID for a deleted account
SidTypeInvalid	Indicates an invalid SID
SidTypeUnknown	Indicates an unknown SID type
SidTypeComputer	Indicates a SID for a computer account on the domain

Although *LookupAccountName* takes a computer name as a parameter, its search is not limited to the machine indicated. The search should be thought of as being performed from the perspective of the machine indicated. That said, the following list shows the search order for a trustee:

1. Well-known SID names

2. Built-in and defined accounts on the local machine

3. The primary domain of the system

4. Trusted domains

5. Any domain in a domain forest

NOTE

Although you can use *LookupAccountName* to find well-known SIDs, you should always use *AllocateAndInitializeSid* to directly build well-known SIDs.

As you can see, the namespace for trustees is shared among users, groups, computers, and domains. Although the system will not allow you to create a user or a group with the same name as another user or group, the system does allow a user or a group to have the same name as a computer or a domain. This creates potential problems, because *LookupAccountName* will return the SID for a computer before it returns the SID for a user.

Your software must account for the possibility of a user or a group having the same name as a computer or a domain. The best way to deal with this is to explicitly indicate the computer name for the account as part of the account name. For example, if both a computer and a user are named "JClark", and the "JClark" account exists on the "JClark" computer, the following code will return a SID for the user account:

```
BOOL fRet = LookupAccountName(NULL, "JClark\\JClark",
   &sid, &dwSizeSid, szBuffer, &dwSizeBuf, &use);
```

Retrieving a Trustee's Account Name

If your software needs to find the account name for a trustee when it already has a binary SID, it should use *LookupAccountSid*:

```
BOOL LookupAccountSid(
   PCTSTR        pszSystemName,
   PSID          pSid,
   PTSTR         pszName,
   PDWORD        pcbName,
   PTSTR         pszReferencedDomainName,
   PDWORD        pcbReferencedDomainName,
   PSID_NAME_USE peUse);
```

The *pszSystemName* parameter is the name of the machine from which you perform the search. When *LookupAccountSid* searches for a trustee name, it follows the same search order and rules as does *LookupAccountName*.

The *pSid* parameter is a pointer to an existing SID for which you want to retrieve the corresponding account name. This SID can be any valid SID built by you or previously returned by the system in a token or some other structure.

The *pszName* and *pszReferencedDomainName* should point to buffers that you provide to receive the trustee name and its domain name from the system. These buffers should be large enough to hold the return values; their sizes are passed via pointers to DWORD variables, where *pcbName* indicates the length of the buffer pointed to by *pszName,* and *pcbReferencedDomainName* indicates the length of the buffer pointed to by *pszReferencedDomainName*. If you do not pass buffers of sufficient size to receive these names from the system, the system returns the required buffer sizes in these two variables.

You should pass the address of a SID_NAME_USE variable for *peUse* to receive from the system the use for the SID. For an explanation of the SID_NAME_USE enumeration type, see Table 9-9.

Copying SIDs

Many system functions return SIDs, and likewise many security-related functions expect you to build structures including SIDs or pointers to SIDs. This wouldn't be a problem except that SID structures are variable in length and the system designers have asked us to treat the structure as opaque. Luckily, the system provides a function to copy a SID:

```
BOOL CopySid(
   DWORD dwDestinationSidLength,
   PSID  pSidDestination,
   PSID  pSidSource);
```

This easy-to-use function simply takes the length of your buffer (in bytes), a pointer to a destination SID, and a pointer to the original SID.

Although the system knows the length of the SID that it is copying, it still needs to be sure that it does not write past the length of your buffer. This is why *CopySid* requires that you pass a length as the first parameter. More importantly, you will need to know how large a buffer to allocate. You can find the length of a SID in bytes using the *GetLengthSid* function:

```
DWORD GetLengthSid(PSID pSid);
```

Using the value returned from this function, you can allocate a buffer for your new SID.

Textual and Binary SID Conversion

At this point, you know how to perform the most commonly needed tasks concerning SIDs. However, our discussion of SIDs wouldn't be complete unless I told you how to convert a textual representation of a SID to a binary representation and vice versa.

Before proceeding, let me clarify what I mean by "textual SID." I am *not* referring to the string representing the account name of the trustee; rather, I am referring to the string representing the binary SID structure. For example, "S-1-1-0" is a textual SID representing the SID whose trustee account name is Everyone.

Textual SIDs can be useful when storing SID names in persistent storage, such as the system registry, or when representing a SID in a user interface. The functions you use to convert to and from textual SIDs are *ConvertSidToStringSid* and *ConvertStringSidToSid*, respectively. Here are their prototypes:

```
BOOL ConvertSidToStringSid(
   PSID   pSid,
   PTSTR* StringSid);

BOOL ConvertStringSidToSid(
   PCTSTR StringSid,
   PSID*  ppSid);
```

Both of these functions allocate buffers for you, and it is your job to free them when you are finished with the returned data. Use the *LocalFree* function to free the buffers returned by these functions.

Now that you have an exhaustive understanding of the system's binary identification for trustee accounts, you have all the information necessary to exploit both methods of manipulating membership of local groups as described earlier in this chapter. So let's begin our discussion on assigning and removing privileges for trustee

accounts.

[Previous] [Next]

Understanding Privileges and Account Rights

A privilege is a right assigned to a trustee that has system-wide ramifications. (In contrast, an access right assigned to a trustee defines access allowed to a specific object in the system.) Examples of privileges include the rights to log on locally, back up files and directories, and shut down the system.

Another way of looking at a privilege is as a way to secure a system function. For example, certain system functions, such as *LogonUser* and *InitiateSystemShutdown*, require that the calling software have the proper privileges or else the function will fail.

Privileges have two textual representations you should be aware of. These representations are the *friendly name* that shows up in the user interface for Windows and the *programmatic name* used by your software. Each of the programmatic names is defined in a header file in the SDK.

Table 9-10 lists all the privileges available in Windows 2000 at the time of this book's printing. A partial list is given in the Platform SDK documentation.

Table 9-10. *Privileges and account rights*

Programmatic Name	Display Name	Header File
SeAssignPrimaryTokenPrivilege SE_ASSIGNPRIMARYTOKEN_NAME	Replace a process level token	WinNT.h
SeAuditPrivilege SE_AUDIT_NAME	Generate security audits	WinNT.h
SeBackupPrivilege SE_BACKUP_NAME	Back up files and directories	WinNT.h
SeChangeNotifyPrivilege SE_CHANGE_NOTIFY_NAME	Bypass traverse checking	WinNT.h
SeCreatePagefilePrivilege SE_CREATE_PAGEFILE_NAME	Create a pagefile	WinNT.h
SeCreatePermanentPrivilege SE_CREATE_PERMANENT_NAME	Create permanent shared objects	WinNT.h
SeCreateTokenPrivilege SE_CREATE_TOKEN_NAME	Create a token object	WinNT.h
SeDebugPrivilege SE_DEBUG_NAME	Debug programs	WinNT.h
SeEnableDelegationPrivilege SE_ENABLE_DELEGATION_NAME	Enable computer and user accounts to be trusted for delegation	WinNT.h
SeIncreaseBasePriorityPrivilege SE_INC_BASE_PRIORITY_NAME	Increase scheduling priority	WinNT.h
	Increase quotas	WinNT.h

SeIncreaseQuotaPrivilege SE_INCREASE_QUOTA_NAME		
SeLoadDriverPrivilege SE_LOAD_DRIVER_NAME	Load and unload device drivers	WinNT.h
SeLockMemoryPrivilege SE_LOCK_MEMORY_NAME	Lock pages in memory	WinNT.h
SeMachineAccountPrivilege SE_MACHINE_ACCOUNT_NAME	Add workstations to domain	WinNT.h
SeProfileSingleProcessPrivilege SE_PROF_SINGLE_PROCESS_NAME	Profile single process	WinNT.h
SeRemoteShutdownPrivilege SE_REMOTE_SHUTDOWN_NAME	Force shutdown from a remote system	WinNT.h
SeRestorePrivilege SE_RESTORE_NAME	Restore files and directories	WinNT.h
SeSecurityPrivilege SE_SECURITY_NAME	Manage auditing and security log	WinNT.h
SeShutdownPrivilege SE_SHUTDOWN_NAME	Shut down the system	WinNT.h
SeSyncAgentPrivilege SE_SYNC_AGENT_NAME	Synchronize directory service data	WinNT.h
SeSystemEnvironmentPrivilege SE_SYSTEM_ENVIRONMENT_NAME	Modify firmware environment values	WinNT.h
SeSystemProfilePrivilege SE_SYSTEM_PROFILE_NAME	Profile system performance	WinNT.h
SeSystemtimePrivilege SE_SYSTEMTIME_NAME	Change the system time	WinNT.h
SeTakeOwnershipPrivilege SE_TAKE_OWNERSHIP_NAME	Take ownership of files or other objects	WinNT.h
SeTcbPrivilege SE_TCB_NAME	Act as part of the operating system	WinNT.h
SeUndockPrivilege SE_UNDOCK_NAME	Remove computer from docking station	WinNT.h
SeUnsolicitedInputPrivilege SE_UNSOLICITED_INPUT_NAME	Receive unsolicited device input	WinNT.h
SeBatchLogonRight SE_BATCH_LOGON_NAME	Log on as a batch job	NTSecAPI.h
SeDenyBatchLogonRight SE_DENY_BATCH_LOGON_NAME	Deny logon as a batch job	NTSecAPI.h
SeDenyInteractiveLogonRight SE_DENY_INTERACTIVE_LOGON_NAME	Deny logon locally	NTSecAPI.h
		NTSecAPI.h

SeDenyNetworkLogonRight SE_DENY_NETWORK_LOGON_NAME	Deny access to this computer from the network	
SeDenyServiceLogonRight SE_DENY_SERVICE_LOGON_NAME	Deny logon as a service	NTSecAPI.h
SeInteractiveLogonRight SE_INTERACTIVE_LOGON_NAME	Log on locally	NTSecAPI.h
SeNetworkLogonRight SE_NETWORK_LOGON_NAME	Access this computer from the network	NTSecAPI.h
SeServiceLogonRight SE_SERVICE_LOGON_NAME	Log on as a service	NTSecAPI.h

The system actually distinguishes between the privilege and its close cousin, the account right, but they are assigned and revoked identically. In Table 9-10, the privileges are defined in WinNT.h and the account rights are defined in NTSecAPI.h.

Now that you have some familiarity with what privileges are and how the system represents them, let's dive into assigning and revoking privileges on a system.

The LSA Functions

To assign privileges programmatically, you need to become familiar with a set of functions known as the LSA functions. LSA stands for Local Security Authority, which handles user logon and authentication on the local machine. The LSA functions can be used to do a number of tasks, of which we will only be concerned with privilege assignment in this chapter.The first step in using the LSA functions is to retrieve a handle to a *policy object*. A policy object represents the system on which you will be performing account management functions. To obtain a handle to a policy object, you must call *LsaOpenPolicy*.

```
NTSTATUS LsaOpenPolicy(
   PLSA_UNICODE_STRING    plsastrSystemName,
   PLSA_OBJECT_ATTRIBUTES pObjectAttributes,
   ACCESS_MASK            DesiredAccess,
   PLSA_HANDLE            pPolicyHandle);
```

This function might look a bit strange at first, because the data types aren't commonly used in other Win32 functions. So let's examine them one by one.

The *plsastrSystemName* parameter is a pointer to an LSA_UNICODE_STRING structure, which is defined as follows:

```
typedef struct _LSA_UNICODE_STRING {
   USHORT Length;
   USHORT MaximumLength;
   PWSTR  Buffer;
} LSA_UNICODE_STRING
```

Like the Net functions, the LSA functions deal only with strings in Unicode format. However, unlike the Net functions, the LSA functions require that all strings be managed in terms of the LSA_UNICODE_STRING structure, which is simply a string length, a buffer length, and a pointer to a buffer.

NOTE

It is important to remember that the *Length* (indicating the length of the string) and *MaximumLength* (indicating the length of the buffer) members are stored in terms of bytes,

not characters.

The string pointed to by the *Buffer* member of the LSA_UNICODE_STRING structure is not explicitly defined to be zero-terminated. Although you can choose to end a string with a zero, you must be sure not to include the null termination in your calculation of the length of the string. Also keep in mind that when the system returns an LSA_UNICODE_STRING structure to your software, it cannot count on zero termination of the string pointed to by the *Buffer* member.

Dealing with the LSA_UNICODE_STRING type can be a little awkward because it is length-delimited rather than zero-terminated. Because of this, I think that the LSA_UNICODE_STRING virtually begs to be wrapped in a simple C++ class. I've made a class called CLSAStr, which does this, and use it liberally in the TrusteeMan sample application described later in this chapter.

To call *LsaOpenPolicy*, initialize an instance of the LSA_UNICODE_STRING structure and point it to a Unicode string containing the name of the system for which you wish to manipulate privileges. Passing NULL as the *plsastrSystemName* parameter indicates the local machine.

The *pObjectAttributes* parameter points to an instance of the LSA_OBJECT_ATTRIBUTES structure, which is unused at this time and should have each of its members initialized to 0.

The *DesiredAccess* parameter is declared as type ACCESS_MASK, which maps to a 32-bit unsigned integer. If you will be viewing privilege information on your system, combine POLICY_VIEW_LOCAL_INFORMATION and POLICY_LOOKUP_NAMES as in the example that follows this discussion. If you will be setting privilege information, you should also combine the POLICY_CREATE_ACCOUNT access flag.

The final parameter of *LsaOpenPolicy* requires that you pass the address of a variable of type LSA_HANDLE. Upon success, the system will place the handle to the open LSA policy object in this variable. Once you have finished with the policy object, you should pass this handle to the *LsaClose* function:

```
NTSTATUS LsaClose(LSA_HANDLE hObjectHandle);
```

All LSA functions return a value of type NTSTATUS, and you should pass this value to *LsaNtStatusToWinError*:

```
ULONG LsaNtStatusToWinError(NTSTATUS Status);
```

The *LsaNtStatusToWinError* function converts the NT status code to a familiar Win32 error code just like those returned from *GetLastError*. After conversion, a successful call to *LsaOpenPolicy* returns a value of ERROR_SUCCESS.

The following code shows the proper way to open an LSA policy object for a system on the network whose name is "JasonsComputer":

```
LSA_OBJECT_ATTRIBUTES   lsaOA = { 0 };
LSA_UNICODE_STRING      lsastrComputer = { 0 };
LSA_HANDLE              hPolicy = NULL;

// Computer Name
WCHAR* pstrName = L"JasonsComputer";

// Set the size of the useless LSA_OBJECT_ATTRIBUTES structure
lsaOA.Length = sizeof(lsaOA);

// Fill in the members of the LSA_UNICODE_STRING structure
lsastrComputer.Length = (USHORT) (lstrlen(pstrName) * sizeof(WCHAR));
lsastrComputer.MaximumLength = lsastrComputer.Length + sizeof(WCHAR);
lsastrComputer.Buffer = pstrName;
```

```
// Retrieve the policy handle
NTSTATUS ntStatus = LsaOpenPolicy(&lsastrComputer, &lsaOA,
   POLICY_VIEW_LOCAL_INFORMATION | POLICY_LOOKUP_NAMES |
   POLICY_CREATE_ACCOUNT, &hPolicy);

ULONG lErr = LsaNtStatusToWinError(ntStatus);
if (lErr != ERROR_SUCCESS){
   // Handle error
}
```

Now that you know how to obtain an open handle to an LSA policy object, you are ready to begin managing the privileges on the system.

Enumerating Privileges

There are two ways to enumerate the privileges on a system: either obtain a list of the privileges held by a specific trustee, or request a list of trustees that hold a specific privilege.

Privileges Held by a Specific Trustee

Let's start by discussing how to obtain a list of privileges held by a specific trustee of the system. To enumerate the privileges, use the *LsaEnumerateAccountRights* function:

```
NTSTATUS LsaEnumerateAccountRights(
   LSA_HANDLE           hPolicy,
   PSID                 psidTrustee,
   PLSA_UNICODE_STRING* pplsastrUserRights,
   PULONG               plCountOfRights);
```

You must pass an open policy object created with POLICY_LOOKUP_NAMES access as the *hPolicy* parameter of *LSAEnumerateAccountRights*. The *psidTrustee* parameter is a pointer to the SID for the trustee for which you want to enumerate privileges. You can use *LookupAccountName* (discussed earlier in the section "Understanding SIDs") to obtain a SID from a trustee's account name. This trustee can be a user account, group account, or computer account.

You should pass the address of a variable of type LSA_UNICODE_STRING as the *pplsastrUserRights* parameter of *LsaEnumerateAccountRights*. The system generates an array of LSA_UNICODE_STRING structures, allocates a buffer to hold the array, and then places a pointer to the buffer in the variable pointed to by the *pplsastrUserRights* parameter. The system then returns the number of privileges in the array in the ULONG variable pointed to by *plCountOfRights*.

Because the system has allocated a buffer on your behalf, you must free the buffer when you are finished with it. Passing a pointer to the buffer to the *LsaFreeMemory* function does this:

```
NTSTATUS LsaFreeMemory(PVOID pvBuffer);
```

If *LsaEnumerateAccountRights* succeeds, the translated status code will be ERROR_SUCCESS. If the account has no privileges assigned to it, the translated error code will be ERROR_FILE_NOT_FOUND. The elements of the array returned by *LsaEnumerateAccountRights* are of type LSA_UNICODE_STRING (defined earlier in the section "The LSA Functions"). Each element will point to a buffer containing a Unicode string representation of an account right, including values such as SeDebugPrivilege and SeEnableDelegationPrivilege. You can refer back to Table 9-10 for a list of all possible returned privileges.

Although it is not directly related to enumerating privileges, the *LookupPrivilegeDisplayName* function should be mentioned. This function translates a programmatic privilege name, such as SeTcbPrivilege, into its

friendly display name, which in this case would be "Act as part of the operating system".

```
BOOL LookupPrivilegeDisplayName(
   PCTSTR pszSystemName,
   PCTSTR pszName,
   PTSTR  pszDisplayName,
   PDWORD cbDisplayName,
   PDWORD pLanguageId);
```

This function takes the system name and the programmatic name for the privilege, and then returns the friendly name in a buffer that you supply.

NOTE

LookupPrivilegeDisplayName will not return the display name for an account right. It works only for privileges. Although our discussion here does not distinguish between these two types of account right, this topic is discussed more fully in Chapter 11.

You can determine whether an account right is a privilege by referring to Table 9-10. If the right in question is defined in the WinNT.h header file, it is a privilege. If it is defined in the NTSecAPI.h header file, it is only an account right, not an actual privilege.

The following sample function shows how to list all the privileges held by a trustee. It takes an LSA policy handle and a PSID as its parameters.

```
BOOL PrintTrusteePrivs(LSA_HANDLE hPolicy, PSID psid) {
   BOOL fSuccess = FALSE;
   WCHAR szTempPrivBuf[256];
   WCHAR szPrivDispBuf[1024];
   PLSA_UNICODE_STRING plsastrPrivs = NULL;

   __try {
      // Retrieve the array of privileges for the given SID
      ULONG lCount = 0;
      NTSTATUS ntStatus = LsaEnumerateAccountRights(hPolicy, psid,
         &plsastrPrivs, &lCount);
      ULONG lErr = LsaNtStatusToWinError(ntStatus);
      if (lErr != ERROR_SUCCESS) {
         plsastrPrivs = NULL;
         __leave;
      }

      ULONG lDispLen = 0;
      ULONG lDispLang = 0;

      for (ULONG lIndex = 0; lIndex < lCount; lIndex++) {
         // Assure zero termination
         lstrcpyn(szTempPrivBuf,
            plsastrPrivs[lIndex].Buffer, plsastrPrivs[lIndex].Length);
         szTempPrivBuf[plsastrPrivs[lIndex].Length] = 0;

         wprintf(L"Programmatic Name: %s\n", szTempPrivBuf);

         // Translate to Display Name
         lDispLen = 1024; // Size of static Display buffer
         if (LookupPrivilegeDisplayName(NULL, szTempPrivBuf,
            szPrivDispBuf, &lDispLen, &lDispLang))
            wprintf(L"Display Name: %s\n\n", szPrivDispBuf);
      }

      fSuccess = TRUE;
   }
```

```
    __finally {
       if (plsastrPrivs) LsaFreeMemory(plsastrPrivs);
    }
    return(fSuccess);
 }
```

Trustees That Hold a Specific Privilege

Now for the second way to retrieve privilege information for a Windows 2000 system: request a list of trustees that hold a specific privilege by calling *LsaEnumerateAccountsWithUserRight*.

```
NTSTATUS LsaEnumerateAccountsWithUserRight(
   LSA_HANDLE          hPolicy,
   PLSA_UNICODE_STRING plsastrUserRight,
   PVOID*              ppvEnumerationBuffer,
   PULONG              CountReturned);
```

This function requires a handle to an LSA policy object much like *LsaEnumerateAccountRights* does. However, instead of filling an LSA_UNICODE_STRING structure with the name of a trustee, you should pass a pointer to an LSA_UNICODE_STRING structure complete with the programmatic name of a privilege or an account right.

The system returns trustee information to your software by allocating a buffer and returning its pointer via the *ppvEnumerationBuffer* parameter. Although this parameter is defined as a pointer to a PVOID, you should pass the address of a pointer to a variable of type PLSA_ENUMERATION_INFORMATION because the system will return the information as an array of LSA_ENUMERATION_INFORMATION structures. It would have been nice if the developers of Windows had defined *LsaEnumerateAccountsWithUserRight* to take a pointer to the correct type, but they chose instead to require a pointer to a PVOID, so you will have to cast this parameter.

The LSA_ENUMERATION_INFORMATION structure is actually very simple and contains only a single member—a pointer to a SID:

```
typedef struct _LSA_ENUMERATION_INFORMATION {
   PSID Sid;
} LSA_ENUMERATION_INFORMATION;
```

The number of elements in the array is returned in the variable pointed to by *CountReturned*, which is the final parameter of *LsaEnumerateAccountsWithUserRight*. The SIDs in the returned array can be used to generate a trustee name by passing the SID to *LookupAccountSid*, discussed earlier in this chapter. When you are finished with the buffer returned by *LsaEnumerateAccountsWithUserRight*, you should pass its pointer to *LsaFreeMemory*.

Assigning and Removing Privileges

It is uncommon to have to create software solely to enumerate the privileges given to a trustee account. However, just about any software that creates trustee accounts will need to assign (and perhaps remove) privileges from an account. Fortunately, the LSA functions provide two simple functions to perform these tasks. The first, *LsaAddAccountRights*, is used to grant privileges to a trustee:

```
NTSTATUS LsaAddAccountRights(
   LSA_HANDLE          hPolicy,
   PSID                psidTrustee,
   PLSA_UNICODE_STRING plsastrUserRights,
   ULONG               lCountOfRights);
```

This function is easy to use. You must pass the handle to an open LSA policy object, as well as a pointer to the SID for the trustee whose privileges you wish to modify. (You can obtain a SID for a user or group account by calling *LookupAccountName*, as discussed earlier in this chapter.)

Before calling *LsaAddAccountRights*, however, you should build an array of LSA_UNICODE_STRING structures, each with the name of a privilege you want the trustee to hold. Pass a pointer to this array as the *plsastrUserRights* parameter of *LsaAddAccountRights*. Finally, pass the number of elements in the array as the final parameter, *lCountOfRights*.

The *LsaAddAccountRights* function ignores any account rights or privileges already held by the trustee. It fails, however, if any of the rights in the array are not valid names for the system.

The following code provides an example of how to assign the SeInteractiveLogonRight account right and the SeTcbPrivilege privilege to a trustee account whose SID is pointed to by the variable *psid*.

```
LSA_UNICODE_STRING lsastrPrivs[2] = { 0 };
lsastrPrivs[0].Buffer = SE_INTERACTIVE_LOGON_NAME;
lsastrPrivs[0].Length =
   lstrlen(lsastrPrivs[0].Buffer) * sizeof(WCHAR);
lsastrPrivs[0].MaximumLength = lsastrPrivs[0].Length + sizeof(WCHAR);

lsastrPrivs[1].Buffer = SE_TCB_NAME;
lsastrPrivs[1].Length =
   lstrlen(lsastrPrivs[1].Buffer) * sizeof(WCHAR);
lsastrPrivs[1].MaximumLength = lsastrPrivs[1].Length + sizeof(WCHAR);

NTSTATUS ntStatus = LsaAddAccountRights(hPolicy, psid,
   lsastrPrivs, 2);
ULONG lErr = LsaNtStatusToWinError(ntStatus);
```

The function used to remove privileges from a trustee account is similar to *LsaAddAccountRights*. It is called *LsaRemoveAccountRights* and is prototyped as follows:

```
NTSTATUS LsaRemoveAccountRights(
   LSA_HANDLE          hPolicy,
   PSID                psidTrustee,
   BOOLEAN             fAllRights,
   PLSA_UNICODE_STRING plsastrUserRights,
   ULONG               lCountOfRights);
```

This function is the same as *LsaAddAccountRights* except that it removes the list of rights from the trustee indicated by *psidTrustee* and has one extra parameter, the Boolean *fAllRights.*

The *fAllRights* parameter allows you to remove all privileges from a trustee without building a list of the privileges held by the trustee. If you pass TRUE for this parameter, the *plsastrUserRights* and *lCountOfRights* parameters should be NULL and 0, respectively. If you pass FALSE for the *fAllRights* parameter, *LsRemoveAccountRights* is used the same way as *LsaAddAccountRights.*

Before wrapping up our discussion of *LsaRemoveAccountRights*, I should point out one detail about removing all rights from an account. The Platform SDK documentation states that if you pass TRUE for the *fAllRights* parameter, the system will remove all privileges from the account and then delete the account from the system. However, this is true only from a certain perspective. Each system running Windows 2000 must maintain a local database of account privilege entries (technically implemented internally as a local account) even if the trustee account lives on another machine such as a domain controller. The *LsaRemoveAccountRights* function will not delete a trustee account (even if the account is local) from the system that hosts the account; it deletes only the account privilege entries from its local database.

[Previous] [Next]

Reasons to Create a Trustee

Before bringing this chapter to a close, I'd like to bring some perspective to the creation of trustee accounts. We've discussed how to create and destroy user and group accounts, as well as how to assign privileges to these accounts. Additionally, we explored the critical topic of the security identifiers or SIDs. However, you might still be wondering why your server software would need to create trustee accounts, since, simply put, many server applications never create trustee accounts and never assign or revoke privileges from existing trustee accounts.

Here are a couple of reasons:

- To restrict or otherwise manage access to resources. This is how your server software is likely to take advantage of trustee management.
- To create accounts for human users to use as a logon identity or to create group accounts to manage live users. Although this is a common reason to create trustee accounts, it is not a usual reason for server software to create trustee accounts. The MMC or some other administrative tool typically manages this function.

We'll be discussing the association between trustees and access rights of securable objects in the next chapter. Chapter 11 will talk about methods your server software can use to act on behalf of a client or any arbitrarily selected trustee account. You'll also learn ways to adjust the rights of an existing trustee by using a second trustee account.

As these topics unfold and you learn more creative ways to restrict and enhance access to objects using the various techniques available in the Windows environment, it is important to remember that, if necessary, your server software has the power to create trustee accounts. And as I discuss these topics in the next couple of chapters, I will point out cases where a trustee account created solely for use by your server software might be appropriate.

[Previous] [Next]

The TrusteeMan Sample Application

The TrusteeMan sample application ("09 TrusteeMan.exe") demonstrates the use of the Net functions and the LSA functions for creating and managing user and group accounts, as well as for assigning and revoking system privileges. The source code and resource files for the application are in the 09-TrusteeMan directory on the companion CD. Figure 9-2 shows the user interface for the TrusteeMan sample application.

Figure 9-2. *User interface for the TrusteeMan sample application*

Usability was an overriding goal for the TrusteeMan sample application. This is because security, in many ways, must be explored to be understood, and a usable tool can help you explore the features of the system. This sample application allows you to create user and group accounts to your heart's content. It also allows you to modify membership of groups, as well as grant and deny privileges to trustee accounts on the system. You can administer trustees on any system to which you have sufficient rights by entering the name of the computer that you want to use.

> **NOTE**
>
> When creating User accounts using TrusteeMan, the application uses the text "Pass2000" as the password for your new account. This password is used to simplify the user interface and focus on the topics of importance. You can use the MMC snap-in to manually set the password for a User account if you are an administrator of the system.

I suggest that you spend some time with the TrusteeMan sample application just to familiarize yourself with how trustee management works on Windows 2000. Also, the remaining chapters in this book assume that you are comfortable using TrusteeMan, the MMC snap-ins, or some other tool for administering user accounts and assigning privileges.

In addition to the TrusteeMan sample application being a useful tool, its source code provides a reference for much of the capabilities of the Net and LSA functions regarding trustee administration. The sample application uses the Net functions to perform the functionality of the left pane, or trustee side, of the application, while using the LSA functions to implement the right pane, or privilege side, of the application.

The sample also implements a reusable common dialog for editing a trustee list. The sample uses this dialog for managing group accounts and holders of a specific privilege. The system does not provide a built-in interface for this, so one was included with the sample.

[Previous] [Next]

Chapter 10

Access Control

Microsoft Windows 2000 provides extensive and exceptionally flexible security functionality. No other popular operating system on the market today offers a finer granularity of control over securable objects. To a great extent, this is due to Windows implementation of *access control*.

[Previous] [Next]

Introduction to Access Control

In general, when people refer to "Windows security," they mean Windows implementation of *access control*. Access control can be defined as the assignment and enforcement of who can and cannot perform certain activities on securable objects. Windows applies access control to a number of system objects and provides a mechanism for securing custom objects as well.

Securable Objects

The system enforces security on a number of objects and features provided by the system. Any object the system secures through access control is considered to be a securable object. Table 10-1 shows the types of securable objects in Windows 2000 as of the writing of this book. (The system also provides an enumerated type called SE_OBJECT_TYPE that includes a value for each type.) As you can see, many of the major components in Windows can (and do) take advantage of access control, including custom and private objects.

Windows 2000 allows your service software to subject application-defined objects of any type to Windows access control. The system manages the security of the object, whereas your software associates the security with the object and manages the functionality of the object itself. This security management is called *private object security* and is discussed in detail later in this chapter.

Table 10-1. *Securable object types in Windows 2000*

Category	SE_OBJECT_TYPE Enumerated Type	Description
File objects	SE_FILE_OBJECT	Files or directories on an NTFS file system.
Service objects	SE_SERVICE	A service installed on a system or the Service Control Manager (SCM) of a system.
Printer objects	SE_PRINTER	A printer or a print server.
Registry keys	SE_REGISTRY_KEY	A registry key on a Windows 2000 system.
Share objects	SE_LMSHARE	A share object indicating a shared directory on a Windows 2000 system.
Kernel objects	SE_KERNEL_OBJECT	The system can secure each of the following kernel objects: process objects, thread objects, job objects, semaphore objects, event objects, mutex objects, file-mapping objects, waitable timer objects, access tokens, named pipes, and

		anonymous pipes.
Window objects	SE_WINDOW_OBJECT	Window station and desktop objects (described later in this chapter).
Directory services objects	SE_DS_OBJECT SE_DS_OBJECT_ALL	Windows 2000 allows you to apply security to objects in Active Directory or directory services.
WMI objects	SE_WMIGUID_OBJECT	Objects exposed to WMI.
Security provider objects	SE_PROVIDER_DEFINED_OBJECT	Windows 2000 supports replaceable security providers, and these can expose securable objects.
Private objects		Custom objects created by a service or application secured by the system.

One strength of access control in Windows is that the process of securing an object is largely the same from object type to object type. This common design makes it easier for administrators and programmers to secure different objects.

Each securable object maintains an *access control list (ACL)* that determines who can and cannot perform certain securable actions on an object. This ACL is the center of Windows 2000 access control, and is associated with a securable object in some combination of the following three ways:

- **Default assignment** Windows 2000 provides a flexible default security mechanism that assigns access control to objects that are not explicitly secured by the creating software. This mechanism is the most common method of assigning security to objects, and it is the one used by all software that ignores security on Windows 2000. Default security is assigned to an object at the time the object is created.

- **Inheritance** System and private objects that exist in a hierarchy of parent-child relationships (such as files and registry keys) might also be subject to security inheritance. Inheritance allows security applied to a parent object to propagate to child objects in the hierarchy. If inherited access control entries (ACEs) are available, they are applied to new objects rather than default security. Inheritance is applied to objects regardless of whether an explicit access control list is also assigned.

- **Explicit assignment** You can explicitly assign access control to a securable object upon creation or after the object has been created. This method of assigning security is much more common in service software than in application and client software.

Building and assigning ACLs for securable objects is the primary topic of this chapter.

Overview of Access Rights

Although I have explained which objects are securable, I have not yet discussed what it really means for an object to be securable. Windows provides a fine level of control over the actions that can be performed on securable objects. Securable actions are performed only if the requesting trustee has the proper *access rights* to the object. Table 10-2 describes the three groups of access rights that can apply to an object.

Table 10-2. *Access rights, defined by the system, that can apply to an object*

Access Type	Description
Standard rights	Standard rights apply to all object types in the system, and include rights such as the ability to delete an object or to read an object's security. For a list of all the standard rights defined by Windows 2000, see Table 10-13.
Specific rights	Specific rights apply only to a specific type of securable object. For example, a specific right for a file object would be the right to append data or to read data from the file.
Generic rights	Generic rights indicate a collection of standard and specific rights for an object. The system defines four generic rights: read, write, execute, and all. The meaning of each generic right differs from object to object.

I will discuss each type of right in more detail throughout this chapter. At this time, however, it is important simply to think of a right as something that either allows or disallows a user to do something to an object. For example, a file might allow a user to read a file but disallow that same user to delete the file; this is possible because the system defines different rights for the reading of and deleting of a file.

When a user attempts to perform a securable task on an object, the system performs an *access check*. The access check searches the rights assigned to the object and compares them to the identity of the user attempting the action. If the system determines that the user has the requested access to the object, the action is performed. If the user does not have proper access, the action is not performed and the software running on the user's behalf receives an "Access Denied" error.

NOTE

The system also provides the ability to report on access attempts (successful and unsuccessful) made to securable objects. This is called auditing. Audit events are reported to the event log. Any access right that can be allowed or denied for an object can also be audited for that object. Although auditing is similar to access control, it is a separate and far less commonly used feature. Much of what you learn regarding access control will apply to auditing, so it might be helpful for you to be aware of the feature now. (I will discuss auditing in the section "Auditing and the SACL" later in this chapter.)

The Security Descriptor

As I mentioned earlier, the security for each type of securable object in the system is largely stored and manipulated in the same manner as security for any other securable object. This is because all access control in Windows 2000 is implemented via a data structure called a *security descriptor*. The security descriptor maintains important security information for an object, such as its owner, and a list of rights associated with users of the system. Each securable object in Windows has a security descriptor. Figure 10-1 shows a representation of a security descriptor, and Table 10-3 describes the components. You'll find that you are most often concerned with the owner and discretionary access control list (DACL) components of the security descriptor.

Figure 10-1. *The security descriptor*

As I discussed, manipulating security for objects in Windows is pretty much the same process from one type to another. In a typical scenario, a security descriptor is created by your software and then passed to a system function that creates the object, which assigns the security to the object.

Table 10-3. *Components of a security descriptor*

Component	Description
Revision	A value indicating the revision level of the security descriptor structure.
Control	A set of flags indicating the meaning of the contents of the security descriptor.
Owner	The security identifier (SID) of the owner of the object. The owner of an object has special rights—the owner can always read and modify the security for the object regardless of whether he has explicitly assigned rights to do so as described in the DACL. All securable objects have an owner. (For a discussion on SIDs, see Chapter 9.)
Group	The SID of the primary group of the object. Windows 2000 does not utilize the primary group in access control. The information is maintained so that Microsoft Windows NT can be used as a file-server platform for operating systems other than Windows.
DACL	The discretionary access control list (DACL) is an access control list that describes all the access rights for an object. This list defines who can and cannot perform securable

actions on an object. If a DACL is not present, everyone has all rights to the object. If a DACL is present but empty, no one but the owner has rights to the object.

SACL — The system access control list (SACL) is a list of access rights associated with users for the purpose of auditing. The SACL does not affect access to an object, but it does cause access to an object to be reported to the event log. See the section "Auditing and the SACL" later in this chapter for more information.

You can also retrieve a copy of the security descriptor of an existing object, use system functions to read or modify the security descriptor, and then use a system function to set the security descriptor of the original object.

NOTE

It might seem less efficient to use system functions to modify the security descriptor once you have retrieved a copy of it from the object. You might ask yourself why you would not access the data directly with your software—after all, you do have a copy of the data structure in your process's memory. Although you could access the data this way, it is important not to because the security descriptor is intended to be accessed as an "opaque" structure so that your code will function properly in future versions of Windows.

ACLs, ACEs, and the DACL

An ACL is little more than a sequential list of structures of variable lengths called *access control entries,* or *ACEs*. Each ACE indicates an access right and the SID of a trustee to which the right is associated for the object.

The DACL and the SACL are ACLs. The DACL is used for access control of an object; its ACEs indicate who is allowed or disallowed access to the object. The SACL is used for access auditing. The structure for the DACL and the SACL are the same, and the code to manipulate them is implemented similarly. I will discuss the SACL later in this chapter in the section "Auditing and the SACL," but much of our discussion here regarding the DACL applies to the SACL.

When an application attempts to access a secured object, the system scans the DACL of the object looking for ACEs, which indicate either the user trustee account under which the application is running or a group trustee account of which the user is a member. If a matching ACE is found, the system checks to see if the ACE grants or denies the right to perform the access requested by the application. The access check is discussed in more detail shortly, but first let's look at the six types of ACE. Table 10-4 describes them. The access-allowed ACE and the access-denied ACE are by far the most common.

NOTE

Object-type ACEs are not used with any securable objects in the system except the directory services objects. However, you can use object-type ACEs with your own private objects. I will focus mostly on the regular ACE types, but I will discuss object-type ACEs briefly throughout this chapter.

Table 10-4. *ACE types in Windows 2000*

ACE Type (value used by the system)	Description

Access allowed (ACCESS_ALLOWED_ACE_TYPE)	Defines a set of access rights allowed to a specified trustee account
Access denied (ACCESS_DENIED_ACE_TYPE)	Defines a set of access rights denied to a specified trustee account
System audit (SYSTEM_AUDIT_ACE_TYPE)	Defines a securable action that will cause an audit report if performed by the specified trustee account
Access-allowed object (ACCESS_ALLOWED_OBJECT_ACE_TYPE)	Defines a single access right that is allowed for a specified trustee account for an object or an object's subobject or property (typically used with directory services objects)
Access-denied object (ACCESS_DENIED_OBJECT_ACE_TYPE)	Defines a single access right that is denied for a specified trustee account for an object or an object's subobject or property (typically used with directory services objects)
System audit object (SYSTEM_AUDIT_OBJECT_ACE_TYPE)	Defines a single access right that will cause an audit report if performed by a specified trustee account for an object or an object's subobject or property (typically used with directory services objects)

Standard access-allowed ACEs and access-denied ACEs are pretty simple. Table 10-5 shows the contents of a standard ACE.

Table 10-5. *Contents of a standard (nonobject) ACE*

ACE Component	Description
ACE type	A numerical value indicating the ACE type. (Table 10-4 shows ACE types in Windows 2000.)
ACE flags	Indicate inheritance rules for ACEs as well as auditing rules for ACEs in a SACL. (For a list of the available inheritance flags, see Table 10-11.)
Access mask	A 32-bit value indicating the access rights described by the ACE.
Trustee's SID	Indicates the trustee user, group, or computer account with which the access rights of the ACE are associated.

Earlier in the chapter you learned about the three types of access rights defined by the system: standard, specific, and generic. The access mask portion of an ACE is a 32-bit value in which each possible access right for a given object is mapped to a bit.

NOTE

A single ACE can be used to indicate multiple access rights for a given trustee, because each access right maps to a single bit in the ACE's access map.

The access mask is divided into sections for each of the three types of access rights supported by Windows. Figure 10-2 shows the bits and their purposes.

Figure 10-2. *Access mask format*

ACEs for system and private objects that exist in a hierarchy of parent/child relationships can indicate inheritance rules. I will cover inheritance in more detail when we begin our discussion on programmatically manipulating ACLs; however, it is useful to know now what types of inheritance are allowed. The following list explains the possibilities:

- A parent object contains an ACE that does not affect the parent but does affect child objects. Conversely, it is possible to have an ACE that affects both the parent object and the child objects.
- A parent has an ACE that affects child objects but does not continue to inherit to grandchild objects, great-grandchild objects, and so on. You can also indicate that an ACE should inherit indefinitely.
- An ACE that inherits to a child (or grandchild) object that is also a container can affect the container object, or it can be set to affect only non-container child objects.
- An ACE can be defined to inherit only to child objects that are also containers. Or it can be defined to inherit to any child object.

Each of these decisions can be made for an ACE irrespective of the other inheritance properties of the ACE, so as you can see, an ACE can be made inheritable in a number of different ways.

NOTE

It is also possible to create an object whose security descriptor does not allow ACEs to inherit from parents. Such a descriptor is called a *protected security descriptor*. It stops inheritance to the object as well as to children of the object. It does not, however, affect the security descriptor of the parent object or any of the other children of the parent.

The system keeps track of which ACEs are inherited and which are explicitly set to an object. This way, if a security descriptor is set protected after the object is created, the system knows which ACEs to remove from the DACL and SACL. ACEs that are explicitly set to an object take precedence over ACEs that are inherited.

Access Checks

You now have a good idea of how the security for a securable object is maintained. We will soon be ready to begin our discussion of the security API, but first it is important that you understand access checks.

When you log on to a system running Windows 2000, you enter your username and password. The system looks up your user account and the group trustee accounts of which you are a member, and stores each trustee account's SID in an internal structure called a *token*. The system also stores a list of privileges that are assigned to your trustee and group trustee accounts. You'll learn more about tokens in Chapter 11, but for now you should think of the token as the structure that stores your identity and privileges.

After the system builds a token for you and launches the shell process, it associates your token with the shell. From this point forward, any process that the shell process launches automatically inherits a copy of your token. This is how the system maintains a sense of your identity.

Processes that are associated with your token are said to be running in your *user context* or *security context*. When a process running in your security context attempts to perform a securable action on a securable object, the system first performs an access check to determine whether you or one of your groups has sufficient rights to perform the task. Figure 10-3 shows the relationship of the entities involved in an access check.

Figure 10-3. *A process accessing a securable object*

NOTE

It is also possible for a thread in a process to run in a user context different from the process. This is called *impersonation* and is covered in detail in the next chapter.

Here are the steps the system takes when it performs an access check:

1. The system maps requested generic rights to standard and specific rights.

2. The system checks your token for relevant privileges in this access check. Most access checks do not take privileges into consideration. However, if you hold the SeTakeOwnershipPrivilege privilege, the system always passes an access check for the WRITE_OWNER standard right. (See Table 10-13.) Also, if the ACCESS_SYSTEM_SECURITY standard right is required, you must hold the SeAuditPrivilege privilege. (See Chapter 9 for a discussion of privileges.)

3. The system compares your SID and the group SIDs in your token with the owner SID of the object. If you are the owner of the object and you request the READ_CONTROL or WRITE_DAC standard access right, the system grants access regardless of the contents of the DACL.

4. The system checks for existence of a DACL in the security descriptor. If one is not present, access is granted.

5. The system checks the SID in the first ACE of the DACL against the SIDs in your token. If a match is found, the access mask of the ACE is checked against the access rights required by this access check.

6. If the ACE is an access-denied ACE, and the access mask matches any of the access rights required, the access check fails immediately.

7. If the ACE is an access-allowed ACE, and it fulfills any or all of the access rights required for the access check, the system makes note of the success.

8. After all the rights required for the access check are met, the system passes the access check.

9. If the last ACE in the DACL is checked, and not all of the rights required by the check are found, the system fails the access check.

NOTE

You can create a token that is restricted to the access rights of additional, arbitrarily selected trustee accounts. This is called a *restricted token*. If your process or thread is associated with a restricted token, the rules of an access check change. This topic is covered in detail in Chapter 11.

The most important steps in this process are steps 4, 6, 8, and 9. In step 4, if the system finds that the object's security descriptor does not include a DACL, the access check succeeds for everyone. In step 6, an access check fails if an access-denied ACE matches your user or group SIDs and any of the access rights for the access check. It fails immediately regardless of whether a later ACE in the DACL would have passed the access check. Step 8 passes the access check after all of the requested rights are found, regardless of whether an access-denied ACE later in the DACL would have failed the access check. And finally, in step 9, not enough ACEs to pass a check indicates implicit failure.

As you can see, in an access check, the order of ACEs in a DACL is very important. You should always place access-denied ACEs before access-allowed ACEs in a DACL. The Windows 2000 user interface enforces ACE ordering; however, in your own software, you can place ACEs in any order. If you place an access-denied ACE after an access-allowed ACE in your object's DACL, you should have a good reason!

NOTE

An access-denied ACE at the end of a DACL is a wasted ACE. If the ACE denies access that was already allowed, the access check would never even see the denied ACE at the end. And if the ACE denies access that was not explicitly allowed, it was not necessary to deny the access in the first place&"95%">**ACE TypeGroup**Access-denied ACEs

Access-denied object ACEs that apply to subobjects or properties of the object

Access-allowed ACEs

Access-allowed object ACEs that apply to subobjects or properties of the object

Explicitly assigned (noninherited) ACEsAccess-denied ACEs

Access-denied object ACEs that apply to subobjects or properties of the object

Access-allowed ACEs

Access-allowed object ACEs that apply to subobjects or properties of the object

Inherited ACEs

The ordering rule in Table 10-6 might look complex, but remember that object ACEs are not used for any securable object in the system except directory services objects (objects in Active Directory). Ignoring object ACEs will simplify the order of ACEs dramatically.

Understanding Custom or Private Object Security

I have discussed system securable objects, the structure of the security descriptor that secures the object, and how the system uses the DACL to check security against a request made by software. However, I have not yet explained how your software can create secure objects that use the Windows security model. Windows refers to this ability as *securing private objects*.

Private object security is a very powerful and flexible feature included in Windows. Later in this chapter I discuss the security API used with private objects. At this point, I would like to explain how private object security ties in to the existing security model in Windows.

Everything you have learned about security descriptors, ACLs, DACLs, and ACEs also applies to private objects, except that your software, rather than the system, must decide which of the standard rights (listed in Table 10-13) apply to your objects. Additionally, you must define specific rights for your objects, and map the four generic rights to appropriate combinations of standard and specific rights.

Your software performs the access check by calling a system function. Typically, your software is a service that passes the token of a connected client to the system along with a security descriptor. The system then indicates whether the required access rights exist for the client. Your service is responsible for performing or refusing to perform the requested action, based on the results of the access check.

The system creates and destroys security descriptors in memory for your private objects. You must associate the security descriptors with the data that they protect. Your service is also responsible for storing the security with the data in persistent storage when your service ends (if the objects persist, that is).

NOTE

Private object security does not automatically secure the data of the objects defined by your software—if your objects are stored in files, you must still secure the files. Private object security does, however, provide a mechanism for allowing you to control your data at a fine level of granularity, without being limited to file security or the security of some other storage mechanism in Windows.

Exploring Security in Windows

To become a successful security developer, it is important to understand and also be comfortable with security in Windows 2000. Spending some time modifying the security of system objects via the user interface will greatly increase your ability to effectively design software that incorporates security.

The following sections walk you through the tools supplied with Windows to help you become familiar with security. In Windows, registry keys are securable and files and directories (on NTFS partitions only) are securable. The file system is probably the most effective way to become familiar with security, because its inheritance model includes both container objects (directories) and non-container objects (files). If you do not have an NTFS partition with which to meddle, the registry will also do just fine.

Access Control for the Registry

These steps describe how security options can be specified in the registry:

1. Log on to your system as Administrator or as a member of the Administrators group.

2. Run the RegEdt32.exe utility. You will see a screen similar to Figure 10-4.

Figure 10-4. *The registry editor (RegEdt32.exe)*

3. Select the window titled HKEY_LOCAL_MACHINE On Local Machine. This window shows the HKEY_LOCAL_MACHINE hive on your system.

4. Open the Software key and select Add Key from the Edit menu. Add a new key named *ANewKey*. (I am calling my new key ANewKey so that it shows up near the top of the list of keys under Software.)

5. Click on the new key, and select Permissions from the Security menu. A Permissions dialog box appears.

6. Uncheck the Allow Inheritable Permissions From Parent To Propagate To This Object check box. This causes the security descriptor for your new key to be protected, which disallows inheritable permissions to propagate from parent keys. The system will ask you whether you wish to copy or remove the current inherited permissions, as shown in Figure 10-5.

Figure 10-5. *Setting inheritable permissions for ANewKey*

7. Click the Remove button to remove the access list.

8. Right now your new key has an empty DACL. If you were to click OK in the dialog box, nobody but you (because you are the key's owner) could do anything with the key. And the only things *you* could do would be read or write the security of the object.

9. Click the Add button to display the Select Users, Computers, Or Groups dialog box. Select Everyone from the list box, click the Add button, and then OK. This adds an ACE for everyone to your key's DACL.

10. Click the check box under Allow to indicate Full Control. Now click OK. The registry editor creates a new security descriptor for your key that allows everyone full control of the key.

11. Add two new subkeys to ANewKey by selecting it and choosing Add Key from the Edit menu. (I named my new keys *First* and *Second*, but you can name them anything you like.)

12. Log off of your machine, log on as the built-in Guest account (you might need to enable the account), and then rerun RegEdt32.exe. You can use any trustee account other than your own for this portion. (An alternative to logging off is to use the RunAs utility to launch RegEdt32.exe as the Guest account. For example: RunAs.exe /env /user:mymachine\Guest RegEdt32.exe.)

13. Find your new key under HKEY_LOCAL_MACHINE\Software and open it. You should see your new subkeys. Check the permissions of both subkeys by clicking on them and then selecting the Permissions option from the Security menu. Notice that both keys have inherited the simple Everyone\Full Control security from their parent key. Do not uncheck the Allow Inheritable Permissions check box this time.

14. By virtue of being a member of the built-in Everyone group, the otherwise poorly endowed Guest account can do whatever it wants to these keys, including change their permissions. So take advantage of this power and open the permissions for one of your subkeys.

15. Check the box under Deny for Full Control, which denies full control to the Everyone group. Don't click OK just yet.

16. What you are doing is adding an access-denied ACE to the DACL for the subkey. Notice, however, that the check boxes under Allow are still checked. This is because the inherited access-allowed ACE allowing Full Control to Everyone is still present in the DACL as well. However, your explicitly applied ACE will take precedence over any inherited ACEs. Click the OK button, and click Yes in the Security dialog box asking if you want to continue.

17. Now you have actually denied everyone (including the account that you logged in as) any access to this key. No one can create subkeys or values under this key. And only the owner (which should be your regular logon account) can modify the security to allow permissions.

18. Before logging off and logging back on with your normal account, try editing the permissions of the registry key that was just modified. The system should tell you that you do not have sufficient access rights to view or edit the security for the key. (Then the system will show you an empty security descriptor, just in case you do have the right to write access to the key, which you do not.)

Access Control for an NTFS Partition

Here are some examples that illustrate how permissions work for files on an NTFS partition:

1. To view permissions for files, right-click on a file or folder in Windows Explorer, and select Properties from the context menu to display the Properties dialog box. Select the Security tab to display permissions for the file. File permissions are set in much the same way that they are for registry keys in RegEdt32.

2. In the Properties dialog box, click the Advanced button to display the Access Control Settings dialog box as shown in Figure 10-6. Add access rights that don't inherit, or add access rights that inherit only to non-containers (or files).

Figure 10-6. *Setting permissions for a file on an NTFS partition*

3. In the Properties dialog box, click the Advanced button. If you or a group that you are a member of is the owner, you should be able to set the owner to yourself or one of your groups. Additionally, if your account has the SeTakeOwnershipPrivilege privilege, you should be able to set the owner of any object in the system. To see the list of owners, click the Owner tab.

4. In Explorer, create several levels of directories, of which the top level is a new directory that you created with a protected security descriptor, and add enough access rights to continue creating directories. Then change the security of a directory in the middle of the hierarchy by adding more access-allowed or access-denied ACEs. Now look at a directory or file at the bottom of your hierarchy, and notice how it has inherited permissions from multiple parents. Try removing some of the permissions from a parent in the hierarchy and notice how it affects the child or grandchild. You can also do this type of testing with a hierarchy of registry keys.

I strongly suggest you spend more time experimenting with permissions of registry keys or files (if you have an NTFS partition). Exploring will work wonders for your abilities as a security programmer.

Review of Access Control Terminology

If you are new to security programming in Windows, you have just been bombarded with new terms and ideas. So now is a good time for us to rehash what we have covered before we dive into the depths of the security API.

- **Access check** A check performed by the system (or your software for private objects) that determines whether the user identified by a token has a requested set of access rights allowed by an object's DACL.
- **Access control** System-provided feature for managing and enforcing security access to objects.
- **Access mask** The 32-bit value within an ACE that contains a bit for each access right for an object, including standard, specific, and generic rights.
- **Access right** A system-defined or software-defined value that indicates a right required to perform some action on or with a securable object.
- **ACE** Stands for access control entry. The ACE contains a SID identifying a trustee of the system and an access mask indicating access rights. An ACE can allow or deny access to an object.
- **ACL** Stands for access control list. The ACL contains ACEs that define the security or security reporting of an object.
- **DACL** Stands for discretionary access control list. The DACL contains ACEs that explicitly allow and deny access to an object.
- **Generic access rights** The right to generically "read," "write," or "execute" an object, as well as the generic "all" access right. Each generic right maps to a set of standard and specific rights; these mappings differ for each type of object.
- **Protected security descriptor** A security descriptor with a control flag that indicates that it and its children should not receive inheritable ACEs from its parent object.
- **SACL** Stands for system access control list. The SACL contains ACEs that specify events to be reported to the event log for associated users.
- **Securable object** Any object, private or system, that is secured with the Windows access control model.
- **Security descriptor** A structure associated with every securable object in the system. A security descriptor includes a SID indicating owner and primary group, as well as optional DACL and SACL.

- **SID** Stands for security identifier, which identifies a trustee account to the system. SIDs are discussed in detail in Chapter 9.
- **Specific access rights** System-defined or software-defined access rights that apply only to a specific type of system or private securable object.
- **Standard access rights** System-defined access rights of which a subset will be applicable to each securable object type in the system.
- **Token** A structure associated with a process or a thread that contains a SID identifying a user and SIDs identifying the user's groups, as well as the privileges held by the user. Tokens are discussed in detail in Chapter 11.
- **User context** If software is running under your token, it is said to be running in your user context.

[Previous] [Next]

Programming for Access Control

In this section, we will discuss how to programmatically manipulate security in Windows. We'll begin with an overview of the steps involved.

Basic Steps for Security Tasks

Modifying the security of a securable object is largely the same no matter what type of object you are dealing with. You usually perform one of two tasks: create a new object with security, or change the security of an existing object.

Here are the basic steps that you will take to create an object with security:

1. Compile a list of SIDs for which you will be creating denied and allowed ACEs.
2. Create and initialize a security descriptor.
3. Create and initialize a DACL large enough to hold the required ACEs.
4. Add ACEs to the DACL.
5. Add the DACL to the security descriptor.
6. Create the object using the new security descriptor.
7. Clean up after yourself.

Only step 6 is different from one securable object type to the next.

To modify the security of an existing object, follow these steps:

1. Compile a list of SIDs for which you will be adding denied and allowed ACEs to the object's DACL.
2. Retrieve the DACL of the object.

3. Check the existing DACL for ACEs that you wish to remove, and remove them.

4. Check the existing DACL for ACEs that you are adding so that you can avoid adding them again and creating unnecessary bulk.

5. Create a new DACL large enough to accommodate the modified "old" DACL in addition to the new ACEs.

6. Copy the old ACEs and add new ACEs to the "new" DACL.

7. Set the DACL to the object.

8. Clean up after yourself.

In this process, only step 2 (in which you retrieve a DACL) and step 7 (in which you apply the DACL) differ from one securable object type to the next.

Don't let yourself be overwhelmed by this process. I will discuss each step in detail and describe alternatives for a few of these steps. The purpose of mentioning the process here is to show the commonality of approach for *any* securable object in Windows.

As you can see, once you are comfortable modifying the security of one object, you have the skills (and perhaps even the code) that you need to modify the security of any object. The differences in your approach to modification are mainly in the getting and the setting of security. If you know which function to use for the type of object you are concerned with, you are all set. Take a look at Table 10-7.

Table 10-7. *Specific security functions for securable objects*

Securable Object	Creation Function	Getting and Setting Security Descriptor
Access tokens	*DuplicateTokenEx*	*GetSecurityInfo*, *SetSecurityInfo*
Desktops	*CreateDesktop*	
Directories (on an NTFS file system)	*CreateDirectory* *CreateDirectoryEx*	*GetNamedSecurityInfo*, *GetSecurityInfo*, *SetNamedSecurityInfo*, *SetSecurityInfo*
Event objects	*CreateEvent*	
Files (on an NTFS file system)	*CreateFile*	
File-mapping objects	*CreateFileMapping*	
Job objects	*CreateJobObject*	
Mutex objects	*CreateMutex*	
Network share objects	*NetShareAdd*	
Pipes, anonymous	*CreatePipe*	*GetSecurityInfo,* *SetSecurityInfo*
Pipes, named	*CreateNamedPipe*	

Printers	*AddPrinter*	*GetNamedSecurityInfo*, *GetSecurityInfo*, *SetNamedSecurityInfo*, *SetSecurityInfo*
Process objects	*CreateProcess*, *CreateProcessAsUser*	*GetSecurityInfo*, *SetSecurityInfo*
Registry keys	*RegCreateKeyEx*	*GetNamedSecurityInfo*, *GetSecurityInfo*, *SetNamedSecurityInfo*, *SetSecurityInfo*
Semaphore objects	*CreateSemaphore*	
Service Control Manager	[cannot create the Service Control Manager]	*QueryServiceObjectSecurity*, *SetServiceObjectSecurity*
Services	[cannot specify security when creating services]	*GetNamedSecurityInfo*, *GetSecurityInfo*, *SetNamedSecurityInfo*, *SetSecurityInfo*
Thread objects	*CreateThread*, *CreateRemoteThread*	*GetSecurityInfo*, *SetSecurityInfo*
Waitable timer objects	*CreateWaitableTimer*	*GetNamedSecurityInfo*, *GetSecurityInfo*, *SetNamedSecurityInfo*, *SetSecurityInfo*
Window stations	*CreateWindowStation*	*GetSecurityInfo*, *SetSecurityInfo*
Private objects		*GetPrivateObjectSecurity*, *SetPrivateObjectSecurity*, *SetPrivateObjectSecurityEx*

Notice in Table 10-7 that although the functions for creating an object differ for each object, only a handful of functions are required for setting and retrieving security information for all securable object types in the system.

NOTE

There are other functions for getting and setting security for specific object types, such as *GetFileSecurity* and *SetKernelObjectSecurity*. However, object-specific functions are no longer the preferred method for retrieving and setting security for objects, because functions such as *GetSecurityInfo* and *SetNamedSecurityInfo* are easier to use and offer a more complete implementation of the inheritance model in Windows 2000. You should always use these functions when possible.

Reading Security Information for an Object

Only *reading* an object's security information is not a common task, and typically you do it so you can modify the security in some way. Understanding how to read an object's security, however, will greatly simplify the more common task of modifying object security, and so I'll cover the topic here.

As you might imagine, not everyone can read the security of an object. Like any other task performed on a securable object, the ability to read the security information itself is securable. To read the security information, one or both of the following conditions must be true:

1. You are the owner of the object.

2. An ACE in the object's DACL grants you, or a group to which you are a member, the standard right READ_CONTROL. (See Table 10-13.)

If neither of these conditions is true, you do not have the right to read an object's security. If either is true, when you obtain a handle to the object, you can ask for READ_CONTROL access in the access required parameter of the function used to acquire a handle, and the system will give you a handle. The following code fragment shows an example of obtaining a handle for reading the security information of a file:

```
HANDLE hFile = CreateFile(TEXT("C:\\Test\\Test.txt"), READ_CONTROL, 0,
   NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

if ((hFile == INVALID_HANDLE_VALUE) &&
   GetLastError() == ERROR_ACCESS_DENIED){
   // You do not have READ_CONTROL access to the file
}
```

If the file C:\Test\Test.txt exists and resides on an NTFS drive, this code fragment will work. If you meet one of the two conditions, this code will give you a valid file handle that you can use to read the security of your object.

Now that you have a handle to a securable object such as a file, let's use that handle to retrieve security information about the object by making a call to the *GetSecurityInfo* function:

```
DWORD GetSecurityInfo(
   HANDLE               hHandle,
   SE_OBJECT_TYPE       objType,
   SECURITY_INFORMATION secInfo,
   PSID                 *ppsidOwner,
   PSID                 *ppsidGroup,
   PACL                 *ppDACL,
   PACL                 *ppSACL,
   PSECURITY_DESCRIPTOR *ppSecurityDescriptor);
```

NOTE

As you might recall from Table 10-7, the *GetSecurityInfo* function is used to retrieve security information for the majority of securable objects in Windows. It is a very flexible and useful function indeed.

When using *GetSecurityInfo*, you pass a handle to an object, which you have opened with READ_CONTROL access as the *hHandle* parameter. The *objType* parameter is an enumerated type that indicates the type of securable object that the handle represents. Table 10-1 shows different enumerated values that can be used with this function. For example, if you passed the handle of a file as the *hHandle* parameter of *GetSecurityInfo*, you would have to pass the enumerated value SE_FILE_OBJECT as the *objType* parameter.

The *secInfo* parameter indicates what information in the object's security descriptor you would like the system to return. Remember that an object's security descriptor maintains an owner, a group, a DACL, and a SACL, and you can use *GetSecurityInfo* to retrieve any or all of these by passing any combination of the values in Table 10-8.

Table 10-8. *SECURITY_INFORMATION values that can be passed for* GetSecurityInfo*'s* secInfo *parameter*

Value	Description
DACL_SECURITY_INFORMATION	Indicates that you wish to retrieve DACL information for a securable object
SACL_SECURITY_INFORMATION	Indicates that you wish to retrieve SACL information for a securable object
OWNER_SECURITY_INFORMATION	Indicates that you wish to retrieve owner information for a securable object
GROUP_SECURITY_INFORMATION	Indicates that you wish to retrieve group information for a securable object

The *ppsidOwner*, *ppsidGroup*, *ppDACL*, and *ppSACL* parameters are optional. If you want the system to return a pointer to the owner's SID, group SID, DACL, or SACL of the object, you can pass the address of a PSID or PACL variable for any or all of these parameters. However, you must pass NULL for any information that you do not request by passing the proper flag as the *secInfo* parameter. Regardless, you can always pass NULL for any or all of these parameters.

The final parameter is the address of a pointer to a SECURITY_DESCRIPTOR structure and is mandatory. The system allocates a buffer large enough to hold the requested security information for the object using *LocalAlloc* and places a pointer to the security descriptor in the variable whose address you supply in *ppSecurityDescriptor*. (It is necessary to pass the value returned in the *ppSecurityDescriptor* parameter to *LocalFree* when you are finished with the security descriptor for the object.)

NOTE

If you should pass the address of a PSID or PACL variable as the *ppsidOwner*, *ppsidGroup*, *ppDACL*, or *ppSACL* parameter, the system will return an address that points to a portion of the security descriptor returned in *ppSecurityDescriptor*. This is why these parameters are optional.

You don't have to call *GetLastError* after calling *GetSecurityInfo*, because *GetSecurityInfo* returns an error code directly. If *GetSecurityInfo* succeeds, it returns ERROR_SUCCESS. Common error values are ERROR_ACCESS_DENIED, which indicates that your handle was not opened with READ_CONTROL access, and ERROR_INVALID_HANDLE, which usually indicates that you passed mismatched handle and object type values for the *hHandle* and *objType* parameters.

The following code could be used with the preceding code fragment (in the beginning of "Reading Security Information for an Object"), which opens a handle to a file using *CreateFile*, to retrieve DACL and object owner information for the file at C:\Test\Test.txt:

```
PSECURITY_DESCRIPTOR pSD;
PSID  pSID;
PACL  pDACL;
ULONG lErr = GetSecurityInfo(hFile, SE_FILE_OBJECT,
   DACL_SECURITY_INFORMATION, &pSID, NULL, &pDACL, NULL, &pSD);
if (lErr != ERROR_SUCCESS){
   // Error case
}

// Perform work here

// Clean up
LocalFree(pSD);
CloseHandle(hFile);
```

In this example, after successfully calling *GetSecurityInfo*, the *pSID* and *pDACL* variables point to a SID structure and a DACL structure contained in the buffer returned in the *pSD* variable. It is neither necessary nor correct to pass the pointers returned in *pSD* or *pDACL* to *LocalFree*, because the entire security descriptor is freed when the pointer returned in the *pSD* variable is passed to *LocalFree*.

You are familiar with SIDs and pointers to SIDs from Chapter 9, and you can use the owner SID returned from *GetSecurityInfo* with functions such as *LookupAccountSid* and *CopySid*, which are discussed in that chapter.

Before dissecting the DACL of a secured object, I would like to introduce *GetSecurityInfo*'s sister function, *GetNamedSecurityInfo*. Whereas *GetSecurityInfo* requires a handle to a securable object, *GetNamedSecurityInfo* requires the textual name of a secure object in the system, and is defined as follows:

```
DWORD GetNamedSecurityInfo(
   LPTSTR               pObjectName,
   SE_OBJECT_TYPE       objType,
   SECURITY_INFORMATION secInfo,
   PSID                 *ppsidOwner,
   PSID                 *ppsidGroup,
   PACL                 *ppDACL,
   PACL                 *ppSACL,
   PSECURITY_DESCRIPTOR *ppSecurityDescriptor);
```

Notice that this function differs from *GetSecurityInfo* only in the first parameter, which takes a pointer to a string containing the name of an object in the system. Fewer secured objects in the system are named. However, for those that are, this function can sometimes be more convenient than *GetSecurityInfo*, because it does not require you to first obtain a handle to an object. For a list of the objects whose security can be retrieved using *GetNamedSecurityInfo*, see Table 10-7.

Our previous two code fragments could be consolidated to create this smaller code fragment by using *GetNamedSecurityInfo*:

```
PSECURITY_DESCRIPTOR pSD;
PSID pSID;
PACL pDACL;
ULONG lErr = GetNamedSecurityInfo(TEXT("C:\\Test\\Test.txt"),
   SE_FILE_OBJECT, DACL_SECURITY_INFORMATION, &pSID, NULL,
   &pDACL, NULL, &pSD);
if (lErr != ERROR_SUCCESS){
   // Error case
}

// Clean up
LocalFree(pSD);
```

NOTE

In the examples I have shown so far, I have used the file at C:\Test\Test.txt to demonstrate how to retrieve an object's security information. However, both *GetNamedSecurityInfo* and *GetSecurityInfo* can be used to retrieve security for other securable objects including registry keys, kernel objects, and user objects. See Table 10-7 for specific security functions for securable objects.

GetSecurityInfo and *GetNamedSecurityInfo* allow you to pass the address of pointer variables to retrieve the address of the owner SID, group SID, DACL, or SACL in the returned security descriptor. However, you might be wondering how you would retrieve this information if you had only retrieved the security descriptor of an object. You can use the *GetSecurityDescriptorOwner*, *GetSecurityDescriptorGroup*, *GetSecurityDescriptorDacl*, and *GetSecurityDescriptorSacl* functions to retrieve pointers to these data structures within a security descriptor. I will discuss these functions further in the section titled "Securing

Private Objects" later in this chapter.

Dissecting the DACL

Now that we have discussed how to retrieve a pointer to the DACL of a secured object, it is time to begin uncovering how the DACL is read. Remember that the DACL is an access control list (ACL), and an ACL is little more than a list of access control entries (ACEs) that indicate access that is denied or allowed to a trustee.

The DACL for a security descriptor can be set to NULL. This indicates that no DACL is present for a particular object, and is referred to as a "null DACL." If an object has a null DACL, all access is implicitly granted to all trustees of the system.

> **NOTE**
>
> When a DACL *is* present for an object, all access to the object is implicitly denied unless explicitly allowed. Sometimes an object will have an empty DACL, which should not be confused with a null DACL. An empty DACL signifies no access for anyone (but the owner) for an object. A null DACL, on the other hand, grants all access to all trustees.

If a DACL is present, retrieving information about the ACEs in the DACL might be necessary. First you must find out how many ACEs the DACL contains by calling *GetAclInformation*:

```
BOOL GetAclInformation(
   PACL                  pACL,
   PVOID                 pACLInformation,
   DWORD                 dwACLInformationLength,
   ACL_INFORMATION_CLASS aclInformationClass);
```

This function allows you to retrieve information about a DACL (or a SACL), which you pass to the function via the *pACL* parameter. You should pass the address of a structure to retrieve ACL information as the *pACLInformation* parameter, and the length of the structure as the *dwACLInformationLength* parameter. The *aclInformationClass* parameter is an enumerated type that indicates what type of information is being returned and also defines which type of structure you should pass as the *pACLInformation* parameter. Possible values for the *aclInformationClass* parameter are listed in Table 10-9.

Table 10-9. *The ACL_INFORMATION_CLASS enumeration*

Enumerated Value	Structure	Definition
AclRevisionInformation	ACL_REVISION_INFORMATION	Returns ACL revision information
AclSizeInformation	ACL_SIZE_INFORMATION	Returns ACL size information including ACE count

The ACL_SIZE_INFORMATION structure is most commonly used with *GetAclInformation* and is defined as follows:

```
typedef struct _ACL_SIZE_INFORMATION {
   DWORD AceCount;
   DWORD AclBytesInUse;
   DWORD AclBytesFree;
} ACL_SIZE_INFORMATION;
```

The following code fragment builds on the last fragment to retrieve the number of ACEs in the DACL for the file at C:\Test\Text.txt:

```
PSECURITY_DESCRIPTOR pSD;
PACL pDACL;
ULONG lErr = GetNamedSecurityInfo(TEXT("C:\\Test\\Test.txt"),
   SE_FILE_OBJECT, DACL_SECURITY_INFORMATION, NULL, NULL,
   &pDACL, NULL, &pSD);
if (lErr != ERROR_SUCCESS){
   // Error case
}

ACL_SIZE_INFORMATION aclSize = {0};
if(pDACL != NULL){
   if(!GetAclInformation(pDACL, &aclSize, sizeof(aclSize),
      AclSizeInformation)){
      // Error case
   }
}

ULONG nAceCount = aclSize.AceCount;
```

NOTE

If the file at C:\Test\Test.txt is on a FAT drive, or if it does not have a DACL, the system returns NULL in the *pDACL* variable. This is why it is important to test for NULL before calling *GetAclInformation*. Passing a NULL pointer value to *GetAclInformation* causes an access violation.

After you have the number of ACEs in a DACL, you can use this information to retrieve individual ACEs. Calling *GetAce* does this.

```
BOOL GetAce(
   PACL  pACL,
   DWORD dwACEIndex,
   PVOID *pACE);
```

This function simply takes the pointer to an ACL, the zero-based index of the ACE you want to retrieve, and the address of a pointer variable in which to store the pointer to the ACE in the DACL.

Notice that the *pACE* parameter is of type PVOID. This is because a number of different ACE types can reside in a DACL, and each of these types is represented by a different structure. *GetAce* is used to retrieve each type. However, ACLs can contain any number of ACEs and any combination of ACE types, so there is no way to know which ACEs will be of which type until after you have retrieved the ACE with *GetAce*.

Although retrieving the ACE count and the ACE itself is simple, the real work in reading a DACL comes in understanding the different types of ACEs.

Understanding ACEs

There are a number of different ACE types, which can be divided into two broad types: standard ACEs and object ACEs. In each group you will find ACE types for allowing and denying access, as well as for auditing access. Each ACE type is guaranteed to share one attribute in common—the first member of its structure will be of type ACE_HEADER, defined as follows:

```
typedef struct _ACE_HEADER {
   BYTE   AceType;
   BYTE   AceFlags;
   USHORT AceSize;
} ACE_HEADER;
```

The ACE header is key to reading the DACL, because when you retrieve a pointer to an ACE from *GetAce*, you do not know what type of ACE it is or what type of structure makes up the ACE until you have read the ACE's header.

As you might have guessed, the *AceType* member of the ACE header indicates what type of ACE the header represents. The possible values for *AceType* are listed in Table 10-10. These six ACE types are currently available in Windows 2000.

Table 10-10. *ACE types*

ACE Type	Description
ACCESS_ALLOWED_ACE_TYPE	Used in a DACL. Indicates a set of access rights that are explicitly granted to a trustee. Uses the ACCESS_ ALLOWED_ACE structure.
ACCESS_DENIED_ACE_TYPE	Used in a DACL. Indicates a set of access rights that are explicitly denied to a trustee. Uses the ACCESS_DENIED_ACE structure.
SYSTEM_AUDIT_ACE_TYPE	Used in a SACL (see "Auditing and the SACL"). Indicates a set of access rights that should cause an audit event when requested by a trustee. Uses the SYSTEM_AUDIT_ACE structure.
ACCESS_ALLOWED_OBJECT_ACE_TYPE	Used in a DACL of a directory services object. Indicates a set of access rights that are explicitly granted to a trustee for an object in Active Directory. Uses the ACCESS_ALLOWED_OBJECT_ACE structure.
ACCESS_DENIED_OBJECT_ACE_TYPE	Used in a DACL of a directory services object. Indicates a set of access rights that are explicitly denied to a trustee for an object in Active Directory. Uses the ACCESS_DENIED_OBJECT_ACE structure.
SYSTEM_AUDIT_OBJECT_ACE_TYPE	Used in a SACL (see "Auditing and the SACL") of a directory services object. Indicates a set of access rights that should cause an audit event when requested by a trustee. Uses the SYSTEM_AUDIT_OBJECT_ACE structure.

The *AceFlags* member of the ACE_HEADER structure contains inheritance information about the ACE, as well as auditing information. The *AceFlags* member will be some combination of the values in Table 10-11.

Table 10-11. *Values for the* AceFlags *member of the ACE_HEADER structure*

Value	Description
Inheritance Flags	
INHERITED_ACE	This flag is set in the ACE if the flag was inherited from a parent object. This allows the system to distinguish between ACEs that were directly applied and ACEs that were automatically applied due to inheritance.

CONTAINER_INHERIT_ACE	Container objects that are children of the object owning this ACE will inherit this ACE as an effective ACE. This inheritance continues indefinitely by default; however, inheritance can be configured to stop with direct children if the NO_PROPAGATE_INHERIT_ACE flag is also set. If this bit is not set, but the OBJECT_INHERIT_ACE bit is set, the ACE will inherit to container objects, but the INHERIT_ONLY_ACE flag will be set for the container's ACE.
OBJECT_INHERIT_ACE	This ACE is inherited to and effective on non-container and child objects of the owning object. For child objects that are containers, the ACE is inherited as an inherit-only ACE unless the CONTAINER_INHERIT_ACE flag is also set. If the NO_PROPAGATE_INHERIT_ACE flag is set, and the CONTAINER_INHERIT_ACE flag is not set, an ACE with the OBJECT_INHERIT_ACE bit set will not inherit to container objects.
INHERIT_ONLY_ACE	This ACE is not effective on the object owning the ACE. However, it might be effective on child container or child non-container objects. It is invalid for an ACE to have the INHERIT_ONLY_ACE flag set unless one or both of the CONTAINER_INHERIT_ACE and OBJECT_INHERIT_ACE flags are set.
NO_PROPAGATE_INHERIT_ACE	Setting this flag in an ACE causes it to be inherited only once. The inherited ACEs will have their OBJECT_INHERIT_ACE and CONTAINER_INHERIT_ACE flags cleared so that the new ACEs will not also be inherited.
Auditing Flags	
FAILED_ACCESS_ACE_FLAG	In a SACL, an ACE with this bit set will cause an audit event if access is requested and the result is ACCESS_DENIED.
SUCCESSFUL_ACCESS_ACE_FLAG	Used with system-audit ACEs in a SACL to generate audit messages for successful access attempts.

In addition to the ACE type and flags, the *AceSize* member of the ACE_HEADER structure indicates the size of the ACE in question. As you can see, the ACE_HEADER contains a fair amount of information about the ACE. In fact, if you wanted to retrieve only inheritance and type information about an ACE, when calling *GetAce*, you could look no farther than the ACE_HEADER structure. The following code fragment shows how to call *GetAce* to retrieve the ACE's type:

```
ACE_HEADER* pACEHeader ;
GetAce(pDACL, 0, (PVOID*)&pACEHeader);

switch( pACEHeader->AceType ){
   // Do work depending on the ACE's type
}
```

Since you will typically require more information from an ACE than just its type and inheritance style, you will have to go beyond the ACE_HEADER. There are two main types of ACEs, standard and object. Standard

ACEs are ACEs that are not object ACEs. Object ACEs are used only with Active Directory. They are known as object ACEs because they contain GUIDs that make it possible for them to work with the many types of objects available with directory services. Because standard ACEs are far and away the most common, we will cover them first.

Standard ACEs Most securable objects in the system deal exclusively with standard ACEs. The exceptions are objects that exist in Active Directory, which are referred to as *directory services objects*.

Like all ACEs, the standard ACEs are represented by three structures, one each for allowing access, denying access, and auditing. Here is the definition of each of these structures:

```
typedef struct _ACCESS_DENIED_ACE {
   ACE_HEADER  Header;
   ACCESS_MASK Mask;
   DWORD       SidStart;
} ACCESS_DENIED_ACE;

typedef struct _ACCESS_ALLOWED_ACE {
   ACE_HEADER  Header;
   ACCESS_MASK Mask;
   DWORD       SidStart;
} ACCESS_ALLOWED_ACE;

typedef struct _SYSTEM_AUDIT_ACE {
   ACE_HEADER  Header;
   ACCESS_MASK Mask;
   DWORD       SidStart;
} SYSTEM_AUDIT_ACE;
```

The definition of each of these structures is identical, and each includes the promised *Header* member, which is the now-familiar type ACE_HEADER.

The remaining two members are relatively easy to understand. The first is *Mask*, which is a 32-bit mask indicating the access rights that are being denied, allowed, or set for auditing.

I will discuss access rights in great detail shortly, but for now remember that the access rights are broken up into generic, standard, and specific rights for a given object type, and are combined to form a single 32-bit value. This 32-bit value is the *Mask* member of an ACE. See Figure 10-2 for the access mask format.

The last member of the ACE structure is *SidStart*, which indicates the beginning of the SID of the trustee account for which the ACE is denying, allowing, or auditing access for the object. This member requires some discussion.

Remember that a SID (discussed in Chapter 9) is a variable-length binary structure indicating a trustee account of the system. Because each ACE contains a SID, the ACE structure is also a variable-length structure. The *SidStart* member is a placeholder for the beginning of the SID contained within the ACE. The value (and the type) of *SidStart* is irrelevant and should never be accessed, because it is actually the first couple of bytes of a SID structure.

You might find that a macro such as the following is useful in extracting a SID from an ACE structure:

```
#define PSIDFromPACE(pACE) ((PSID)(&((pACE)->SidStart)))
```

This particular macro takes a pointer to any ACE structure—object or standard—and returns a pointer to the SID structure for the ACE.

The standard ACE type is broken into three separate structures, which are used primarily as logical placeholders for you when you create ACEs for an ACL. However, because the structure of each ACE is

identical, when reading ACEs from a DACL, it is very common to write code that uses only one of the ACE types but uses the *AceType* member of the ACE_HEADER structure to maintain a type for the ACE. The following function uses this technique and prints information about each ACE in a DACL:

```
void DumpACL( PACL pACL ){
   __try{
      if (pACL == NULL){
         _tprintf(TEXT("NULL DACL\n"));
         __leave;
      }

      ACL_SIZE_INFORMATION aclSize = {0};
      if (!GetAclInformation(pACL, &aclSize, sizeof(aclSize),
         AclSizeInformation))
         __leave;
      _tprintf(TEXT("ACL ACE count: %d\n"), aclSize.AceCount);

      struct{
         BYTE  lACEType;
         PTSTR pszTypeName;
      }aceTypes[6] = {
         {ACCESS_ALLOWED_ACE_TYPE, TEXT("ACCESS_ALLOWED_ACE_TYPE")},
         {ACCESS_DENIED_ACE_TYPE, TEXT("ACCESS_DENIED_ACE_TYPE")},
         {SYSTEM_AUDIT_ACE_TYPE, TEXT("SYSTEM_AUDIT_ACE_TYPE")},
         {ACCESS_ALLOWED_OBJECT_ACE_TYPE,
            TEXT("ACCESS_ALLOWED_OBJECT_ACE_TYPE")},
         {ACCESS_DENIED_OBJECT_ACE_TYPE,
            TEXT("ACCESS_DENIED_OBJECT_ACE_TYPE")},
         {SYSTEM_AUDIT_OBJECT_ACE_TYPE,
            TEXT("SYSTEM_AUDIT_OBJECT_ACE_TYPE")}};

      struct{
         ULONG lACEFlag;
         PTSTR pszFlagName;
      }aceFlags[7] = {
         {INHERITED_ACE, TEXT("INHERITED_ACE")},
         {CONTAINER_INHERIT_ACE, TEXT("CONTAINER_INHERIT_ACE")},
         {OBJECT_INHERIT_ACE, TEXT("OBJECT_INHERIT_ACE")},
         {INHERIT_ONLY_ACE, TEXT("INHERIT_ONLY_ACE")},
         {NO_PROPAGATE_INHERIT_ACE, TEXT("NO_PROPAGATE_INHERIT_ACE")},
         {FAILED_ACCESS_ACE_FLAG, TEXT("FAILED_ACCESS_ACE_FLAG")},
         {SUCCESSFUL_ACCESS_ACE_FLAG,
            TEXT("SUCCESSFUL_ACCESS_ACE_FLAG")}};

      for (ULONG lIndex = 0;lIndex < aclSize.AceCount;lIndex++){
         ACCESS_ALLOWED_ACE* pACE;
         if (!GetAce(pACL, lIndex, (PVOID*)&pACE))
            __leave;

         _tprintf(TEXT("\nACE #%d\n"), lIndex);

         ULONG lIndex2 = 6;
         PTSTR pszString = TEXT("Unknown ACE Type");
         while (lIndex2--){
            if(pACE->Header.AceType == aceTypes[lIndex2].lACEType){
               pszString = aceTypes[lIndex2].pszTypeName;
            }
         }
         _tprintf(TEXT("  ACE Type =\n  \t%s\n"), pszString);

         _tprintf(TEXT("  ACE Flags = \n"));
         lIndex2 = 7;
         while (lIndex2--){
            if ((pACE->Header.AceFlags & aceFlags[lIndex2].lACEFlag)
               != 0)
```

```
                _tprintf(TEXT("  \t%s\n"),
                   aceFlags[lIndex2].pszFlagName);
            }

            _tprintf(TEXT("  ACE Mask (32->0) =\n  \t"));
            lIndex2 = (ULONG)1<<31;
            while (lIndex2){
               _tprintf(((pACE->Mask & lIndex2) != 0)?TEXT("1"):TEXT("0"));
               lIndex2>>=1;
            }

            TCHAR szName[1024];
            TCHAR szDom[1024];
            PSID pSID = PSIDFromPACE(pACE);
            SID_NAME_USE sidUse;
            ULONG lLen1 = 1024, lLen2 = 1024;
            if (!LookupAccountSid(NULL, pSID,
               szName, &lLen1, szDom, &lLen2, &sidUse))
               lstrcpy(szName, TEXT("Unknown"));
            PTSTR pszSID;
            if (!ConvertSidToStringSid(pSID, &pszSID))
               __leave;
            _tprintf(TEXT("\n  ACE SID =\n  \t%s (%s)\n"), pszSID, szName);
            LocalFree(pszSID);
         }
      }__finally{}
   }
```

Notice for all ACE types of the ACEs that are dumped the use of the ACCESS_ALLOWED_ACE structure. This structure will work for any standard ACE type (although this function will fail for object ACEs) because all standard ACE structures are the same.

The following code fragment could be used with the *DumpACL* function to show the ACEs in a directory on an NTFS share:

```
PSECURITY_DESCRIPTOR pSD;
PACL pDACL;
ULONG lErr = GetNamedSecurityInfo(TEXT("C:\\Test\\Test"),
   SE_FILE_OBJECT, DACL_SECURITY_INFORMATION, NULL, NULL,
   &pDACL, NULL, &pSD);
if (lErr == ERROR_SUCCESS){
   DumpACL(pDACL);
}
```

NOTE

The *DumpACL* sample function demonstrates how to read ACE information in a DACL. You might find code like this useful in your own production code. You might also find the *DumpACL* function useful as a debugging tool, particularly when you are beginning to write code that modifies the DACL of securable objects. Use it to dump out the DACL of the object before and after you make your modifications to see whether your code is producing the desired results.

Techniques such as this one have advantages over viewing security by using the Security Properties page in Explorer, because the system's user interface does not display the ACE information of an object verbatim. In fact, the user interface will not display a DACL that is not properly ordered without first ordering the ACEs in the DACL.

Because of the nature of the ACE structures, you might find it helpful to define a union that represents each ACE type. Then you can deal with ACEs returned from *GetAce* in terms of a pointer to the union type. The following code shows an example of such a technique:

```
typedef union _ACE_UNION{
   ACE_HEADER         aceHeader;
   ACCESS_ALLOWED_ACE aceAllowed;
   ACCESS_DENIED_ACE  aceDenied;
   SYSTEM_AUDIT_ACE   aceAudit;
}*PACE_UNION;

PACE_UNION pACE ;
GetAce(pDACL, 0, (PVOID*)&pACE);

switch (pACE->aceHeader.AceType){
```

Object ACEs Unless you are writing code to secure and modify the security of objects in Active Directory, you are not likely to use object ACEs in your own applications. However, Active Directory is an important component of Windows 2000, and there is no harm in understanding how it secures objects. We won't spend much time on the topic, though, to avoid clouding your understanding of standard ACEs.

Like standard ACEs, object ACEs come in three flavors: one to deny, one to allow, and one to audit access rights for a trustee. Here are the structures for the object ACEs:

```
typedef struct _ACCESS_ALLOWED_OBJECT_ACE {
   ACE_HEADER  Header;
   ACCESS_MASK Mask;
   DWORD       Flags;
   GUID        ObjectType;
   GUID        InheritedObjectType;
   DWORD       SidStart;
} ACCESS_ALLOWED_OBJECT_ACE, *PACCESS_ALLOWED_OBJECT_ACE;

typedef struct _ACCESS_DENIED_OBJECT_ACE {
   ACE_HEADER  Header;
   ACCESS_MASK Mask;
   DWORD       Flags;
   GUID        ObjectType;
   GUID        InheritedObjectType;
   DWORD       SidStart;
} ACCESS_DENIED_OBJECT_ACE, *PACCESS_DENIED_OBJECT_ACE;

typedef struct _SYSTEM_AUDIT_OBJECT_ACE {
   ACE_HEADER  Header;
   ACCESS_MASK Mask;
   DWORD       Flags;
   GUID        ObjectType;
   GUID        InheritedObjectType;
   DWORD       SidStart;
} SYSTEM_AUDIT_OBJECT_ACE, *PSYSTEM_AUDIT_OBJECT_ACE;
```

Notice that like standard ACEs, each structure type is the same and the members of each structure are the same except for the addition of the *Flags*, *ObjectType*, and *InheritedObjectType* members.

Take care not to confuse the *Flags* member of the object ACE structures with the *AceFlags* member of the ACE_HEADER structure. The *Flags* member can be 0 or any combination of the values in Table 10-12.

NOTE

Unlike similar structures in the Win32 SDK, the flags in Table 10-12 do not indicate whether the *ObjectType* and *InheritedObjectType* members are *used* in the structure, but rather whether these members exist in the structure at all!

Structures that do not have a fixed member list are referred to as *amorphous*. Each of the object ACE structures is amorphous, which is very uncommon for a structure in the Win32

SDK. This is by far the most difficult aspect of dealing with object ACEs.

Table 10-12. *Object ACE* Flags *member values*

Value	Description
ACE_OBJECT_TYPE_PRESENT	Indicates that the *ObjectType* member is present in the object ACE structure
ACE_INHERITED_OBJECT_TYPE_PRESENT	Indicates that the *InheritedObjectType* member is present in the object ACE structure

Both *ObjectType* and *InheritedObjectType* are GUIDs. If you are unfamiliar with GUIDs, you can find a complete discussion of the topic in the Platform SDK documentation or *Inside COM* (Dale Rogerson, Microsoft Press, 1997). For our purposes, you need to understand only that a GUID is a 128-bit value that can be used to uniquely identify just about anything.

Fortunately, the Active Directory Service Interfaces (ADSI) provide interfaces for manipulating security on Active Directory objects that remove from your code the responsibility of dealing with object ACEs. For this reason, it is highly unlikely that you will have to write code that directly manipulates object ACEs in a DACL. For more information on this topic, see the Platform SDK discussion on the *IADsSecurityDescriptor*, *IADsAccessControlList*, and *IADsAccessControlEntry* interfaces.

NOTE

Object ACEs are never found in the ACLs of securable objects outside of Active Directory, and they significantly complicate ACL manipulation code. For this reason the remainder of this chapter (and all the code in this chapter) assumes ACLs do not contain object ACEs. ACLs without object ACEs ease my job as a writer and simplify your job of understanding a complicated topic.

Access Rights

Access rights might at first seem like a simple concept to grasp, but some key points might not be obvious. The best way to understand the nuances of access rights is to discuss where they are used and to examine each group. First let's review what we already know:

- A trustee must hold a combination of access rights before the system will perform any securable task on a securable object on behalf of that trustee.
- All access rights are packed into a 32-bit access mask. (See Figure 10-2.)
- Access rights are divided into three groups: generic rights (bits 31-28), standard rights (bits 22-16), and specific rights (bits 15-0).
- Two "odd" bits in an access mask do not fit into an access right group: the auditing bit and the maximum allowed bit.

Up to this point, I have discussed access rights in terms of the access mask stored in an ACE, which is only one side of the transaction. I said that rights are necessary to perform tasks, but I have not specifically stated how you *ask* the system for a right. Asking for a right is the other side of the transaction.

You typically ask for a right when you ask the system for a handle to an object. Most functions that return handles to securable objects have a required access parameter to which you pass a value indicating how you

intend to use the handle. Look at the following example:

```
HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, TEXT("MyEvent"));
```

You are probably so familiar with code like this that it never occurred to you that you were asking the system for a specific access right to the event object, but you are. The flag bits that you pass in the desired access parameter of *OpenEvent* (and other similar functions) are the same bits that are stored in the ACEs that secure the object. In fact, the system performs an access check before returning the handle to the event.

NOTE

Many secured objects in Windows are accessed through handles. With these objects, security is checked when the handle is requested, and your rights are stored with the handle. This means that once you have received a handle to an object, the access rights on the object itself might change, but the access provided to you by your handle remains the same until the handle is closed.

Some secure actions are performed on objects for which you never see a handle. A perfect example of this is every call to *GetNamedSecurityInfo*. The convenience of this function is that you do not have to first retrieve a handle to an object. In this case, the system internally performs an access check before performing any securable activity.

Now that you are familiar with both the ACE side and the access request side of access rights, let's discuss the types of access rights in more detail. First, we need to get the "odd" bits in the access mask out of the way.

The auditing bit represents the access right ACCESS_SYSTEM_SECURITY, which must be held by a trustee before that user can modify the SACL on an object. (I will discuss auditing in more detail later in this chapter.)

The maximum allowed bit represents the "non-access right" MAXIMUM_ALLOWED. I call it a non-access right because it will never be found in an ACE stored in a DACL. It is used only in calls to functions requesting access. For example, the following code fragment uses the MAXIMUM_ALLOWED flag in retrieving a handle to a named event object:

```
HANDLE hEvent = OpenEvent(MAXIMUM_ALLOWED, FALSE, TEXT("MyEvent"));
```

In this case, the system performs an access check on the object, and rather than allowing or denying the access check, it returns an access mask that is the maximum rights awarded to you for the object. This might seem like a very convenient feature (and in certain situations it can be), as it allows you to always pass MAXIMUM_ALLOWED when requesting a handle rather than forcing you to figure out exactly which rights you need for an object. However, there is a very good reason not to do this.

When you obtain a handle to an object, typically you are being told by the system that you are allowed to use the object in a certain way. If you are denied a request for a handle, this typically means that you are not allowed to do whatever you have requested, and your software can take the appropriate measures.

If you pass MAXIMUM_ALLOWED each time you need access to an object, your code might work well for a long time. But the first time your code encounters an object for which you are not awarded sufficient access, your software might not catch the error until long after it has retrieved a handle to the object. The failing function might not even return ERROR_ACCESS_DENIED. It is entirely possible that a function will return another error such as ERROR_INVALID_PARAMETER, because you passed a handle to it with insufficient access to an object. Code with this type of problem can be drastically harder to debug.

Standard access rights Five standard rights are defined by the system. In addition to these, another five "composite" access rights are defined by the system that map to some combination of the five standard rights.

Not all standard rights are relevant for all securable objects in the system, but each standard right is special in that its meaning does not change from object to object. For example, not all objects can be deleted, so the standard right DELETE has no meaning for certain objects. However, for objects that can be deleted, the standard right DELETE carries the same meaning from one object type to the next: the holder of this right can delete the object. Table 10-13 lists all the standard and composite rights.

Table 10-13. *Standard and composite rights*

Right	Description
Standard Rights	
DELETE (bit 16)	Delete access.
READ_CONTROL (bit 17)	Read access to the security information of a securable object, including the owner SID, the group SID, the DACL, and the security descriptor revision and control. It does not include the access required to read the SACL of a secured object.
WRITE_DAC (bit 18)	Write access to the DACL and group SID of a secured object. WRITE_DAC access is implicitly granted to the owner of an object.
WRITE_OWNER (bit 19)	Write access to the owner SID of a secured object. Unless you also hold the SE_BACKUP_NAME privilege, you can write a SID that only denotes your user SID or one of your group SIDs to the owner SID of an object.
SYNCHRONIZE (bit 20)	Synchronize access can be thought of as the right to wait on an object. Such an object can be a synchronization object (for example, an event or mutex), or it can be one of the other waitable kernel objects (for example, a file or a process handle).
Standard Right Composites	
STANDARD_RIGHTS_READ STANDARD_RIGHTS_WRITE STANDARD_RIGHTS_EXECUTE	Maps to READ_CONTROL access.
STANDARD_RIGHTS_REQUIRED	Maps to bits 16-19; or to DELETE, READ_ CONTROL, WRITE_DAC, and WRITE_OWNER. This access right is typically included with the "all access" defines for specific objects such as EVENT_ALL_ACCESS or FILE_ALL_ACCESS.
STANDARD_RIGHTS_ALL	Defined as 0x001F0000, this access right includes all the standard rights.

As Table 10-13 shows, the standard rights include some powerful rights. For example, if you hold WRITE_OWNER access to an object, but are denied all other access, you might not be able to do much, as is. But if you choose to use this access to set the owner SID of the object to your SID, you have implicit WRITE_DAC access. You can then use this access to modify the DACL of the object, allowing yourself all the access you want to this object.

Specific rights Specific rights inhabit bits 15-0 of the access mask, and therefore each securable object in the system can have 16 different access rights defined for it.

Because the specific rights differ from object to object, the greatest challenge in working with specific rights is finding a comprehensive list of all specific rights available for a given object. You should look for specific rights in two places: the Platform SDK documentation and the Platform SDK header files. You should check the function that gives you a handle to an existing object of the type you are interested in. For example, if you were interested in finding the specific rights available for a registry key, you might check the Platform SDK documentation for the *RegOpenKeyEx* function, where you would find a description of rights such as KEY_READ, KEY_SET_VALUE, and KEY_ALL_ACCESS. After you have the names of a couple of access rights for an object, you can search the Platform SDK's include directory for a header file that defines one or more of the specific rights that you know, and you will find the remaining access rights for that object.

Table 10-14 lists the specific rights, as of this writing, for the most common securable objects in Windows.

SPECIFIC_RIGHTS_ALL is defined as 0x0000FFFF, which includes all 16 of the bits, reserved for specific rights in an access mask. Object Type Specific Right

Table 10-14. *Specific rights for common securable objects*

Object Type	Specific Right
File (WinNT.h)	
	FILE_ALL_ACCESS (STANDARD_RIGHTS_REQUIRED | SYNCHRONIZE | 0x1FF) (0x1FF includes all currently defined standard rights for files.)
	FILE_READ_DATA
	FILE_WRITE_DATA
	FILE_APPEND_DATA
	FILE_READ_EA
	FILE_WRITE_EA
	FILE_EXECUTE
	FILE_READ_ATTRIBUTES
	FILE_WRITE_ATTRIBUTES
Directory (WinNT.h)	
	FILE_ALL_ACCESS (STANDARD_RIGHTS_REQUIRED | SYNCHRONIZE | 0x1FF) (0x1FF includes all currently defined standard rights for files.)
	FILE_LIST_DIRECTORY
	FILE_ADD_FILE
	FILE_ADD_SUBDIRECTORY
	FILE_READ_EA
	FILE_WRITE_EA
	FILE_TRAVERSE
	FILE_DELETE_CHILD
	FILE_READ_ATTRIBUTES
	FILE_WRITE_ATTRIBUTES
Service (WinSvc.h)	

SERVICE_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
SERVICE_QUERY_CONFIG |
SERVICE_CHANGE_CONFIG |
SERVICE_QUERY_STATUS |
SERVICE_ENUMERATE_DEPENDENTS |
SERVICE_START |
SERVICE_STOP |
SERVICE_PAUSE_CONTINUE |
SERVICE_INTERROGATE |
SERVICE_USER_DEFINED_CONTROL)
SERVICE_CHANGE_CONFIG
SERVICE_ENUMERATE_DEPENDENTS
SERVICE_INTERROGATE
SERVICE_PAUSE_CONTINUE
SERVICE_QUERY_CONFIG
SERVICE_QUERY_STATUS
SERVICE_START
SERVICE_STOP
SERVICE_USER_DEFINED_CONTROL

Printer
(WinSpool.h)

SERVER_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
SERVER_ACCESS_ADMINISTER |
SERVER_ACCESS_ENUMERATE)
PRINTER_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
PRINTER_ACCESS_ADMINISTER |
PRINTER_ACCESS_USE)
JOB_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
JOB_ACCESS_ADMINISTER)
SERVER_ACCESS_ADMINISTER
SERVER_ACCESS_ENUMERATE
PRINTER_ACCESS_ADMINISTER
PRINTER_ACCESS_USE
JOB_ACCESS_ADMINISTER

Registry key
(WinNT.h)

KEY_ALL_ACCESS
(STANDARD_RIGHTS_ALL |
KEY_QUERY_VALUE |
KEY_SET_VALUE |
KEY_CREATE_SUB_KEY |
KEY_ENUMERATE_SUB_KEYS |
KEY_NOTIFY |
KEY_CREATE_LINK)
KEY_QUERY_VALUE
KEY_SET_VALUE
KEY_CREATE_SUB_KEY

KEY_ENUMERATE_SUB_KEYS
KEY_NOTIFY
KEY_CREATE_LINK

Share object
(LMShare.h)

PERM_FILE_READ
PERM_FILE_WRITE
PERM_FILE_CREATE

Process
(WinNT.h)

PROCESS_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
SYNCHRONIZE |
0xFFF)
(0xFFF includes all currently defined standard rights for processes.)
PROCESS_TERMINATE
PROCESS_CREATE_THREAD
PROCESS_SET_SESSIONID
PROCESS_VM_OPERATION
PROCESS_VM_READ
PROCESS_VM_WRITE
PROCESS_DUP_HANDLE
PROCESS_CREATE_PROCESS
PROCESS_SET_QUOTA
PROCESS_SET_INFORMATION
PROCESS_QUERY_INFORMATION

Thread
(WinNT.h)

THREAD_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
SYNCHRONIZE |
0x3FF)
(0x3FF includes all currently defined standard rights for threads.)
THREAD_TERMINATE
THREAD_SUSPEND_RESUME
THREAD_GET_CONTEXT
THREAD_SET_CONTEXT
THREAD_SET_INFORMATION
THREAD_QUERY_INFORMATION
THREAD_SET_THREAD_TOKEN
THREAD_IMPERSONATE
THREAD_DIRECT_IMPERSONATION

Job
(WinNT.h)

JOB_OBJECT_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
SYNCHRONIZE |
0x1F)
(0x1F includes all currently defined standard rights for job objects.)

JOB_OBJECT_ASSIGN_PROCESS
JOB_OBJECT_SET_ATTRIBUTES
JOB_OBJECT_QUERY
JOB_OBJECT_TERMINATE
JOB_OBJECT_SET_SECURITY_ATTRIBUTES

Semaphore
(WinNT.h)

SEMAPHORE_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
SYNCHRONIZE |
0x3)
(0x3 includes all currently defined standard rights for semaphores.)
SEMAPHORE_MODIFY_STATE
MUTANT_QUERY_STATE

Event
(WinNT.h)

EVENT_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
SYNCHRONIZE |
0x3)
(0x3 includes all currently defined standard rights for events.)
EVENT_MODIFY_STATE
MUTANT_QUERY_STATE

Mutex
(WinBase.h)

MUTEX_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
SYNCHRONIZE |
0x3)
(0x3 includes all currently defined standard rights for mutexes.)
MUTEX_MODIFY_STATE
MUTANT_QUERY_STATE

File map object
(WinBase.h)

FILE_MAP_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
FILE_MAP_COPY |
FILE_MAP_WRITE |
FILE_MAP_READ |
SECTION_MAP_EXECUTE
SECTION_EXTEND_SIZE)
FILE_MAP_WRITE
FILE_MAP_READ
FILE_MAP_COPY
SECTION_EXTEND_SIZE

Waitable timer
(WinNT.h)

TIMER_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
SYNCHRONIZE |
TIMER_QUERY_STATE |
TIMER_MODIFY_STATE)
TIMER_QUERY_STATE
TIMER_MODIFY_STATE

Token
(WinNT.h)

TOKEN_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
TOKEN_ASSIGN_PRIMARY |
TOKEN_DUPLICATE |
TOKEN_IMPERSONATE |
TOKEN_QUERY |
TOKEN_QUERY_SOURCE |
TOKEN_ADJUST_PRIVILEGES |
TOKEN_ADJUST_GROUPS |
TOKEN_ADJUST_SESSIONID |
TOKEN_ADJUST_DEFAULT)
TOKEN_ASSIGN_PRIMARY
TOKEN_DUPLICATE
TOKEN_IMPERSONATE
TOKEN_QUERY
TOKEN_QUERY_SOURCE
TOKEN_ADJUST_PRIVILEGES
TOKEN_ADJUST_GROUPS
TOKEN_ADJUST_DEFAULT
TOKEN_ADJUST_SESSIONID

Pipe
(WinNT.h)

FILE_ALL_ACCESS
(STANDARD_RIGHTS_REQUIRED |
SYNCHRONIZE |
0x1FF)
(0x1FF includes all currently defined standard rights for files.)
FILE_READ_DATA
FILE_WRITE_DATA
FILE_CREATE_PIPE_INSTANCE
FILE_READ_ATTRIBUTES
FILE_WRITE_ATTRIBUTES

Window station
(WinUser.h)

WINSTA_ACCESSCLIPBOARD
WINSTA_ACCESSGLOBALATOMS
WINSTA_CREATEDESKTOP
WINSTA_ENUMDESKTOPS
WINSTA_ENUMERATE
WINSTA_EXITWINDOWS
WINSTA_READATTRIBUTES

	WINSTA_READSCREEN WINSTA_WRITEATTRIBUTES
Desktop (WinUser.h)	
	DESKTOP_CREATEMENU DESKTOP_CREATEWINDOW DESKTOP_ENUMERATE DESKTOP_HOOKCONTROL DESKTOP_JOURNALPLAYBACK DESKTOP_JOURNALRECORD DESKTOP_READOBJECTS DESKTOP_SWITCHDESKTOP DESKTOP_WRITEOBJECTS

With these specific rights and the standard rights discussed earlier, you have all the rights required to secure an object. However, there is still the issue of generic rights.

Generic rights and default security Previously in this chapter I mentioned that default security is assigned to securable objects when the software creating the object does not explicitly set the security for the object. This assignment is typically characterized by passing NULL for the security attributes parameter of a function that creates a securable object such as a file or an event. Before we can discuss generic rights, you must understand how default security is implemented.

Remember that your executing code is associated with an internal structure known as a token. Up to this point, I have said that the token contains your identifying SID and group SIDs, as well as a list of privileges assigned to you. In addition to this information, every token also stores a DACL that is used for creating objects with default security, known as the *default DACL*, and can be set by your code. (This and many other topics regarding tokens are covered in detail in the next chapter.)

For the moment, you need to be aware that a DACL exists (which you can modify and set) and that it can be applied to objects created with default security. The important issue to note is that the DACL can be applied to any type of object. So you can see that this creates a problem for the system.

NOTE

Neither the DACL nor the standard ACE structures maintain information about the object type that they are protecting. This complicates the creation of ACEs for a default DACL, since specific rights for one type of object might not be appropriate if applied to another type of object. The creators of Windows solved this problem by creating generic rights.

Generic rights are intended for use in the default DACL for a token. The system defines the generic rights GENERIC_READ, GENERIC_WRITE, GENERIC_EXECUTE, and GENERIC_ALL. When a default DACL is applied to an object, the system maps any generic rights found in the DACL's ACEs to a combination of standard and specific rights appropriate for that object. In this way, an ACE that will appropriately affect the security on any object that uses that ACE can be added to the default DACL.

NOTE

You cannot directly assign a DACL to an object that contains ACEs whose access masks include generic rights. The system automatically clears these rights from the ACE before setting the security on the object.

NOTE

You can specify generic rights when requesting access to an object, but you should never do this. It is more appropriate to request the standard and specific rights that will meet the needs of your code.

Knowing what you know about generic rights, you might be wondering how you can find out what generic rights map to for a specific object in the system. This is a good question, and the only good answer is to create a default DACL with an ACE that includes a single generic right (such as GENERIC_READ), and then create an object and inspect the resulting ACE. (See Chapter 11.) I did this to produce Table 10-15.

Table 10-15. *Generic mappings for common securable objects*

Object Type	Generic Right	Standard and Specific Mapped Rights
File		
	GENERIC_READ	FILE_GENERIC_READ
	GENERIC_WRITE	FILE_GENERIC_WRITE
	GENERIC_EXECUTE	FILE_GENERIC_EXECUTE
	GENERIC_ALL	FILE_ALL_ACCESS
Directory		
	GENERIC_READ	STANDARD_RIGHTS_READ | FILE_LIST_DIRECTORY | FILE_ADD_FILE
	GENERIC_WRITE	STANDARD_RIGHTS_WRITE | FILE_ADD_SUBDIRECTORY | FILE_READ_EA
	GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE | FILE_LIST_DIRECTORY | FILE_ADD_FILE
	GENERIC_ALL	STANDARD_RIGHTS_REQUIRED | FILE_LIST_DIRECTORY | FILE_ADD_FILE | FILE_ADD_SUBDIRECTORY | FILE_READ_EA
Service		
	GENERIC_READ	STANDARD_RIGHTS_READ | SERVICE_QUERY_CONFIG | SERVICE_QUERY_STATUS | SERVICE_ENUMERATE_DEPENDENTS | SERVICE_INTERROGATE | SERVICE_USER_DEFINED_CONTROL
	GENERIC_WRITE	STANDARD_RIGHTS_WRITE | SERVICE_CHANGE_CONFIG
	GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE | SERVICE_START | SERVICE_STOP | SERVICE_PAUSE_CONTINUE | SERVICE_INTERROGATE |

		SERVICE_USER_DEFINED_CONTROL
	GENERIC_ALL	SERVICE_ALL_ACCESS
Printer		
	GENERIC_READ	PRINTER_READ
	GENERIC_WRITE	PRINTER_WRITE
	GENERIC_EXECUTE	PRINTER_EXECUTE
	GENERIC_ALL	PRINTER_ALL_ACCESS
Registry key		
	GENERIC_READ	KEY_READ
	GENERIC_WRITE	KEY_WRITE
	GENERIC_EXECUTE	KEY_EXECUTE
	GENERIC_ALL	KEY_ALL_ACCESS
Process		
	GENERIC_READ	STANDARD_RIGHTS_READ | PROCESS_VM_READ | PROCESS_QUERY_INFORMATION
	GENERIC_WRITE	STANDARD_RIGHTS_WRITE | PROCESS_CREATE_PROCESS | PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION | PROCESS_VM_WRITE | PROCESS_DUP_HANDLE | PROCESS_TERMINATE | PROCESS_SET_QUOTA | PROCESS_SET_INFORMATION
	GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE | SYNCHRONIZE
	GENERIC_ALL	PROCESS_ALL_ACCESS
Thread		
	GENERIC_READ	STANDARD_RIGHTS_READ | THREAD_GET_CONTEXT | THREAD_QUERY_INFORMATION
	GENERIC_WRITE	STANDARD_RIGHTS_WRITE | THREAD_TERMINATE | THREAD_SUSPEND_RESUME | THREAD_SET_INFORMATION | THREAD_SET_CONTEXT
	GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE | SYNCHRONIZE
	GENERIC_ALL	THREAD_ALL_ACCESS
Job		
	GENERIC_READ	STANDARD_RIGHTS_READ | JOB_OBJECT_QUERY

	GENERIC_WRITE	STANDARD_RIGHTS_WRITE | JOB_OBJECT_ASSIGN_PROCESS | JOB_OBJECT_SET_ATTRIBUTES | JOB_OBJECT_TERMINATE
	GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE | SYNCHRONIZE
	GENERIC_ALL	JOB_OBJECT_ALL_ACCESS
Semaphore		
	GENERIC_READ	STANDARD_RIGHTS_READ | MUTANT_QUERY_STATE
	GENERIC_WRITE	STANDARD_RIGHTS_WRITE | SEMAPHORE_MODIFY_STATE
	GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE | SYNCHRONIZE
	GENERIC_ALL	SEMAPHORE_ALL_ACCESS
Event		
	GENERIC_READ	STANDARD_RIGHTS_READ | MUTANT_QUERY_STATE
	GENERIC_WRITE	STANDARD_RIGHTS_WRITE | EVENT_MODIFY_STATE
	GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE | SYNCHRONIZE
	GENERIC_ALL	EVENT_ALL_ACCESS
Mutex		
	GENERIC_READ	STANDARD_RIGHTS_READ | MUTANT_QUERY_STATE
	GENERIC_WRITE	STANDARD_RIGHTS_WRITE
	GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE | SYNCHRONIZE
	GENERIC_ALL	MUTEX_ALL_ACCESS
File map object		
	GENERIC_READ	STANDARD_RIGHTS_READ | FILE_MAP_COPY | FILE_MAP_READ
	GENERIC_WRITE	STANDARD_RIGHTS_WRITE | FILE_MAP_WRITE
	GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE | FILE_MAP_EXECUTE
	GENERIC_ALL	FILE_MAP_ALL_ACCESS

Waitable timer		
	GENERIC_READ	STANDARD_RIGHTS_READ | TIMER_QUERY_STATE
	GENERIC_WRITE	STANDARD_RIGHTS_WRITE | TIMER_MODIFY_STATE
	GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE | SYNCHRONIZE
	GENERIC_ALL	TIMER_ALL_ACCESS
Token		
	GENERIC_READ	TOKEN_READ
	GENERIC_WRITE	TOKEN_WRITE
	GENERIC_EXECUTE	TOKEN_EXECUTE
	GENERIC_ALL	TOKEN_ALL_ACCESS
Window station		
	GENERIC_READ	WINSTA_ENUMDESKTOPS | WINSTA_READATTRIBUTES | WINSTA_ENUMERATE | WNSTA_READSCREEN | STANDARD_RIGHTS_READ
	GENERIC_WRITE	WINSTA_ACCESSCLIPBOARD | WINSTA_CREATEDESKTOP | WINSTA_WRITEATTRIBUTES | STANDARD_RIGHTS_WRITE
	GENERIC_EXECUTE	WINSTA_ACCESSGLOBALATOMS | WINSTA_EXITWINDOWS | STANDARD_RIGHTS_EXECUTE
	GENERIC_ALL	WINSTA_ENUMDESKTOPS | WINSTA_READATTRIBUTES | WINSTA_ENUMERATE | WINSTA_READSCREEN | WINSTA_ACCESSCLIPBOARD | WINSTA_CREATEDESKTOP | WINSTA_WRITEATTRIBUTES | WINSTA_ACCESSGLOBALATOMS | WINSTA_EXITWINDOWS | STANDARD_RIGHTS_REQUIRED
Desktop		
	GENERIC_READ	DESKTOP_READOBJECTS | DESKTOP_ENUMERATE | STANDARD_RIGHTS_READ
	GENERIC_WRITE	DESKTOP_WRITEOBJECTS | DESKTOP_CREATEWINDOW | DESKTOP_CREATEMENU |

	DESKTOP_HOOKCONTROL |
	DESKTOP_JOURNALRECORD |
	DESKTOP_JOURNALPLAYBACK |
	STANDARD_RIGHTS_WRITE
GENERIC_EXECUTE	DESKTOP_SWITCHDESKTOP |
	STANDARD_RIGHTS_EXECUTE
GENERIC_ALL	DESKTOP_READOBJECTS |
	DESKTOP_WRITEOBJECTS |
	DESKTOP_ENUMERATE |
	DESKTOP_CREATEWINDOW |
	DESKTOP_CREATEMENU |
	DESKTOP_HOOKCONTROL |
	DESKTOP_JOURNALRECORD |
	DESKTOP_JOURNALPLAYBACK |
	DESKTOP_SWITCHDESKTOP |
	STANDARD_RIGHTS_REQUIRED

Before we proceed, I must make one more point about default security. If you are dealing with objects for which security is inheritable (such as registry keys or files), the default security is applied only to objects for which no ACEs are inherited from parent objects. In both registry and file system hierarchies, ACEs are inherited by default, so *default security* is not applied in the common case.

Now that you are familiar with all types of access rights, as well as the ACE and ACL types that contain these rights, you have all the tools necessary to read and interpret the security of any object in the system.

The AccessMaster Sample Application

The AccessMaster sample application ("10 AccessMaster.exe") demonstrates the usage of the *GetNamedSecurityInfo*, *SetNamedSecurityInfo*, *GetSecurityInfo*, and *SetSecurityInfo* functions as well as the *ISecurityInformation* interface for the *EditSecurity* common dialog. The source code and resource files for the application are in the AccessMaster directory on the companion CD. Figure 10-7 shows AccessMaster being used to set the security for a file.

Figure 10-7. *AccessMaster being used to set the security for a file*

In addition to showing how to programmatically get and set the security for common objects in the system, the AccessMaster sample application can be a very useful tool for understanding access control for securable

objects. You can access objects by name or by process ID and numeric handle value (use decimal values). You can also view ACEs with raw binary access masks rather than map them to standard and specific rights for the objects.

The AccessMaster tool also lets you set the security on securable objects in the system. Remember that because you might not have rights to all objects in the system, you have to take ownership of an object before you can read or write its other security information. To accomplish this, you can use the TrusteeMan sample application from Chapter 9 to assign the SE_TAKE_ OWNERSHIP_NAME privilege to your user account.

If you are unfamiliar with the ramifications of access control on objects in the system, including objects created by your software, you can use the AccessMaster tool to view and modify the access rights assigned to these objects.

Setting Security Information for an Object

The next step in programming for access control is setting the security of a securable object. For most securable objects in the system, you do this by calling *SetSecurityInfo* or *SetNamedSecurityInfo*. (For a list of securable objects and the functions used to get and set their security, see Table 10-7.)

Here is the definition of the *SetSecurityInfo* function:

```
DWORD SetSecurityInfo(
   HANDLE               handle,
   SE_OBJECT_TYPE       objType,
   SECURITY_INFORMATION secInfo,
   PSID                 psidOwner,
   PSID                 psidGroup,
   PACL                 pDACL,
   PACL                 pSACLl);
```

And here is the definition of the *SetNamedSecurityInfo* function:

```
DWORD SetNamedSecurityInfo(
   LPTSTR               pObjectName,
   SE_OBJECT_TYPE       objType,
   SECURITY_INFORMATION secInfo,
   PSID                 psidOwner,
   PSID                 psidGroup,
   PACL                 pDACL,
   PACL                 pSACL);
```

As you can see, these functions are very similar, except that *SetSecurityInfo* sets the security of an object for which you have a handle, and *SetNamedSecurityInfo* sets the security of a named object in the system and does not require a handle.

These functions also have a strong resemblance to their cousins, *GetSecurityInfo* and *GetNamedSecurityInfo*. Notice that the *objType* parameter of the "SetSecurity" functions defines the type of object for which you are setting security information, and uses the same object types as the "GetSecurity" functions. (See Table 10-7.)

The *secInfo* parameter indicates which components of the object's security descriptor you wish to set. These components can be any combination of the object's owner SID, group SID, DACL, and SACL. Additionally, this parameter is used to set a security descriptor as protected from inherited ACEs in parent DACLs and SACLs. Table 10-16 shows all values that can be passed as the *secInfo* parameter to *SetNamedSecurityInfo* and *SetNamedSecurity.* You can combine any of these values to indicate exactly what and how to apply the security to your object.

Table 10-16. *Values that can be passed as the* secInfo *parameter to* SetNamedSecurityInfo *and* SetNamedSecurity

Value	Description
DACL_SECURITY_INFORMATION	Indicates that you wish to set DACL information for a securable object.
SACL_SECURITY_INFORMATION	Indicates that you wish to set SACL information for a securable object.
OWNER_SECURITY_INFORMATION	Indicates that you wish to set the owner SID for a securable object.
GROUP_SECURITY_INFORMATION	Indicates that you wish to set the group SID for a securable object.
UNPROTECTED_DACL_SECURITY_INFORMATION	Indicates that you want inherited ACEs in parent objects to propagate to this object's DACL. This flag must be used with DACL_SECURITY_INFORMATION and cannot be used with PROTECTED_DACL_ SECURITY_INFORMATION.
PROTECTED_DACL_SECURITY_INFORMATION	Indicates that you do not want inherited ACEs in parent objects to propagate to this object's DACL. This flag must be used with DACL_SECURITY_INFORMATION and cannot be used with UNPROTECTED_DACL_ SECURITY_INFORMATION.
UNPROTECTED_SACL_SECURITY_INFORMATION	Indicates that you want inherited ACEs in parent objects to propagate to this object's SACL. This flag must be used with SACL_SECURITY_INFORMATION and cannot be used with PROTECTED_SACL_SECURITY_INFORMATION.
PROTECTED_SACL_SECURITY_INFORMATION	Indicates that you do not want inherited ACEs in parent objects to propagate to this object's SACL. This flag must be used with SACL_SECURITY_INFORMATION and cannot be used with UNPROTECTED_SACL_SECURITY_INFORMATION.

Depending on which flags you pass as the *secInfo* parameter when calling *SetSecurityInfo* or *SetNamedSecurityInfo*, the values you pass for the *psidOwner*, *psidGroup*, *pDACL*, and *pSACL* parameters are ignored or indicate the object's new owner SID, group SID, DACL, and SACL, respectively.

You should pass NULL for any parameters to indicate portions of the security descriptor for which you are not setting information.

NOTE

It can be appropriate to indicate DACL_SECURITY_INFORMATION or SACL_SECURITY_INFORMATION in the *secInfo* parameter and still pass NULL for the *pDACL* or *pSACL* parameter. This indicates a NULL DACL or a NULL SACL.

The following code fragment shows the use of *SetNamedSecurityInfo* to assign a NULL DACL to a registry key and protect its DACL from inherited ACEs found in parent keys' DACLs.

```
ULONG lErr = SetNamedSecurityInfo(
   TEXT("Machine\\Software\\Jason'sKey"), SE_REGISTRY_KEY,
   DACL_SECURITY_INFORMATION|PROTECTED_DACL_SECURITY_INFORMATION,
   NULL, NULL, NULL, NULL);
   if (lErr != ERROR_SUCCESS){
      // Error case
   }
```

NOTE

Remember that a NULL DACL indicates that everyone has all access to the object (even the ability to write the security of the object). In the rare cases where it is appropriate to set NULL security to an object, you must protect the object's DACL (or lack of DACL) from inherited ACEs. Otherwise, the system is forced to create a DACL that will include the ACEs inherited from parent objects, which immediately undermines your goal of NULL security for the object. On the other hand, this side effect of inherited security can be a convenient way to set an object's DACL to include no ACEs except those inherited from parent objects.

As you can see, setting an object's security can be simple. However, it can become significantly more difficult when including a DACL with meaningful ACEs. Whether you are setting the security of a new object (one you create) or an object that already exists, you can take two basic approaches to create the DACL for the object: create a new DACL for the object, or modify an existing DACL.

You will typically create a DACL for new objects, but not always. You can also use an existing DACL from an object and make modifications to it, and then use the modified DACL to create a new object of the same type as the original securable object.

Similarly for existing objects, it is very common to make modifications to the existing DACL and far less common to completely overwrite the security. However, this does not mean that you cannot completely replace the DACL of an existing object.

Because you can be flexible about the approaches depending on your needs, I will cover both, starting with the simpler task of creating a brand new DACL.

Creating a DACL

You will typically follow these steps when creating a DACL:

1. Gather the SIDs for the trustees for which you will be creating ACEs for the DACL.
2. Calculate the size of the new DACL by using the size of the SIDs and the size of the ACE structure that you are using.
3. Allocate memory for the DACL.
4. Initialize the DACL.
5. Use the SIDs to add ACEs to the DACL, taking care to add your access-denied ACEs before adding your access-allowed ACEs.

For existing objects, you will use one of the "SetSecurity" functions to apply the new DACL to the objects. For most new objects, you are required to also create and initialize a security descriptor and a security attributes structure to pass when calling a create function such as *CreateEvent* or *CreateFile*. I will show you how to do this in a moment.

The most complicated part of creating a DACL is calculating its size. The *CalculateACLSize* sample function shows how to do this. To use this function, you pass an array of pointers to SIDs, which are used to calculate the necessary size of the new DACL. I also included an optional pointer to an existing DACL, which, if NULL, is ignored, and if not NULL, adds the size of the ACEs in the existing DACL and gives you the size of a new DACL created from an existing DACL. More on this in a moment, but for now, see the *CalculateACLSize* function here.

```
ULONG CalculateACLSize( PACL pACLOld, PSID* ppSidArray, int nNumSids,
   PACE_UNION* ppACEs, int nNumACEs ){
   ULONG lACLSize = 0;

   try{
      // If we are including an existing ACL, then find its size
      if (pACLOld != NULL){
         ACL_SIZE_INFORMATION aclSize;
         if(!GetAclInformation(pACLOld, &aclSize, sizeof(aclSize),
            AclSizeInformation)){
            goto leave;
         }
         lACLSize = aclSize.AclBytesInUse;
      }

      if (ppSidArray != NULL){
         // Step through each SID
         while (nNumSids--){
            // If a SID isn't valid, then we bail
            if (!IsValidSid(ppSidArray[nNumSids])){
               lACLSize = 0;
               goto leave;
            }
            // Get the SID's length
            lACLSize += GetLengthSid(ppSidArray[nNumSids]);
            // Add the ACE structure size, minus the
            // size of the SidStart member
            lACLSize += sizeof(ACCESS_ALLOWED_ACE) -
               sizeof(((ACCESS_ALLOWED_ACE*)0)->SidStart);
         }
      }

      if (ppACEs != NULL){
         // Step through each ACE
         while (nNumACEs--){
            // Get the SIDs length
            lACLSize += ppACEs[nNumACEs]->aceHeader.AceSize;
         }
      }
      // Add in the ACL structure itself
      lACLSize += sizeof(ACL);

   leave:;
   }catch(...){
      // An exception means we fail the function
      lACLSize = 0;
   }
   return (lACLSize);
}
```

NOTE

The *CalculateACLSize* function uses the size of the ACCESS_ ALLOWED_ACE structure, which will work for all standard ACE types. This function will not calculate an ACL size properly if it includes object ACE types. To properly calculate the size, replace the reference to ACCESS_ALLOWED_ACE in this function with an object ACE structure such as ACCESS_ALLOWED_OBJECT_ACE. For a discussion on object ACEs, see the section

earlier in this chapter titled "Object ACEs."

When you use a function such as *CalculateACLSize*, finding the amount of memory necessary to create a new ACL is a simple task. You should use the value returned from the function to allocate memory using an allocator such as *new*, *HeapAlloc*, or *malloc*.

After you have memory for your new ACL, you should initialize the ACL using *InitializeAcl*:

```
BOOL InitializeAcl(
   PACL  pACL,
   DWORD dwAclLength,
   DWORD dwAclRevision);
```

InitializeAcl is a very simple function that sets the length of the buffer in the structure of the ACL and the ACL's revision. You should pass ACL_REVISION for the *dwAclRevision* parameter. An ACL can also have the revision of ACL_REVISION_DS, which indicates that the ACL contains object ACEs. However, you do not need to explicitly set this revision, because when you add object ACEs to an ACL, the system updates the revision of the ACL.

After calling *InitializeAcl*, your memory buffer contains an empty ACL containing no ACEs. If you wish to remove all ACEs from an existing ACL, you can also use *InitializeAcl* to do this.

Now you are ready to begin adding ACEs to your empty ACL. Assuming you have properly calculated the size of your finished ACL, adding ACEs should be as simple as making repeated calls to *AddAccessDeniedAceEx* for all your access-denied ACEs, followed by calls to *AddAccessAllowedAceEx* for all of your access-allowed ACEs.

AddAccessDeniedAceEx and *AddAccessAllowedAceEx* build ACEs of the appropriate type and add them to the end of a DACL (assuming that there is room in the DACL to accompany the ACE). *AddAccessDeniedAceEx* is defined as follows:

```
BOOL AddAccessDeniedAceEx(
   PACL  pDACL,
   DWORD dwACERevision,
   DWORD dwACEFlags,
   DWORD dwAccessMask,
   PSID  psidTrustee);
```

Here is *AddAccessAllowedAceEx*:

```
BOOL AddAccessAllowedAceEx(
   PACL  pDACL,
   DWORD dwACERevision,
   DWORD dwACEFlags,
   DWORD dwAccessMask,
   PSID  psidTrustee);
```

The declarations for these functions are identical, which is nice. The *pDACL* parameter indicates the DACL to which you are adding an ACE. The *dwACERevision* parameter should be set to ACL_REVISION. The *dwACEFlags* parameter indicates the value that will be set in the *AceFlags* member of the new ACE's ACE_HEADER. This is used to indicate the ACE's inheritance properties. See Table 10-11 for an explanation of the different ACE flags. You should not pass the flags indicating audit type for the *dwACEFlags* parameter.

The *dwAccessMask* parameter indicates the access mask of the new ACE, which describes which rights you are denying or allowing to the trustee. And finally, you should pass a pointer to a SID structure for the *psidTrustee* parameter to indicate the trustee for which access is being denied or allowed.

If there is insufficient room in the DACL, the *AddAccessDeniedAceEx* and *AddAccessAllowedAceEx*

functions return FALSE, and *GetLastError* returns ERROR_ALLOTTED_SPACE_EXCEEDED.

NOTE

For objects that do not support inheritance, you can use the simpler non-"Ex" versions of *AddAccessDeniedAce* and *AddAccessAllowedAce*. The only difference with the non-"Ex" functions is that they do not allow you to set the flags for the new ACE.

When you understand *AddAccessDeniedAceEx* and *AddAccessAllowedAceEx*, you can see that calling them repeatedly to create a new DACL is no big deal. After you create the new DACL, you can pass it to *SetSecurityInfo* or *SetNamedSecurityInfo* to apply it to an existing object. (Don't forget to free the memory for your new DACL after you have called one of these functions.)

You usually create a new DACL for an object that you are about to create. Creating an object typically requires you to assign the new DACL to a security descriptor, and then assign the security descriptor to a security attributes structure before calling the "create" function. The following code shows how to create a new DACL for a new event object. The code creates and initializes a security descriptor with a DACL, and then uses it to create a named event. (Note that this code uses the previous *CalculateACLSize* function to calculate the size of the new DACL.) The two challenges in this process are calculating the DACL's size and adding your ACEs in the proper order.

```
PSID psidEveryone ;

// Create a SID for the built-in "Everyone" group
SID_IDENTIFIER_AUTHORITY sidAuth = SECURITY_WORLD_SID_AUTHORITY;
if (!AllocateAndInitializeSid( &sidAuth, 1, SECURITY_WORLD_RID,
   0, 0, 0, 0, 0, 0, 0, &psidEveryone )){
   // Error
}

// We are creating two ACEs, both using the "Everyone" group
PSID psidArray[2];
psidArray[0] = psidEveryone;
psidArray[1] = psidEveryone;

// Get the size of the new ACL
ULONG lACLSize = CalculateACLSize( NULL, psidArray, 2, NULL, 0);
if (lACLSize == 0){
   // Error
}

// Allocate memory for the ACL
PACL pDACL = (PACL)HeapAlloc(GetProcessHeap(), 0, lACLSize);
if (pDACL == NULL){
   // Error
}

// Initialize the ACL
if (!InitializeAcl(pDACL, lACLSize, ACL_REVISION)){
   // Error
}

// Make sure to add our denied ACE first
if (!AddAccessDeniedAce(pDACL, ACL_REVISION,
   WRITE_OWNER|WRITE_DAC, psidArray[0])){
   // Error
}

// Then add our allowed ACE
if (!AddAccessAllowedAce(pDACL, ACL_REVISION,
   STANDARD_RIGHTS_ALL|SPECIFIC_RIGHTS_ALL, psidArray[1])){
```

```
   GetLastError();//Error "winerror.h"
}

// Allocate space for a security descriptor
PSECURITY_DESCRIPTOR pSD = HeapAlloc(GetProcessHeap(), 0,
   SECURITY_DESCRIPTOR_MIN_LENGTH);
if (pSD == NULL){
   // Error
}
// We now have an empty security descriptor
if (!InitializeSecurityDescriptor(pSD, SECURITY_DESCRIPTOR_REVISION)){
   // Error
}
// To which we assign our DACL
if (!SetSecurityDescriptorDacl(pSD, TRUE, pDACL, FALSE)){
   // Error
}

// Then we point to our SD from a SECURITY_ATTRIBUTES structure
SECURITY_ATTRIBUTES sa = {0};
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = pSD;

// Which we pass to CreateEvent
HANDLE hEvent = CreateEvent(&sa, TRUE, FALSE, TEXT("SecureEvent"));

// Clean up
HeapFree(GetProcessHeap(), 0, pSD);
HeapFree(GetProcessHeap(), 0, pDACL);
FreeSid(psidEveryone);
```

NOTE

An alternative to adding the ACEs in the proper order would be to write a function to order the ACL for you so that you could add ACEs without concern for order. You would then order the entire ACL before applying the ACL to an object. I implement such a function later in this chapter.

This sample not only shows how to build a DACL and security descriptor for use in object creation, but it also illustrates an important technique for sharing objects with services.

The default security for a service in the LocalSystem user context will cause objects such as named events and named pipes to be secured against use by processes running in the security context of a logged-on user. But sometimes this is not the desired behavior. At times you might be tempted to create an object with a NULL DACL, which allows everyone all access to the object, but doing so creates a security hole. All access includes the ability to adjust the object's security, and would allow anyone to change the security of the object. Having the security changed would almost definitely cause the object to stop working as expected.

A better solution to controlling access is to deny everyone access to the security of the object (which allows only the owner of the object to modify the object's security). After adding the denied ACE, you can comfortably allow any trustees access to the object, knowing that the deny ACE will take precedence. In this case, I allow everyone all access.

Modifying a DACL

Creating secure objects involves as much access control as many services ever need to implement. It is surprising how many services you can implement without ever having to modify the security of an existing object. Of course, some programs have to deal with existing objects that already have security applied to them. In such a case, your software must be able to read and modify a DACL for an object. Here are the steps

you might take to do this:

1. Retrieve the DACL of the object that you wish to modify.
2. If you are removing ACEs, search the DACL for these ACEs and delete them.
3. Gather the SIDs of the trustees for which you will be adding ACEs.
4. Create the ACEs that you will be adding.
5. Search the DACL for ACEs that are the same as ACEs you will be adding. If any exist, remove them from the group of ACEs you will be adding.
6. Calculate the new DACL size.
7. Allocate memory for and initialize the new DACL.
8. Copy the old DACL to the new DACL.
9. Insert new ACEs in the proper position in the new DACL.
10. Assign the DACL back to the object.

As you can see, the process for reading and modifying an object's DACL is not so simple, but I should tell you that I've offered the worst-case scenario. Often you are not removing ACEs at all, so step 2 can be ignored. And commonly you add only a single ACE; if that ACE is already in the DACL, you can abort the whole procedure, simplifying step 5.

NOTE

It is important to search ACLs for ACEs that already exist, even though duplicate ACEs do not affect the *security* of an object. The reason is that ACEs take memory in the system, and some system objects (such as a window station) can hold surprisingly few ACEs. Any software that adds an ACE every time it is run, without checking the necessity of the task, will eventually exhaust system resources.

Now I will tell you about the tools required to tackle the task of reading and modifying a DACL. Let's start with a simple function. (You already know how to read information in a DACL, and those skills will come into play here.)

```
BOOL DeleteAce(
   PACL  pACL,
   DWORD dwACEIndex);
```

The *DeleteAce* function removes an ACE from a DACL. You must pass it the ACL and the index of the ACE that you wish to remove.

NOTE

If the only modification that you are making to an object is to remove ACEs, you do not have to allocate memory for a new DACL. You can simply remove the ACEs in the DACL returned by *GetSecurityInfo*, and then set the modified DACL back to the object.

The next steps in the process require that you have an ACE handy because you will use the *AddAce* function (discussed shortly), which requires an ACE that is already built. Also, because you might search the ACL for

the existence of the ACE, it is convenient to have an ACE structure with which to compare ACEs in the DACL.

Unfortunately the system does not provide a function that helps you build ACEs in a simple manner. The task of building ACEs is complicated because each ACE includes a SID structure, and SID structures are variable in length. It is best to use your SID's length and the length of the ACE structure to allocate a buffer for the ACE, copy the SID into the ACE, and then set the ACE fields. The following function shows how to do this:

```
PACE_UNION AllocateACE( ULONG bACEType, ULONG bACEFlags,
   ULONG lAccessMask, PSID pSID ){
   PACE_UNION pReturnACE = NULL;
   PBYTE pbBuffer = NULL;
   try{
      // Get the offset of the SID in the ACE
      ULONG lSIDOffset = (ULONG)(&((ACCESS_ALLOWED_ACE*)0)->SidStart);
      // Get the size of the ACE without the SID
      ULONG lACEStructSize = sizeof(ACCESS_ALLOWED_ACE) -
         sizeof(((ACCESS_ALLOWED_ACE*)0)->SidStart);
      // Get the length of the SID
      ULONG lSIDSize = GetLengthSid(pSID);

      // Allocate a buffer for the ACE
      pbBuffer = (PBYTE)LocalAlloc(LPTR, lACEStructSize + lSIDSize);
      if (pbBuffer == NULL)
         goto leave;

      // Copy the SID into the ACE
      if(!CopySid(lSIDSize, (PSID)(pbBuffer+lSIDOffset), pSID)){
         goto leave;
      }
      pReturnACE = (PACE_UNION) pbBuffer;
      pReturnACE->aceHeader.AceSize = (USHORT)(lACEStructSize + lSIDSize);
      pReturnACE->aceHeader.AceType = (BYTE)bACEType;
      pReturnACE->aceHeader.AceFlags = (BYTE)bACEFlags;
      pReturnACE->aceAllowed.Mask = lAccessMask;
   leave:;
   }catch(...){}
   // Free the buffer in an error case
   if (pbBuffer != (PBYTE)pReturnACE){
      LocalFree(pbBuffer);
   }
   return (pReturnACE);
}
```

When using *AllocateAce*, you simply pass the returned ACE to *LocalFree* when finished with it.

NOTE

AllocateAce returns a PACE_UNION pointer, which is a type I defined earlier in this chapter. I chose to use this technique to represent ACEs in a generic way. It is also very common to simply choose one of the specific ACE types, such as ACCESS_ALLOWED_ACE, to use generically for your ACE manipulation. Either approach works; however, for a number of the sample functions in this chapter, I use the PACE_UNION.

Now that you have an ACE built, you must search your object's existing DACL for ACEs that match. You can do this by using the ACL reading techniques discussed earlier in this chapter. I wrote code to perform this task using two functions: one to compare ACEs and one to search an ACL for a matching ACE.

```
BOOL IsEqualACE( PACE_UNION pACE1, PACE_UNION pACE2 ){
   BOOL fReturn = FALSE;
```

```
    try{{
       if(pACE1->aceHeader.AceType != pACE2->aceHeader.AceType)
          goto leave;

       // Get the offset of the SID in the ACE
       ULONG lSIDOffset = (ULONG)(&((ACCESS_ALLOWED_ACE*)0)->SidStart);
       // Get the size of the ACE without the SID
       ULONG lACEStructSize = sizeof(ACCESS_ALLOWED_ACE) -
          sizeof(((ACCESS_ALLOWED_ACE*)0)->SidStart);

       PBYTE pbACE1 = (PBYTE)pACE1;
       PBYTE pbACE2 = (PBYTE)pACE2;
       fReturn = TRUE;
       while(lACEStructSize--)
          fReturn = (fReturn && ((pbACE1[lACEStructSize] ==
             pbACE2[lACEStructSize])));
       // Check SIDs
       fReturn = fReturn && EqualSid((PSID)(pbACE1+lSIDOffset),
          (PSID)(pbACE2+lSIDOffset));
    }leave:;
    }catch(...){
    }

    return (fReturn);
 }
```

Here is the second function:

```
 int FindACEInACL( PACL pACL, PACE_UNION pACE ){
    int nACEIndex = -1;

    try{{
       ACL_SIZE_INFORMATION aclSize;
       if (!GetAclInformation(pACL, &aclSize, sizeof(aclSize),
          AclSizeInformation)){
          goto leave;
       }

       while (aclSize.AceCount--){
          PACE_UNION pACETemp;
          if(!GetAce(pACL, aclSize.AceCount, (PVOID *)&pACETemp))
             goto leave;

          if(IsEqualACE(pACETemp, pACE)){
             nACEIndex = (int)aclSize.AceCount;
             break;
          }
       }
    }leave:;
    }catch(...){
    }
    return (nACEIndex);
 }
```

The *FindAceInACL* function returns the index of a matching ACE in the DACL, or -1 if no match is found. You should use this function or one like it to find out whether the ACEs you intend to add to your object's DACL already exist in the DACL. (The *FindACEInACL* function can also be useful for finding ACEs that you wish to delete using *DeleteAce*).

Now that you know which ACEs you must add your DACL and you have built the ACEs, it is time to calculate the size of your new DACL. You can use the *CalculateACLSize* function, shown earlier in this chapter. It allows you to calculate an ACL size using an existing ACL, an array of SIDs, and an array of ACEs. You can pass NULL for any of these parameters. In our example, you are likely to pass an existing ACL and an array of ACEs to calculate the new size of the ACL after your ACEs have been added.

After you allocate memory for your new ACL, you initialize the new ACL using *InitializeAcl* (discussed earlier in this chapter). You have to copy the ACEs from your old ACL to your new ACL. The following function shows how to do this:

```
BOOL CopyACL( PACL pACLDestination, PACL pACLSource ){
   BOOL fReturn = FALSE;
   try{{
      // Get the number of ACEs in the source ACL
      ACL_SIZE_INFORMATION aclSize;
      if (!GetAclInformation(pACLSource, &aclSize,
         sizeof(aclSize), AclSizeInformation)){
         goto leave;
      }

      // Use GetAce and AddAce to copy the ACEs
      for(ULONG lIndex=0;lIndex < aclSize.AceCount;lIndex++){
         ACE_HEADER* pACE;
         if(!GetAce(pACLSource, lIndex, (PVOID*)&pACE))
            goto leave;
         if(!AddAce(pACLDestination, ACL_REVISION, MAXDWORD,
            (PVOID*)pACE, pACE->AceSize))
            goto leave;
      }
      fReturn = TRUE;
   }leave:;
   }catch(...){
   }
   return (fReturn);
}
```

This *CopyACL* function is fairly simple. It simply iterates through the ACEs in the existing DACL and copies each one to the new DACL. C*opyACL* uses the *AddAce* system function, which is defined as follows:

```
BOOL AddAce(
   PACL  pACL,
   DWORD dwACERevision,
   DWORD dwStartingACEIndex,
   PVOID pACEList,
   DWORD dwACEListLength);
```

Notice that *AddAce* is different from *AddAccessAllowedAce* and *AddAccessDeniedAce*. First you have to supply the ACE; the system does not compile the ACE into the DACL for you. This means that you define the type of ACE as well, so *AddAce* can add any type of ACE to the DACL. Second, *AddAce* allows you to decide where in the DACL you wish to insert the new ACE by using an index starting with zero. Third, *AddAce* allows you to add more than one ACE by passing a sequential list of ACEs as the *pACEList* parameter and the size of the list as the *dwACEListLength* parameter (not to be confused with the number of ACEs in the list).

NOTE

The *CopyACL* sample function could have been implemented more efficiently by allowing *AddAce* to copy all of the source ACEs over in a single function call. However, doing this assumes information about the ACL structure, which you should treat as opaque.

After you copy your old DACL into your new, more spacious DACL, you can begin adding ACEs to it using *AddAce*. Take care to insert your ACEs in the proper index for your new DACL so that proper ACE ordering is preserved. (See Table 10-6 regarding ACE order.) You can use the following function to determine the appropriate index for a new ACE in an ACL:

```
ULONG GetACEInsertionIndex( PACL pDACL, PACE_UNION pACENew){
   ULONG lIndex = (ULONG) -1;
```

```
    try{{
       // ACE types by ACL order
       ULONG lFilterType[] = { ACCESS_DENIED_ACE_TYPE,
                               ACCESS_DENIED_OBJECT_ACE_TYPE,
                               ACCESS_ALLOWED_ACE_TYPE,
                               ACCESS_ALLOWED_OBJECT_ACE_TYPE};

       // Determine which group the new ACE should belong to
       ULONG lNewAceGroup;
       for(lNewAceGroup = 0; lNewAceGroup<4 ; lNewAceGroup++){
          if(pACENew->aceHeader.AceType == lFilterType[lNewAceGroup])
             break;
       }
       // If group == 4, the ACE type is no good
       if(lNewAceGroup==4)
          goto leave;
       // If new ACE is an inherited ACE, then it goes after other ACEs
       if((pACENew->aceHeader.AceFlags & INHERITED_ACE) != 0)
          lNewAceGroup+=4;

       // Get ACE count
       ACL_SIZE_INFORMATION aclSize;
       if (!GetAclInformation(pDACL, &aclSize,
          sizeof(aclSize), AclSizeInformation)){
          goto leave;
       }

       // Iterate through ACEs
       lIndex = 0;
       for(lIndex = 0;lIndex < aclSize.AceCount;lIndex++){
          ACE_HEADER* pACE;
          if(!GetAce(pDACL, lIndex, (PVOID*)&pACE))
             goto leave;

          // Get the group of the ACL ACE
          ULONG lAceGroup;
          for(lAceGroup = 0; lAceGroup<4 ; lAceGroup++){
             if(pACE->AceType == lFilterType[lAceGroup])
                break;
          }
          // Test for bad ACE
          if(lAceGroup==4){
             lIndex = (ULONG) -1;
             goto leave;
          }
          // Inherited adjustment
          if((pACE->AceFlags & INHERITED_ACE) != 0)
             lAceGroup+=4;

          // If this is the same group, then insertion point found
          if(lAceGroup>=lNewAceGroup)
             break;
       }

    }leave:;
    }catch(...){
    }
    return (lIndex);
}
```

The *GetACEInsertionIndex* function finds the index in which to insert your new ACE, and in the process considers object ACEs and ACE inheritance. After you know the proper index, you can call *AddAce* to add the new ACE into the ACL.

Do this for each new ACE, and then use the appropriate function to set the new DACL to the securable object. Don't forget to clean up after yourself and free any memory that you have allocated.

Earlier in this chapter, I promised to include a sample function that orders the ACEs in a DACL. This function might be useful if you don't want to concern yourself with ACE ordering at all until after you completely finish adding ACEs to your DACL. The following *OrderDACL* function is fairly simple, and it builds on the *GetACEInsertionIndex* sample function:

```
BOOL OrderDACL( PACL pDACL ){
   BOOL fReturn = FALSE;

   try{{
      // Get ACL size and ACE count
      ACL_SIZE_INFORMATION aclSize;
      if (!GetAclInformation(pDACL, &aclSize,
         sizeof(aclSize), AclSizeInformation)){
         goto leave;
      }
      // Get memory for temporary ACL
      PACL pTempDACL = (PACL) _alloca(aclSize.AclBytesInUse);
      if (pTempDACL==NULL)
         goto leave;
      // Initialize temporary ACL
      if (!InitializeAcl(pTempDACL, aclSize.AclBytesInUse,
         ACL_REVISION))
         goto leave;

      // Iterate through ACEs
      for (ULONG lAceIndex = 0;
         lAceIndex < aclSize.AceCount ; lAceIndex++){
         // Get ACE
         PACE_UNION pACE;
         if (!GetAce(pDACL, lAceIndex, (PVOID*)&pACE))
            goto leave;
         // Find location, and add ACE to temp DACL
         ULONG lWhere = GetACEInsertionIndex(pTempDACL, pACE);
         if (!AddAce(pTempDACL, ACL_REVISION,
            lWhere, pACE, pACE->AceSize))
            goto leave;
      }
      // Copy temp DACL to original
      CopyMemory(pDACL, pTempDACL, aclSize.AclBytesInUse);
      fReturn = TRUE;
   }
   leave:;
   }catch(...){
   }
   return (fReturn);
}
```

Let's take a look at a real-world example that implements the techniques we have been discussing.

Modifying a DACL Example

Our example shows how to modify an existing DACL while providing a useful set of functions for the service developer. I used the window station and the desktop-securable objects in Windows.

A *window station* is a secure object in Windows that contains a clipboard, a set of global atoms, and a collection of desktop objects. A window station can be interactive, meaning its "desktops" can be seen by a user. An interactive window station also maintains keyboard and mouse information. A process can have a single window station associated with it.

A *desktop* is a secure object contained within a window station. A desktop maintains a logical display surface and contains menus, windows, and other objects visible on screen.

Services that do not interact with the user are not associated with the interactive desktop. When a user logs on to a system, the DACL for the interactive window station (named WinSta0) and its default desktop (named Default) are reset, and the user is given access to the objects. Ultimately, only the logged-on user and the system are granted access to the objects.

Problem description Sometimes a service has to create another process under an arbitrary trustee account (using *CreateProcessAsUser* discussed in the next chapter) to create a process under a user context that is not currently interacting with the system. If this process needs to interact with a user, and the user account does not have access to the interactive window station and its default desktop, the system fails the call to *CreateProcessAsUser*.

Herein lies the problem. Before creating the process, you have to check the DACLs of these objects for sufficient access rights for the user. If these rights aren't found, they must be added. It is important to check for the rights first, because adding an ACE blindly to the user objects can eventually exhaust resources in the system. (Typically you can add only about 80 ACEs to a window station.)

Solution Now let's tackle the solution. I used the tools and concepts (and some of the sample functions) discussed throughout this section to implement two functions: one that allows a trustee access to a window station and one that allows a trustee access to a desktop. These functions are pretty straightforward. Although the process of modifying an object's DACL can seem somewhat daunting, in the real world the process tends to be less complex than we imagine. The following code allows a trustee access to a window station:

```
BOOL AllowAccessToWinSta( PSID psidTrustee, HWINSTA hWinSta ){
   BOOL fReturn = FALSE;
   PSECURITY_DESCRIPTOR psdWinSta = NULL;
   PACE_UNION pACENew = NULL;

   try{{
      // Get the DACL for the window station
      PACL pDACLWinSta;
      if(GetSecurityInfo(hWinSta, SE_WINDOW_OBJECT,
         DACL_SECURITY_INFORMATION, NULL, NULL, &pDACLWinSta,
         NULL, &psdWinSta) != ERROR_SUCCESS)
         goto leave;

      // Allocate our new ACE
      // This is the access awarded to a user who logged on interactively
      PACE_UNION pACENew = AllocateACE(ACCESS_ALLOWED_ACE_TYPE, 0,
         DELETE|WRITE_OWNER|WRITE_DAC|READ_CONTROL|
         WINSTA_ENUMDESKTOPS|WINSTA_READATTRIBUTES|
         WINSTA_ACCESSCLIPBOARD|WINSTA_CREATEDESKTOP|
         WINSTA_WRITEATTRIBUTES|WINSTA_ACCESSGLOBALATOMS|
         WINSTA_EXITWINDOWS|WINSTA_ENUMERATE|WINSTA_READSCREEN,
         psidTrustee);

      // Is the ACE already in the DACL?
      if (FindACEInACL(pDACLWinSta, pACENew) == -1){
         // If not, calculate new DACL size
         ULONG lNewACL = CalculateACLSize( pDACLWinSta, NULL, 0,
            &pACENew, 1 );

         // Allocate memory for the new DACL
         PACL pNewDACL = (PACL)_alloca(lNewACL);
         if (pNewDACL == NULL)
            goto leave;

         // Initialize the ACL
         if (!InitializeAcl(pNewDACL, lNewACL, ACL_REVISION))
            goto leave;

         // Copy the ACL
```

```
         if (!CopyACL(pNewDACL, pDACLWinSta))
            goto leave;

         // Get location for new ACE
         ULONG lIndex = GetACEInsertionIndex(pNewDACL, pACENew);

         // Add the new ACE
         if (!AddAce(pNewDACL, ACL_REVISION, lIndex,
            pACENew, pACENew->aceHeader.AceSize))
            goto leave;

         // Set the DACL back to the window station
         if (SetSecurityInfo(hWinSta, SE_WINDOW_OBJECT,
            DACL_SECURITY_INFORMATION, NULL, NULL,
            pNewDACL, NULL)!=ERROR_SUCCESS)
            goto leave;
      }
      fReturn = TRUE;
   }leave:;
   }catch(...){
   }
   // Clean up
   if(pACENew != NULL)
      LocalFree(pACENew);
   if(psdWinSta != NULL)
      LocalFree(psdWinSta);
   return (fReturn);
}
```

The next sample function allows a trustee access to a desktop:

```
BOOL AllowAccessToDesktop( PSID psidTrustee, HDESK hDesk ){
   BOOL fReturn = FALSE;
   PSECURITY_DESCRIPTOR psdDesk = NULL;
   PACE_UNION pACENew = NULL;

   try{{
      // Get the DACL for the desktop
      PACL pDACLDesk;
      if(GetSecurityInfo(hDesk, SE_WINDOW_OBJECT,
         DACL_SECURITY_INFORMATION, NULL, NULL, &pDACLDesk,
         NULL, &psdDesk) != ERROR_SUCCESS)
         goto leave;

      // Allocate our new ACE
      // This is the access awarded to a user who logged on interactively
      PACE_UNION pACENew = AllocateACE(ACCESS_ALLOWED_ACE_TYPE, 0,
         DELETE|WRITE_OWNER|WRITE_DAC|READ_CONTROL|
         DESKTOP_READOBJECTS|DESKTOP_CREATEWINDOW|
         DESKTOP_CREATEMENU|DESKTOP_HOOKCONTROL|
         DESKTOP_JOURNALRECORD|DESKTOP_JOURNALPLAYBACK|
         DESKTOP_ENUMERATE|DESKTOP_WRITEOBJECTS|DESKTOP_SWITCHDESKTOP,
         psidTrustee);

      // Is the ACE already in the DACL?
      if (FindACEInACL(pDACLDesk, pACENew) == -1){
         // If not, calculate new DACL size
         ULONG lNewACL = CalculateACLSize( pDACLDesk, NULL, 0,
            &pACENew, 1 );

         // Allocate memory for the new DACL
         PACL pNewDACL = (PACL)_alloca(lNewACL);
         if (pNewDACL == NULL)
            goto leave;

         // Initialize the ACL
```

```
         if (!InitializeAcl(pNewDACL, lNewACL, ACL_REVISION))
            goto leave;

         // Copy the ACL
         if (!CopyACL(pNewDACL, pDACLDesk))
            goto leave;

         // Get location for new ACE
         ULONG lIndex = GetACEInsertionIndex(pNewDACL, pACENew);

         // Add the new ACE
         if (!AddAce(pNewDACL, ACL_REVISION, lIndex,
            pACENew, pACENew->aceHeader.AceSize))
            goto leave;

         // Set the DACL back to the window station
         if (SetSecurityInfo(hDesk, SE_WINDOW_OBJECT,
            DACL_SECURITY_INFORMATION, NULL, NULL,
            pNewDACL, NULL)!=ERROR_SUCCESS)
            goto leave;
      }
      fReturn = TRUE;
   }leave:;
   }catch(...){
   }
   // Clean up
   if(pACENew != NULL)
      LocalFree(pACENew);
   if(psdDesk != NULL)
      LocalFree(psdDesk);
   return (fReturn);
}
```

The following code fragment shows an example of how these functions are used. The code creates a SID for the built-in Everyone group and passes it to the *AllowAccessToWinSta* and *AllowAccessToDesktop* functions:

```
PSID psidEveryone;

// Create a SID for the built-in "Everyone" group
SID_IDENTIFIER_AUTHORITY sidAuth = SECURITY_WORLD_SID_AUTHORITY;
if (!AllocateAndInitializeSid( &sidAuth, 1, SECURITY_WORLD_RID,
   0, 0, 0, 0, 0, 0, 0, &psidEveryone )){
   // Error
}

HWINSTA hWinSta = GetProcessWindowStation();
if (hWinSta == NULL){
   // Error
}
AllowAccessToWinSta(psidEveryone, hWinSta);

HDESK hDesk = GetThreadDesktop(GetCurrentThreadId());
if (hDesk == NULL){
   // Error
}
AllowAccessToDesktop(psidEveryone, hDesk);
```

NOTE

Here's a tip. The *AllowAccessToWinSta* and *AllowAccessToDesktop* functions use a function called *_alloca* defined by the C run-time library in the header file Malloc.h. The *_alloca* function allocates a block of memory on the thread's stack. The beauty of this function is that it is very fast, requires no internal thread synchronization, and returns memory that does not need to be freed by your application. The system frees the memory when you exit the function in which it was called.

> For a security programmer who must make repeated small allocations, a function like this one can be a lifesaver. It will speed up your code and help you avoid memory leaks.

I strongly suggest that you take the time to walk through the *AllowAccessToWinSta* and *AllowAccessToDesktop* functions until you understand how they work and feel confident using the techniques that they use. These sample functions perform tasks that are about as complex as you will see in access control programming, and if you are comfortable with them, you probably won't have trouble implementing access control that meets your needs.

Options for Implementing Access Control

Microsoft and other parties have attempted to ease the burden on the access control programmer by creating higher-level functions that wrap the low-level functions we have been covering. Third-party vendors have also created solutions to ease this task of access control. Awareness of the low-level functionality of access control in Windows, as well as of some of the pitfalls of these higher-level functions, should enable you to make decisions that will address the needs of your code.

The implementers of the higher-level packages have been faced with these challenges:

- Simplifying the extremely flexible access control system, without restricting flexibility or the features you are likely to need in your project
- Creating robust and usable code

The first challenge&"5" width="95%">

```
DWORD GetEffectiveRightsFromAcl(
   PACL         pACL,
   PTRUSTEE     pTrustee,
   PACCESS_MASK pAccessRights);
```

The *GetEffectiveRightsFromAcl* function is intended to search an ACL and return an access mask that indicates access allowed to a trustee by a DACL. Sounds very convenient! Such a function could potentially remove our need to search a DACL for ACEs before adding our own ACEs. *GetEffectiveRightsFromAcl*, however, attempts to do too much, and thus ends up doing almost nothing of use.

GetEffectiveRightsFromAcl figures out the access allowed based on a composite of the matching access-allowed ACEs, and then subtracts a composite of the matching access-denied ACEs. What this means is that *GetEffectiveRightsFromAcl* can return a set of access rights indicating that the ACL does not award the access I want for my trustee, and I'm left still not knowing how to fix the problem. Is the access not granted because of the absence of access-allowed ACEs, or is it not granted because of the presence of access-denied ACEs? I would hate to add an access-allowed ACE, just to find out that I still don't have access because of an overriding access-denied ACE.

GetEffectiveRightsFromAcl not only searches for ACEs that match the trustee you supply, but also searches for any ACEs for group accounts of which your trustee is a member. But the function excludes built-in groups such as Everyone and Authenticated Users. And it fails if it finds ACEs for groups whose member trustees your code does not have rights to enumerate. Finally, there is no way to limit the search to finding only the access explicitly assigned to your trustee.

GetEffectiveRightsFromAcl was intended to enable your code to find out whether an access check on an object would succeed or fail for a trustee. However, access checks require a token, not a trustee SID. Tokens contain privileges; tokens can also be adjusted or restricted. (See Chapter 11.) *GetEffectiveRightsFromAcl* does not take privileges into account when checking access. Access checks can succeed based on object ownership, but

GetEffectiveRightsFromAcl has no knowledge of object ownership. In these ways, *GetEffectiveRightsFromAcl* has limited use.

The more useful functions *SetEntriesInAcl* and *GetExplicitEntriesFromAcl* are intended to relieve you of the responsibility of allocating memory for ACEs and ACLs while still allowing you to deal with the ACL directly. The goal of these functions is great, but the functions have a sordid history of bugs and performance issues. Some of these problems have been cleared up, but if you choose to use these functions in your projects, it is important that you thoroughly test the code that uses them.

The *BuildExplicitAccessWithName* function, which should be used with *SetEntriesInAcl*, does not return a value. Instead it potentially defers an error scenario (that you would have caught when using low-level functions to call *LookupAccountName*) to the *SetEntriesInAcl* function. Because *SetEntriesInAcl* has no way of reporting back which entry failed, you are left with a failure case from which it is difficult to recover.

You also have other options. The developers of the Active Template Library designed a C++ class called *CSecurityDescriptor* defined in the header file AtlCom.h. The bonus of *CSecurityDescriptor* is that the entire source code for the class is provided. Although this class provides a wealth of functionality, it comes with its own pitfalls. For example, the functions used to add ACEs do not follow the ACE ordering guidelines set forth for Windows 2000, although they do follow the guidelines for Windows NT 4.0. Additionally, the *AttachObject* function for retrieving the security from a kernel object uses *GetKernelObjectSecurity* instead of *GetSecurityInfo*, which is the suggested function for use with Windows 2000. (The class also suffers from an unlikely but potential race condition.)

As I mentioned, the great thing about the *CSecurityDescriptor* class is that you have the source code. If it works for your needs, great! If it doesn't, you have the option of modifying the class. And if you find any bugs, you have the option of fixing them yourself using the knowledge you now have about low-level access control in Windows.

Securing Private Objects

I have mentioned several times that you can secure private objects created by your software by using Windows access control. This feature is a powerful one indeed, and it is particularly likely to be of use to service developers. The DACL building techniques you have learned so far apply to securing private objects as well as securing system objects, so there is little to learn about implementing access control on your own objects.

The tasks involved in securing your objects are shared between your software and the system. Your software must perform the following tasks to implement private security for objects:

- Your software must define specific rights for your objects (using the low 16 bits of the access mask).
- Your software must decide which of the standard rights are relevant for your objects.
- Your software must decide to which standard and specific rights the generic rights map.
- Your software must associate security descriptors (created by the system) with objects.
- Your software must store security descriptors with objects in persistent storage if appropriate for your objects.
- Your software must perform access checks before performing securable actions on secured objects.

The system provides the following features:

- Functions that create and destroy security descriptors for your objects
- Functions that get and set specific parts of these security descriptors
- A function that performs an access check in a manner consistent with other secured objects in Windows

Before beginning our discussion of specific functions, I need to clarify two points:

- Private object security should be used for service software that is serving client software running in a different security context.
- Private object security requires your software to use tokens to indicate client security context. Remember that tokens contain a trustee's SID as well as its group SIDs, privileges, and default DACL. (I will discuss the token in greater detail in Chapter 11. You might find it helpful to refer to that chapter while continuing your reading in this chapter.)

When creating an object that is to be secured privately, you should create the object's security descriptor. Call *CreatePrivateObjectSecurity* to do this:

```
BOOL CreatePrivateObjectSecurity(
   PSECURITY_DESCRIPTOR psdParentDescriptor,
   PSECURITY_DESCRIPTOR psdCreatorDescriptor,
   PSECURITY_DESCRIPTOR *ppsdNewDescriptor,
   BOOL                 fIsDirectoryObject,
   HANDLE               hToken,
   PGENERIC_MAPPING     gmGenericMapping);
```

The *CreatePrivateObjectSecurity* function takes an optional parent security descriptor, as well as an optional creator descriptor. Both parameters can be NULL. If neither are provided, *CreatePrivateObjectSecurity* creates a security descriptor by using the default DACL found in the provided token. The generic rights found in the default DACL are mapped to the object's specific and standard rights using the information found in the passed GENERIC_MAPPING structure. The *CreatePrivateObjectSecurity* function returns the new security descriptor by assigning its address to the PSECURITY_DESCRIPTOR variable, whose address you passed as the *ppsdNewDescriptor* parameter. This is a new security descriptor that you should now associate with your privately secured object. When you need to free the memory allocated by this function, you should pass the address of a variable that points to the security descriptor you want to free to *DestroyPrivateObjectSecurity*.

```
BOOL DestroyPrivateObjectSecurity(
   PSECURITY_DESCRIPTOR *ppsdObjectDescriptor);
```

The GENERIC_MAPPING structure that you initialize and pass to *CreatePrivateObjectSecurity* is defined as follows:

```
typedef struct _GENERIC_MAPPING {
   ACCESS_MASK GenericRead;
   ACCESS_MASK GenericWrite;
   ACCESS_MASK GenericExecute;
   ACCESS_MASK GenericAll;
} GENERIC_MAPPING;
```

You can fill in this simple structure by setting each member to the appropriate combination of standard and specific rights for each generic right. The system uses this information when building the DACL from the token's default DACL.

NOTE

You should never directly modify the security descriptor returned from *CreatePrivateObjectSecurity*. Although the system provides functions for you to request specific security information from this descriptor, your software should treat the returned pointer as a black box, as if the security descriptor were stored in system memory.

When the client attempts to act on a secure object in your software, you must first subject the client to an access check. To do this, you pass the rights required to perform the securable task, as well as the security descriptor and the client's token, to the *AccessCheck* function.

```
BOOL AccessCheck(
   PSECURITY_DESCRIPTOR pSecurityDescriptor,
   HANDLE               hClientToken,
   DWORD                dwDesiredAccess,
   PGENERIC_MAPPING     gmGenericMapping,
   PPRIVILEGE_SET       pPrivilegeSet,
   PDWORD               pdwPrivilegeSetLength,
   PDWORD               pdwGrantedAccess,
   PBOOL                pfAccessStatus);
```

You must also include an address of a GENERIC_MAPPING structure for the object, as well as the address of an array of PRIVILEGE_SET structures. The system uses the PRIVILEGE_SET structure to report the privileges that were used to grant the access. The system uses privileges to grant access in several cases, and you should provide a sufficiently large buffer to receive several returned privileges. The PRIVILEGE_SET structure is defined as follows:

```
typedef struct _PRIVILEGE_SET {
   DWORD               PrivilegeCount;
   DWORD               Control;
   LUID_AND_ATTRIBUTES Privilege[ANYSIZE_ARRAY];
} PRIVILEGE_SET;
```

The ANYSIZE_ARRAY value is defined as 1, and you should create a buffer large enough to receive the relevant privileges. You can also call *AccessCheck* once to receive the required size of the privilege buffer, and then allocate a buffer of sufficient length.

NOTE

Privileges can be used to grant access when your client requests WRITE_OWNER, READ_CONTROL, WRITE_DAC, or ACCESS_SYSTEM_SECURITY access for an object. If these access rights are not explicitly assigned to your client's trustee accounts, the system checks the client's token for privileges that override the security of the object. An example of such a privilege is SE_TAKE_OWNERSHIP_NAME, which allows the client to set the ownership of any object in the system.

The access mask that your client was granted is returned via the *pdwGrantedAccess* parameter, and a Boolean value is returned via the *pfAccessStatus* parameter indicating whether the *AccessCheck* function succeeded.

The *AccessCheck* function is the centerpiece when implementing private object security. If your software is secure, and it religiously calls *AccessCheck* before performing a secure task on an object, Windows takes care of remaining details of deciding when your software's clients are allowed access.

NOTE

You can also create audit events with private security objects by using the *AccessCheckAndAuditAlarm* function. Auditing will be covered later in this chapter.

At this point, you know how to create and destroy private security, and you also know how and when to call

AccessCheck. The only remaining piece of the private object puzzle is how to modify the components of a security descriptor.

To modify the security of a private object, you should first get the component of the security descriptor that you wish to adjust. Typically this is the DACL, but it could be the object's owner or SACL. Use *GetPrivateObjectSecurity* to do this.

```
BOOL GetPrivateObjectSecurity(
   PSECURITY_DESCRIPTOR psdObjectDescriptor,
   SECURITY_INFORMATION secInfo,
   PSECURITY_DESCRIPTOR psdResultantDescriptor,
   DWORD                dwDescriptorLength,
   PDWORD               pdwReturnLength);
```

The *GetPrivateObjectSecurity* function takes a pointer to the private security object's security descriptor, and returns the requested security descriptor. You use the familiar *secInfo* parameter to indicate which portions of the object's security descriptor you want to retrieve. You must supply a buffer large enough to contain a security descriptor that will hold the requested information. You can call *GetPrivateObjectSecurity* once to find the size of the required buffer, and again to retrieve the information.

After you have the security descriptor for the object, you must modify it, and then reset the modified security descriptor to the private object using *SetPrivateObjectSecurity*.

```
BOOL SetPrivateObjectSecurity(
   SECURITY_INFORMATION secInfo,
   PSECURITY_DESCRIPTOR psdModificationDescriptor,
   PSECURITY_DESCRIPTOR *ppsdObjectsSecurityDescriptor,
   PGENERIC_MAPPING     gmGenericMapping,
   HANDLE               hClientToken);
```

The *secInfo* parameter indicates which information in the "modification security descriptor" is to be set in the object's private security descriptor. You pass the address of a PSECURITY_DESCRIPTOR variable, which contains a pointer to the object's private security descriptor. *SetPrivateObjectSecurity* frees the security descriptor and replaces it with the new security descriptor, whose address is returned in the provided pointer variable. You must also pass *hClientToken* so that the system can confirm object owner settings, as well as the generic mapping structure.

The modification of a security descriptor is typically performed using these steps:

1. Retrieve the object's security descriptor.
2. Create and initialize a new security descriptor (shown earlier in this chapter).
3. Copy security information from the original security descriptor to the new security descriptor.
4. Modify the new security descriptor (using techniques described throughout this chapter).
5. Set the new security descriptor back to the private object.

To get the individual components of a security descriptor, you should use the following functions: *GetSecurityDescriptorOwner*, *GetSecurityDescriptorDacl*, *GetSecurityDescriptorSacl*, and *GetSecurityDescriptorGroup*. To set the components in a new security descriptor, you can use *SetSecurityDescriptorOwner*, *SetSecurityDescriptorDacl*, *SetSecurityDescriptorSacl*, and *SetSecurityDescriptorGroup*. These functions are similar, so I will show and discuss the most common and complex two, *GetSecurityDescriptorDacl* and *SetSecurityDescriptorDacl*. Here is *GetSecurityDescriptorDacl*:

```
BOOL GetSecurityDescriptorDacl(
   PSECURITY_DESCRIPTOR pSecurityDescriptor,
   PBOOL                pfDACLPresent,
   PACL                 *pDACL,
   PBOOL                pfDACLDefaulted);
```

This function retrieves a pointer to the DACL in a security descriptor. It also indicates whether a DACL is present, and whether it was originally created via default security. You supply the security descriptor as well as the address of a Boolean value indicating whether a DACL is present, the address of a PACL variable to retrieve the DACL, and the address of another Boolean value, which the system uses to indicate whether the DACL was defaulted.

To set the DACL back to a security descriptor, you can use *SetSecurityDescriptorDacl*:

```
BOOL SetSecurityDescriptorDacl(
   PSECURITY_DESCRIPTOR pSecurityDescriptor,
   BOOL                 bDaclPresent,
   PACL                 pDacl,
   BOOL                 bDaclDefaulted);
```

This function simply takes the pointer to a security descriptor object that you wish to modify, and Boolean values indicating whether the DACL is present and whether it is the result of default security (for which you will typically pass FALSE). The *SetSecurityDescriptorDacl* function requires a pointer to a DACL that becomes the new DACL for the security descriptor.

NOTE

You should never attempt to set the DACL or any other component of a security descriptor that is returned for any object in the system. You should always create a new security descriptor, to which you can set components such as the DACL and SACL. The reason for this is security descriptors returned from the system are packaged in *self-relative* format. This means that the data comes in a single contiguous block of memory, which leaves no room for modification.

When you allocate memory and initialize a new security descriptor by using the *InitializeSecurityDescriptor* function, the system initializes a security descriptor that is in *absolute* format. A security descriptor in absolute format uses pointers to reference its components, allowing the components to be set and reset.

The RoboService Sample Service

The RoboService sample service ("10 RoboService.exe") demonstrates how private security objects and client/server access control are employed using named pipes. The source code and resource files for the application are in the 10-RoboService directory on the companion CD.

When you launch RoboService with the "/install" switch, the application installs itself as a service on the system. (See Figure 10-8.) You can then use any service control program to start and stop the service. Also, if you are executing the service from a debugger, you can pass the "/debug" switch, which causes the service to execute in debug mode, bypassing service functionality.

Figure 10-8. *The command-line options for RoboService*

After the service is running, you can run the RoboClient sample application, shown in Figure 10-9, and enter the computer name of the system running the service. If you don't enter a computer name, the client attempts to connect to the service on the local machine.

Once connected, you will see a list of "virtual robots" created by the service. These robots are secured using private security functionality on the service side. You may add and remove robots. You can perform several defined actions with the robots, including editing robot security.

Figure 10-9. *User interface for the RoboClient sample application*

I suggest you run the service and then launch the client from multiple user contexts, perhaps by using the RunAs.exe utility packaged with Windows. This will allow you to experiment with object ownership and access control.

The client is very thin, and defers nearly all functionality to the service. All security code (except for ACL editing) is implemented on the service side.

The service uses *impersonation* to obtain a token for each connecting client. (Impersonation is discussed in detail in Chapter 11.) The service then stores the token for use in future securable requests from the client and in robot creation. The service uses *CreatePrivateObjectSecurity*, *DestroyPrivateObjectSecurity*, and *AccessCheck*, as well as many other security-related functions, to implement the service.

In addition to security functionality, the service uses I/O completion ports, discussed in Chapter 2, to communicate efficiently with the client, implementing a model for scalable communication.

Auditing and the SACL

Finally, we're ready to discuss auditing and the SACL. Unlike the DACL, the SACL of an object has no effect on who can access an object. However, when access is requested of an object, the SACL can cause an event to be added to the event log. Specifically, you can add ACEs to an object's SACL, which adds events to the event log under two conditions:

- When an access check succeeds for any of a set of access rights
- When an access check fails for any of a set of access rights

For example, you could create a SACL for a file on an NTFS partition that contains a single SYSTEM_AUDIT_ACE. This would indicate that each time a certain trustee failed to write to the file because of denied access, an event should be added to the event log by the system.

Auditing is disabled by default in Windows 2000, so you should enable auditing if you wish to begin writing software that audits. Follow these steps to enable auditing for your system:

1. Run the Microsoft Management Console (MMC) with the /a switch, *mmc /a*.
2. On the Console menu, select Add/Remove Snap-in, and then click the Add button.
3. In the Add Standalone Snap-in dialog box, select Group Policy, and then click Add.
4. In the Select Group Policy Object dialog box, the Local Computer object should be selected, so click Finish.
5. Click the Close button, and then click OK. The Local Computer Policy object should be displayed in the MMC.
6. Expand the Local Computer Policy object as follows: Local Computer Policy\Computer Configuration\Windows Settings\Security Settings\Local Policies\Audit Policy.
7. In the right pane, right-click Audit Object Access, and then select Security.
8. In the Local Security Policy Setting dialog box, check the Success and Failure check boxes as shown in Figure 10-10 and click OK.

Figure 10-10. *Enabling success and failure auditing for object access*

After you have followed these steps, the system begins adding audit events to the event log under the Security log.

NOTE

A domain policy can override your local policy, preventing auditing from being set. You must have domain administrative rights to adjust domain policy.

Believe it or not, you've already finished the hard work for implementing auditing. You know how to create ACEs and modify ACLs from the work we did with DACLs. Just about everything you have learned so far applies to creating ACEs and SACLs.

Here is the ACE structure used for the SACL:

```
typedef struct _SYSTEM_AUDIT_ACE {
   ACE_HEADER  Header;
   ACCESS_MASK Mask;
   DWORD       SidStart;
} SYSTEM_AUDIT_ACE;
```

This structure should look familiar, because it is exactly the same as the ACCESS_ALLOWED_ACE and ACCESS_DENIED_ACE structures that we have been discussing. The *AceFlags* member of the ACE_HEADER structure typically contains inheritance information, which is also true for audit ACEs. But you should also include one or both of the SUCCESSFUL_ACCESS_ACE_FLAG and FAILED_ACCESS_ACE_FLAG audit flags to indicate auditing for successful and failed access, respectively.

To modify SACLs for objects, use the functions that you are already familiar with, such as *InitializeAcl* and *AddAce*. Getting and setting the SACL for a system object is still handled using *GetSecurityInfo* and *SetSecurityInfo*, in much the same manner you would use them for the DACL. Creating a SACL is somewhat simpler than creating a DACL, because the order of ACEs in a SACL has no effect on the object. You can add ACEs in any order that is convenient for your software.

NOTE

One difference you'll notice in dealing with DACLs is that a trustee cannot set the SACL of an object unless it has the SE_AUDIT_NAME privilege.

Access Control Programming Considerations

You now understand what is possible with access control and how to implement access control in your own software, but you've still got more to learn to become a successful security programmer—it has been argued that the flexibility of Windows access control gives the security developer more than enough rope to hang himself. In this section, I discuss some issues you should consider in your security development. My overall advice to you is to plan the access control of your service. With a little forward planning, you will save yourself a great deal of headache.

Memory Management

Unlike 90 percent of the programming you do at the application level, security programming requires a fair amount of buffer management, including the allocating and moving of structures within memory. Although this is what many developers focused on in school, it is not a terribly common task in Windows development. When dealing with security, you should come up with ways to simplify your memory management because so many functions require the allocation and reallocation of temporary buffers. Here are some tips:

- Consider using functions such as *_alloca* that do not require memory to be freed.
- Consider writing a class such as the CAutoBuf class, which is discussed in Appendix B. (You will also find the sample applications in Appendix B.) This class automatically adjusts its buffer and frees itself when it goes out of scope.
- Avoid static buffers; they are a lazy programmer's technique and will eventually break your application.
- Think about the task at hand, and design an approach that requires as few buffer allocations as possible. Place as much of the buffer allocation at the beginning of the algorithm as possible.
- Use structured exception handling to clean up after yourself.

Code Reuse

The more code you write, the more bugs you write. With access control programming, this is doubly true. Plan ahead and write code that can be used generally throughout your service. Try to design the security of different aspects of your service to be consistent with one another so that you can depend on certain patterns and reuse functions as much as possible. This reuse will greatly ease the task of debugging your security code.

Finally, I can't express enough how advantageous it is to implement your security code in a set of thin C++ classes. Doing so buys you code reuse and encapsulation that can greatly simplify a task such as access control.

Keep It Simple

Simple security is often the best security. If an object has dozens of ACEs, your software and users are probably going to have trouble keeping tabs on who really has access to the object. If an object has only a few

simple ACEs, both the software and the users will be clear about who has access. An object that is owned by the system and has only an empty DACL (not a NULL DACL) is as simple and secure as an object gets in Windows. Add a single ACE to allow one trustee access and the object is still secure, and it is clear who has access to the object.

Try to implement as much of your access control as possible using only access-allowed ACEs. Remember that access is denied implicitly unless it is explicitly allowed. If you must use both access-allowed ACEs and access-denied ACEs, allow access to groups and individuals, and deny access to individuals and smaller groups. This way you won't find yourself needing to override a group access-denied ACE for a single user in the group.

Keep it simple to avoid holes in your access control.

Use Default Security and Inherited Security

If at all possible, use default security and inherited security. If your service can avoid manipulating ACLs on individual objects, great! Many services can get by with creating only a single DACL and setting it as the default DACL for the service (discussed in more detail in Chapter 12). An object created by the service can use the default security, avoiding the need to create a DACL for each object created. This approach can greatly simplify a service that must create many secure objects.

> **NOTE**
>
> Remember that an object that inherits security will not use ACEs from the default DACL. You might find that you'll need to protect a parent directory or registry key's security descriptor if you want to use default security on child objects.

Inherited security is another way to get a DACL for free. You can set the DACL on a single parent object in a file structure (or registry tree), and each object you create under that node can inherit a whole DACL's worth of ACEs. This is a very powerful technique, and it can free your code from creating dozens of individual DACLs.

> **NOTE**
>
> Remember that an object that is protected from inheritable ACEs can itself have inheritable ACEs. So you can create a root directory or registry key that is protected from inherited ACEs, and the object's ACEs can define the inherited security for every object below it. This is a common and powerful tactic for implementing inherited security on files and registry keys.

[Previous] [Next]

Chapter 11

User Context

In the previous chapter, we focused on the Microsoft Windows implementation of *access control*. Access control in Windows is user-based, allowing system administrators and security programmers to control who can access objects in the system to a very fine granularity. For this control to work, the system must maintain an identity for any code executing in the system. This identity is referred to as *user context*. Sometimes user context is referred to as *security context* or *user security context*.

In addition to the system's need to maintain a sense of identity, a secure service must also be able to maintain a sense of its clients' identities. Two goals for your service software are as follows:

1. Identify the client connecting to your service.

2. Ensure that the service executes code *appropriate* for the client connecting to your service, regardless of the requests the client makes.

This chapter describes the features you can use to develop your service so that Windows security meets these goals.

[Previous] [Next]

Understanding User Context

The system maintains user context information in a token (introduced in Chapter 10). Before I discuss how the system uses the token to maintain a sense of user identity, I will discuss how the system builds a token.

Authentication and Token Contents

When you interactively log on to a workstation or server machine running Microsoft Windows 2000, you typically enter a username and a password. The username is a trustee account that exists either in Active Directory located on a domain controller or as a local user account maintained in a security database on the local machine. The password is the user account's *credentials,* which the system uses to *authenticate* the logon.

After the user logs on, the system begins to gather information about the user account (unless of course the username or password is incorrect, in which case the logon fails). If the user account is a domain account, the system gathers domain-level information such as the domain groups of which the user is a member. Regardless of whether the account is a domain account, the system begins gathering information about the user relative to the local machine, such as the local group accounts of which the user is a member (including nested memberships via domain groups that are members of local groups). The system creates a list of privileges that are assigned to the user account as well as all the users' group accounts. All this information is compiled into a data structure that is maintained in a system kernel object, which is the token.

After the system has authenticated a logon attempt, the token *is* the user's identity as far as Windows security is concerned. This identity consists of the security identifier (SID) for the user trustee account and all the information shown in Table 11-1. (See Chapter 9 for information on SIDs and trustee accounts.)

As you can see, a wealth of information is maintained in a token. Think of the token as a very detailed "virtual security badge," where the token's user SID is the name printed on the badge and the group SIDs are the membership information printed on the badge. The privileges in the token indicate certain system-wide rights; the remaining data is mainly informational. The system references this virtual badge every time you attempt to access a secure component in the system.

Tokens and Executing Code

So far, I have covered how a token is built (in general terms) and what information is stored in it. Now I need to clarify how the information in a token is used to assure that code is executed securely.

Every process in Windows 2000 owns a single token, and as you know, this token is the user context for the process. When a process calls *CreateProcess*, the system makes a duplicate of the calling process's token and associates the new token with the new process. In this way, your user context propagates throughout the system, from the moment that you log on to the moment that the last process owning one of your tokens terminates. Every time you log on, the system builds a token for your user context and then launches Microsoft Windows Explorer and associates the token with Explorer's process. Each time you launch a program from Explorer, the new process inherits a copy of the original token. You are not logged off the system until each process has terminated and the Explorer process has exited, closing the last handle to the last token object for your user context.

Table 11-1. *Contents of a token*

Token Information	Description
Token user SID	A SID representing the user account for which the token was created. This account is commonly referred to as the *token user*.
Token group SIDs	SIDs for the groups of which the token user is a member. This includes built-in accounts such as Everyone and Authenticated Users.
Logon SID	A unique SID created at the time of authentication. This SID uniquely identifies a logon session from other logon sessions, even if the other sessions are for the same token user. I will discuss the logon SID in more detail later in this chapter.
Privileges	A list of privileges held by the token user and group trustee accounts. Privileges are discussed in detail in Chapter 9, as well as later in this chapter.
Default owner SID	A SID indicating the owner of objects created with default security by code running under this token. Default security is discussed in Chapter 10 and later in this chapter.
Default group SID	A SID indicating the primary group of objects created with default security by code running under this token. Default security is discussed later in this chapter and in Chapter 10.
Default DACL	A SID indicating the discretionary access control list (DACL) of objects created with default security by code running under this token. Default security is discussed in Chapter 10 and later in this chapter.
Token source	An eight-character "source name" set by the system component that created the token.
Token type	The type of token, which can be either primary or impersonation. Primary and impersonation tokens will be discussed in more detail later in this chapter.
Restricting SIDs	Optional trustee accounts used to restrict access for code running under this token. Restricted SIDs are discussed in detail later in this chapter.
Impersonation level	The impersonation level of a token indicates to what degree a server can act as the client. Impersonation levels are discussed later in this chapter.
Security descriptor	The token is a secured object in Windows, and like all secured objects, it has a security descriptor, which controls access to the object.
Credential information	Some tokens have credential information. These tokens can act on behalf of

the token user on the network, as well as on the local machine. Credential information is discussed in more detail throughout this chapter.

When any thread in your process executes a system function, which requires knowledge of security, the system checks your process's token for the information. This means that each time you ask the system for a handle to a secured object such as a registry key or a mutex, the system performs an access check, looking up necessary information in your token. Also, each time you call a function such as *GetUserName*, or any other function that must identify the user context of the executing code, the system retrieves the information from the token.

Although you can retrieve the handle to a token object for a process, setting the token for a process is impossible. (Actually, the system has the power to set the token for a process, but this is not exposed to the developer via the API.) So a process's token, or identity, is the one that the process is stuck with until it dies. Windows does, however, provide a very cool feature that allows you to change the token under which your code is executing, called *impersonation*.

Overview of Impersonation

Impersonation is implemented by associating a token with a thread of execution in a process. When you do this, this thread (for the most part) takes on the identity associated with its *impersonation token*. This thread is said to be "impersonating" a user or a security context. All other threads in the process will continue to execute code on behalf of the process's token unless they are also impersonating another user context. When the thread no longer needs to behave as though it were another user, it can revert to its normal state, which causes the thread to become itself again and use the process's token. As you can imagine, impersonation is very convenient in a client/server environment, in which a server is performing tasks on behalf of a client with a different security context.

Before moving on, I should clarify my assertion that a thread (for the most part) takes on the identity associated with its impersonation token. Typically, when you call a function that requires token information, the system checks for a thread token, and if one is not found, the system defaults to the process's token. Since threads don't naturally have tokens, the process's token is the natural security context for the process. However, in some cases a system function will still use a process's token even when the calling thread is running under an impersonation token.

For example, if an impersonating thread calls *CreateProcess* to create a new process in the system, the new process inherits the handle associated with the calling process, not with the impersonation token associated with the calling thread. Additionally, any function that requires the SE_TCB_NAME privilege (of which we will be discussing at least one, *LogonUser*, later in this chapter) or the SE_AUDIT_NAME privilege uses the process's token rather than checking the thread for an impersonation token. We'll talk more about impersonation later in this chapter.

You now understand how the system builds a token for you when you log on. You know how the system executes code that runs under your token (or security context), and you know that a thread can use impersonation to temporarily take on another identity. However, up to this point we have discussed only tokens that were created as a result of a logon with a user account such as "JClark" or "v-JeffrR". Now you'll learn how code that isn't running under a user account gets its token.

The LocalSystem Account

If you read Chapter 3, you are aware that services can be configured to execute regardless of whether a user is interactively logged on to the system. Also remember that you can configure a service to run as a user account, which requires you to enter the account name and password for the trustee account. It is easy to

imagine the system logging on this user behind the scenes and building a token to use when running the process that executes the service's code. However, you can also configure a service to run as the LocalSystem account, which does not require an account name or a password.

The LocalSystem account is a special account on a system running Windows 2000 that is set aside for services and other processes running as "part of the operating system." As a result, the token that any process in the LocalSystem account has assigned to it is very powerful indeed. Here are some qualities of the LocalSystem account:

- The LocalSystem account has nearly every privilege in the system assigned to it.
- The LocalSystem account is an implicit member of the built-in Administrators group, and as such has all the access that an administrator has.
- If the LocalSystem account does not have access to an object explicitly granted to it, it can always take ownership of the object and modify its DACL as desired.
- The LocalSystem account has the SE_TCB_NAME or "Act as part of the operating system" privilege, so it can log on a user (to obtain a token), and execute code on behalf of that user, at will.
- The LocalSystem account has direct or indirect access to everything on the local machine.

NOTE

In Microsoft Windows NT the LocalSystem account lived a life of irony—it had limitless access to its local machine and almost no access to securable objects on other machines on the network. The reason was that the LocalSystem account for one machine had no identity on another machine (in the same way that the Administrator account for one machine has no meaning on another machine). So objects on other machines were naturally secured against the LocalSystem account. Like Aladdin's genie, the LocalSystem account had "phenomenal cosmic powers! Itty bitty living space."

This limitation has been lifted, however, in network environments where the domain controller is a server running Windows 2000 using Active Directory. LocalSystem accounts now have trustee account status in Active Directory, and therefore can be given rights to securable objects on other machines in the domain.

The LocalSystem "genie" is now free from the single machine that used to be its prison. This change adds greatly to the flexibility of the LocalSystem account, but also opens potential security risks that must be closely administered.

Notice that a number of the functions provided for manipulation of tokens and other user context_related activities require privileges commonly granted only to the LocalSystem account. This is not a real problem, since your service will most likely run in the LocalSystem account. However, this need for privileges can be a bother when you are testing code and trying functions such as *LogonUser* and *CreateProcessAsUser* (both of which are discussed later in this chapter). We will discuss ways of dealing with the privilege issue and other problems arising from coding for the LocalSystem account later in this chapter.

User Context and Access Control

Although the previous chapter was entirely devoted to access control, I spent very little time focusing on the importance of user context. However, the security context that your process's (or thread's) token represents is absolutely critical to the function of access control.

Each time the system performs an access check, it scans the object's DACL for access control entries (ACEs) that match the user SID and group SIDs found in your token. If you are not explicitly allowed to read or modify an object's security, but you request permission to do so, the system compares the object's owner SID with the SIDs in your token, looking for a match. If one is found, you are granted access because you are an owner of the object.

As you can see, access control is completely reliant on the token. In fact, we will be discussing ways that you can exploit the token to have dramatic effects on access control in your software and in your system. Gaining an in-depth understanding of how these components work together to implement security under Windows 2000 will make you a powerful security developer indeed.

[Previous] [Next]

Programming User Context

We will be learning a number of powerful and interesting techniques that can be performed by manipulating tokens in Windows 2000. The first step down the path to all the fun is to acquire a handle to a token. So let's begin our journey by obtaining a handle to a process's token:

```
BOOL OpenProcessToken(
   HANDLE  hProcessHandle,
   DWORD   dwDesiredAccess,
   PHANDLE phTokenHandle);
```

The *OpenProcessToken* function retrieves a handle to a process's token. The *hProcessHandle* parameter is the handle to the process for which we wish to open the token. Unfortunately the system does not allow you to pass NULL to indicate your desire to retrieve a handle to your own process's token. But a quick call to *GetCurrentProcess* returns a pseudo-handle that can be passed as the *hProcessHandle* parameter. The *dwDesiredAccess* parameter indicates how the token is to be used. You should request only the rights that your code needs. Table 11-2 describes the available token access rights. Many of the functions listed in the table will be discussed later in this chapter.

The variable whose address you pass as *phTokenHandle* receives a handle to the requested token. If the return value of *OpenProcessToken* is TRUE, the function succeeded; otherwise the function failed, in which case you should call *GetLastError* for more information.

Table 11-2. *Token access rights*

Value	Description
TOKEN_ADJUST_DEFAULT	Required when calling *SetTokenInformtion* (discussed shortly) to change features of the token, such as the default owner, primary group, or default DACL.
TOKEN_ADJUST_GROUPS	Required in a call to *AdjustTokenGroups*.
TOKEN_ADJUST_PRIVILEGES	Required in a call to *AdjustTokenPrivileges*.
TOKEN_ADJUST_SESSIONID	Required to adjust the session ID of token and requires the SE_TCB_NAME privilege.
TOKEN_ASSIGN_PRIMARY	Required when using the token in calls to *CreateProcessAsUser*.
TOKEN_DUPLICATE	Required to duplicate the token.
TOKEN_EXECUTE	

	Equals STANDARD_RIGHTS_EXECUTE. See Chapter 10 for further discussion of standard rights.
TOKEN_IMPERSONATE	Required to use this token with *ImpersonateLoggedOnUser*.
TOKEN_QUERY	Required to read any token information other than its source using *GetTokenInformation*.
TOKEN_QUERY_SOURCE	Required to read the token's source using *GetTokenInformation*.
TOKEN_READ	Combines STANDARD_RIGHTS_READ and TOKEN_QUERY. See Chapter 10 for further discussion of standard rights.
TOKEN_WRITE	Combines STANDARD_RIGHTS_WRITE, TOKEN_ADJUST_PRIVILEGES, TOKEN_ADJUST_GROUPS, and TOKEN_ADJUST_DEFAULT. See Chapter 10 for further discussion of standard rights.
TOKEN_ALL_ACCESS	Complete access to the token, combining all rights.

The most likely reason for *OpenProcessToken* to fail is insufficient access rights; however, if your service is running in the LocalSystem user context, it will probably have sufficient access to any process's token. (For more information on access control, see Chapter 10.)

Token objects are kernel objects, and as with most kernel objects, you should pass the token object's handle to *CloseHandle* when you are finished using the resource.

NOTE

The handle you receive when calling *OpenProcessToken* is the handle of the token, which affects the user context of that process. Anything you do to adjust the token can have dramatic and immediate effects on the secure behavior of the process from which you retrieved the token. I will be discussing what you can do to a token shortly. For information on how to obtain the handle to a process (other than the current process), see the discussion of *OpenProcess* in Chapter 22 of *Programming Applications for Microsoft Windows, Fourth Edition* (Jeffrey Richter, Microsoft Press, 1999).

If you guessed that there must also be a way to obtain a thread's token (assuming the thread is impersonating at the time), you are correct. You can do this using *OpenThreadToken*:

```
BOOL OpenThreadToken(
   HANDLE  hThreadHandle,
   DWORD   dwDesiredAccess,
   BOOL    fOpenAsSelf,
   PHANDLE phTokenHandle);
```

Notice that *OpenThreadToken* is very similar to *OpenProcessToken* except that the first parameter is a handle to a thread rather than a process, and it has the additional parameter *fOpenAsSelf*. This parameter indicates to the system *who* you want to open the token as. Let me explain.

Remember that when you call *OpenThreadToken* you are requesting a token, which is a secured object in Windows. This means that you might or might not have access to the object. Remember also that you are running in a thread that could be impersonating a security context other than your process's security context. Because it is very common for a thread to retrieve a handle to its own token when it is impersonating, the system allows you to indicate that you want all access checks that are necessary to open a thread's token to be performed against your process's token. You should pass TRUE for the *fOpenAsSelf* parameter of

OpenThreadToken if you want access checks to be performed against your process's token. You should pass FALSE if you want access checks to be performed against your thread's impersonation token. This parameter has no meaning if your thread is not currently impersonating. (I will discuss impersonation in more detail later in this chapter.)

Like *OpenProcessToken*, the *phTokenHandle* parameter of *OpenThreadToken* returns a handle to the requested token. If the return value of *OpenThreadToken* is TRUE, the function succeeded; otherwise the function failed, in which case you should call *GetLastError* for more information.

If the thread is not impersonating, the *OpenThreadToken* function will fail and *GetLastError* will return ERROR_NO_TOKEN.

So now you have a handle to a token. What can you do with it? you ask. Read on.

Reading Token Information

Most of the token information listed in Table 11-1 can also be read from a token, assuming that the calling code has TOKEN_QUERY (or TOKEN_QUERY_SOURCE) access to the object. You should use the *GetTokenInformation* function to find the contents of a token:

```
BOOL GetTokenInformation(
   HANDLE hTokenHandle,
   TOKEN_INFORMATION_CLASS tokenInformationClass,
   PVOID  pTokenInformation,
   DWORD  dwTokenInformationLength,
   PDWORD pdwReturnLength);
```

You should pass a handle to a token as the *hTokenHandle* parameter. The *tokenInformationClass* parameter indicates what information you wish to obtain from the token, and the *pTokenInformation* parameter is a pointer to a buffer that the system fills with the requested information. The *dwTokenInformationLength* parameter indicates the length of the buffer you passed, and the *pdwReturnLength* parameter points to a variable that receives the size of the buffer necessary to retrieve the information.

The buffer pointed to by the *pTokenInformation* parameter varies in type, depending on the *tokenInformationClass* parameter. The following list describes the information that you can retrieve from a token: the information class value and the data type used. (The TOKEN_INFORMATION_CLASS is also used with the *SetTokenInformation* function, discussed later in this chapter.)

- ***TokenUser*** Returns the SID of the token user. This is the user whose account name was used to create the token during logon. (This value is not used with *SetTokenInformation*.)

```
typedef struct _TOKEN_USER {
   SID_AND_ATTRIBUTES User;
} TOKEN_USER;
```

- ***TokenDefaultDacl*** Used to read or set a token's default DACL. See Chapter 10 for further discussion of default DACLs as well as detailed discussion on building a DACL. The TOKEN_ADJUST_DEFAULT access right is required to set information in the DACL.

```
typedef struct _TOKEN_DEFAULT_DACL {
   PACL DefaultDacl;
} TOKEN_DEFAULT_DACL;
```

- ***TokenOwner*** Used to read or set the default owner of the token object. (See Chapter 10 for discussion of object ownership.) The TOKEN_ADJUST_DEFAULT access right is necessary to set the owner of a token.

```
typedef struct _TOKEN_OWNER {
   PSID Owner;
} TOKEN_OWNER;
```

- ***TokenPrimaryGroup*** Reads or sets the token's primary group. The TOKEN_ADJUST_DEFAULT access right is required in calls to *SetTokenInformation*.

```
typedef struct _TOKEN_PRIMARY_GROUP {
   PSID PrimaryGroup;
} TOKEN_PRIMARY_GROUP;
```

- ***TokenGroups*** Returns the SIDs of the groups associated with the token. (This value is not used with *SetTokenInformation*.)

```
typedef struct _TOKEN_GROUPS {
   DWORD GroupCount;
   SID_AND_ATTRIBUTES Groups[ANYSIZE_ARRAY];
} TOKEN_GROUPS;
```

- ***TokenPrivileges*** Returns the privileges associated with the token. (This value is not used with *SetTokenInformation*.)

```
typedef struct _TOKEN_PRIVILEGES {
   DWORD PrivilegeCount;
   LUID_AND_ATTRIBUTES Privileges[ANYSIZE_ARRAY];
} TOKEN_PRIVILEGES;
```

- ***TokenSource*** Returns the token's source. This is a textual string representing the creating entity of the token. The TOKEN_QUERY_SOURCE access right is necessary to retrieve this information. (This value is not used with *SetTokenInformation*.)

```
typedef struct _TOKEN_SOURCE {
   CHAR SourceName[8];
   LUID SourceIdentifier;
} TOKEN_SOURCE;
```

- ***TokenType*** Returns the token's type. The possible values are TokenPrimary and TokenImpersonation. The *pTokenInformation* parameter of *GetTokenInformation* will return a single TOKEN_TYPE indicating the token's type. (This value is not used with *SetTokenInformation*.)

```
typedef enum _TOKEN_TYPE {
   TokenPrimary = 1,
   TokenImpersonation
} TOKEN_TYPE;
```

- ***TokenImpersonationLevel*** Returns impersonation level. See Table 11-3 for more information. (This value is not used with *SetTokenInformation*.)

```
typedef enum _SECURITY_IMPERSONATION_LEVEL {
   SecurityAnonymous,
   SecurityIdentification,
   SecurityImpersonation,
   SecurityDelegation
} SECURITY_IMPERSONATION_LEVEL;
```

- ***TokenStatistics*** Returns general information about the token. Members of note include *GroupCount*, *PrivilegeCount*, and *ModifiedId*. The *ModifiedId* member is a locally unique identifier (LUID) that changes each time the token is modified. Your code can use this value to detect whether or not a token has changed since the last time you checked. This detection ability can be very useful in writing fault-tolerant code that calls into third party DLLs or libraries. (This value is not used with *SetTokenInformation*.)

```
typedef struct _TOKEN_STATISTICS {
   LUID TokenId;
   LUID AuthenticationId;
   LARGE_INTEGER ExpirationTime;
   TOKEN_TYPE TokenType;
   SECURITY_IMPERSONATION_LEVEL ImpersonationLevel;
   DWORD DynamicCharged;
   DWORD DynamicAvailable;
   DWORD GroupCount;
   DWORD PrivilegeCount;
   LUID ModifiedId;
} TOKEN_STATISTICS;
```

- ***TokenRestrictedSids*** Returns the token's restricted SIDs. Restricted tokens are covered in detail later in this chapter. (This value is not used with *SetTokenInformation*.)

```
typedef struct _TOKEN_GROUPS {
   DWORD GroupCount;
   SID_AND_ATTRIBUTES Groups[ANYSIZE_ARRAY];
} TOKEN_GROUPS;
```

- ***TokenSessionId*** Indicates a Terminal Server session ID for the token. The *pTokenInformation* parameter of *GetTokenInformation* will return a DWORD. The value of the DWORD is 0 if Terminal Server is not installed on the local machine or if the token is associated with the Terminal Server console. Otherwise the token is associated with a Terminal Server client, and the value of the DWORD is a session ID for the client. (This value is not used with *SetTokenInformation*.)

```
DWORD
```

As the previous list shows, you can read a great deal of information from a token. Here are the most common items to retrieve from a token:

- **The token user SID** The user account that the token represents. This information is commonly received from a token to find out who is executing the code. You can pass this SID to *LookupAccountSid* (discussed in Chapter 9) to get the textual name of the user account.

- **The logon SID** This SID is buried in with the token groups, so it takes a little digging to retrieve (which I will discuss in a moment). However, the logon SID can be very convenient for uniquely identifying the session. If a user logs on to a machine running Windows 2000 more than once (either interactively or through other means), the system creates a unique logon SID for each session, regardless of whether the token user is the same from session to session.

- **Token groups** Useful for finding the groups associated with a token. However, if you want to find out whether a token has a single group, you can use the *CheckTokenMembership* function.

- **Token default DACL** Useful for finding out exactly what the DACL of a newly created object will be if you pass NULL for the security attributes when creating the object.

When retrieving information from a token, you will typically have to call *GetTokenInformation* to find out the required length of the buffer, and then call it again to actually retrieve the information. To create truly fault-tolerant code, you should be prepared to call *GetTokenInformation* repeatedly until you have successfully retrieved the desired information. This is because the size of token information (such as the default DACL) can change between the time you call *GetTokenInformation* to retrieve the buffer size and the time you call *GetTokenInformation* to retrieve the actual data. This condition in multithreaded programming is known as a *race condition* and can cause those hard-to-find bugs that appear only once every several months or so.

The following function shows how to call *GetTokenInformation* properly to receive information about a token. It also returns the token information in a buffer allocated using *LocalAlloc*. If you use this function in your code, you should free the returned buffer using *LocalFree* when you are finished with the buffer.

```
LPVOID AllocateTokenInformation(HANDLE hToken,
   TOKEN_INFORMATION_CLASS tokenClass ){
   PVOID    pvBuffer = NULL;
   __try{
      BOOL fSuccess;
      // Initial buffer size
      ULONG    lSize = 0 ;
      do
      {
         // Do we have a size yet?
         if (lSize != 0)
         {
            // Do we already have a buffer?
            if (pvBuffer != NULL)
               LocalFree(pvBuffer);// Then free it
            // Allocate a new buffer
            pvBuffer = LocalAlloc(LPTR, lSize) ;
            if(pvBuffer == NULL)
               __leave;
         }

         // Try again
         fSuccess = GetTokenInformation( hToken, tokenClass,
            pvBuffer, lSize, &lSize ) ;
      // Still not enough buffer?
      }while( !fSuccess && (GetLastError() ==
         ERROR_INSUFFICIENT_BUFFER)) ;

      // If we failed for some other reason, back out
      if(!fSuccess)
      {
         if(pvBuffer)
            LocalFree(pvBuffer) ;
         pvBuffer = NULL;
      }
   }__finally{}
   // Return locally allocated buffer
   return (pvBuffer) ;
}
```

The following code fragment shows the use of this function to retrieve the token's user SID and default DACL for the current process.

```
HANDLE hToken;

if(!OpenProcessToken( GetCurrentProcess(), TOKEN_QUERY, &hToken)){
   // Error
}

TOKEN_USER* ptUser = (TOKEN_USER*)AllocateTokenInformation(hToken,
   TokenUser);
if ( ptUser != NULL){
   // Do something with the SID pointed to by ptUser->User
}

TOKEN_DEFAULT_DACL* ptDACL =
   (TOKEN_DEFAULT_DACL*)AllocateTokenInformation(hToken,
   TokenDefaultDacl);
if ( ptDACL != NULL){
   // Do something with the DACL pointed to by ptDACL->DefaultDacl
}
```

Notice that a number of the structures returned by *GetTokenInformation* use the SID_AND_ATTRIBUTES structure, which is defined as follows:

```
typedef struct _SID_AND_ATTRIBUTES {
   PSID  Sid;
   DWORD Attributes;
} SID_AND_ATTRIBUTES ;
```

This structure includes a SID for a trustee account and an *Attributes* member, which includes information about the SID. See the Platform SDK documentation for the meaning of the *Attributes* member for specific structures that include a SID_AND_ATTRIBUTES structure.

The following function, *AllocateTokenLogonSID*, uses the *AllocateTokenInformation* sample function to retrieve the token's group SIDs, and then iterates through the SIDs to find the logon SID discussed earlier in this section. The function finds the logon SID by checking the *Attributes* member for the SE_GROUP_LOGON_ID flag. The returned PSID should be freed using *LocalFree*.

```
PSID AllocateTokenLogonSID(HANDLE hToken){
   PSID psidLogon = NULL;
   TOKEN_GROUPS* ptGroups = NULL;
   __try{
      // Get the token groups
      ptGroups = (TOKEN_GROUPS*)
         AllocateTokenInformation(hToken, TokenGroups);
      if (ptGroups == NULL)
         __leave;

      // Find the logon SID
      int nCount = ptGroups->GroupCount;
      while (nCount--){
         if ((ptGroups->Groups[nCount].Attributes & SE_GROUP_LOGON_ID)
            != 0)
            break;
      }
      if (nCount == -1)
         __leave; // No logon SID found

      // Get memory for the returned SID
      ULONG lLen = GetLengthSid(ptGroups->Groups[nCount].Sid);
      psidLogon = (PSID)LocalAlloc(LPTR, lLen);
      if (psidLogon == NULL)
         __leave;

      // Copy the logon SID
      if(!CopySid(lLen, psidLogon, ptGroups->Groups[nCount].Sid)){
         LocalFree(psidLogon);
         psidLogon = NULL;
         __leave;
      }
   }__finally{
      if (ptGroups != NULL)
         LocalFree(ptGroups);
   }
   return (psidLogon);
}
```

The TokenMaster Sample Application

The TokenMaster sample application ("11 TokenMaster.exe") demonstrates the use of the token-related system functions, including *GetTokenInformation* and *SetTokenInformation*. The source code and resource files for the sample application are in the 11-TokenMaster directory on the companion CD. The program

allows you to acquire a token from one of four sources: a process, a thread, a user's credentials, or via duplication. The program also allows you to view and modify token information. When the user executes TokenMaster, the dialog box in Figure 11-1 appears.

Figure 11-1. *User interface for the TokenMaster sample application*

The source code for this sample will help you understand how to apply the concepts discussed in this chapter. As a user of the TokenMaster sample application, you can explore the many capabilities of user context in Windows 2000. I will describe TokenMaster before I discuss some of the coding techniques so that you can use the sample to test some of the concepts in the chapter.

When TokenMaster is executed, its first task is to check its own user identity. If it is running under any account other than the LocalSystem account, it does the following in an attempt to upgrade itself:

1. Enumerates the processes running in the system, and locates the System process.

2. Uses *OpenProcessToken* to acquire a handle to the System process's token.

3. Uses *CreateProcessAsUser* (which is discussed later in this chapter) to re-execute itself under the LocalSystem user context.

4. If all three steps succeed, TokenMaster exits knowing that a new and more powerful instance of itself has been launched.

The features of TokenMaster are far more educational for you if you run the sample application under the LocalSystem account, but special privileges are required to take the first three steps. If you are interested in executing TokenMaster to its full potential, you must do the following:

1. Log on as an administrator of the system.

2. Add the "Increase Quotas" and "Replace a Process Level Token" privileges to your user account using the TrusteeMan sample application from Chapter 9 or the Microsoft Management Console (MMC) Group Policy snap-in.

3. Log out, and then log on again so that the new privileges take effect.

4. Launch TokenMaster.

If you are successful, when you execute TokenMaster, the Status window in the application should contain the following message: "Token Master, Status - Token Master running as SYSTEM". Otherwise, the window will report that TokenMaster is running as the account you used to launch the application.

The primary function of the application is to view information about a token and modify the information that can be changed. The following are ways to acquire a token:

- **Open a token from an existing process or thread** You can select a process and additionally a thread (if you are interested in opening an impersonation token) by using the Processes and Threads combo boxes. Click the button labeled OpenProcessToken or OpenThreadToken to grab the token.

- **Retrieve a process from the system by using a user's credentials** You can type in a username and password and select a logon type and provider. TokenMaster uses *LogonUser* (discussed later in this chapter) to retrieve a token from the system.

- **Duplicate an existing token** If you have already acquired a token for use by TokenMaster, you can make a duplicate token by clicking the DuplicateTokenEx button. This duplication option allows you to set the impersonation level and the token type of the new token. If TokenMaster has a token that can be duplicated, the Token Information window will display the token information.

After you have acquired a token, you will see a listing of information about the token in the Token Information window. You can view all retrievable information for a token by using the *GetTokenInformation* function.

Using TokenMaster, you can modify and adjust the token in a variety of ways:

- Adjust token groups and privileges, including the ability to enable or disable individual items.

- Modify a token's default DACL. See Chapter 10 for more information on default DACLs.

- Create a restricted token.

- Launch executables using a token. This is a particularly useful feature of TokenMaster. You can create a new process by using a token that you have modified using TokenMaster. This allows you to change a token or restrict it in some way, and then launch code to see how the process is affected.

I strongly suggest that you spend some time using the TokenMaster sample application to become familiar with the user context-related features of Windows.

You will also find that TokenMaster can be used to ease the process of debugging. For example, I often launch the application and "steal" a token from the LocalSystem process. I then use this token to execute the Microsoft Visual Studio development environment, with which I can compile and test code from the LocalSystem user account. In terms of security, using the LocalSystem token when testing approximates the behavior of code running in a service.

The source code for this sample application demonstrates a number of useful programming tactics. First, it calls nearly every system function covered in this chapter. It also offers some tips that might be useful to you if you are new to security programming for Windows.

Windows security functions often require the allocation and reallocation of small buffers. This requirement

can often be viewed as troublesome by developers, including myself. As a result, it is often tempting to hard-code buffer sizes or make other concessions that can lead to code that isn't airtight. The TokenMaster sample application addresses this requirement by making use of a simple template class known as CAutoBuf. You will see that this class greatly simplifies code where variable-sized buffers are required for system functions.

[Previous] [Next]

Modifying Token Information

Most token information is fixed. For example, you can't set a token's user to a SID that is different from the one it has, and you can't add group SIDs or privileges to a token. However, the modifications that you *can* make fall into two groups:

- Information that can be *adjusted*
- Information that can be *set*

NOTE

Although very little that already exists in a token can be changed, it is possible to make a new token based on an existing token and add restrictions. This technique is referred to as creating a restricted token, and is discussed in detail later in this chapter.

The process of setting information works the way you might imagine. You build a structure and pass it and a token handle to a system function, and the system sets information in the token.

Adjusting information works somewhat differently than setting information. Both the privileges and the groups in a token can be in two states: enabled and disabled. For example, although you can't "set" the privilege list in a token, you can enable and disable individual privileges that already exist in that list.

The most common modifications you will make to a token are to set the token default DACL (which is used for default security and was described in the previous chapter) and to adjust token privileges. You will find that the process of setting the token's default owner and adjusting token groups is very similar to setting the default DACL and adjusting token privileges, respectively.

Adjusting a Token's Privileges

Before discussing how to adjust a token's privileges, let's recap what we already learned about privileges in Chapter 9 and then build on that knowledge:

- Privileges are assigned to trustees (users and groups).
- When a user is logged on by the system, the user's privileges are copied into the user's token.
- Privileges are identified by a display name, a programmatic name, and a LUID.

Before you can adjust a token's privileges, you have to know how to get from a privilege's programmatic name to its LUID used by the system. This is done via a call to the *LookupPrivilegeValue* function:

```
BOOL LookupPrivilegeValue(
   LPCTSTR lpSystemName,
   LPCTSTR lpName,
   PLUID   lpLuid);
```

You should pass as the *lpSystemName* parameter a string representing the system name for which you wish to receive a LUID. Passing NULL indicates the local machine. The *lpName* parameter indicates the programmatic name of a system privilege, and you should pass a *"5" width="95%"*>

```
BOOL AdjustTokenPrivileges(
   HANDLE           hTokenHandle,
   BOOL             fDisableAllPrivileges,
   PTOKEN_PRIVILEGES pNewState,
   DWORD            dwBufferLength,
   PTOKEN_PRIVILEGES pPreviousState,
   PDWORD           pdwReturnLength);
```

The first parameter, *hTokenHandle*, indicates the token whose privileges' states you want to modify. The *fDisableAllPrivileges* parameter, when TRUE, causes all privileges in the token to be disabled, disregarding anything passed in the *pNewState* parameter. In such a case you should pass NULL for *pNewState*. Typically you use the *pNewState* parameter when you are only enabling or disabling a subset of the privileges awarded to your token. The *pNewState* parameter is a pointer to a TOKEN_PRIVILEGES structure, which is defined here:

```
typedef struct _TOKEN_PRIVILEGES {
   DWORD PrivilegeCount;
   LUID_AND_ATTRIBUTES Privileges[ANYSIZE_ARRAY];
} TOKEN_PRIVILEGES;
```

Notice that *Privileges* is a variable-sized array of LUID_AND_ATTRIBUTES structures. One structure should be in the array for each privilege you wish to adjust in the token, and the *PrivilegeCount* member should reflect the number of privileges. Here is the definition of the LUID_AND_ATTRIBUTES structure:

```
typedef struct _LUID_AND_ATTRIBUTES {
   LUID   Luid;
   DWORD  Attributes;
} LUID_AND_ATTRIBUTES;
```

The *Luid* member of each structure in the array should be set to the LUID for the privilege in question. And finally, the *Attributes* member should be set to zero to disable the privilege or to SE_PRIVILEGE_ENABLED to enable the privilege.

The *dwBufferLength* parameter of the *AdjustTokenPrivileges* function indicates the length, in bytes, of the buffer pointed to by *pPreviousState*, which points to a buffer that will receive the privileges' previous states. You can pass NULL for *pPreviousState* if no previous state information is needed. If previous state information is needed, you must supply a buffer of sufficient size. A pointer to required size is returned in the *pdwReturnLength* parameter. The buffer pointed to by *pPreviousState* will be structured identically to the TOKEN_PRIVILEGES structure and the embedded array you build for the *pNewState* parameter.

As you can see, enabling and disabling a single privilege is not a trivial task. However, you will find it to be a very common task. I have often thought that the developers of Windows should have provided a shorthand function, defined something like the following *EnablePrivilege* function, that offered easy access to a token's privileges.

```
BOOL EnablePrivilege(
   HANDLE hToken,
   PTSTR szPriv,
   BOOL fEnabled);
```

The *szPriv* parameter would be a string representing the programmatic name of the privilege to enable or disable, and the *fEnabled* parameter would indicate simply whether to enable or disable the privilege. I implemented such a function in the TokenMaster sample application. Here is the implementation of the function:

```
BOOL EnablePrivilege(HANDLE hToken, LPTSTR szPriv, BOOL bEnabled)
{
   TOKEN_PRIVILEGES tp;
   LUID             luid;
   BOOL             bRet = FALSE ;

   __try{
      // First look up the system-unique LUID for the privilege
      if(!LookupPrivilegeValue(NULL, szPriv /*SE_DEBUG_NAME*/, &luid))
      {
         // If the name is bogus...
         __leave ;
      }

      // Set up our token privileges "array" (in our case an array of one)
      tp.PrivilegeCount           = 1;
      tp.Privileges[0].Luid       = luid;
      tp.Privileges[0].Attributes = bEnabled?SE_PRIVILEGE_ENABLED:0;

      // Adjust our token privileges by enabling or disabling this one
      if(!AdjustTokenPrivileges(
         hToken,
         FALSE,
         &tp,
         sizeof(TOKEN_PRIVILEGES),
         NULL,
         NULL ))
      {
         __leave ;
      }
      bRet = TRUE ;
   }__finally{}

   return(bRet);

}
```

NOTE

The system has implemented another function named *AdjustTokenGroups* whose calling convention is very similar to *AdjustTokenPrivileges* except that it allows the enabling and disabling of token groups instead of privileges. However, enabling and disabling of token groups is rarely necessary. The TokenMaster sample application shows an example of *AdjustTokenGroups*.

Setting the Default DACL

The less common task of setting a token's default DACL is simpler than adjusting a token's privilege. A token's default DACL defines what the security access will be for securable objects created with default security. Securable objects include constructs such as files, mutexes, and threads. Securable objects are typically created with default security by passing NULL as the LPSECURITY_ATTRIBUTES parameter of the creating function. (For more information on this topic, see Chapter 10.) Modifying the default DACL can greatly simplify server code by removing the necessity to explicitly apply security to each object the code creates.

You can set a token's default DACL as well as its default owner and default primary group (neither of which should be confused with the token's user identity, which cannot be changed) by using the *SetTokenInformation* function:

```
BOOL SetTokenInformation(
   HANDLE                  hTokenHandle,
   TOKEN_INFORMATION_CLASS TokenInformationClass,
   PVOID                   pTokenInformation,
   DWORD                   dwTokenInformationLength);
```

The *hTokenHandle* parameter is the handle to the token you wish to modify. The *TokenInformatonClass* is used to indicate which piece of the token you want to change. Your options were discussed previously in the bulleted list.

The *dwTokenInformationLength* parameter of *SetTokenInformation* indicates the length of the buffer pointed to by *pTokenInformation*.

For a complete example of *SetTokenInformation* in action, see the TokenMaster sample application.

[Previous] [Next]

Using a Token to Execute Code

So far I have discussed what you can find out from a token, as well as what adjustments you can make to a token. But tokens don't really get interesting until you begin executing code under a token other than your process token. There are two ways to do this:

- Create a new process with a new token.
- Impersonate a token with a thread in your process.

First I will discuss how to create a new process under a different user context than your own, because it is the less sophisticated way of executing code on behalf of a user other than yourself. Then I will discuss impersonation at length.

To create a process using a token, you should call *CreateProcessAsUser*:

```
BOOL CreateProcessAsUser(
   HANDLE                hToken,
   PCTSTR                pszApplicationName,
   PTSTR                 pszCommandLine,
   PSECURITY_ATTRIBUTES  psaProcessAttributes,
   PSECURITY_ATTRIBUTES  psaThreadAttributes,
   BOOL                  fInheritHandles,
   DWORD                 dwCreationFlags,
   PVOID                 pEnvironment,
   PCTSTR                pszCurrentDirectory,
   PSTARTUPINFO          pStartupInfo,
   PPROCESS_INFORMATION  pProcessInformation);
```

CreateProcessAsUser is parameter-for-parameter identical to *CreateProcess* except for the first parameter, *hToken*, which accepts a primary token for the user context under which the new process will run. (Please refer to *Programming Applications for Microsoft Windows, Fourth Edition,* for an extensive discussion of *CreateProcess*.)

Now let's look at its differences. *CreateProcessAsUser* returns TRUE if the new process was created and

FALSE if the function failed. In three common cases *CreateProcessAsUser* might fail, where *CreateProcess* would not.

In the first case, the calling process might not have access to the executable file or directory of the executable file. A service running in the LocalSystem account is unlikely to run into this problem, although it is possible. To get around this problem, you must temporarily impersonate the user for whom the new process is being created by using the same token that is being passed to *CreateProcessAsUser*. This temporary impersonation must be done before making the call to *CreateProcessAsUser*. Naturally, if the new user does not have access to the executable or directory, the function fails.

In the second case, *CreateProcessAsUser* requires the calling process to have the SE_ASSIGNPRIMARYTOKEN_NAME and SE_INCREASE_QUOTA_NAME privileges granted to its token. The exception to this rule is the case in which the token being passed to *CreateProcessAsUser* is a restricted token created with the calling process's token. In this situation, the privilege SE_ASSIGNPRIMARYTOKEN_NAME is not required. (Using restricted tokens is a very powerful technique that is discussed in some detail later in this chapter.) The rationale here is that the restricted token will have less access to the system than the current process.

In the third scenario, *CreateProcessAsUser* fails when the token passed as the *hToken* parameter is not a primary token. If you hold a handle to an impersonation token, you can convert it to a primary token with a call to *DuplicateTokenEx*:

```
BOOL DuplicateTokenEx(
   HANDLE      hExistingToken,
   DWORD       dwDesiredAccess,
   PSECURITY_ATTRIBUTES pTokenAttributes,
   SECURITY_IMPERSONATION_LEVEL ImpersonationLevel,
   TOKEN_TYPE TokenType,
   PHANDLE     phNewToken);
```

The *hExistingToken* parameter is the token that you wish to duplicate. This token can be either a primary token or an impersonation token. The *dwDesiredAccess* parameter indicates the access that you want for the new token. You should ask only for the rights needed to accomplish the task at hand. See Table 11-2 for a list of access rights.

The *pTokenAttributes* parameter indicates the security descriptor and inheritance attributes of the new token. Do not confuse this parameter with the token's default DACL—this parameter sets the access control for the new object. (Security descriptors and access control are discussed at length in Chapter 10. The topic of token default DACLs is discussed earlier in this chapter as well as in Chapter 10.)

The *ImpersonationLevel* parameter indicates the new token's impersonation level and can be any of the values in Table 11-3.

Table 11-3. *Members of the SECURITY_IMPERSONATION_LEVEL enumerated type*

Value	Description
SecurityAnonymous	A token created with the SecurityAnonymous impersonation level cannot be used to create a process with impersonation.
SecurityIdentification	A token created with this impersonation level can be used only as a means of identification. The token's user and groups can be queried, but the token cannot be used with impersonation or in calls to *CreateProcessAsUser*.
SecurityImpersonation	This impersonation level creates a fully functional token that can be queried as well as used to execute code through impersonation. A token must be an impersonation token before a thread can impersonate that token.

SecurityDelegation	A token with the SecurityDelegation impersonation level can be used to access network resources. This is called *delegation* and allows a server to become a client on behalf of its client in a connection to a second server. Windows NT 4.0 and earlier versions did not support delegation. Windows 2000 does, but it requires that delegation be allowed on the server attempting to create a token of delegation type.

The *TokenType* can be either TokenPrimary or TokenImpersonation, both members of the TOKEN_TYPE enumeration. And the final parameter, *phNewToken*, receives the handle to the new token. Remember to call *CloseHandle* on the handle when you are finished with the object.

If *DuplicateTokenEx* returns FALSE, the function has failed. It usually fails because of insufficient access rights on the original token. (See Chapter 10 for a complete discussion of access control and access rights.)

NOTE

In previous versions of Windows, tokens created on a server machine for the purpose of acting on behalf of a client were called *network tokens* and typically had the SecurityImpersonation impersonation level. These tokens did not contain a copy of the trustee's credentials and could not be used to access resources outside of the machine that created the tokens.

However, with Windows 2000 the SecurityDelegation impersonation level is supported using the Kerberos security protocol (discussed in Chapter 12). Now it is possible to have a network token that does not limit access to the local machine.

Acquiring a Token Using *LogonUser*

You now know how to obtain a token from a process and how to create a process using an arbitrary token. However, we have not discussed how to retrieve a handle to a token when you have only a username and a password. You do this by calling *LogonUser*:

```
BOOL LogonUser(
   PTSTR   pszUsername,
   PTSTR   pszDomain,
   PTSTR   pszPassword,
   DWORD   dwLogonType,
   DWORD   dwLogonProvider,
   PHANDLE phToken);
```

The *pszUsername* and *pszDomain* parameters indicate the username and the domain of the user for which you wish to receive a token. You can pass the period character (".") for the *pszDomain* parameter, which searches only the local system's account database. If you pass NULL for the *pszDomain* parameter, the local system is searched for the user followed by the domains trusted by the system. The third parameter, *pszPassword*, is the password for this account.

The *dwLogonType* parameter of *LogonUser* describes to the system how the token will be used and what type of token you want to receive. Table 11-4 lists the values you can pass as the *dwLogonType* parameter.

Table 11-4. *Values you can pass for* LogonUser's dwLogonType *parameter*

Value	Description
LOGON32_LOGON_INTERACTIVE	Passing this value to *LogonUser* causes the

	system to check for the existence of the SE_INTERACTIVE_LOGON_NAME privilege in the account of the user for whom a token is being requested. *LogonUser* will fail if the user does not have this privilege assigned. Additionally, tokens received with this logon will be cached with the system. This means that the local system can lose connection with the authenticating machine but still make future successful calls to *LogonUser* using cached credentials. *LogonUser* will return a primary token.
LOGON32_LOGON_BATCH	Passing this value to *LogonUser* causes the system to check for the existence of the SE_BATCH_LOGON_NAME privilege in the account of the user for whom a token is being requested. *LogonUser* will fail if the user does not have this privilege assigned. Tokens received with this logon type are not cached, increasing the performance of *LogonUser* and making the logon type appropriate for high-performance servers. *LogonUser* will return a primary token.
LOGON32_LOGON_NETWORK	Passing this value to *LogonUser* causes the system to check for the existence of the SE_NETWORK_LOGON_NAME privilege in the account of the user for whom a token is being requested. *LogonUser* will fail if the user does not have this privilege assigned. Tokens received with this logon type are not cached. In addition, this token will be an impersonation token and a "network token."
LOGON32_LOGON_NETWORK_CLEARTEXT	Like the LOGON32_LOGON_NETWORK logon type, this logon type checks the account in question for the SE_NETWORK_LOGON_NAME account right. However, unlike LOGON32_LOGON_NETWORK, this logon type returns an impersonation token while *preserving* a copy of the trustee's credentials so that network access is possible using the resulting token. The token has to be duplicated to a primary token before it can be used in calls to *CreateProcessAsUser*.
LOGON32_LOGON_NEW_CREDENTIALS	This intriguing logon type is new in Windows 2000. It requires that you use LOGON32_PROVIDER_WINNT50 as the value for *dwLogonProvider* in calls to *LogonUser*. This logon type makes a copy of the calling thread's process token and adds a second identity to the token. This second identity will be the token's identity for all network access, where the token's

	identity for the local machine will remain the same as that of the original token. This makes the LOGON32_LOGON_NEW_CREDENTIALS unique in that it uses an existing token to build a new token with extra credentials. For an example of this logon type, see the RunAs.exe utility provided with Windows 2000. The "/NetOnly" switch uses the LOGON32_LOGON_NEW_CREDENTIALS logon type to create a token for the new process. The resulting token is a primary token.
LOGON32_LOGON_SERVICE	Passing this value to *LogonUser* causes the system to check for the existence of the SE_SERVICE_LOGON_NAME privilege in the account of the user for whom a token is being requested. *LogonUser* will fail if the user does not have this privilege assigned. This token will be cached for future calls to *LogonUser* if the machine loses connection to the authenticating agent. *LogonUser* returns a primary token.
LOGON32_LOGON_UNLOCK	This logon type is used by GINA to handle the unlocking of a workstation. If auditing is turned on for the system, calling *LogonUser* with this logon type creates an entry in the event log. The resulting token is a primary token.

As a developer or a network administrator, you can tighten your security and make systems more secure against misuse by creating user accounts that can be used only with the LOGON32_LOGON_SERVICE or LOGON32_LOGON_INTERACTIVE type. For example, you could grant *only* the SE_SERVICE_LOGON_NAME or the SE_INTERACTIVE_LOGON_NAME privilege, respectively.

NOTE

Although logon types such as LOGON32_LOGON_SERVICE and LOGON32_LOGON_INTERACTIVE require different privileges, they do not actually produce drastically different tokens. (The major difference among the tokens is the existence of a SID in the token groups, which indicates what type of logon was used. The logon source will also differ.) Requiring different privileges allows the system to provide an extra level of authentication without complicating the architecture of a token.

A token created with the LOGON32_LOGON_BATCH or LOGON32_LOGON_SERVICE can still run a process that interacts with the user, creating windows and other GUI objects.

However, when the system creates a process to run as a service, the token it uses is logged on using the LOGON32_LOGON_SERVICE logon type. And when the system logs a user on interactively, it uses the LOGON32_LOGON_INTERACTIVE logon type.

The value you pass for *dwLogonProvider* decides which method is used to authenticate the credentials passed to *LogonUser*. You should pass LOGON32_PROVIDER_DEFAULT for all uses of *LogonUser*, unless you are using the LOGON32_LOGON_NEW_CREDENTIALS logon type. In this case you should pass the value of LOGON32_PROVIDER_WINNT50 as the *dwLogonProvider* parameter.

You receive a handle to your new token via the *phToken* parameter of *LogonUser*.

The *LogonUser* function will return TRUE if the system has succeeded in authenticating the credentials passed, and FALSE if it has failed. A call to *GetLastError* will return the reason for failure.Three common errors that result when *LogonUser* fails are ERROR_LOGON_FAILURE, ERROR_LOGON_TYPE_NOT_GRANTED, and ERROR_PRIVILEGE_NOT_HELD. When *GetLastError* returns ERROR_LOGON_FAILURE, the credentials passed to *LogonUser* are not recognized. If *GetLastError* returns ERROR_LOGON_TYPE_NOT_GRANTED, the requested account does not have the proper account right for the requested logon type. The error code ERROR_PRIVILEGE_NOT_HELD suggests that the process calling *LogonUser* does not have the SE_TCB_NAME privilege granted. By default the LocalSystem account has this privilege granted. No other trustee in the system has this privilege by default.

When you are finished with the token, call *CloseHandle* on the object.

When you call *LogonUser*, you are asking the system to build a token for you. You can use this token in calls to *CreateProcessAsUser* and in impersonation, which is the subject of the next section.

Impersonation

When you are writing server software in Windows, you have the option of creating a process, on behalf of your client, to do the bidding of your client. However, creating a process for each client simply isn't a scalable technique. Impersonation is the solution to this problem, because it lets a single thread act under the security context of your client for an arbitrary length of time and revert back to the process's security context when it has finished. Impersonation can be done in a manner that is exceptionally efficient and scalable.

I love impersonation—the system handles the details for you, and it is implemented simply by making a call to *ImpersonateLoggedOnUser*. I think this is really cool because it ensures the thread is acting under the client's security context. What better way to ensure that you don't accidentally allow a client to cause a service to misuse the awesome power granted to it via the LocalSystem account? Here is *ImpersonateLoggedOnUser*:

```
BOOL ImpersonateLoggedOnUser(
   HANDLE hToken);
```

If the token passed as the *hToken* parameter is of type TokenPrimary, *ImpersonateLoggedOnUser* creates a duplicate token of type TokenImpersonation and assigns it to the calling thread. If the original token is already an impersonation token, the token is directly assigned to the thread. In the first case, the calling thread must have TOKEN_QUERY and TOKEN_DUPLICATE access to the token. In the second case, only TOKEN_QUERY is required.

If *ImpersonateLoggedOnUser* succeeds, it returns TRUE. Otherwise your service should call *GetLastError* to find the reason for failure.

The token passed to *ImpersonateLoggedOnUser* can be a token received via a call to *LogonUser*, *DuplicateTokenEx*, *OpenThreadToken*, or any of the other similar functions we have discussed so far. It can also be a token retrieved by some other means (of which I will discuss a few shortly). Upon success, most securable functions called by the impersonating thread will recognize the new security represented by the token.

NOTE

If a thread currently in the impersonation state calls *CreateThread* to create another thread, the new thread will not be impersonating. Said another way, all threads created using *CreateThread* use the process's token for securable activity, unless the created thread

explicitly impersonates a token.

This can cause some pretty tough-to-find bugs, because your code can impersonate a user and then call a function that creates a new thread to perform work. This new thread will not represent the creating thread's security context. In the case of service development, this more commonly grants more access, not less access, to the thread than was intended and can create a security "hole" in your service.

When your thread has finished acting on behalf of your client, the thread can return to using the process s token by making a call to RevertToSelf:

```
BOOL RevertToSelf(VOID);
```

NOTE

To improve performance when impersonating, it is preferable to avoid obtaining a token more often than necessary regardless of whether you are calling *LogonUser*, *OpenProcessToken*, or some other function to get the token you are using for impersonation. Typically your service retrieves or builds a handle once, saving the handle to the token in the state data for the connection. The service can then call *ImpersonateLoggedOnUser* and *RevertToSelf* as needed using the stored handle, closing the handle only when the connection has been terminated and the token is no longer necessary.

Now your service can act on behalf of clients connecting via any communication mechanism, so long as the client is able to pass its credentials to the service. Alternatively, your service could store a set of preconfigured credentials that it uses for its various client accounts. Depending on the needs of your service, this approach can be very effective. In many cases, however, a more seamless approach is desirable.

Impersonating a Connected Client

Windows offers a truly seamless form of impersonation that does not require your service to acquire a set of credentials. This form of impersonation is connection-oriented, but it is otherwise similar to the impersonation technique that we already discussed. If a trusted authority has authenticated the client and the communication medium is one that supports impersonation, your service can impersonate the client automatically!

Table 11-5 lists connection methods supported by impersonation as well as the functions used to initiate the impersonation and revert to the process's token.

To discuss each of these forms of impersonation, I would need to digress to the topic of network communication, which is worthy of an entire book. However, the RoboService sample application in the previous chapter fully implements impersonation using named pipes, and the SSPIChat sample application (in Chapter 12) implements impersonation using the Security Support Provider Interface (SSPI). (SSPI is also discussed in Chapter 12.)

Two of the functions mentioned in Table 115 warrant further mention. They are *ImpersonateSelf* and *SetThreadToken*:

```
BOOL ImpersonateSelf(
   SECURITY_IMPERSONATION_LEVEL ImpersonationLevel);
```

The *ImpersonateSelf* function duplicates your process's token, creating a token of type TokenImpersonation, and assigns the token to the calling thread.

In practice you won't commonly use *ImpersonateSelf* to adjust the impersonation level of the token. Rather, you'll use it to create an impersonation so that adjustments to the thread, such as enabling and disabling privileges or disabling and enabling groups in the token, affect only a single thread rather than every thread in the process. The *RevertToSelf* function ends the impersonation.

Table 11-5. *Connection methods supported by impersonation*

Connection Method	Impersonating Functions

Connectionless	`BOOL ImpersonateLoggedOnUser( HANDLE hToken);`
	or
	`BOOL ImpersonateSelf( SECURITY_IMPERSONATION_LEVEL ImpersonationLevel);`
	and
	`BOOL RevertToSelf(VOID);`
	and/or
	`BOOL SetThreadToken( PHANDLE Thread, HANDLE Token);`

Named pipes	`BOOL ImpersonateNamedPipeClient( HANDLE hNamedPipe);`
	and
	`BOOL RevertToSelf(VOID);`

Dynamic data exchange (DDE)	`BOOL DdeImpersonateClient( HCONV hConv);`
	or
	`BOOL ImpersonateDdeClientWindow( HWND hWndClient, HWND hWndServer);`
	and
	`BOOL RevertToSelf(VOID);`

Remote procedure calls (RPC)	`RPC_STATUS RPC_ENTRY RpcImpersonateClient( RPC_BINDING_HANDLE BindingHandle);`
	and
	`RPC_STATUS RPC_ENTRY RpcRevertToSelfEx( RPC_BINDING_HANDLE BindingHandle);`

Sockets or any other transport mechanism (via SSPI)&"temp0089.html">Chapter 12	`SECURITY_STATUS ImpersonateSecurityContext( PCtxtHandle phContext );` and

```
SECURITY_STATUS RevertSecurityContext(
  PCtxtHandle phContext);
```

The *SetThreadToken* function, shown next, allows you to arbitrarily choose an impersonation token for any thread by using the thread's handle. The *pThread* parameter is a pointer to a handle to the thread that you will adjust, where passing NULL indicates the current thread. The *hToken* parameter indicates the token to be used for impersonation, where a NULL value will cause the thread to revert to the process-level token.

```
BOOL SetThreadToken(
   PHANDLE pThread,
   HANDLE  hToken);
```

At first glance, the *SetThreadToken* function might not seem necessary. Why not just use the impersonation function appropriate for your communication mechanism? The answer is the one missing feature with impersonation—the notion of an *impersonation stack*. Let me explain.

If you were to call *ImpersonateLoggedOnUser* with token A, and then call a function that calls *ImpersonateLoggedOnUser* with token B, when the function called *RevertToSelf*, the system would revert the thread's token all the way back to the process token. The system has no memory of the thread's association with token A. Normally you have control of your code and can avoid situations like this. But here are two noteworthy exceptions:

- Large projects, where the state before entry into your function is unsure or inconsistent
- DLL projects that are used as extensions to other software, such as ISAPI extensions to Microsoft Internet Information Services

To fake an impersonation stack, your function would have to retrieve a handle to its current token via a call to *OpenThreadToken* (which we have discussed), store it (most likely on the stack), and then call the appropriate impersonation function. Rather than calling the complementary "revert" function, your code would use *SetThreadToken* to restore the stored token. This way your function gives the thread back to the calling code the way it found it.

NOTE

In a high-performance server environment, the overhead of communicating impersonation information over the network is undesirable. In many cases, you can achieve the best performance by calling the appropriate impersonation function upon connection and then making a call to *OpenThreadToken* to retrieve and store a handle to the impersonating thread token. Thereafter you can use *SetThreadToken* or *ImpersonateLoggedOnUser* to impersonate the stored token, as threads in your service fulfill further requests for that client.

[Previous] [Next]

Restricted Tokens

As I mentioned earlier, it is impossible for your service or any application to build a token from scratch—the system must build it for you. But Windows will allow you to create a new token, based on an existing token, with some additional restrictions. This new token is known as a *restricted token*. The restricted token can help your software meet some complex security needs with a very clean solution.

When you create a restricted token, you are not adding restrictions to an existing token. Instead you are using a token as a kind of template to create a new token, although this new token has additional restrictions. There are three restrictions that you can place on your token. Each restriction is optional, and you can choose any

combination of them in the creation of your new token. Two of the restrictions rely on knowledge of access control, which I covered in detail in Chapter 10. You can perform any combination of the following actions on a token to create a new restricted token:

- Delete privileges
- Disable token SIDs for trustee accounts
- Add "restricted SIDs" of trustee accounts

Deleting Privileges

When creating a restricted token, you can choose a set of privileges that you do *not* want granted to the new token, which is not the same as disabling privileges. Rather, you are explicitly removing a privilege from the new token. There might be situations in which you want to use an existing security context but delete certain privileges. For example, you might use your own security context or one received via impersonation and remove the SE_SHUTDOWN_NAME or SE_TCB_NAME privileges.

Disabling Token SIDs

When creating a restricted token, you can select which of the trustees in the existing token will be disabled. The disabled trustees can be any combination of the token user and the groups of which the user is a member. Any trustees whose SIDs are selected to be added to the disabled list will be used for denied access only, when access checks are performed.

For example, suppose the token user's membership in the TEMP USERS group awarded the user access to the C:\Temp directory while denying the user access to the C:\Permanent directory. If the SID for the TEMP USERS group was selected to be a disabled SID in a restricted token, membership in the TEMP USERS group would no longer allow access to the C:\Temp directory, but it would still deny access to the C:\Permanent directory.

You can disable only those SIDs that already exist in the source token, and disabling trustees in a token does not in any way affect the identity of the token. Disabling the token user, for example, only affects access to objects. The token still identifies that particular user.

Adding Restricted SIDs

In addition to disabling existing SIDs, you can create a set of trustees, or SIDs, known as *restricted SIDs*, to dynamically add to your new token. You might be thinking that the ability to dynamically add a trustee to a token is awfully powerful. But the catch is that the new SIDs are used as a cross-check for access to a securable object. Not only does an access check have to clear the token's "natural" trustees before access can be granted, but it also has to clear the new set of restricted trustees.

Using restricted SIDs is similar to creating a second token and making sure that both tokens have access to a securable object before performing an action on the object. But with restricted tokens, this second check is handled automatically for you by the system. Restricted SIDs change access checks somewhat in that they must actually be performed twice: once for the natural SIDs in the token and once for the restricted SIDs. Only when both succeed is the access check granted. (See Chapter 10 for a detailed discussion of access checks.)

Now you are familiar with the restrictions you can apply to your new restricted copy of a token. Let's look at the function that creates the token, *CreateRestrictedToken*:

```
BOOL CreateRestrictedToken(
   HANDLE              hExistingTokenHandle,
   DWORD               dwFlags,
   DWORD               dwDisableSidCount,
   PSID_AND_ATTRIBUTES  pSidsToDisable,
   DWORD               dwDeletePrivilegeCount,
   PLUID_AND_ATTRIBUTES pPrivilegesToDelete,
   DWORD               dwRestrictedSidCount,
   PSID_AND_ATTRIBUTES  pSidsToRestrict,
   PHANDLE             phNewTokenHandle);
```

The *hExistingTokenHandle* parameter is the source token from which the new token will be created. *CreateRestrictedToken* creates a new token based on the token whose handle is passed as the *hExistingTokenHandle* parameter. The handle must have TOKEN_DUPLICATE access. Any token is legal as the source token for a restricted token with the exception of a token that already has a list of restricting SIDs. You can ascertain whether a token contains restricted SIDs by passing its handle to the *IsTokenRestricted* function:

```
BOOL IsTokenRestricted(
   HANDLE TokenHandle);
```

You can pass DISABLE_MAX_PRIVILEGE as the *dwFlags* parameter if you want the new token to allow only functionality that requires no privileges. You can pass zero if you want to delete only a subset of the privileges. Passing zero is the more common case.

The *dwDisableSidCount* parameter indicates the number of trustees you want to ensure are disabled in the new token. The next parameter, *pSidsToDisable*, points to an array of SID_AND_ATTRIBUTES structures indicating the groups (as well as perhaps the token user) to disable in the new token. The *dwDisableSidCount* parameter refers to the number of entries passed as this parameter. Disabled SIDs are characterized by the existence of the SE_GROUP_USE_FOR_DENY_ONLY attributes information for the SID in the new token.

NOTE

You can pass a superset of the trustees represented in the source token as the *pSidsToDisable* parameter. The system will ignore any SIDs not in the source token. In this way you can generically apply a single list of disabled SIDs to more than one token with different underlying token users and groups.

The SID_AND_ATTRIBUTES structure is defined as follows:

```
typedef struct _SID_AND_ATTRIBUTES {
   PSID  Sid;
   DWORD Attributes;
} SID_AND_ATTRIBUTES ;
```

The *Attributes* member of the structure is ignored by *CreateRestrictedToken*. Other functions such as *GetTokenInformation* use the *Attributes* member to report attributes of a SID such as SE_GROUP_USE_FOR_DENY_ONLY and SE_GROUP_MANDATORY. The *Sid* member of SID_AND_ATTRIBUTES points to a SID structure indicating the trustee that you wish to disable in the new token.

Your server should pass a fully populated array of SID_AND_ATTRIBUTES structures as the *pSidsToDisable* parameter of *CreateRestrictedToken*. If you do not want to disable any trustees in your new token, you should pass zero for the *dwDisableSidCount* and NULL for the *pSidsToDisable* parameter.

The *dwDeletePrivilegeCount* and *pPrivilegesToDelete* parameters of *CreateRestrictedToken* work similarly to the "disabled SID" parameters, except that *pPrivilegesToDelete* points to an array of LUID_AND_ATTRIBUTES structures. As with disabled SIDs, passing zero and NULL for *dwDeletePrivilegeCount* and *pPrivilegesToDelete* is appropriate when you don't want to delete any privileges. For a detailed discussion of the LUID_AND_ATTRIBUTES structure, refer to the section "Adjusting a Token's Privileges" earlier in this chapter.

The *dwRestrictedSidCount* and *pSidsToRestrict* parameters represent the count and the array of the trustees that you want to add as "restricted SIDs" to the new token, respectively. You should use the same rules applied to *dwDisableSidCount* and *pSidsToDisable* to build your restricted trustee list. Remember that you are not required to include any restricted SIDs in your new token. If no restricted SIDs are desired, pass zero for the *dwRestrictedSidCount* parameter.

The difference between the disabled SIDs array that you pass to *CreateRestrictedToken* and the restricted SIDs array is that with restricted SIDs the function does not ignore trustees that do not exist in the original token. In fact, you will often include trustees not included in the original token to further restrict access checks.

The last parameter of *CreateRestrictedToken*, named *phNewTokenHandle*, points to a HANDLE variable, which receives a handle to the new, restricted token.

CreateRestrictedToken is a powerful function allowing you flexible restriction of existing tokens. Consider, for example, the following scenario, which illustrates this flexibility. Imagine that you wanted to secure a group of objects for which you explicitly allow or deny users certain access, in a somewhat unorthodox manner. You want to restrict access to some objects only on Tuesdays, regardless of what the typical access is for this object. You can take the following steps to implement this functionality without undermining the typical non-Tuesday access to the objects. Here are the administrative tasks:

1. Create a user account named Tuesday for the sole purpose of adding restrictions to securable objects.

2. Modify the objects' DACLs to include the restrictions desired on Tuesdays. Assign these access-denied ACEs to the SID of the Tuesday account.

Here are the server tasks:

1. When a user connects to your server, use the *GetSystemTime* function to determine whether it is Tuesday.

2. If it is Tuesday, rather than use the impersonation token for the user, create a restricted token before executing code on behalf of the client.

3. Build a list of restricting SIDs that matches the groups in the source token, and include the user SID for the token. Additionally include in the list of restricting SIDs an entry for the Tuesday trustee.

4. Impersonate the new restricted token using *ImpersonateLoggedOnUser*.

Using this process you can easily enforce additional restrictions without requiring a change to the access control of all of your secured objects every Tuesday.

The TokenMaster sample application allows you to retrieve a token and then create a restricted version of the token. You can use this restricted version to launch applications. This can be a very useful tool in furthering your understanding of restricted tokens.

After reading this chapter, you are familiar with tools and techniques that allow your software to maintain and

manage a sense of identity in a flexible manner that is consistent with the Windows access control model. Your software will greatly benefit from your understanding of the token as the system's way of enforcing user context. Your software will also greatly benefit from your knowledge of impersonation, which allows you to act on behalf of your client. (Chapter 12 takes this topic to the next level with the SSPI.) The restricted token is another powerful feature you can use in secure projects that require a great deal of flexibility in their access control.

The better you understand the features discussed in this chapter, as well as the capabilities and features covered in the last two chapters, the better you will be able to design software that makes security work for you.

[Previous] [Next]

Chapter 12

Secure Connectivity

The ability to communicate in a secure manner is absolutely essential to server software. No matter how much time and effort you spend applying access control and security to the objects on your system, your investment is virtually worthless if your server is not constantly sure of two things:

- Your server must be sure who its client is. That is to say, your server must be able to authenticate the party with which it is communicating.
- Your server must be able to ensure that the information passing between itself and its client has not been modified (or perhaps even viewed) by a third party.

Server software usually has many rights on the system that hosts it—if your server can be sure of these two things, you can be assured that its power is not being abused and the system's security is in tact.

The challenge for the developer of a service running on Microsoft Windows 2000 is to authenticate clients and enable the service to communicate securely with these clients in a manner that integrates well with the Windows security model. Meeting this challenge is the focus of this chapter, but first I will be covering some history and defining some terms.

[Previous] [Next]

Encryption

Encryption forms the foundation of secure communication. Encryption is a huge topic, and neither this chapter nor any single book can completely cover it. The goals of this section are to provide some background on the topic and explain some key concepts and important terms.

Two or more communicating parties can use encryption to share information securely. In general terms, this is accomplished when one party modifies data in such a way that the data cannot feasibly be restored to its original state without the use of a *key*. A key is a digital value that is used by an encryption algorithm to encrypt data. A key is also a digital value that is used to decrypt data. The modification of data is *encryption*. The receiving party is able to restore the data (and therefore understand it) using the key, a process known as *decryption*. The study of data encryption and decryption is called *cryptography*.

Typically cryptography is used to ensure one or more of the three goals are met over insecure communication medium:

- **Privacy** Nobody but the intended party can understand the data being communicated.
- **Authentication** You have established with whom you are communicating.
- **Integrity** The data you have received has not been modified by a third party in transit.

Symmetric Key Encryption

Commonly, when people speak of encrypted communication, they are referring to a conversation between two parties that share a single secret key. The same key is used for both encryption and decryption of data. This is called *symmetric key encryption*.

Symmetric key encryption is appropriate only in environments where it is reasonable for two parties to share a secret. As this chapter unfolds, you will see that both the Microsoft Windows NT LAN Manager (NTLM) and Kerberos security protocols used by Windows 2000 (discussed later in this chapter) make use of symmetric key encryption.

There are, however, scenarios in which symmetric key encryption falls short. Here are some reasons:

- Two parties must hold symmetric keys, requiring the parties to communicate these keys. This communication of keys can become a security risk.
- Both parties must be trusted to use the key. What one party can do with a key, another party can do with the key. This is a concern when a known principal is communicating with an unknown principal.

These points might seem to create insurmountable problems in an environment in which many parties will need to communicate with one another in a secure manner, such as in an enterprise LAN. However, in an environment with known entities, such as a corporate network or a Windows domain, symmetric key encryption can be used to create very elegant and secure communication solutions. You will see an example of this in my discussion on Kerberos later in this chapter.

Asymmetric, or Public Key, Encryption

The Internet has created an environment in which communication must sometimes be secure, even when one or more of the communicating principals have no previous knowledge of one another. Clearly, in an environment such as the Internet, it is infeasible for each party to share a secret key. This is where *asymmetric,* or *public key,* encryption shines.

Public key encryption makes use of two keys: a *public key* that is intended to be shared, and a *private key* that must be kept secret. Data encrypted with the public key in a pair can only be decrypted using the private key. Conversely, data encrypted with the private key can only be decrypted using the public key.

You can't derive the private key in a pair from the public key. However, using brute-force techniques you can enumerate all possible private keys until a match is found for a public key. This approach, however, is computationally infeasible and becomes exponentially more difficult as the width of the keys increases.

As you can see, a system like this has a great deal of potential. It is now possible for you to publish your public key so that an agent who wishes to communicate with you can encrypt his data with your public key,

knowing that only you can read the data because only you hold the private key.

Unfortunately, algorithms to encrypt and decrypt data using asymmetric (public/private) key pairs are very slow in comparison to algorithms using symmetric key encryption. Because of this, encrypting large amounts of data using asymmetric key encryption is usually unreasonable.

To get around this problem, many protocols make use of public key encryption to communicate a symmetric key. Then symmetric key encryption is used during the remainder of the conversation, or "session." This way, the advantages of public key encryption can be realized while enjoying the efficiency of symmetric key encryption.

Digital Certificates

Digital certificates, or *certificates,* are a means of packaging or publishing the public key in an asymmetric key pair. Digital certificates can also contain additional information such as the owner of the key pair and the allowed usage of the certificate.

> **NOTE**
>
> The private key in an asymmetric key pair is never packaged in a certificate, because it should never be published. It is tightly held by the owning entity.

If you are able to trust the validity of a certificate, you can use the public key held within the certificate to decrypt data from the owner of the certificate. In this way, you can trust the data and the source of the data. Similarly, if you trust the validity of the certificate, you can encrypt data using the public key found in the certificate, knowing that only the rightful owner of the certificate can decrypt the data. This type of technique is used when you communicate credit card information using your Web browser on a secure commercial Web site such as Amazon.com or Ebay.com.

A great deal is riding on your software's ability to trust the validity of a certificate, and therefore to trust the validity of a public key. This is where an entity known as a *certificate authority,* or CA, steps in.

Certificates are issued by a CA, which signs the certificate with its own public key. By signing the certificate, the CA promises that the information held in the certificate, including the public key, is correct. If you hold the public key of the CA (which I will discuss in more detail in a moment), you can assure that the signature of the certificate is intact, at least as far as this CA is concerned.

The entity you are trusting is the certificate authority, which issued the certificate in question. A client trusts a certificate authority by holding a copy of the CA's public key and using it to verify the signature of certificates that it receives. Once the client verifies the signature, it can choose to trust the information found in the certificate based on its trust relationship with the CA.

Examples of two certificate authorities include VeriSign, which can be found at www.Verisign.com, and Thawte, which can be found at www.Thawte.com. There are dozens more.

Depending on your needs, you might find it to your advantage to set up your own CA using Microsoft Certificate Services or similar software. Doing so will allow you to create and distribute certificates for use in your own enterprise or Internet environment, potentially increasing your flexibility with certificates and saving you money on certificate licensing fees.

I can't include a truly comprehensive discussion of certificates in this chapter, but it is an important technology to be aware of. Certificates will come up again later in this chapter when we discuss the Secure Sockets Layer (SSL) security protocol. For a more comprehensive discussion on the topic, I suggest you

consult the Platform SDK documentation or research the topic using the World Wide Web.

[Previous] [Next]

Security Protocols

You probably won't be surprised to find out that protocols have been defined to initiate and implement secure communication. Different protocols can be used to meet different needs, and we will discuss three such protocols in this chapter: Kerberos, NTLM, and SSL. I selected these protocols because of the exceptional support Windows 2000 provides for them and their significance in existing environments such as the Internet.

Security protocols are primarily designed to use encryption techniques as well as network communication techniques, in a defined manner, with the objective of implementing one or more of the three goals mentioned earlier in this chapter: privacy, authentication, and integrity. All three protocols that we will cover achieve these goals to varying degrees. Additional features of some of the protocols are delegation and mutual authentication.

The Kerberos and NTLM security protocols are an integral part of security in Windows 2000. As such, these protocols can be used to directly authenticate a user in a domain. Said another way, a server can retrieve an access token for a client by using the Kerberos and NTLM security protocols. (Tokens are discussed in detail in Chapter 11.)

SSL is a security protocol that is not as tightly integrated with the Windows security model as are the other protocols. SSL is a certificate-based protocol, and Windows must do a mapping of certificate information before a server can impersonate. (Impersonation is discussed in detail in Chapter 11.)

NTLM

NTLM is the existing security protocol of the Windows family. It is understood by Windows 2000, Windows NT, and Windows 95/98. Although Kerberos offers some noteworthy improvements over NTLM, you need to understand NTLM.

The NTLM protocol is fairly straightforward and is based on what is known as a challenge/response sequence. The sequence, illustrated in Figure 12-1, works like this:

- **Step 1** The client sends its username and its domain name to the server.
- **Step 2** The server forwards this information to the domain controller (DC).
- **Step 3** The DC creates a *challenge,* which is randomly generated and created with the client's password (known only by the client and the DC).
- **Step 4** The DC sends the challenge to the server.
- **Step 5** The server forwards the challenge to the client.
- **Step 6** The client examines the challenge using its password, and performs a *known modification* to the challenge, creating a *response*. The known modification is defined by the NTLM protocol.
- **Step 7** The client sends the response back to the server.

- **Step 8** The server forwards the response to the DC.
- **Step 9** The DC examines the response and verifies that it is the original challenge modified using the known modification. The client is now authenticated.
- **Step 10** The DC informs the server that the client is authenticated.

Figure 12-1. *NTLM authentication*

As you can see from Figure 12-1, the network traffic generated by the NTLM protocol is significant, and the server bears a great deal of the responsibility even though it is largely acting as a proxy between the client and the domain controller (DC). But this protocol does get the job done.

Kerberos

As I mentioned in the last section, NTLM is the only security protocol supported on all of Microsoft's desktop operating systems. For this reason, it still plays a very important role in server development in an enterprise. However, Kerberos is the shining star of Windows 2000 and offers a lot of great features:

- **Mutual authentication** Kerberos offers a mechanism by which a client can be sure of the server's identity, in addition to the more common client authentication by the service. This authentication is achieved efficiently by Kerberos without requiring a second authentication by the client.
- **Delegation** With Kerberos, a server can use its client's credentials to contact a second-tier server on behalf of the client. Through delegation, the second-tier server authenticates the server as though its user context were that of the client's.
- **Efficient authentication** Kerberos authentication requires minimal communication legs in its journey, and only a single leg of communication must be performed by the server (or two communication legs, if the client is mutually authenticating). This efficiency removes a great deal of the load from the server, shifting the responsibility to the client.
- **Kerberos is a standard** The Kerberos authentication system was designed at MIT as part of project Athena. Windows 2000 implements Kerberos V5.0 as defined in RFC 1510. As such, Windows 2000 can interoperate with other operating systems that are capable of communicating with Kerberos.

Kerberos is the name of the three-headed dog in Greek mythology that guarded the entrance to the underworld. The Kerberos protocol was named after this mythical animal because it too has three components:

- **Client** This is the entity or principal that is initiating an authenticated conversation.
- **Server** This is the entity or principal with which the client wishes to communicate.
- **Key Distribution Center (KDC)** An entity, trusted by both the client and the server, that manages authentication for them.

Each of these entities plays an important role in the Kerberos specification. The following description of Kerberos will define some important terms and concepts for you, but you don't need a complete understanding of Kerberos before you can use it in your applications.

Kerberos Authentication

Kerberos relies on symmetric key encryption. A server using Kerberos to authenticate a client decrypts a packet known as an *authenticator*, which is sent by the client and contains an encrypted copy of the client's identification and a timestamp (which is used to avoid authenticator duplication and reuse across the wire). Only the client and the server can encrypt or decrypt the authenticator with this key, which is guaranteed to be a secret known only by them. This secret key is a *session key*. This session key is also the key that the client and server use to encrypt data as they continue to communicate for the duration of the *session*.

Kerberos authentication is pretty simple so far, but as you can imagine, the real magic of Kerberos is in the negotiation of the session key. The session key must be unique and is a secret known only by the communicating entities—the client and the server. The open questions are as follows:

- Who creates the session key?
- How is it communicated to the client and the server in a secure manner?

I will go through the process that answers these questions one step at a time, but first it is important that you understand the Kerberos protocol's concept of a ticket.

The ticket A *ticket* is a packet of information whose contents are held by a client, and that contains information that allows a server to authenticate the client. This packet was given the name *ticket* because its information is the client's "ticket" to "use" the server. Here are some important items contained within a ticket:

- The identification of the client holding the ticket. This is username and domain information for Windows 2000.
- A session key that is encrypted in the server's password. This is the secret key that is used to encrypt and decrypt any data communicated between the client and the server.
- A creation time and an expiration time.
- Additional information and flags.

Tickets are generated by the Key Distribution Center or KDC. The secret session key found inside the ticket is encrypted with the server's password. Only the KDC and the server know this password. Although the KDC grants tickets to clients, the encrypted portions of a ticket are opaque to a client and will be used only by the requested server.

The client requests tickets for use with specific servers, and the KDC grants these tickets. A ticket can be used and reused for authenticated communication with a server until the expiration time has passed, at which time the client must be granted a new ticket from the KDC.

Because the session key is encrypted using the server's password, the key can be used only by the server. Additionally, encrypted and clear-text versions of the ticket holder's identification are included on the ticket. This allows anybody to know the holder of the ticket, but the encrypted copy allows the server to ensure that the ticket holder has not been changed since the creation of the ticket.

The duration of a ticket's usefulness defines a session with a server. This session continues until the ticket expires or until the client discards the ticket. The server does not store the ticket, so authentication (from the server's point of view) is stateless.

Now I would like to describe a very important session. This is the client's session with the KDC. This is also the first of three phases that a client must go through to authenticate itself with a server. Figure 12-2 shows the entire authentication process using the Kerberos protocol from start to finish.

Figure 12-2. *Kerberos protocol authentication process*

The ticket-granting ticket (First Phase) As I mentioned before, the KDC creates tickets and communicates them to a requesting client. The KDC itself is a server, however, and the client must maintain a ticket for use with the KDC. Because the client will be using the KDC to retrieve tickets for use with other servers in the system, the ticket that the client holds for use with the KDC is called a *ticket-granting ticket,* or TGT. This is the very first ticket that a client requests from the KDC, and it defines a session with the KDC. Here are the steps in acquiring a TGT:

- **Step 1** The client sends a request for a ticket-granting ticket to the KDC. (Microsoft has defined extra data that accompanies this request to preauthenticate the user to the KDC. This information is not part of the Kerberos definition, and it is not important for our discussion.)

- **Step 2** The KDC sends a ticket-granting ticket to the client along with a secret key encrypted with the client's password. This secret key is a session key for future communication with the KDC.

  Now the client holds a session key and a ticket that can be used with the KDC in requests for future tickets. The session key returned to the client is encrypted with the client's password. If the client has lied about its identity, it will not be able to decrypt the session key, and the ticket-granting ticket will be useless to it. This is because the session key is used in all future communication with the KDC.

The ticket-granting ticket, like all tickets, has a session key encrypted with the server's password and expiration information. As long as the client holds an unexpired TGT for the KDC, it is participating in a session with the server.

Requesting a ticket (Second Phase) When a client wishes to initiate an authenticated session with a server, it must first request a ticket for this session from the KDC. Before it can do this, it must hold a ticket-granting ticket for use with the KDC. This was established in the first phase. Here are the steps the client takes to request a ticket for use with a server:

- **Step 1** The client sends a request to the KDC for a ticket. The request includes the following information:
  - An *authenticator* (the client's identity and a timestamp), which is encrypted with the session key that the client holds for communication with the KDC
  - The ticket-granting ticket
  - The server for which the client is requesting a ticket
- **Step 2** The KDC opens the TGT by decrypting the session key held within (using its own secret password). The KDC decrypts the authenticator with this session key. If the decryption succeeds, the client is authenticated to the KDC, and the KDC builds the requested ticket for the client.
- **Step 3** The KDC sends the requested ticket back to the client. (Remember that the ticket contains a copy of the session key encrypted in the server's password known only by the KDC and the server.) The KDC also communicates the session key (for use with the server) to the client, encrypted in the secret session key associated with the client's TGT.

  At this point, the client holds a ticket for authenticating with the requested server. The client also holds its own copy of the session key used when communicating with the server.

Authenticating with a server (Third Phase) Finally, the client begins its communication with the server, which can happen only if the client holds a ticket for authenticating with the server. The ticket is granted in the second phase. Here are the steps that the client takes to authenticate with a server:

- **Step 1** The client sends a request for authentication to the server. This request includes the following information:
  - An authenticator (the client's identity and a timestamp), which is encrypted with the secret key that the client holds for this session
  - The ticket that the client holds for the session
- **Step 2** The server opens the ticket by decrypting, using its secret password, the session key held within. The server then uses the session key to decrypt the authenticator included with the request. If the decryption works, the client has been authenticated to the server.
- **Step 3** If the client has requested mutual authentication, the server sends an authenticator to the client, encrypted using the session key. The server could not have encrypted the authenticator unless it held the session key, proving that the server is the requested server.

At this point, the client and the server can communicate using their shared session key to encrypt data as it crosses the network.

Like the KDC, the server does not save its ticket, and in this way uses what is known as stateless authentication. This means that the third phase is necessary each successive time the client wishes to communicate with the server.

Notice in Figure 12-2 that communication with the server does not begin until the third phase. Notice also that the server and the KDC never communicate. These are noteworthy scalability improvements over the authentication scheme presented by NTLM.

As I mentioned before, this coverage of the Kerberos protocol does not cover every detail of the protocol. However, this description provides some perspective, which you will find useful when we begin our discussion of programming using security protocols.

[Previous] [Next]

Windows 2000 Developer Services

Before discussing code, I would like to mention the different APIs, or function groups, that Windows 2000 provides to help the developer deal with cryptography, certificates, and security protocols.

CryptoAPI Overview

As you can imagine from our previous discussion on cryptography, software that uses encryption has a fair number of details to consider. At minimum it must be able to create and maintain keys. And it must be able to use these keys to encrypt and decrypt data.

If asymmetrical key encryption is needed, software must also be able to create and manage public/private key pairs. As mentioned earlier, certificates have become an important part of public key management, and software might also have to manage certificates.

Fortunately, the CryptoAPI (also referred to as CAPI) provides all the functionality needed to manage keys and certificates as well as to encrypt and decrypt data. The CryptoAPI is useful if you will be rolling your own security protocol for use with your client and server software.

Typically, you will want to use an existing security protocol, which, thanks to the Security Support Provider Interface (SSPI), you can use without worrying about the details of cryptography and key management.

NOTE

I will cover the CryptoAPI functions used for certificate management in the section "CryptoAPI" later in this chapter. For more information on the CryptoAPI, see the Platform SDK documentation.

Security Support Provider Interface (SSPI)

Windows is able to communicate using different security protocols such as NTLM and Kerberos, which we have already discussed. (I will discuss SSL later in this chapter.) The system's internal implementations of these protocols are in modular units known as Security Support Providers, or SSPs. (An individual SSP is sometimes called a *security package,* or *package*). This implementation allows the system to use multiple security protocols in a standardized way. It also allows the developers of Windows to create new SSPs and incorporate them into the system. (In fact, it is also possible for third-party vendors to create SSPs for use with

Windows.)

You might have guessed that the system also provides a programming interface for SSPs that allows your software to take advantage of a security protocol provided by the system. This interface is called the Security Support Provider Interface, or the SSPI. The SSPI is a very powerful set of functions. Here are some of the features it provides:

- Communication transport independence
- Common interface to multiple SSPs
- Authentication (including impersonation)
- Message privacy (encryption)
- Message integrity (signing)

Each of these features is managed by the SSPI functions, so your code doesn't need to deal with the details of encryption and implementing a security protocol. Your code can do what it is already good at, which is communicating information to and from its clients and performing tasks on their behalf.

Before discussing the protocols provided by the SSPI, I would like to say a little more about transport independence. The SSPI builds *security blobs* that must be sent from client to server and vice-versa. The SSPI will also take your software's data and modify it, creating encrypted or signed security blobs to be sent to and from server and client software. You send a blob returned by an SSPI function, the receiving end passes the blob to its SSPI function, and the function translates the information. The SSPI functions do not actually communicate this data—your software does.

The transport independence of the SSPI awards your software a great deal of flexibility. Because your software's job is to communicate data built by the SSPI, you can use any communication medium you want, including sockets, named pipes, NetBEUI, and IPX/SPX. In fact, one of the designers of the SSPI likes to point out that you could tape the security blobs returned by the SSPI functions to the backs of turtles, so long as a computer can read the blob and pass it back to the SSPI. (Imagine being the author of the first server to impersonate a client, both of which are communicating via a reptile!)

It is difficult to imagine a common client/server environment in which the SSPI won't meet your needs. The most likely scenario is one in which you are forced to adhere to an existing protocol that is not supported by the SSPI. In this case, you might be forced to use the CryptoAPI or some other set of functions to manage cryptography and other details on your own. However, if you have the luxury of designing your client and server from the ground up, or if you are integrating with software that is already using a protocol supported by the SSPI, you should use the SSPI in your code.

Table 12-1 shows the security protocols supported by the SSPI as of the writing of this book, including on which platforms each protocol is available and the special features of each protocol. This information can help you decide which protocol is best for you.

Table 12-1. *SSPI protocols*

Protocol	Supported Operating Systems	Description
Kerberos	Windows 2000	

		• Windows integrated authentication (that is, token retrieval and client impersonation). • Mutual authentication. • Delegation. • Message encryption and signing. • Efficiency and scalability.
NT LAN Manager (NTLM)	Windows 2000, Windows NT, Windows 95/98	• Windows integrated authentication (that is, token retrieval and client impersonation). • No mutual authentication. • No delegation. • Message encryption and signing. • Reduced efficiency and scalability as compared to Kerberos.
Secure Sockets Layer/Transport Layer Security (SSL/TLS)	Windows 2000, Windows NT	• Authentication using certificates and public key infrastructure (PKI). • Optional impersonation by mapping certificates to domain accounts. • Mutual authentication. • Message encryption and signing. • Good scalability.
Negotiate	Windows 2000, Windows NT, Windows 95/98	• A special protocol that allows the client and server to negotiate the most appropriate protocol to use. The client should never use Negotiate but should select its best supported protocol. The server can use the Negotiate protocol, regardless of the protocol selected by the client.

As you can see from Table 12-1, you have several choices to make when developing a secure server. For enterprise software, you will most likely choose to use Kerberos, NTLM, or Negotiate to communicate between client and server software. For Internet server software, there are some compelling reasons to use SSL, and I will discuss those later.

[Previous] [Next]

Programming for Secure Connectivity

The SSPI provides a common set of functions that can be used to authenticate clients and servers as well as ensure data privacy and integrity in your software's communication. The SSPI was designed to create a common interface for multiple SSPs supported by Windows, and to a great extent it does this.

This section discusses how to use the SSPI to initiate a conversation between a client and a server by using the Kerberos and NTLM protocols. The SSPI is designed in such a way that the same code, with only minor modifications, can be used to take advantage of Kerberos and NTLM. Because of the variety of security protocols, some SSPs will cause your SSPI code to vary greatly from the SSPI code for other protocols. SSL is an example of such an SSP. Because of this, I will discuss SSL in a section of its own later in this chapter.

NOTE

The SSPI set of functions is available on Windows 95/98, Windows NT, and Windows 2000. You should use the SSPI in both your server and client software, so it is very helpful that the SSPI functions are available on all platforms. Use of these functions is basically the same from one platform to the next, but there are also some differences in the details.

This book is dedicated to the software development on Windows 2000 and will be covering the SSPI as used on this platform. The concepts discussed in this chapter can be used as a guide for your SSPI code running on other platforms. You should consult Platform SDK docmentation for information on the differences between platforms.

NOTE

If you are going to use the SSPI functions in your code, you should be sure to include the Security.h header file with your source code. Additionally, you must link Secur32.lib with your application.

Credentials, Contexts, and Blobs

The SSPI relies on two data constructs with which you will become intimately familiar. They are *credentials* and *contexts.* Credentials are data that allow authentication. This data can be a username and password or signed information using public key infrastructure (PKI). You will see that in any SSPI exchange, both the client and the server will establish the credentials they want to use in the exchange. The credentials themselves exist outside of the SSPI, and you will use the SSPI to retrieve a handle to the credentials.

The SSPI functions will use the credentials that your client and/or server software establishes to create a user context. The client and server software will communicate their contexts to each other, allowing authentication. Server software will also be able to impersonate a user context sent to it by the client. Contexts do not contain credential information, but they contain information derived from credentials and used for a communication session. This information can include session keys, Kerberos tickets, and other security-relevant data.

Your software will maintain handles to credentials and handles to contexts that are managed by the SSPI. Also, your software will communicate context information to its counterpart so that it can build context information on the other side of the conversation. The context that a server sends to its client (or vice versa) comes in the form of a security blob generated by the SSPI. When your software receives a blob from the SSPI, it should communicate it across the wire. When your software receives a blob from the wire, it passes the blob to an SSPI function.

When using the SSPI, you will see that the authentication stage of a conversation becomes little more than a series of sending and receiving blobs, which enables the SSPI to build a user context for each side. Because the responsibility of completing an authentication is so completely shared between the server and the client, the flow of logic can be confusing. But it helps to know the components and functions involved.

First I am going to introduce the SSPI functions used by the server, and then those used by the client, and I'll show how they interact. Then we'll look at the details of calling these functions. This way you start with a big picture.

The Server and the SSPI

A server using the SSPI for secure connectivity is most likely doing so to achieve one or more of these goals:

- Authentication/session initiation
- Impersonation
- Data privacy or encryption
- Data integrity or message signing

Table 12-2 shows the functions that your server will use, grouped by which task they help to perform.

Table 12-2. *SSPI functions for servers*

Task	Function	Description
Authentication/session initiation	*AcquireCredentialsHandle*	Used to retrieve a handle indicating your credentials. One or more contexts of clients can be associated with a credentials handle. Also selects the security protocol that you are using.
	AcceptSecurityContext	Called repeatedly. Blobs returned from the client function *InitializeSecurityContext* are passed to *AcceptSecurityContext* and vice versa, until a context has been completed.
Impersonation	*ImpersonateSecurityContext*	Your current thread impersonates a completed context.
Data privacy/encryption	*EncryptMessage*	Encrypts data and creates blobs to be communicated.
	DecryptMessage	Takes blobs created by *EncryptMessage* and returns decrypted data.
Data integrity and message signing	*MakeSignature*	Signs data provided by the application and creates blobs intended for communication.
	VerifySignature	Takes blobs returned from *MakeSignature* and checks the signature of the message contained within the blobs.
Cleanup	*DeleteSecurityContext*	Used to clean up handles when you are

FreeCredentialsHandle	finished with them.

The functions in Table 12-2 are the tools your server uses to converse using the SSPI. Much of the time your server will be receiving blobs from these functions that it must communicate across the wire to its client. Other times your server will be waiting for blobs from the client. The flowchart in Figure 12-3 shows a common example of a conversation from the server's point of view.

NOTE

In this diagram, and throughout this chapter, I will be referring to made-up functions called "*SendData*" and "*ReceiveData*," which you should view as placeholders for communication functions (such as *ReadFile* and *WriteFile* or *WSARecv* and *WSASend*). I do this to emphasize the point that the SSPI is transport layer_independent.

Figure 12-3. *A secure conversation from the server's point of view. The numbered squares call out the points at which the server is communicating with the client. They are numbered so that you can match them up with the points of communication on the client's flowchart, shown in Figure 12-4.*

The conversation illustrated in Figure 12-3 shows the building of the user context, how the user context impersonates the client, and the sending and receiving of encrypted messages. Of course, a real conversation is likely to have many more messages and a real server would perform actions based on those messages.

NOTE

Not all servers using the SSPI will encrypt every transaction with their client. Encryption might not be necessary, so the server might sign messages instead. This type of detail—whether encryption is necessary—is your decision as the developer of the service. You don't have to sign or encrypt your data at all. If you did not, you would be using the SSPI only for an initial authentication.

The Client and the SSPI

A client uses the SSPI to achieve the same goals as a server except for impersonation, which a client cannot do. Like the server, the client calls certain functions to achieve these goals. These functions are listed in Table 12-3.

Table 12-3. *SSPI functions for clients*

Task	Function	Description
Authentication/session initiation	*AcquireCredentialsHandle*	Used to retrieve a handle indicating your credentials. One or more contexts of clients can be associated with a credentials handle. Also selects the security protocol that you are using.
	InitializeSecurityContext	Called repeatedly. Blobs returned from this function are sent to the server's function *AcceptSecurityContext* and vice versa until a context has been completed.
Data Privacy/encryption to be communicated.	*EncryptMessage*	Encrypts data and creates blobs.
	DecryptMessage	Takes blobs created by *EncryptMessage* and returns decrypted data.
Data integrity and message signing	*MakeSignature*	Signs data provided by the application and creates blobs intended for communication.
	VerifySignature	Takes blobs returned from *MakeSignature* and checks the signature of the message contained within the blobs.
Cleanup	*DeleteSecurityContext* *FreeCredentialsHandle*	Used to clean up handles when you are finished with them.

Notice that the key difference between the server functions (described in Table 12-2) and the client functions is that the client makes repeated calls to *InitializeSecurityContext* rather than to *AcceptSecurityContext*. Figure 12-4 shows the conversation that was illustrated in Figure 12-3, but from the client's point of view.

Figure 12-4. *Secure conversation from the client's point of view. Notice that the numbered squares in this illustration match those in Figure 12-3.*

Notice in Figure 12-4 that the flow of logic for the client is similar to that of the server, but the client does call some different functions, and it is the initiator. (The client's first communication is a send rather than a receive.)

The SSPI Rationale

Once you understand that the communication between your client and your server will be managed within a framework, the SSPI is a piece of cake. Writing code flexible enough to loop at seemingly arbitrary times with the goal of building a "context" can be a challenge, so before jumping in, let me present the rationale.

As you know, the SSPI allows you to converse using different security protocols, which are packaged in the form of SSPs. Different protocols have different needs in terms of what must be communicated and when, which is the reason for some of the looping and blob management of the SSPI.

Take NTLM, for example. You saw in Figure 12-1 that NTLM requires three communication legs between the client and the server: a request for authentication, a challenge forward, and a response return. When your SSPI code is authenticating using NTLM, it repeats the *InitializeSecurityContext* loop twice, requiring your client to communicate with the server three times. These loops correlate directly with the legs of communication seen in Figure 12-1.

If you are authenticating using Kerberos, however, the SSPI will require only one pass through the entire SSPI process. This may come as a surprise, but revisit the Kerberos protocol in Figure 12-2, which shows the client communicating with the server only once. (If mutual authentication is required, the client communicates with the server twice.) The flow of your code through the SSPI sequence will reflect this.

> **NOTE**
>
> The SSPI only exposes communication between the client and the server to your software. Any communication with third parties such as the KDC or the DC is handled behind the scenes in the SSPI.

OK, I have said enough about how this is all going to look. Let's examine the functions used to implement the SSPI code.

Acquiring Credentials

The first step in developing the SSPI code is to establish your own credentials to the SSPI, and to establish which security protocol you wish to use. You call *AcquireCredentialsHandle* to do this:

```
SECURITY_STATUS AcquireCredentialsHandle(
   SEC_CHAR*   pszPrincipal,
   SEC_CHAR*   pszPackage,
   ULONG       lCredentialUse,
   PLUID       pvLogonId,
   PVOID       pAuthData,
   PVOID       pGetKeyFn,
   PVOID       pvGetKeyArgument,
   PCredHandle phCredential,
   PTimeStamp  ptsExpiration);
```

Notice that the *AcquireCredentialsHandle* function has some new data types. The SEC_CHAR data type is the string type used by the SSPI functions, and it boils down to a TCHAR. The SECURITY_STATUS type is an error value returned by all SSPI functions. Table 12-6 shows some of the currently defined errors.

AcquireCredentialsHandle might look daunting because of the number of parameters you must pass, but you will typically pass NULL for most of them. The *pszPrincipal* parameter is the name of the principal or entity for which you are retrieving a credential handle. You will typically pass NULL for this value, indicating the identity of the current thread (or process) token.

You pass a string indicating the security protocol you wish to use for the *pszPackage* parameter. The Platform SDK has values that map to the string names representing Kerberos, NTLM, Negotiate, and SSL. Table 12-4 shows these values.

Table 12-4. *Security package values that can be passed for* AcquireCredentialsHandle's pszPackage *parameter*

Value	Protocol
MICROSOFT_KERBEROS_NAME	Kerberos security support provider
NTLMSP_NAME	NTLM security support provider
NEGOSSP_NAME	Negotiate security support provider
UNISP_NAME	SChannel security support provider (SSL)

You should pass one of the defines in Table 12-4 for the *pszPackage* parameter.

The *lCredentialsUse* parameter can be set to one of the values in Table 12-5 to indicate how a particular credentials handle will be used.

Table 12-5. *Values that can be passed for* AcquireCredentialsHandle's lCredentialUse *parameter that indicate how credentials will be used*

Value	Description
SECPKG_CRED_INBOUND	Your software will validate an incoming user context. (This is a common setting for a server.)
SECPKG_CRED_OUTBOUND	Your software will be authenticated to a remote party. (This is a common setting for a client.)

SECPKG_CRED_BOTH	The value indicates both uses for your credentials handle, which is a common setting for a server that supports mutual authentication.

The *pvLogonId* parameter is used by file-system services that make calls to *AcquireCredentialsHandle*, and you should always pass NULL.

The *pAuthData* parameter is used to supply protocol-specific credentials that you wish to use to build a credentials handle. If you pass NULL, a credentials handle is returned for your token's credentials. The NTLM, Kerberos, and Negotiate security providers also allow you to pass, for the *pAuthData* parameter, a pointer to an instance of the following structure:

```
typedef struct {
   SEC_CHAR* User;
   ULONG     UserLength;
   SEC_CHAR* Domain;
   ULONG     DomainLength;
   SEC_CHAR* Password;
   ULONG     PasswordLength;
   ULONG     Flags;
} SEC_WINNT_AUTH_IDENTITY;
```

If you use the _SEC_WINNT_AUTH_IDENTITY structure and pass it as the *pAuthData* parameter, you should set the structure as follows:

```
SEC_WINNT_AUTH_IDENTITY authIdentity = {0};
authIdentity.User = TEXT("UserName");
authIdentity.UserLength = lstrlen(TEXT("UserName"));
authIdentity.Domain = TEXT("Domain");
authIdentity.DomainLength = lstrlen(TEXT("Domain"));
authIdentity.Password = TEXT("Password");
authIdentity.PasswordLength = lstrlen(TEXT("Password"));
authIdentity.Flags = SEC_WINNT_AUTH_IDENTITY_UNICODE;
```

Of course you should use a real username and domain and real password values. And if you are compiling for ANSI instead of Unicode, you should assign the value SEC_WINNT_AUTH_IDENTITY_ANSI to the *Flags* member of the structure.

The *AcquireCredentialsHandle* function allows you to define a callback function that is used to create keys for use with SSPI. Most software will not need this feature and will pass NULL for the *pGetKeyFn* and *pvGetKeyArgument* parameters.

You should pass the address of a PCredHandle variable as the *phCredential* parameter. The *AcquireCredentialsHandle* function returns the newly created credentials handle in this variable. And finally, you should pass the address of a TimeStamp structure as the *ptsExpiration* parameter.

NOTE

Several of the SSPI functions return a TimeStamp structure. This structure is interchangeable with the standard FILETIME structure used with functions such as *FileTimeToSystemTime*.

All SSPI functions should return TimeStamp (or FILETIME) information in local time. See the Platform SDK documentation for more information on the FILETIME structure.

Like all SSPI functions, when *AcquireCredentialsHandle* succeeds, it returns SEC_E_OK. Table 12-6 lists some relevant SECURITY_STATUS values.

Table 12-6. *Relevant SECURITY_STATUS values*

Status Code	Description
SEC_E_OK	All is well.
SEC_E_NOT_OWNER	The caller of the function is not the owner of the desired credentials.
SEC_E_INVALID_HANDLE	An invalid handle was passed to a function.
SEC_E_INVALID_TOKEN	An invalid token was passed to a function.
SEC_E_NOT_SUPPORTED	A requested feature is not supported by a specified support provider.
SEC_E_QOP_NOT_SUPPORTED	A quality-of-protection attribute is not supported by a specified support provider.
SEC_E_NO_IMPERSONATION	The provided context does not have an impersonation token.
SEC_E_TARGET_UNKNOWN	The target is unknown.
SEC_E_SECPKG_NOT_FOUND	An unknown security package was specified.
SEC_E_NO_IMPERSONATION	Impersonation is not allowed for a context.
SEC_E_LOGON_DENIED	A principal is unable to log on because it does not possess the required credentials.
SEC_E_UNKNOWN_CREDENTIALS	The provided credentials were not recognized.
SEC_E_NO_CREDENTIALS	Credentials are not available.
SEC_E_MESSAGE_ALTERED	A message supplied for verification or decryption has been modified in transit.
SEC_E_OUT_OF_SEQUENCE	A message supplied for verification is out of sequence.
SEC_E_NO_AUTHENTICATING_AUTHORITY	A KDC or DC was unreachable.
SEC_E_CONTEXT_EXPIRED	A context expired and is now invalid.
SEC_E_INCOMPLETE_MESSAGE	An incomplete message was supplied.
SEC_I_CONTINUE_NEEDED	A context was not completed, and a function must be called again.
SEC_I_COMPLETE_NEEDED	The function completed, but you must call *CompleteAuthToken.*
SEC_I_COMPLETE_AND_CONTINUE	You must call *CompleteAuthToken* and loop and call the function again.
SEC_I_INCOMPLETE_CREDENTIALS	The remote party is requesting more complete credentials. This status code can apply when the client's current credentials are anonymous.
SEC_I_RENEGOTIATE	The remote party is requesting that credentials be renegotiated.
SEC_E_INSUFFICIENT_MEMORY	A supplied buffer is too small.

This code fragment shows a common call to *AcquireCredentialsHandle*:

```
CredHandle hCredentials;
TimeStamp tsExpires;
ss = AcquireCredentialsHandle( NULL, MICROSOFT_KERBEROS_NAME,
   SECPKG_CRED_BOTH, NULL, NULL, NULL,
   NULL, &hCredentials, &tsExpires );
ReportSSPIError(L"AcquireCredentialsHandle", ss);
if(ss != SEC_E_OK){
   // Error
}
```

If this call succeeds, the *hCredentials* variable will hold a valid handle for use in a mutually authenticating Kerberos conversation. When you are finished with the handle, pass it to *FreeCredentialsHandle*:

```
SECURITY_STATUS FreeCredentialsHandle(
   PCredHandle phCredential);
```

Now that you have a handle for your credentials, you are ready to begin the authentication process.

Authentication—The Client's Role

As you already know, the authentication process differs for the client and the server. I will start explaining the authentication process with the client function *InitializeSecurityContext*, because an authentication typically starts with the client.

```
SECURITY_STATUS InitializeSecurityContext(
   PCredHandle    phCredential,
   PCtxtHandle    phContext,
   SEC_CHAR       *pszTargetName,
   ULONG          lContextReq,
   ULONG          lReserved1,
   ULONG          lTargetDataRep,
   PSecBufferDesc pInput,
   ULONG          lReserved2,
   PCtxtHandle    phNewContext,
   PSecBufferDesc pOutput,
   PULONG         plContextAttr,
   PTimeStamp     ptsExpiration);
```

The first parameter to the *InitializeSecurityContext* function is a handle to the credentials you received with *AcquireCredentialsHandle*. The *InitializeSecurityContext* function has a fair number of parameters, and it is meant to be called multiple times in a loop. Some of these parameters, however, are not relevant each time you call the function. I will discuss the parameters in terms of the first time you call the function, and then I will point out differences in the successive calls to *InitializeSecurityContext*.

The *phContext* structure is NULL in your first call to the function. (In future calls, it will be a pointer to a *CtxtHandle* variable, which holds the handle of a context "in progress.") The *pszTargetName* parameter is the username of the server to which you are authenticating. If the server is running in the system account on its host machine, the *pszTargetName* is the machine name.

The *lContextReq* parameter is your way of indicating to the SSPI and to the server what you would like to get out of a secure conversation. Table 12-7 lists the flags that you can pass for *lContextReq*.

Table 12-7. *Flags you can pass for* InitializeSecurityContext*'s* lContextReq *parameter*

Flag	Description

ISC_REQ_DELEGATE	The server is allowed to delegate the client's user context. This delegation enables the server to act as a client to yet another server on behalf of the client.
ISC_REQ_MUTUAL_AUTH	If you set this flag, the server must also be able to authenticate itself to the client. Mutual authentication is not supported by the NTLM protocol, but it is supported by Kerberos.
ISC_REQ_REPLAY_DETECT	This flag indicates that you want the security package to sign messages making it impossible for a malicious third party to perform a replay attack. This flag implies the ISC_REQ_ INTEGRITY flag.
ISC_REQ_SEQUENCE_DETECT	The SSPI will detect out-of-sequence messaging for this session context. This also requires message signing and implies the ISC_REQ_ INTEGRITY flag.
ISC_REQ_CONFIDENTIALITY	You will use the context to generate encrypted messages. In Kerberos, you must have mutual authentication before you can have message confidentiality. NTLM does not support mutual authentication, so this restriction is not imposed.
ISC_REQ_USE_SESSION_KEY	You wish to generate a new session key with the remote principal.
ISC_REQ_PROMPT_FOR_CREDS	If the client is an interactive user, the security package will attempt to prompt the user for the appropriate credentials. This feature is not implemented for all security providers.
ISC_REQ_USE_SUPPLIED_CREDS	You are supplying package-specific credentials as an input buffer to the function.
ISC_REQ_ALLOCATE_MEMORY	The security package will allocate memory for your out buffers.
ISC_REQ_USE_DCE_STYLE	You want a three-leg authentication transaction.
ISC_REQ_DATAGRAM	Your communication layer is using data gram_style communication.
ISC_REQ_CONNECTION	Your communication layer is using connection-oriented communication.
ISC_REQ_STREAM	Your communication layer is using stream-style communication.
ISC_REQ_EXTENDED_ERROR	If the context fails, you want to receive extended error information.
ISC_REQ_INTEGRITY	Message signing is requested, but not specifically applied by the package for replay detection or message-ordering purposes.

Many of the flags in Table 12-7 will never be used in your SSPI code, but you can see that the flexibility of the package is fairly high. I will discuss some of these flags throughout this section.

The *lTargetDataRep* parameter indicates what byte-ordering scheme you will be using when communicating across the network. Your options are SECURITY_NATIVE_DREP and SECURITY_NETWORK_DREP. If you have the choice, you should always use SECURITY_NETWORK_DREP, to promote operating system interoperability. The *pInput* parameter indicates input information that you are passing to

InitializeSecurityContext. With this parameter, you pass blobs and other information to the function. On your initial call to *InitializeSecurityContext*, you will typically pass NULL for your input buffers because you have no blobs to start with. This process is similar to that of the *pOutput* parameter, which returns to your software blobs that will be communicated to the server. I will discuss input and output buffers in more detail shortly.

You should pass the pointer to a CtxtHandle variable as the *phNewContext* parameter. The system returns the context handle to you via this parameter. This is the same CtxtHandle that you will pass to *InitializeSecurityContext* via the *phContext* field in successive calls to the function.

The *plContextAttr* parameter returns the attributes of your session as imposed by the security provider. This will be a composite of the attributes that you requested in the *lContextReq* parameter and the attributes imposed by the provider. You should always check the value returned in this parameter to ensure that your session has the attributes that are important to your software.

The *ptsExpiration* parameter will return a timestamp that indicates the expiration time of the session represented by the context that you are building.

Successive Calls to *InitializeSecurityContext*

So far, we have been discussing your client's first call to *InitializeSecurityContext*. In successive calls to this function, you'll notice some differences in the process:

- The *phContext* parameter must point to a variable holding the context handle previously returned by the *phNewContext* parameter.
- The SSPI ignores the *pszTargetName* parameter on successive calls to *InitializeSecurityContext*.
- You will pass blobs that you received from the server into *InitializeSecurityContext* via the *pInput* parameter.
- The server's context requirements are returned via the *plContextAttr* parameter, and they should be checked for a mismatch with your client's needs.

You should make successive calls to *InitializeSecurityContext* based on the function's return value. If it returns SEC_I_CONTINUE_NEEDED, your client should loop and call *InitializeSecurityContext* again. When the function returns SEC_E_OK, you have a completed context. Other return values indicate error scenarios.

Input and Output Buffers

Our discussion of *InitializeSecurityContext* is not finished until we really nail down the topic of input and output buffers. Input and output buffers are found throughout the SSPI functions and show up as the *pInput* and *pOutput* parameters.

Input and output buffers provide a way for software to give information, as needed, to a security support provider, in a way that is flexible enough to meet the needs of any supported protocol. This is why they are used with most of the SSPI functions that send or receive data to or from the system.

Let me explain how you use these buffers. You create an array of SecBuffer variables, which you define, and you point them to memory buffers, which you have allocated. You then put the address of the array in an instance of the SecBufferDesc type, which indicates how many buffers are in your array. Here is the definition of the SecBufferDesc structure:

```
typedef struct _SecBufferDesc {
   ULONG      ulVersion; // Set to SECBUFFER_VERSION
   ULONG      cBuffers;
   PSecBuffer pBuffers;
} SecBufferDesc;
```

Following is the definition of the SecBuffer structure:

```
typedef struct _SecBuffer {
   ULONG cbBuffer;
   ULONG BufferType;
   PVOID pvBuffer;
} SecBuffer;
```

The *cbBuffer* member of SecBuffer indicates the size of the memory block pointed to by the *pvBuffer* member. The *ulVersion* member of SecBufferDesc should always be set to SECBUFFER_VERSION.

NOTE

For output buffers, you can set *cbBuffer* to 0 and *pvBuffer* to NULL, and the system will allocate buffers for you to return data. When you are finished with the returned buffers, you pass them to *FreeContextBuffer*. You can request that the system allocate buffers for you in this way by passing the ISC_REQ_ALLOCATE_MEMORY context requirement to the *InitializeSecurityContext* function.

The illustration in Figure 12-5 shows the relationship between SecBuffer and SecBufferDesc, your actual memory blocks, and *InitializeSecurityContext*.

Figure 12-5. *SSPI input and output buffers*

InitializeSecurityContext and Buffers

Trying to understand the way the SSPI deals with buffers can be confusing until you learn about the specifics. Then things clear up quite a bit. As I mentioned before, *InitializeSecurityContext* uses buffers to input and output security blobs that are intended for communication to the server. On the first call to

InitializeSecurityContext, you can pass NULL for the *pInput* parameter. However, as you receive data from the server, you will be constructing buffers and passing them to the function.

For both input and output, each call to *InitializeSecurityContext* takes an array of one SecBuffer of type SECBUFFER_TOKEN. This buffer type indicates to the system that this buffer is either an input blob for building a context or an output blob for building a context.

Here is a code fragment that shows how you will construct these buffers for use with *InitializeSecurityContext*:

```
// Set up "Out" buffers
SecBuffer secBufferOut[1];
secBufferOut[0].BufferType = SECBUFFER_TOKEN;
secBufferOut[0].cbBuffer = cbBlockToSend;
secBufferOut[0].pvBuffer = pbBlockToSend;

// Set up "Out" buffer descriptor
SecBufferDesc secBufDescriptorOut;
secBufDescriptorOut.cBuffers = 1;
secBufDescriptorOut.pBuffers = secBufferOut;
secBufDescriptorOut.ulVersion = SECBUFFER_VERSION;

// Set up "In" buffers
SecBuffer secBufferIn[1];
secBufferIn[0].BufferType = SECBUFFER_TOKEN;
secBufferIn[0].cbBuffer = cbBlockReceived;
secBufferIn[0].pvBuffer = pbBlockReceived;

// Set up "In" buffer descriptor;
SecBufferDesc secBufDescriptorIn;
secBufDescriptorIn.cBuffers = 1;
secBufDescriptorIn.pBuffers = secBufferIn;
secBufDescriptorIn.ulVersion = SECBUFFER_VERSION;
```

The *pbBlockReceived* and *pbBlockToSend* buffers are allocated before calling *InitializeSecurityContext*. Alternatively, you could choose to have the function allocate a block for your out buffer.

After calling *InitializeSecurityContext*, the *cbBuffer* member of the out buffer contains the size of the blob to be sent to the server. If the size is a nonzero value, you should send to the server as many bytes as indicated by the value from the buffer pointed to by the *pvBuffer* member. This is the "handshake" for authentication.

The SSPI defines a number of different buffer types. I will describe them as they relate to our discussion. For a list of the currently defined buffer types, see the Platform SDK documentation.

InitializeSecurityContext—Putting It All Together

There is no question about it—the number of details you have to remember when dealing with the SSPI can be daunting. Seeing how it all works in a simple function, however, will begin to make the process clearer.

Here is a sample function that uses the techniques we have discussed to build a completed context:

```
BOOL ClientHandshakeAuth(CredHandle* phCredentials,
   PULONG plAttributes, CtxtHandle* phContext, PTSTR pszServer){
   BOOL fSuccess = FALSE;

   __try{
      SECURITY_STATUS ss;

      // Declare in and out buffers
```

```
SecBuffer secBufferOut[1];
SecBufferDesc secBufDescriptorOut;
SecBuffer secBufferIn[1];
SecBufferDesc secBufDescriptorIn;

// Set up some "loop state" information
BOOL fFirstPass = TRUE;
ss = SEC_I_CONTINUE_NEEDED;
while ( ss == SEC_I_CONTINUE_NEEDED ){

   // In blob pointer
   PBYTE pbData = NULL;
   if(fFirstPass){ // First pass, no in buffers
      secBufDescriptorIn.cBuffers = 0;
      secBufDescriptorIn.pBuffers = NULL;
      secBufDescriptorIn.ulVersion = SECBUFFER_VERSION;
   }else{ // Successive passes
      // Get size of blob
      ULONG lSize;
      ULONG lTempSize = sizeof(lSize);
      ReceiveData(&lSize, &lTempSize);
      // Get blob
      pbData = (PBYTE)alloca(lSize);
      ReceiveData(pbData, &lSize);

      // Point "In Buffer" to blob
      secBufferIn[0].BufferType = SECBUFFER_TOKEN;
      secBufferIn[0].cbBuffer = lSize;
      secBufferIn[0].pvBuffer = pbData;
      // Point "In" BufDesc to in buffer
      secBufDescriptorIn.cBuffers = 1;
      secBufDescriptorIn.pBuffers = secBufferIn;
      secBufDescriptorIn.ulVersion = SECBUFFER_VERSION;
   }

   // Set up out buffer (The SSPI will be allocating buffers for us)
   secBufferOut[0].BufferType = SECBUFFER_TOKEN;
   secBufferOut[0].cbBuffer = 0;
   secBufferOut[0].pvBuffer = NULL;
   // Point "Out" Bufdesc to out buffer
   secBufDescriptorOut.cBuffers = 1;
   secBufDescriptorOut.pBuffers = secBufferOut;
   secBufDescriptorOut.ulVersion = SECBUFFER_VERSION;

   ss=
      InitializeSecurityContext(
         phCredentials,
         fFirstPass?NULL:phContext,
         pszServer,
         *plAttributes | ISC_REQ_ALLOCATE_MEMORY,
         0, SECURITY_NETWORK_DREP,
         &secBufDescriptorIn, 0,
         phContext,
         &secBufDescriptorOut,
         plAttributes, NULL);

   // No longer first pass through the loop
   fFirstPass = FALSE;

   // Was a blob output? If so, send it.
   if (secBufferOut[0].cbBuffer!=0){
      // Server communication!!!
      // Send the size of the blob.
      SendData(&secBufferOut[0].cbBuffer, sizeof(ULONG));
      // Send the blob itself
      SendData(secBufferOut[0].pvBuffer,
```

```
                secBufferOut[0].cbBuffer);

             // Free out buffer
             FreeContextBuffer(secBufferOut[0].pvBuffer);
          }
       }// Loop if ss == SEC_I_CONTINUE_NEEDED;

       // Final result
       if (ss != SEC_E_OK){
          __leave;
       }

       fSuccess = TRUE;
    }__finally{
       // Clear the context handle if we fail
       if (!fSuccess){
          ZeroMemory(phContext, sizeof(*phContext));
       }
    }
    return (fSuccess);
}
```

The *ClientHandshakeAuth* function takes a credentials handle returned from *AcquireCredentialsHandle*, some flags, and a server name, and if successful at building a context, returns a completed context.

Notice that *ClientHandshakeAuth* function communicates with the server in two spots in the code. The functions I use to communicate are fictional functions named *SendData* and *ReceiveData*. They take a buffer and a size, and are placeholders for whatever communication mechanism you choose for your software.

Notice also that when I communicate a blob, I send the blob's size before I send the blob, and when I receive a blob, I read the blob's size before receiving it.

> **NOTE**
>
> Because the SSPI is communication transport-independent, it is necessary for you to create some sort of protocol by which you communicate the blobs to the principal on the other side of the wire.

Pay special attention to the buffer management in this function, and notice how *InitializeSecurityContext* communicates to our software when a blob is to be sent across the wire and when the function needs to loop again. This is the essence of the client-side responsibility in the authentication handshake with the SSPI.

The following code fragment could be used to get a credentials handle and to call *ClientHandshakeAuth* to begin authenticating with the Kerberos protocol:

```
CredHandle hCredentials;
TimeStamp tsExpires;
SECURITY_STATUS ss = AcquireCredentialsHandle( NULL,
   MICROSOFT_KERBEROS_NAME, SECPKG_CRED_BOTH, NULL, NULL,
   NULL, NULL, &hCredentials, &tsExpires );
if(ss != SEC_E_OK){
   // Error
}

ULONG lAttributes =
   ISC_REQ_STREAM|ISC_REQ_CONFIDENTIALITY|ISC_REQ_MUTUAL_AUTH;
CtxtHandle hContext = {0};
if(!ClientHandshakeAuth(&hCredentials, &lAttributes,
   &hContext, TEXT("jclark-piii600"))){
   // Error
}
```

```
    // If successful, we have authenticated at this point

 DeleteSecurityContext(&hContext);
 FreeCredentialsHandle(&hCredentials);
```

In this code, *AcquireCredentialsHandle* returns a credentials handle representing the identity of the calling function. Then we call our sample function *ClientHandshakeAuth* to indicate that we want mutual authentication and encryption capability, and that we will be communicating using a streaming technology. I am now going to talk about the server side of the authentication in the SSPI. Before you move on, you might find it useful to take a moment to review Figures 12-3 and 12-4, which illustrate the client and server sides of an SSPI conversation. Think about these figures in the context of the code fragments we've just examined.

Authentication—The Server's Role

Understanding the client's role in an authentication handshake using the SSPI is helpful in understanding the server's role. In fact, once either side is understood, the other side becomes very approachable. The server function for managing blobs is *AcceptSecurityContext*:

```
 SECURITY_STATUS AcceptSecurityContext(
    PCredHandle    phCredential,
    PCtxtHandle    phContext,
    PSecBufferDesc pInput,
    ULONG          lContextReq,
    ULONG          lTargetDataRep,
    PCtxtHandle    phNewContext,
    PSecBufferDesc pOutput,
    PULONG         pfContextAttr,
    PTimeStamp     ptsExpiration);
```

Notice that *AcceptSecurityContext* has the same parameters as *InitializeSecurityContext*, except that it does not have the two reserved parameters and the parameter indicating the server's name. Naturally the server name is not needed, because it is the server who is calling *AcceptSecurityContext*.

Like the role of its client counterpart, *AcceptSecurityContext*'s role is to receive and produce blobs communicated with the opposite principal. Both *AcceptSecurityContext* and *InitializeSecurityContext* must be allowed to loop until they return SEC_E_OK. There are two noteworthy differences when using *AcceptSecurityContext*:

- The first time you call *AcceptSecurityContext*, you have already received your first blob from your client, so there is always an input buffer used with this function. (This is unlike *InitializeSecurityContext*, whose initial pass has no input buffer.)

- *AcceptSecurityContext* uses the same context requirements as *InitializeSecurityContext* (as listed in Table 12-7), except that the values for the *lContextReq* parameter of *AcceptSecurityContext* have a different prefix. Instead of starting with "ISC_REQ_", which stands for "*InitializeSecurityContext* Requirement", *AcceptSecurityContext*'s requirements start with "ASC_REQ_". For example, the equivalent of ISC_REQ_CONFIDENTIALITY would be ASC_REQ_CONFIDENTIALITY.

Aside from these two differences, you use *AcceptSecurityContext* largely the same way you use *InitializeSecurityContext*. However, the resulting context with *AcceptSecurityContext* is more capable in that the server can use it to impersonate or otherwise obtain a handle to a token, which we will discuss in a moment. Let's look at a sample function that shows the use of *AcceptSecurityContext* in server code:

```
 BOOL ServerHandshakeAuth(CredHandle* phCredentials,
    PULONG plAttributes, CtxtHandle *phContext){
    BOOL fSuccess = FALSE;
```

```
__try{
   SECURITY_STATUS ss;

   // Declare in and out buffers
   SecBuffer secBufferIn[1];
   SecBufferDesc secBufDescriptorIn;

   SecBuffer secBufferOut[1];
   SecBufferDesc secBufDescriptorOut;

   // Set up some "loop state" information
   BOOL fFirstPass = TRUE;
   ss = SEC_I_CONTINUE_NEEDED;
   while (ss == SEC_I_CONTINUE_NEEDED){

      // Client communication!!!
      // Get size of blob.
      ULONG lSize;
      ULONG lTempSize = sizeof(lSize);
      ReceiveData(&lSize, &lTempSize);
      // Get blob
      PBYTE pbTokenBuf = (PBYTE)alloca(lSize);
      ReceiveData(pbTokenBuf, &lSize);

      // Point "In Buffer" to blob
      secBufferIn[0].BufferType = SECBUFFER_TOKEN;
      secBufferIn[0].cbBuffer = lSize;
      secBufferIn[0].pvBuffer = pbTokenBuf;
      // Point "In" BufDesc to in buffer
      secBufDescriptorIn.ulVersion = SECBUFFER_VERSION;
      secBufDescriptorIn.cBuffers = 1;
      secBufDescriptorIn.pBuffers = secBufferIn;

      // Set up out buffer
      // (The SSPI will be allocating buffers for us)
      secBufferOut[0].BufferType = SECBUFFER_TOKEN;
      secBufferOut[0].cbBuffer = 0;
      secBufferOut[0].pvBuffer = NULL;
      // Point "Out" BufDesc to out buffer
      secBufDescriptorOut.ulVersion = SECBUFFER_VERSION;
      secBufDescriptorOut.cBuffers = 1;
      secBufDescriptorOut.pBuffers = secBufferOut;

      // Here is our blob management function
      ss =
         AcceptSecurityContext(
            phCredentials,
            fFirstPass?NULL:phContext,
            &secBufDescriptorIn,
            *plAttributes | ASC_REQ_ALLOCATE_MEMORY,
            SECURITY_NETWORK_DREP,
            phContext,
            &secBufDescriptorOut,
            plAttributes, NULL);

      // No longer first pass through the loop
      fFirstPass = FALSE;

      // Was a blob output?  If so, send it.
      if (secBufferOut[0].cbBuffer != 0){

         // Client communication !!!
         // Send the size of the blob
         SendData(&secBufferOut[0].cbBuffer, sizeof(ULONG));
         // Send the blob itself
```

```
            SendData(secBufferOut[0].pvBuffer,
               secBufferOut[0].cbBuffer);

            // Free out buffer
            FreeContextBuffer(secBufferOut[0].pvBuffer);
         }
      }// Loop if ss == SEC_I_CONTINUE_NEEDED;

      // Final result
      if(ss != SEC_E_OK){
         __leave;
      }

      fSuccess = TRUE;
   }__finally{
      // Clear the context handle if we fail
      if (!fSuccess){
         ZeroMemory(phContext, sizeof(*phContext));
      }
   }
   return (fSuccess);
}
```

Notice that *ServerHandshakeAuth* is using pseudo-functions named *SendData* and *ReceiveData*, which are meant to represent any communication mechanism. Also, *ServerHandshakeAuth* sends and receives the sizes of blobs so that the other side knows how much data to receive.

Again, pay particular attention to the buffer management code. Notice that the blobs that are received are passed into the function via in buffers. Meanwhile, the existence of an out buffer is checked and then sent across the wire if it exists. Also, this use of *AcceptSecurityContext* allows the function to allocate memory buffers for output blobs, and these buffers are freed with *FreeContextBuffer* in the same way they are with *InitializeSecurityContext*.

The following code fragment could be used to set up a credentials handle for use in calling the *ServerHandshakeAuth* sample function:

```
CredHandle hCredentials;
TimeStamp tsExpires;
SECURITY_STATUS ss = AcquireCredentialsHandle( NULL,
   MICROSOFT_KERBEROS_NAME, SECPKG_CRED_BOTH, NULL, NULL,
   NULL, NULL, &hCredentials, &tsExpires );
if(ss != SEC_E_OK){
   // Error
}

ULONG lAttributes = ASC_REQ_STREAM;
CtxtHandle hContext = {0};
if(!ServerHandshakeAuth(&hCredentials,
   &lAttributes, &hContext)){
   // Error
}
ss = ImpersonateSecurityContext(&hContext);
if(ss != SEC_E_OK){
   __leave;
}

DeleteSecurityContext(&hContext);
FreeCredentialsHandle(&hCredentials);
```

Impersonation and Token Acquisition

Impersonation is an important part of authentication, from the server's point of view. Fortunately, when you have a completed context handle, impersonation is easy. You use *ImpersonateSecurityContext* and *RevertSecurityContext* as shown here:

```
SECURITY_STATUS ImpersonateSecurityContext(
   PCtxtHandle phContext);

SECURITY_STATUS RevertSecurityContext(
   PCtxtHandle phContext);
```

You can also request a token handle from a security context using the *QuerySecurityContextToken* function:

```
SECURITY_STATUS QuerySecurityContextToken(
   PCtxtHandle phContext,
   HANDLE      *phToken);
```

With the useful *QuerySecurityContextToken* function, you can get a client's token and store it with the connection information, without first impersonating the client.

Impersonation is a great way for a server to manage security and is covered in detail in Chapter 11. The SSPI allows you to impersonate a client over *any* communication medium. You are no longer limited to pipes or RPC. You can impersonate a client with sockets, IPX/SPX, or a serial connection if it serves your software's needs.

You can find a complete implementation of client and server SSPI programming in the SSPIChat sample application discussed later in this chapter. But for now I would like to talk more about message signing and encryption.

Message Signing and Encryption

Once you have negotiated an authentication between the client and server, the client and server can begin a systematic exchange of messages as expected in client/server communication. To take full advantage of this exchange, you should sign or encrypt the messages that are communicated. Here are guidelines to follow:

1. Sign messages when privacy is not needed but assuredness of message integrity is needed (which is just about always).

2. Encrypt messages when privacy is needed. (With encryption, signing is not necessary.)

Signing and encryption use parallel functions in the same way that authentication does, but the mechanisms are the same for client and server. The function pair for creating a signed message is *MakeSignature* and *VerifySignature*. The *MakeSignature* function is defined as follows:

```
SECURITY_STATUS MakeSignature(
   PCtxtHandle    phContext,
   ULONG          lQOP,
   PSecBufferDesc pMessage,
   ULONG          MessageSeqNo);
```

You pass a data buffer and a completed context to *MakeSignature*, and the function returns a signature that you pass (along with the data buffer) across the wire. The receiving side of the communication takes this information and verifies the data buffer by using the signature. Once again, you should view the data returned by *MakeSignature* as an opaque blob intended for communication only.

The *lQOP* parameter is security protocol_specific and allows you to specify the details of the signing algorithm used. You will usually pass zero for this parameter. If you are keeping track of message sequence numbers, you should pass the message's sequence number as the *MessageSeqNo* parameter. If you pass zero, sequence does not matter to you.

As with the other SSPI functions we have dealt with, data is passed to and received from *MakeSignature* via data buffers. In the case of *MakeSignature*, two buffers are passed with a single buffer descriptor, so you must make an array of two SecBuffer variables. You will set one to type SECBUFFER_TOKEN, which will receive the signature for the message. The second buffer is of type SECBUFFER_DATA, which indicates the data itself. Here is a function that signs a data buffer and sends it using the pseudo-communication function *SendData*:

```
BOOL SendSignedMessage(CtxtHandle* phContext, PVOID pvData, ULONG lSize)
{
   BOOL fSuccess = FALSE;

   __try{
      SECURITY_STATUS ss;

      // Find some important max buffer size information
      SecPkgContext_Sizes sizes;
      ss = QueryContextAttributes( phContext, SECPKG_ATTR_SIZES, &sizes );
      if(ss != SEC_E_OK){
         __leave;
      }

      PVOID pvSignature = alloca(sizes.cbMaxSignature);

      SecBuffer secBuffer[2];
      // Set up buffer to receive signature
      secBuffer[0].BufferType = SECBUFFER_TOKEN;
      secBuffer[0].cbBuffer = sizes.cbMaxSignature;
      secBuffer[0].pvBuffer = pvSignature;
      // Set up buffer to point to message data
      secBuffer[1].BufferType = SECBUFFER_DATA;
      secBuffer[1].cbBuffer = lSize;
      secBuffer[1].pvBuffer = pvData;
      // Set up buffer descriptor
      SecBufferDesc secBufferDesc;
      secBufferDesc.cBuffers = 2;
      secBufferDesc.pBuffers = secBuffer;
      secBufferDesc.ulVersion = SECBUFFER_VERSION;
      // Make signature
      ss = MakeSignature( phContext, 0, &secBufferDesc, 0 );
      if(ss != SEC_E_OK){
         __leave;
      }

      // Send signature
      SendData(&secBuffer[0].cbBuffer, sizeof(ULONG));
      SendData(secBuffer[0].pvBuffer, secBuffer[0].cbBuffer);

      // Send message
      SendData(&secBuffer[1].cbBuffer, sizeof(ULONG));
      SendData(secBuffer[1].pvBuffer, secBuffer[1].cbBuffer);

      fSuccess = TRUE;
   }__finally{
   }
   return (fSuccess);
}
```

NOTE

Unlike *InitializeSecurityContext* and *AcceptSecurityContext*, *MakeSignature* does not have separate in and out buffers, and it will not allocate buffers for output. This means that you must supply a buffer large enough for the generated signature. You can find the maximum signature size by using the *QueryContextAttributes* function with the SECPKG_ATTR_SIZES attribute. The *QueryContextAttributes* function is defined as follows:

```
SECURITY_STATUS QueryContextAttributes(
   PCtxtHandle phContext,
   ULONG       lAttribute,
   PVOID       pBuffer);
```

In this case you pass an instance of a SecPkgContext_Sizes structure to be populated by the function. Here is the structure's definition:

```
typedef struct _SecPkgContext_Sizes {
   ULONG cbMaxToken;
   ULONG cbMaxSignature;
   ULONG cbBlockSize;
   ULONG cbSecurityTrailer;
} SecPkgContext_Sizes;
```

The SecPkgContext_Sizes structure conveniently contains some information about maximum sizes, including *cbMaxSignature*, which you will use with *MakeSignature*, as shown in the preceding code sample.

The other side of the transaction must read the signature and the message data, and pass it to *VerifySignature*:

```
SECURITY_STATUS VerifySignature(
   PCtxtHandle    phContext,
   PSecBufferDesc pMessage,
   ULONG          MessageSeqNo,
   PULONG         plQOP);
```

The *VerifySignature* function is similar to *MakeSignature* in that it takes the same buffers and types. However, *VerifySignature* assures only that the message has not been modified and does not modify the buffers. Here is a sample function that receives a signature and a message and calls *VerifySignature*:

```
PVOID GetSignedMessage(CtxtHandle* phContext, PULONG plSize)
{
   PVOID pvMessage = NULL;

   __try{
      SECURITY_STATUS ss;

      ULONG lSigLen;
      PVOID pvDataSig;
      // Get signature length
      ULONG lTempSize = sizeof(lSigLen);
      ReceiveData(&lSigLen, &lTempSize);
      pvDataSig = alloca(lSigLen);
      // Get signature
      ReceiveData(pvDataSig, &lSigLen);

      ULONG lMsgLen;
      PVOID pvDataMsg;
      // Get message length
      lTempSize = sizeof(lMsgLen);
      ReceiveData(&lMsgLen, &lTempSize);
      pvDataMsg = alloca(lMsgLen);
      // Get message
      ReceiveData(pvDataMsg, &lMsgLen);
```

```
      SecBuffer secBuffer[2];
      // Set up signature buffer
      secBuffer[0].BufferType = SECBUFFER_TOKEN;
      secBuffer[0].cbBuffer = lSigLen;
      secBuffer[0].pvBuffer = pvDataSig;
      // Set up message buffer
      secBuffer[1].BufferType = SECBUFFER_DATA;
      secBuffer[1].cbBuffer = lMsgLen;
      secBuffer[1].pvBuffer = pvDataMsg;
      // Set up buffer descriptor
      SecBufferDesc secBufferDesc;
      secBufferDesc.cBuffers = 2;
      secBufferDesc.pBuffers = secBuffer;
      secBufferDesc.ulVersion = SECBUFFER_VERSION;
      ULONG lQual=0;
      // Verify signature
      ss = VerifySignature(phContext, &secBufferDesc, 0, &lQual);
      if (ss != SEC_E_OK){
         __leave;
      }
      // Return a buffer that must be freed, containing message
      pvMessage = LocalAlloc(LPTR, secBuffer[1].cbBuffer);
      if (pvMessage != NULL){
         CopyMemory(pvMessage, secBuffer[1].pvBuffer,
            secBuffer[1].cbBuffer);
      }
   }__finally{};
   return pvMessage;
}
```

If the message has been modified in transit, *VerifySignature* will return SEC_E_MESSAGE_ALTERED or SEC_E_OUT_OF_SEQUENCE.

Encrypting Messages

The process of encrypting and decrypting messages is very similar to that of signing and verifying messages, but the message is encrypted and decrypted in place. This means that the data buffer you pass will be modified by these functions.

Because some encryption algorithms require that encrypted data come in multiples of certain block sizes (such as 8 bytes or 16 bytes), it is necessary to check the block size and include a third buffer as a sort of "encryption overflow" that must also be sent across the wire. The functions we use are *EncryptMessage* and *DecryptMessage*. *EncryptMessage* is defined as follows:

```
SECURITY_STATUS EncryptMessage(
   PCtxtHandle    phContext,
   ULONG          lQOP,
   PSecBufferDesc pMessage,
   ULONG          MessageSeqNo);
```

DecryptMessage is defined like this:

```
SECURITY_STATUS DecryptMessage(
   PCtxtHandle    phContext,
   PSecBufferDesc pMessage,
   ULONG          MessageSeqNo,
   PULONG         plQOP
);
```

Here is a sample function that encrypts a message and sends it across the wire by using *SendData*:

```
BOOL SendEncryptedMessage(CtxtHandle* phContext,
   PVOID pvData, ULONG lSize){
   BOOL fSuccess = FALSE;

   __try{
      SECURITY_STATUS ss;

      // Get some important size information
      SecPkgContext_Sizes sizes;
      ss = QueryContextAttributes(phContext, SECPKG_ATTR_SIZES, &sizes);
      if(ss != SEC_E_OK){
         __leave;
      }

      // Allocate our buffers
      PVOID pvPadding = alloca(sizes.cbBlockSize);
      PVOID pvSignature = alloca(sizes.cbSecurityTrailer);
      // Best to copy the message buffer, since it is encrypted in place
      PVOID pvMessage = alloca(lSize);
      CopyMemory(pvMessage, pvData, lSize);

      SecBuffer secBuffer[3] = {0};
      // Set up the signature buffer
      secBuffer[0].BufferType = SECBUFFER_TOKEN;
      secBuffer[0].cbBuffer = sizes.cbSecurityTrailer;
      secBuffer[0].pvBuffer = pvSignature;
      // Set up the message buffer
      secBuffer[1].BufferType = SECBUFFER_DATA;
      secBuffer[1].cbBuffer = lSize;
      secBuffer[1].pvBuffer = pvMessage;
      // Set up the padding buffer
      secBuffer[2].BufferType = SECBUFFER_PADDING;
      secBuffer[2].cbBuffer = sizes.cbBlockSize;
      secBuffer[2].pvBuffer = pvPadding;
      // Set up buffer descriptor
      SecBufferDesc secBufferDesc;
      secBufferDesc.cBuffers = 3;
      secBufferDesc.pBuffers = secBuffer;
      secBufferDesc.ulVersion = SECBUFFER_VERSION;
      // Encrypt message
      ss = EncryptMessage( phContext, 0, &secBufferDesc, 0 );
      if(ss != SEC_E_OK){
         __leave;
      }

      // Send token
      SendData(&secBuffer[0].cbBuffer, sizeof(ULONG));
      SendData(secBuffer[0].pvBuffer, secBuffer[0].cbBuffer);

      // Send message
      SendData(&secBuffer[1].cbBuffer, sizeof(ULONG));
      SendData(secBuffer[1].pvBuffer, secBuffer[1].cbBuffer);

      // Send padding
      SendData(&secBuffer[2].cbBuffer, sizeof(ULONG));
      SendData(secBuffer[2].pvBuffer, secBuffer[2].cbBuffer);

      fSuccess = TRUE;
   }__finally{}

   return fSuccess;
}
```

Notice that this code is very similar to code that you are already familiar with from message signing. The next function, *GetEncryptedMessage*, is a complementary sample function to decrypt a message and return a buffer:

```
PVOID GetEncryptedMessage(CtxtHandle* phContext, PULONG plSize){
   PVOID pvMessage = NULL;

   __try{
      SECURITY_STATUS ss;

      ULONG lSigLen;
      PVOID pvDataSig;
      // Get signature length
      ULONG lTempSize = sizeof(lSigLen);
      ReceiveData(&lSigLen, &lTempSize);
      pvDataSig = alloca(lSigLen);
      // Get signature
      ReceiveData(pvDataSig, &lSigLen);

      ULONG lMsgLen;
      PVOID pvDataMsg;
      // Get message length
      lTempSize = sizeof(lMsgLen);
      ReceiveData(&lMsgLen, &lTempSize);
      pvDataMsg = alloca(lMsgLen);
      // Get message
      ReceiveData(pvDataMsg, &lMsgLen);

      ULONG lPadLen;
      PVOID pvDataPad;
      // Get padding length
      lTempSize = sizeof(lPadLen);
      ReceiveData(&lPadLen, &lTempSize);
      pvDataPad = alloca(lPadLen);
      // Get padding
      ReceiveData(pvDataPad, &lPadLen);

      SecBuffer secBuffer[3] = {0};
      // Set up signature buffer
      secBuffer[0].BufferType = SECBUFFER_TOKEN;
      secBuffer[0].cbBuffer = lSigLen;
      secBuffer[0].pvBuffer = pvDataSig;
      // Set up message buffer
      secBuffer[1].BufferType = SECBUFFER_DATA;
      secBuffer[1].cbBuffer = lMsgLen;
      secBuffer[1].pvBuffer = pvDataMsg;
      // Set up padding buffer
      secBuffer[2].BufferType = SECBUFFER_PADDING;
      secBuffer[2].cbBuffer = lPadLen;
      secBuffer[2].pvBuffer = pvDataPad;
      // Set up buffer descriptor
      SecBufferDesc secBufferDesc;
      secBufferDesc.cBuffers = 3;
      secBufferDesc.pBuffers = secBuffer;
      secBufferDesc.ulVersion = SECBUFFER_VERSION;
      ULONG lQual=0;
      ss = DecryptMessage( phContext, &secBufferDesc, 0, &lQual );
      if (ss != SEC_E_OK){
         __leave;
      }

      // Return a buffer that must be freed, containing message
      pvMessage = LocalAlloc(LPTR, secBuffer[1].cbBuffer);
      if (pvMessage != NULL){
         CopyMemory(pvMessage, secBuffer[1].pvBuffer,
            secBuffer[1].cbBuffer);
      }
   }__finally{}

   return (pvMessage);
```

```
}
```

Now that you are familiar with encrypting and signing messages, you have the tools to implement a full session with the SSPI. You know how to negotiate an authentication, and your server can impersonate the client. And you know how to send and receive data in a safe way.

The SSPIChat Sample Application

The SSPIChat sample application ("12 SSPIChat.exe") demonstrates all the SSPI-related technologies we have discussed so far, including client and server authentication negotiation, impersonation, message signing, and encryption. The source code and resource files for the sample application are in the 12-SSPIChat directory on the companion CD. Figure 12-6 shows the user interface for the SSPIChat sample application.

Figure 12-6. *User interface for the SSPIChat sample application*

To use the sample, you should run it more than once, either on the same machine or on different machines connected on a network. The sample application communicates using TCP/IP. You can select which security provider you want to use before initiating conversation. You can also select whether you want mutual authentication, encryption, or delegation.

The delegation feature causes the server to create a second chat window assuming the client's identity. The new window can then be used to act as a client to a third server, which will see it as the original client. (This feature will fail if the server machine has not been trusted for delegation.)

In the spirit of communication layer independence, the communication functionality in this sample application is abstracted to a class known as CTransport, which includes *SendData* and *ReceiveData* functions. These two functions are much like the pseudo-functions I have been using in the code fragments in this chapter.

I strongly suggest you become familiar with the sample code for this application before attempting to write SSPI code using SSL (which is covered later in this chapter). The SSPI programming model is complex, and it comes with a handful of gotchas. Comfort with the overall approach will make the SSL model much more palatable.

CryptoAPI

The CryptoAPI, or CAPI, provides a full set of functions for generating keys used to encrypt and decrypt data, as well as for exporting keys and sharing them securely. The CryptoAPI also includes full support for certificates and certificate management. The cryptography and certificate functions provided by the CryptoAPI are defined in WinCrypt.h; your code should include this header. You should also be sure to link with the Crypt32.lib library file.

The CryptoAPI provides such a rich set of features that you could use its functions to implement your own fully secure protocol for communicating over insecure network environments. However, the SSPI already does this for us for free. In fact, the SSPI uses the CryptoAPI under the covers to implement its cryptographic needs. If you are interested in creating your own secure protocol, you can use the CryptoAPI. However, for most projects you should use the known protocols supported by the SSPI.

In two instances it is valid to use the CryptoAPI because the SSPI is not able to do the job for you. They are as follows:

- **Non-session-related cryptography** This includes file cryptography or any other encryption of persistent data that is not being communicated in a session-oriented environment.
- **Certificate management** When we discuss SSL using the SSPI, you will see how critical certificates are to the protocol. Certificate management is performed by the CryptoAPI set.

Encryption for persistent storage is not a common need, and because it is fully supported by the Encrypted File System (EFS) technology of Windows 2000, I will not cover this particular use of the CryptoAPI here. If you are interested in using the CryptoAPI for encryption of this type, you can find a full description of its features in the Platform SDK documentation.

However, certificate management *is* an integral part of the SSL security protocol. In fact, before I can move into my discussion of SSL, I have to spend some time discussing certificate management using the CryptoAPI.

Certificate Stores

Windows manages certificates in terms of *stores*. The CryptoAPI allows you to manage many different kinds of stores, ranging from personal stores for a single user, to stores for machines, to temporary stores that exist only in memory. When you create a store, you can designate the medium on which it persists. Or it does not have to persist at all.

Stores contain certificates, and as you know, certificates are used for authentication, signing, and encryption. When dealing with SSL, however, you are unlikely to need to deal with any store other than the machine store (which you will use to store server certificates), or perhaps a personal store (which would be used to store client certificates). Allow me to describe several scenarios.

- **Scenario 1** You are designing a service that will communicate by using SSL, which is running as the LocalSystem account on a machine. Let's say that the client will authenticate the server, but not vice versa. You would most likely create a certificate (and matching private key) for this service and store them in the machine store for your host system. The service knows which certificate it is by its common name. When the service connects to a client, it uses the CryptoAPI to look up its certificate, and then it uses this certificate to initiate a communication session using SSPI and SSL. In this case, the server knows that it will look up the certificate in the machine store.

After authentication, the client has a copy of the certificate, communicated over the wire. Although the client is not sending a certificate back, it must still decrypt the information in the certificate to find out whether it has connected to the server it was looking for. With Windows 2000, SSL automatically verifies the certificate chain against your client's trusted CAs. It also does a comparison between the server name (as requested by the client) and the common name on the server's certificate.

Your client, however, can choose to use the CryptoAPI functions to open the certificate, extract the common name, and compare it against the known common name of the public certificate that it expected to receive. If the names don't match, the client can choose to close the connection, because the server might have been spoofed by a malicious third party. Although Windows 2000 performs this test for you, earlier versions of Windows will not, so it is a good skill to be familiar with.

Trust enters into the situation here, because you must be sure that the CA does not issue certificates with the same common name to multiple parties. Policy in this area could vary from one certificate provider to the next.

NOTE

Note that the common name of a certificate is simply a text string. However, remember that certificates are signed with the certificate authority's private key. This means that no information on the certificate has changed when you open it using the CA's known public key.

Browser software uses the common name of a certificate to perform a character-for-character comparison of the certificate and the URL to which the browser is connected. If the names do not match, the browser assumes that security has been breached.

- **Scenario 2** Everything is the same as Scenario 1, except that the server is not running in the LocalSystem account but rather is running in a special domain account created just for the service. In terms of certificates, the only major difference is that you (the server's administrator) would want to store a certificate in the personal store for the server's account. This is because the server's account might not be an administrator of its host machine, and thus would not have access to the machine certificate store. So you must know how to access personal accounts.

- **Scenario 3** Whether or not your server will be accessing machine accounts or personal accounts, a third scenario must be considered. If your server requires a certificate from your client, your client will have to follow a similar certificate-finding approach as your server.

  When a connection is made, your client will look up a certificate on its machine and use that certificate to initiate an SSL conversation. Since most client software is being run under an interactive user account, your client software will most likely have to know how to look up a certificate from a personal store, by common name, and then send it across the wire.

If you will be writing software to handle any of these scenarios, here are the tasks you must be able to perform with the CryptoAPI:

1. Open a certificate store.

2. Look up a certificate by common name.

3. Decrypt a certificate (communicated to you).

4. Look up a decrypted certificate's common name (and compare it to a known name).

You can perform all these tasks using the CryptoAPI. However, there is one implied task I have not mentioned that you must be able to do—get a certificate.

Getting a Certificate

Getting a certificate is a complex topic, largely because of the different ways that it can be performed. I will give you some starting points, but you will have to research your options and decide which is best for you. Here are the two main approaches:

1. Purchase a certificate from a public certificate authority.

2. Run your own certificate authority using Microsoft Certificate Services.

The first approach involves interacting with a third party such as VeriSign or Thawte (among many others) to get a certificate for your server software. These third parties will store information about your company as well as the uses for your certificate, and they will give you a certificate, signed by them, for a fee.

The advantage of this approach is that known certificate authorities are trusted by most operating systems by default. So the client software (or human user) does not have to go to any extra trouble to recognize your server's certificate.

Some of these public certificate authorities will happily give you a temporary certificate signed by a test authority (also owned by them), with which you can test their services. Some companies offer these temporary certificates for free.

The second approach involves running Microsoft Certificate Services. This approach is great, except that you must make sure your client software trusts the certificate authority that you set up. If your client and server are running in the same enterprise, you will most likely follow this approach. If your client and server run on the Internet, you can still take this approach, but your client software must know how to help the user establish trust with your CA.

Once you have a certificate, the certificate can be managed and moved from one system store to another using the Certificates snap-in for Microsoft Management Console (MMC). This allows you to test different scenarios with a single certificate. For example, you could create a server certificate, drag it into the machine's Personal certificates folder, and then test your server under the machine account. Later you might decide to run your server under a user account, at which time you would use the snap-in to drag the certificate to the Personal certificates folder for that user's account.

Using the CryptoAPI to Open a Certificate Store

Certificate stores are identified by a textual name. The personal store for a machine account or a user account is named "MY". You use *CertOpenStore* to open a certificate store:

```
HCERTSTORE WINAPI CertOpenStore(
   PCSTR      pszStoreProvider,
   DWORD      dwMsgAndCertEncodingType,
   HCRYPTPROV hCryptProv,
   DWORD      dwFlags,
   const void *pvPara);
```

This function is very flexible and will let you open and create stores in many ways. We will focus our discussion on the machine's personal store or a user account's personal store. Both of these stores are known as *system stores*. To open a system store, you pass CERT_STORE_PROV_SYSTEM as the *pszStoreProvider*

parameter. (Pass CERT_STORE_PROV_SYSTEM_A if you are using ANSI.) If you are interested in dealing with other certificate stores, such as stores kept only in memory, reference the Platform SDK documentation.

You'll always pass X509_ASN_ENCODING | PKCS_7_ASN_ENCODING for the *dwMsgAndCertEncodingType* parameter. At this time, no other encoding types are supported. You should pass NULL for the *hCryptProv* parameter to indicate usage of the default cryptographic provider.

The *dwFlags* parameter is where you indicate which type of store you want to open. Table 12-8 shows selected values for the *dwFlags* parameter.

Table 12-8. *Selected values for* CertOpenStore*'s* dwFlags *parameter*

Flag	Description
CERT_SYSTEM_STORE_CURRENT_SERVICE	Indicates a store for the account under which the current service is running
CERT_SYSTEM_STORE_CURRENT_USER	Indicates a store for the current user account of the calling code
CERT_SYSTEM_STORE_LOCAL_MACHINE	Indicates a store for the local machine

The *pvPara* parameter provides specific information used to find the store. For system stores, you pass a string indicating the name of the store. If you are using the personal store, you would use the text "MY". If you pass a name that is not recognized by the system, the system will create this store for you. This can be a convenient way of logically grouping certificates on a machine.

The *CertOpenStore* function returns an HCERTSTORE variable, which is a handle to a certificate store. When you are finished using the certificate store you should close this handle using the *CertCloseStore* function:

```
BOOL WINAPI CertCloseStore(
   HCERTSTORE hCertStore,
   DWORD      dwFlags);
```

You will typically pass zero for the *dwFlags* value.

Finding a Certificate

Certificates contain a fair amount of information, and the CryptoAPI allows you to look certificates up using just about any information stored in the certificate. However, most of the information in certificates is not unique, which creates ambiguities when you're conducting your lookup. Certificates are generally referenced by their common names, so you should look them up by their common names.

The common name of a certificate is an *attribute* of the certificate. When looking up a certificate in a store, you can use one or more of its attributes as criteria. You do this by creating an array of structures, one per attribute, and then passing them to *CertFindCertificateInStore*:

```
PCCERT_CONTEXT WINAPI CertFindCertificateInStore(
   HCERTSTORE     hCertStore,
   DWORD          dwCertEncodingType,
   DWORD          dwFindFlags,
   DWORD          dwFindType,
   const void     *pvFindPara,
   PCCERT_CONTEXT pPrevCertContext);
```

The *hCertStore* parameter is the handle to an open store containing the certificate. The *dwCertEncodingType* parameter should always be X509_ASN_ ENCODING | PKCS_7_ASN_ENCODING. You should pass zero for the *dwFindFlags* parameter for most situations. For exceptions, consult the Platform SDK documentation.

The *dwFindType* parameter indicates how you want to find a certificate. If you want to find a certificate by one or more attributes of the certificate, you should pass CERT_FIND_SUBJECT_ATTR.

NOTE

You can find a certificate using many other approaches as well. For example, if you are sure that your certificate is the only one in the open store, you can pass CERT_FIND_ANY for the *dwFindType* parameter. The search options are discussed in the Platform SDK documentation. For this chapter, I am focusing on approaches that are commonly used.

You pass the actual search criteria to the *pvFindPara* parameter. If you are using the CERT_FIND_SUBJECT_ATTR flag to find a certificate by looking up its attributes, you will pass a pointer to a CERT_RDN structure for this parameter.

CertFindCertificateInStore can be used to enumerate certificates in a store. The *pPrevCertContext* parameter indicates the last context found. You should pass NULL for the first certificate context, or when you are looking up only one certificate.

If your search is successful, *CertFindCertificateInStore* will return a pointer to a certificate context structure. When you are finished with the structure, you must free it by using *CertFreeCertificateContext*:

```
BOOL WINAPI CertFreeCertificateContext(
   PCCERT_CONTEXT pCertContext);
```

The CERT_RDN structure, which you pass as the *pvFindPara* structure when calling *CertFindCertificateInStore*, is defined as follows:

```
typedef struct _CERT_RDN {
   DWORD          cRDNAttr;
   PCERT_RDN_ATTR rgRDNAttr;
} CERT_RDN;
```

The *cRDNAttr* member indicates the number of attributes you are using to find the certificate. The *rgRDNAttr* member is a pointer to an array of CERT_RDN_ATTR structures that are filled with attribute information. Following is the definition of CERT_RDN_ATTR:

```
typedef struct _CERT_RDN_ATTR {
   PSTR                pszObjId;
   DWORD               dwValueType;
   CERT_RDN_VALUE_BLOB Value;
} CERT_RDN_ATTR;
```

This structure is designed to be very flexible because it needs to contain the many types of data that can make up an attribute in a certificate. The *pszObjId* member is the ID of the attribute we are concerned with. For the common name of a certificate, you should use the value szOID_COMMON_NAME. The *dwValueType* member indicates the data type of this value. For the common name, set this value to CERT_RDN_PRINTABLE_STRING. (Note that you must always use ANSI strings with certificate common names.)

The *Value* member is a blob structure that contains two members: *cbData* and *pbData*. They are the size of the data and a pointer to the data, respectively.

NOTE

Some of the attribute types and their data types are discussed in the Platform SDK documentation. If the structures you want to populate are not described in the documentation, you might have to use another approach to find a certificate. For example, you could open a certificate and look at its RDN values so that you can use similar data for looking up certificates in the future.

The following code sample shows the use of the certificate functions we have discussed so far. It opens the personal store of the local machine and looks up a certificate whose common name is "Jason's Test Certificate".

NOTE

Although "Jason's Test Certificate" is a perfectly legal common name for a certificate, it does not represent common usage. Usually, a certificate is named after the network location of the server. This way client software can compare the certificate's common name to the network location with which it is attempting to initiate a secure session.

```
// Open the personal certificate store for the local machine
HCERTSTORE hMyCertStore =
   CertOpenStore(CERT_STORE_PROV_SYSTEM_A,
      X509_ASN_ENCODING | PKCS_7_ASN_ENCODING,
      0,
      CERT_SYSTEM_STORE_LOCAL_MACHINE,
      "MY");
if(hMyCertStore==NULL){
   // Error
}

// Fill in an attribute structure for the certificate common name
PSTR pszCommonName = "Jason's Test Certificate";
CERT_RDN_ATTR certRDNAttr[1];
certRDNAttr[0].pszObjId = szOID_COMMON_NAME;
certRDNAttr[0].dwValueType = CERT_RDN_PRINTABLE_STRING;
certRDNAttr[0].Value.pbData = (PBYTE) pszCommonName;
certRDNAttr[0].Value.cbData = lstrlen(pszCommonName);
CERT_RDN certRDN = {1, certRDNAttr};

// Find the certificate context
PCCERT_CONTEXT pCertContext =
   CertFindCertificateInStore(hMyCertStore,
      X509_ASN_ENCODING | PKCS_7_ASN_ENCODING,
      0,
      CERT_FIND_SUBJECT_ATTR,
      &certRDN,
      NULL);

if (pCertContext == NULL){
   // Error
}
```

The code that performs the task of looking up a certificate is fairly short and simple, but you might be daunted by the new concepts behind the code. If a server is authenticating itself to a client and is not concerned with client authentication, the previous code is essentially all the certificate code the server need worry about. The rest of the job will be handled by SSL, which I will discuss shortly.

Client or server code that is authenticating a client certificate, however, must open a certificate context and find information about the certificate. Specifically, the code has to identify the certificate by using its common name.

Reading Certificate Information

Your software will usually read certificate information that it has received from a remote principle using SSL. However, this is not always the case, and your software can read certificate information from a certificate that you received from *CertFindCertificateInStore*. Either way, the approach is the same, and you will be starting with a PCCERT_CONTEXT structure returned from SSL or the CryptoAPI. Here are the steps the system must take to read attribute information from a certificate:

1. Decode the "subject" blob of the certificate.
2. Find the RDN attribute for the attribute you wish to read.
3. Extract the attribute information.

Although there are sufficient CryptoAPI functions to allow you to take this three-step approach, you will also find that there are higher-level functions for extracting most useful information from a certificate that avoid the hassle. To extract the common name (or any other name information) from a certificate, you use *CertGetNameString*:

```
DWORD WINAPI CertGetNameString(
   PCCERT_CONTEXT pCertContext,
   DWORD          dwType,
   DWORD          dwFlags,
   void           *pvTypePara,
   PTSTR          pszNameString,
   DWORD          cchNameString);
```

The *pCertContext* parameter is a pointer to a certificate context for which you wish to extract a name. The *dwType* parameter indicates what type of name you wish to return. To retrieve a name that is an attribute of the certificate, you would pass CERT_NAME_ATTR_TYPE. To get the friendly name of the certificate, you would pass CERT_NAME_FRIENDLY_DISPLAY_ID.

The value that you pass to *dwType* also indicates the purpose of the data that you pass as the *pvTypePara* parameter. For example, in the case of CERT_NAME_FRIENDLY_DISPLAY_ID, you would pass NULL. In the case of CERT_NAME_ATTR_TYPE, you would pass the string representing the object identifier (OID) of the object you want returned; szOID_COMMON_NAME would be used to get the common name string. For more *dwType* and *pvTypePara* usage combinations, see the Platform SDK documentation.

You will typically pass zero for the *dwFlags* parameter of *CertGetNameString*; however, if you want information related to the "issuer" of the certificate rather than the certificate subject itself, you can pass CERT_NAME_ISSUER_FLAG for *dwFlags*.

The *pszNameString* and *cchNameString* parameters of *CertGetNameString* indicate a buffer to retrieve the string name and the size of the buffer, respectively. If you pass NULL and zero for these parameters, *CertGetNameString* returns the required size of the buffer.

The following code fragment shows how this function is used to retrieve the common name of a certificate context:

```
TCHAR szCommonName[1024];

ULONG lBytes =
   CertGetNameString(pCertContext,
      CERT_NAME_ATTR_TYPE, 0,
      szOID_COMMON_NAME,
      szCommonName, 1024);
if(lBytes != 1){   // If not empty string
```

```
    // Do something with the string
}
```

Using the concepts presented in this chapter, you will be able to manage certificates sufficiently for your needs in communicating via SSL. You will also find a complete implementation of the topics discussed in this chapter in the SSLChat sample application, which I cover later in this chapter.

[Previous] [Next]

SSL with the SSPI

We have discussed the SSPI and authentication with Windows trustee accounts using NTLM and Kerberos. You can also use SSPI to authenticate via digital certificates. You do this by using the SSPI and the SChannel security provider for SSL. If you plan to use SSL, be sure to include both the Security.h and SChannel.h header files in your code. You should also link your project with the Secur32.lib library file.

Unlike the protocols we have discussed so far, SSL focuses less on authenticating a client to a server and typically starts by authenticating the server to the client. It is possible to authenticate the client to the server as well, in which case impersonation is an option. Let's look at the more common scenario first.

A client wants to attach to some site, most likely on the Internet. Before it sends the site critical information, it wants proof that it is communicating with the desired entity. The server sends the client its certificate. Certificates are signed and contain the public key of a public/private key pair.

The client can tell from the certificate who issued it, and this helps the client decide whether to trust this certificate. If it trusts the certificate's CA, the client can trust that the information within the certificate is valid. After the certificate is verified, the client can read the certificate information, such as the URL of the server, and compare the information to the server it is communicating with. If the comparison is favorable, the client can be comfortable that it has contacted the correct server.

> **NOTE**
>
> There are many issues with the certificate model that have not yet been worked out by the industry. How can client software really trust the information in a certificate? Is there a good way to shift the burden of trust from the client software to the human user? How is the information relayed? What root authorities are trusted? Did the human user choose these authorities? Should all authorities be trusted for any type of certificate? Should certificate authorities also be industry authorities of specific groups such as doctors' offices or book retailers? Answers to these questions will surface as the technology matures. For now, I will focus on the technology.

Depending on the environment you are designing for secure communication, you might decide to be your own certificate authority. Microsoft Certificate Services allows you to issue certificates yourself, and as long as your client trusts your CA, you can create certificates that meet your exact standards.

If you will be your own certificate authority, you could even go so far as to write client software that contains a hard-coded root certificate for trusting your CA. When a client connects to your server, you could use this implicit trust to begin a session for the client, and then generate a certificate for the client (where the client generates the private key and communicates the public key to your server). Your server uses the public key to generate a certificate from your own CA, and then it stores the information. Now each time this client connects to your server, you can use each other's public keys, in the form of certificates, to quickly authenticate each other, negotiate a session key, and get down to business.

Assuming that the client has authenticated the server's certificate, the client can extract the public key from the certificate and use it to encrypt a symmetric session key, which it sends to the server. The server is the only entity holding the matching private key, so it can decrypt the session key. Now the client and server can communicate, safely assured that they are communicating with the proper party.

In the standard scenario requiring secure communication, a browser or other client would send credit card or other sensitive information across the wire to a server. You can use the SSPI to implement SSL for transactions such as these.

Programming for SSL

Much of what you already know about the SSPI applies to communication with SSL. This section assumes knowledge of the SSPI with Kerberos and NTLM; in this section, we will build upon much of the information presented in this chapter thus far.

As with the NTLM and Kerberos protocols, you call *AcquireCredentialsHandle* on both the client and server sides of the communication. However, with SSL you must establish a certificate for at least one side of the communication by passing in an SSP-specific authentication structure, SCHANNEL_CRED. The SCHANNEL_CRED structure is similar to the SEC_WINNT_AUTH_ IDENTITY structure (discussed earlier in this chapter), which provides a username, password, and domain name, except that SCHANNEL_CRED includes certificate information. The SCHANNEL_CRED structure is defined as follows:

```
typedef struct _SCHANNEL_CRED {
   DWORD           dwVersion;
   DWORD           cCreds;
   PCCERT_CONTEXT  *paCred;
   HCERTSTORE      hRootStore;
   DWORD           cMappers;
   struct _HMAPPER **aphMappers;
   DWORD           cSupportedAlgs;
   ALG_ID *        palgSupportedAlgs;
   DWORD           grbitEnabledProtocols;
   DWORD           dwMinimumCipherStrength;
   DWORD           dwMaximumCipherStrength;
   DWORD           dwSessionLifespan;
   DWORD           dwFlags;
   DWORD           reserved;
} SCHANNEL_CRED;
```

Typically the server uses a certificate to initialize SCHANNEL_CRED before passing it to *AcquireCredentialsHandle*. Here is a code fragment showing how to do this:

```
// Open the personal certificate store for the local machine
HCERTSTORE hMyCertStore =
   CertOpenStore(
      CERT_STORE_PROV_SYSTEM_A,
      X509_ASN_ENCODING | PKCS_7_ASN_ENCODING,
      0,
      CERT_SYSTEM_STORE_LOCAL_MACHINE,
      "MY");
if(hMyCertStore==NULL){
   // Error
}

// Fill in an attribute structure for the certificate common name
PSTR pszCommonName = "Jason's Test Certificate";
CERT_RDN_ATTR certRDNAttr[1];
certRDNAttr[0].pszObjId = szOID_COMMON_NAME;
certRDNAttr[0].dwValueType = CERT_RDN_PRINTABLE_STRING;
```

```
certRDNAttr[0].Value.pbData = (PBYTE) pszCommonName;
certRDNAttr[0].Value.cbData = lstrlen(pszCommonName);
CERT_RDN certRDN = {1, certRDNAttr};

// Find the certificate context
PCCERT_CONTEXT pCertContext =
   CertFindCertificateInStore(
      hMyCertStore,
      X509_ASN_ENCODING | PKCS_7_ASN_ENCODING,
      0,
      CERT_FIND_SUBJECT_ATTR,
      &certRDN,
      NULL);
if (pCertContext == NULL){
   // Error
}

// Fill in an SCHANNEL_CRED variable with certificate information
SCHANNEL_CRED sslCredentials = {0};
sslCredentials.dwVersion = SCHANNEL_CRED_VERSION;
sslCredentials.cCreds = 1;
sslCredentials.paCred = &pCertContext;
sslCredentials.grbitEnabledProtocols = SP_PROT_SSL3;

// Get a credentials handle
CredHandle hCredentials;
TimeStamp tsExpires;
SECURITY_STATUS ss =
   AcquireCredentialsHandle(
      NULL,
      UNISP_NAME,
      SECPKG_CRED_INBOUND,
      NULL,
      &sslCredentials,
      NULL, NULL,
      &hCredentials,
      &tsExpires );
if(ss != SEC_E_OK){
   // Error
}
```

NOTE

Notice that this code is using certificate functions discussed in the last section to retrieve a PCCERT_CONTEXT structure for a certificate whose common name is "Jason's Test Certificate".

NOTE

When you use a certificate to get credentials by calling *AcquireCredentialsHandle*, the certificate must match the implied purpose. For example, for a server, the certificate must be signed to allow server authentication. Otherwise *AcquireCredentialsHandle* returns an error indicating unknown credentials.

The client might start communication with a certificate by using the preceding approach. It is also common for a client to use a NULL certificate that indicates anonymous authentication to the server. The following code fragment shows how to do this:

```
SCHANNEL_CRED sslCredentials = {0};
sslCredentials.dwVersion = SCHANNEL_CRED_VERSION;
sslCredentials.grbitEnabledProtocols = SP_PROT_SSL3;
sslCredentials.dwFlags =
```

```
    SCH_CRED_NO_DEFAULT_CREDS|SCH_CRED_MANUAL_CRED_VALIDATION;

 CredHandle hCredentials;
 TimeStamp tsExpires;
 SECURITY_STATUS ss =
    AcquireCredentialsHandle(
       NULL,
       UNISP_NAME,
       SECPKG_CRED_OUTBOUND,
       NULL,
       &sslCredentials,
       NULL,
       NULL,
       &hCredentials,
       &tsExpires);
 if(ss != SEC_E_OK){
    // Error
 }
```

So far, using SSL is not too different from the authentication process we are used to. However, from this point forward things change somewhat, even though the client still calls *InitializeSecurityContext* and the server still calls *AcceptSecurityContext*.

SSL Is a Stream

Unlike the other protocols we have discussed, for which you are supposed to wrap the blobs passed to and from *InitializeSecurityContext* and *AcceptSecurityContext* before communicating them, SSL expects your software *not* to wrap the blobs at all! When you receive a blob from *InitializeSecurityContext*, instead of sending size information and then the data, you send only the data verbatim.

> **NOTE**
>
> Although SSL expects to send and receive data verbatim, there is nothing stopping you from implementing a "meta protocol" to indicate the blob sizes as you send them. Doing this can drastically simplify your SSL code, though technically you won't be following the HTTPS specification. If you pass extra size data across the wire, your client or server probably won't be able to communicate with a client or server that has implemented HTTPS, which is commonly used by Web browsers and Web servers.

You might be wondering how the receiving side of the communication knows how much data to read, and the answer is that it doesn't. With SSL your communication is a *stream*. The following scenario illustrates the effect SSL has on the flow of information. It shows how you read information until a message is complete.

1. One side (for example, the client side) sends a blob across the wire, unmodified.
2. The receiving side reads as much information as is available and passes it to its "blob-handling" function. (In our example, the server passes the data to *AcceptSecurityContext*.)
3. If the receiving side did not read enough information from the wire to finish a leg of the transaction, the blob-handling function reports SEC_E_INCOMPLETE_MESSAGE.
4. If the incomplete message value is returned, you must save the data you already have and go back to the wire to receive as much additional data as you can until you get a return value other than SEC_E_INCOMPLETE_MESSAGE.

You might have also realized that the streaming nature of SSL creates a complication other than the possibility of receiving too little data. Your software can read *too much* information across the wire. In this

scenario, *AcceptSecurityContext* or *InitializeSecurityContext* report that you have passed extra data, which must be saved for use in the future (presumably after reading more data across the wire).

So as you can see, there are more scenarios to watch for with SSL. Here are two situations you must be aware of at all times:

1. You might have too little information for an SSPI function to process, so you need to receive more data.

2. You might have too much information for an SSPI function, so you need to save the information for a future call to an SSPI function.

    A. You must get more information from the wire.

    B. You must not get more information from the wire.

Notice that the second scenario has two subscenarios. Scenario 2A is somewhat to be expected, but scenario 2B might come as a surprise. There are cases in which *AcceptSecurityContext* or *InitializeSecurityContext* report that you have passed in extra data and that a "continue" is necessary. And yet in these cases the blob-handling function might not report any blob data to send back across the wire. So you must modify the buffer you just passed in, and then pass it back to the blob-handling function again.

In my opinion, this scenario of having too much data should never have been exposed to the application via the SSPI, because *AcceptSecurityContext* and *InitializeSecurityContext* have all the information they need to process the data. The functions would need to stop only when there is not enough data to process a complete message.

This problem of handling extra data is not as easily solved as you might think. Imagine this scenario: You receive a blob on the wire that is internally broken into three "sections" of which the last section is not complete, and you need to get more data from the wire. You pass this blob to *AcceptSecurityContext* (for example), and it processes the first two sections, internal to the blob. When it finishes, it finds that the last section is not complete and that it needs more data. However, it doesn't want your application to go back to the wire, get more data, and send the *whole* blob back to the blob-handling function since it has already handled the first two sections internal to the blob.

To avoid this situation, the designers of the system decided that if any section or subsection internal to a blob is processed, *AcceptSecurityContext* and *InitializeSecurityContext* would return regardless of whether there is sufficient extra data for more processing. This way you can be aware of the extra data, adjust your buffers, and recall the function.

Because of the complications surrounding extra data, as well as the possibility of reading too little data to be useful to the SSPI, it is best to allocate and use a single buffer for all of your communication with a client. This buffer should be large enough to hold the largest single message transferred across the wire with the SSL protocol. To find out this size, you should call *QuerySecurityPackageInfo* before beginning to communicate:

```
SECURITY_STATUS QuerySecurityPackageInfo(
   SEC_CHAR    *pszPackageName,
   PSecPkgInfo *ppPackageInfo);
```

This function returns a buffer containing a SecPkgInfo structure, which you must free using *FreeContextBuffer* when you are finished with it:

```
typedef struct _SecPkgInfo {
   ULONG    fCapabilities;  // Capability flags for package
   USHORT   wVersion;       // Version of driver
   USHORT   wRPCID;         // Used by the system
```

```
    ULONG    cbMaxToken;     // Max message size for package
    SEC_CHAR *Name;          // Text name
    SEC_CHAR *Comment;       // Comment
} SecPkgInfo;
```

As you can see, *QuerySecurityPackageInfo* returns interesting information in the SecPkgInfo structure. However, you are typically only going to need the *cbMaxToken* value to indicate the minimum size of the buffer that you should be using when negotiating authentication with SSL. In our previous discussion of the SSPI, I implemented functions to perform the authentication handshake between the client and server. The following code shows the functions that perform similar tasks for SSL. You will notice that these functions take a pointer to a buffer and a size of that buffer. They also take a parameter that indicates how much data is already in the buffer. This parameter is used to accommodate the extra data scenarios that arise from SSL. Here is *SSLServerHandshakeAuth*:

```
BOOL SSLServerHandshakeAuth(
   CredHandle* phCredentials,
   PULONG plAttributes,
   PCtxtHandle phContext,
   PBYTE pbExtraData,
   PULONG pcbExtraData,
   ULONG lSizeExtraDataBuf){

   BOOL fSuccess = FALSE;

   __try
   {
      // Set up a buffer in terms of local variables
      ULONG lEndBufIndex = *pcbExtraData;
      ULONG lBufMaxSize = lSizeExtraDataBuf;
      PBYTE pbTokenBuf = pbExtraData;

      // Declare in and out buffers
      SecBuffer secBufferIn[3]={0};
      SecBufferDesc secBufDescriptorIn;

      SecBuffer secBufferOut;
      SecBufferDesc secBufDescriptorOut;

      // Set up loop state information
      BOOL fFirstPass = TRUE;
      SECURITY_STATUS ss = SEC_I_CONTINUE_NEEDED;
      while (ss == SEC_I_CONTINUE_NEEDED ||
         ss == SEC_E_INCOMPLETE_MESSAGE){
         // How much data can be read per pass
         ULONG lReadBuffSize;
         // Reset this if we are not doing an "incomplete" loop
         if (ss != SEC_E_INCOMPLETE_MESSAGE){
            // Reset state for another "blob exchange"
            lEndBufIndex =0;
            lReadBuffSize = lBufMaxSize;
         }

         // Receive blob data from client
         ReceiveData(pbTokenBuf+lEndBufIndex, &lReadBuffSize);

         // Here is how much we have read to date
         lEndBufIndex += lReadBuffSize;

         // Set up our in buffers
         secBufferIn[0].BufferType = SECBUFFER_TOKEN;
         secBufferIn[0].cbBuffer = lEndBufIndex;
         secBufferIn[0].pvBuffer = pbTokenBuf;

         // This becomes a SECBUFFER_EXTRA buffer to let
```

```
// us know if we have extra data afterward
secBufferIn[1].BufferType = SECBUFFER_EMPTY;
secBufferIn[1].cbBuffer = 0;
secBufferIn[1].pvBuffer = NULL;

// Set up in buffer descriptor
secBufDescriptorIn.ulVersion = SECBUFFER_VERSION;
secBufDescriptorIn.cBuffers = 2;
secBufDescriptorIn.pBuffers = secBufferIn;

// Set up out buffer (allocated by the SSPI)
secBufferOut.BufferType = SECBUFFER_TOKEN;
secBufferOut.cbBuffer = 0;
secBufferOut.pvBuffer = NULL;

// Set up out buffer descriptor
secBufDescriptorOut.ulVersion = SECBUFFER_VERSION;
secBufDescriptorOut.cBuffers = 1;
secBufDescriptorOut.pBuffers = &secBufferOut;

// This inner loop handles the "continue case" in which there
// is no blob data to be sent. In this case, there are still
// more "sections" in our last blob entry that must be processed.
BOOL fMoreSections;
// This loop processes all complete "sections" of data in buffer
do{
   fMoreSections  = FALSE;
   // Blob processing
   ss =
      AcceptSecurityContext(
         phCredentials,
         fFirstPass ? NULL : phContext,
         &secBufDescriptorIn,
         *plAttributes|
         ISC_REQ_ALLOCATE_MEMORY|ISC_REQ_STREAM,
         SECURITY_NATIVE_DREP,
         phContext,
         &secBufDescriptorOut,
         plAttributes,
         NULL);
   // Are there more "sections" to process?
   if ((ss == SEC_I_CONTINUE_NEEDED) &&
      (secBufferOut.cbBuffer == 0)){
      fMoreSections = TRUE; //Set state to loop
      // This is how much data was left over
      ULONG lExtraData = secBufferIn[1].cbBuffer;
      // Let's move this data back to the beginning of our buffer
      MoveMemory(pbTokenBuf,
         pbTokenBuf+(lEndBufIndex - lExtraData), lExtraData);
      // This is how much data is in our buffer
      lEndBufIndex = lExtraData;

      // Let's reset input buffers
      secBufferIn[0].BufferType = SECBUFFER_TOKEN;
      secBufferIn[0].cbBuffer = lEndBufIndex;
      secBufferIn[0].pvBuffer = pbTokenBuf;

      secBufferIn[1].BufferType = SECBUFFER_EMPTY;
      secBufferIn[1].cbBuffer = 0;
      secBufferIn[1].pvBuffer = NULL;
   }
}while(fMoreSections);

// This is how much data our next read from the wire
// can bring in without overflowing our buffer
lReadBuffSize = lBufMaxSize - lEndBufIndex;
```

```
         if (ss != SEC_E_INCOMPLETE_MESSAGE){
            // We are never on our first pass again
            fFirstPass = FALSE;
         }

         // Was there data to be sent?
         if (secBufferOut.cbBuffer != 0){
            // Send it then
            ULONG lOut = secBufferOut.cbBuffer;
            SendData(secBufferOut.pvBuffer, lOut);
            // And free up that out buffer
            FreeContextBuffer(secBufferOut.pvBuffer);
         }
      }

      if (ss == SEC_E_OK){
         // If there is extra data, this is encrypted application
         // layer data. We'll put it in a buffer and the application
         // layer can later decrypt it with DecryptMessage.
         int nIndex = 1;
         while(secBufferIn[nIndex].BufferType
            != SECBUFFER_EXTRA && (nIndex-- != 0));

         if ((nIndex != -1) && (secBufferIn[nIndex].cbBuffer != 0)){
            *pcbExtraData = secBufferIn[nIndex].cbBuffer;
            PBYTE pbTempBuf = pbTokenBuf;
            pbTempBuf += (lEndBufIndex - *pcbExtraData);
            MoveMemory(pbExtraData, pbTempBuf, *pcbExtraData);
         } else *pcbExtraData = 0;
         fSuccess = TRUE;
      }

   }__finally{}

   return (fSuccess);
}
```

Here is *SSLClientHandshakeAuth*:

```
BOOL SSLClientHandshakeAuth(
   CredHandle* phCredentials,
   CredHandle* phCertCredentials,
   PULONG plAttributes,
   CtxtHandle* phContext,
   PTSTR pszServer,
   PBYTE pbExtraData,
   PULONG pcbExtraData,
   ULONG lSizeExtraDataBuf)
{
   BOOL fSuccess = FALSE;

   __try
   {
      // Set up our own copy of the credentials handle
      CredHandle credsUse;
      CopyMemory(&credsUse, phCredentials, sizeof(credsUse));

      // Set up buffer in terms of local variables for readability
      ULONG lEndBufIndex = *pcbExtraData;
      ULONG lBufMaxSize = lSizeExtraDataBuf;
      PBYTE pbData = pbExtraData;

      // Declare in and out buffers
      SecBuffer secBufferOut;
      SecBufferDesc secBufDescriptorOut;
```

```
SecBuffer secBufferIn[2];
SecBufferDesc secBufDescriptorIn;

// Set up loop state information
BOOL fFirstPass = TRUE;
SECURITY_STATUS ss = SEC_I_CONTINUE_NEEDED;
while ((ss == SEC_I_CONTINUE_NEEDED) ||
   (ss == SEC_E_INCOMPLETE_MESSAGE)){
   // How much data can we read for a pass
   ULONG lReadBuffSize;
   // Reset if we are not doing an "incomplete" loop
   if (ss != SEC_E_INCOMPLETE_MESSAGE){
      // Reset state for another blob exchange
      lEndBufIndex =0;
      lReadBuffSize = lBufMaxSize;
   }

   // Some stuff we only do after the first pass
   if (!fFirstPass){
      // Receive as much data as we can

      ReceiveData(pbData+lEndBufIndex, &lReadBuffSize);

      // This is how much data we have so far
      lEndBufIndex += lReadBuffSize;

      // Set up in buffer with our current data
      secBufferIn[0].BufferType = SECBUFFER_TOKEN;
      secBufferIn[0].cbBuffer = lEndBufIndex;
      secBufferIn[0].pvBuffer = pbData;

      // This becomes a SECBUFFER_EXTRA buffer to let us
      // know if we have extra data afterward
      secBufferIn[1].BufferType = SECBUFFER_EMPTY;
      secBufferIn[1].cbBuffer = 0;
      secBufferIn[1].pvBuffer = NULL;

      // Set up in buffer descriptor
      secBufDescriptorIn.cBuffers = 2;
      secBufDescriptorIn.pBuffers = secBufferIn;
      secBufDescriptorIn.ulVersion = SECBUFFER_VERSION;
   }

   // Set up out buffer (allocated by SSPI)
   secBufferOut.BufferType = SECBUFFER_TOKEN;
   secBufferOut.cbBuffer = 0;
   secBufferOut.pvBuffer = NULL;

   // Set up out buffer descriptor
   secBufDescriptorOut.cBuffers = 1;
   secBufDescriptorOut.pBuffers = &secBufferOut;
   secBufDescriptorOut.ulVersion = SECBUFFER_VERSION;

   // This inner loop handles the "continue case" in which there is
   // no blob data to be sent. In this case, there are still more
   // "sections" in our last blob entry that must be processed.
   BOOL fNoOutBuffer;
   do {
      fNoOutBuffer = FALSE;
      // Blob processing
      ss =
         InitializeSecurityContext(
            &credsUse,
            fFirstPass ? NULL : phContext,
            fFirstPass ? pszServer : NULL,
```

```
                *plAttributes|
                ISC_REQ_ALLOCATE_MEMORY|ISC_REQ_STREAM,
                0,
                SECURITY_NATIVE_DREP,
                fFirstPass ? NULL : &secBufDescriptorIn,
                0,
                phContext,
                &secBufDescriptorOut,
                plAttributes,
                NULL);
            // Are there more sections to process?
            if ((ss == SEC_I_CONTINUE_NEEDED) &&
               (secBufferOut.cbBuffer == 0)){
               fNoOutBuffer = TRUE; //Set state to loop
               // Here is how much data was left over
               ULONG lExtraData = secBufferIn[1].cbBuffer;
               // We want to move this data back to the
               // beginning of our buffer
               MoveMemory(pbData,
                  pbData+(lEndBufIndex - lExtraData), lExtraData);
               // Now we have a new lEndBufIndex
               lEndBufIndex = lExtraData;

               // Let's reset input buffers
               secBufferIn[0].BufferType = SECBUFFER_TOKEN;
               secBufferIn[0].cbBuffer = lEndBufIndex;
               secBufferIn[0].pvBuffer = pbData;

               secBufferIn[1].BufferType = SECBUFFER_EMPTY;
               secBufferIn[1].cbBuffer = 0;
               secBufferIn[1].pvBuffer = NULL;
            }
            if (ss == SEC_I_INCOMPLETE_CREDENTIALS){
               // Server requested credentials.
               // Copy credentials with certificate.
               // Normally, we would call AcquireCredentialsHandle here
               // to pick up new credentials.
               // However, we have already passed
               // in certificate credentials in this sample function.
               CopyMemory(&credsUse, phCertCredentials, sizeof(credsUse));
               // No input needed this pass
               secBufDescriptorIn.cBuffers = 0;
               // Keep on truckin'
               fNoOutBuffer = TRUE; //Set state to loop
            }
         } while(fNoOutBuffer);

         // This is how much data our next read from the wire
         // can bring in without overflowing our buffer
         lReadBuffSize = lBufMaxSize - lEndBufIndex;

         // Was there data to be sent?
         if (secBufferOut.cbBuffer!=0){
            // Send it then
            ULONG lOut = secBufferOut.cbBuffer;
            SendData(secBufferOut.pvBuffer, lOut);
            // And free up that out buffer
            FreeContextBuffer( secBufferOut.pvBuffer );
         }

         if (ss != SEC_E_INCOMPLETE_MESSAGE){
            fFirstPass = FALSE;
         }
      }

      if(ss == SEC_E_OK){
```

```
            int nIndex = 1;
            while(secBufferIn[nIndex].BufferType
               != SECBUFFER_EXTRA && (nIndex-- != 0));

            if((nIndex != -1) && (secBufferIn[nIndex].cbBuffer != 0)){
               *pcbExtraData = secBufferIn[nIndex].cbBuffer;
               PBYTE pbTempBuf = pbData;
               pbTempBuf += (lEndBufIndex - *pcbExtraData);
               MoveMemory(pbExtraData, pbTempBuf, *pcbExtraData);
            } else *pcbExtraData = 0;
            fSuccess = TRUE;
         }
      } __finally{}

      return (fSuccess);
   }
```

These functions differ from their Kerberos and NTLM counterparts in a couple of ways. Both of these functions check for a return value of SEC_E_INCOMPLETE_MESSAGE and continue to build the message buffer if this happens.

These functions also differ from their counterparts in that the client function takes two credentials handles rather than one. One is the anonymous credentials handle, and the other is a credentials handle created with a client certificate. (You could pass the address of the anonymous handle for both if you did not have a client certificate.) This is because the client presents its anonymous credentials first and then switches to the certificate only if the server requests mutual authentication. The client detects this request by a return value of SEC_I_INCOMPLETE_CREDENTIALS.

NOTE

In a real-world client application, it is likely that the client would not search for a certificate and call *AcquireCredentialsHandle* until after it had found that the server was requiring mutual authentication. My sample function is assuming the possibility of a client certificate from the beginning, primarily as a means of simplifying the already complex logic of the sample.

NOTE

As you can see, the details of SSL complicate the situation dramatically! However, once your authentication has completed, one or both sides of the communication hold certificate information from the other side. You can retrieve the certificate information by calling the function *QueryContextAttributes* with the completed context and passing the SECPKG_ATTR_REMOTE_CERT_CONTEXT value plus a pointer to a PCCERT_CONTEXT structure.

When retrieving a certificate context from an SSL context, it is your responsibility to free the certificate context when you are finished with it. Use *CertFreeCertificateContext* to do this.

After you have retrieved the certificate context, you can use the CryptoAPI functions we discussed in the previous section to extract information from the remote principal's certificate.

Here is a code sample that shows how to retrieve certificate information. This code assumes a completed context and ends up with the common name of the remote certificate stored in the buffer named *szNameBuf*.

```
// Get server's certificate
PCCERT_CONTEXT pCertContext = NULL;
ss = QueryContextAttributes(&hContext, SECPKG_ATTR_REMOTE_CERT_CONTEXT,
   (PVOID)&pCertContext);
```

```
if(ss != SEC_E_OK){
   // Error
}


// Find the size of the block that we are decoding from the certificate
ULONG lSize = 0;
CryptDecodeObject(
   X509_ASN_ENCODING | PKCS_7_ASN_ENCODING,
   X509_NAME,
   pCertContext->pCertInfo->Subject.pbData, // Data to decode
   pCertContext->pCertInfo->Subject.cbData, // Size of data
   0, NULL, &lSize);

// Allocate memory for the block
CERT_NAME_INFO* pcertNameInfo = (CERT_NAME_INFO*)alloca(lSize);

// Actually decode the subject block from the certificate
if(!CryptDecodeObject(
   X509_ASN_ENCODING | PKCS_7_ASN_ENCODING,
   X509_NAME,
   pCertContext->pCertInfo->Subject.pbData, // Data to decode
   pCertContext->pCertInfo->Subject.cbData, // Size of data
   0, pcertNameInfo, &lSize)){
   // Error
}

// Look up the common name attribute of the certificate
PCERT_RDN_ATTR pcertRDNAttr =
   CertFindRDNAttr(szOID_COMMON_NAME, pcertNameInfo);
if(pcertRDNAttr == NULL){
   // Error
}

// Translate the information in the CERT_RDN_ATTR structure to a string
// to use in your application
TCHAR szNameBuf[1024];
if(CertRDNValueToStr(CERT_RDN_PRINTABLE_STRING,
   &pcertRDNAttr->Value, szNameBuf, 1024) == 0){
   // Error
}
```

Encrypting and Decrypting Messages

As you might imagine, encryption is also somewhat more complex because of the stream-oriented approach of SSL. However, once you are comfortable receiving extra data from the receiving end of a function set, encryption and decryption are doable tasks.

The following two functions demonstrate the sending and receiving ends of an encrypted message being sent. The first function is *SendEncryptedMessage*:

```
BOOL SendEncryptedMessage(CtxtHandle* phContext,
   PVOID pvData, ULONG lSize){
   BOOL fSuccess = FALSE;

   __try
   {
      SecPkgContext_StreamSizes Sizes;

      // Get stream data properties
      SECURITY_STATUS ss =
         QueryContextAttributes(
            phContext,
```

```
            SECPKG_ATTR_STREAM_SIZES,
            &Sizes);
      if (ss != SEC_E_OK){
         __leave;
      }

      // Can we handle this much data?
      if (lSize > Sizes.cbMaximumMessage){
         __leave;
      }

      // This buffer is going to be a header, plus message, plus trailer
      ULONG lIOBufferLength = Sizes.cbHeader +
         Sizes.cbMaximumMessage +
         Sizes.cbTrailer;
      PBYTE pbIOBuffer = (PBYTE)alloca(lIOBufferLength);
      if (pbIOBuffer == NULL){
         __leave;
      }

      // The data is copied into the buffer after the header
      CopyMemory(pbIOBuffer+Sizes.cbHeader,(PBYTE)pvData, lSize);

      SecBuffer Buffers[3];
      // Set up the header in buffer
      Buffers[0].BufferType   = SECBUFFER_STREAM_HEADER;
      Buffers[0].pvBuffer     = pbIOBuffer;
      Buffers[0].cbBuffer     = Sizes.cbHeader;

      // Set up the data in buffer
      Buffers[1].BufferType   = SECBUFFER_DATA;
      Buffers[1].pvBuffer     = pbIOBuffer + Sizes.cbHeader;
      Buffers[1].cbBuffer     = lSize;

      // Set up the trailer in buffer
      Buffers[2].BufferType   = SECBUFFER_STREAM_TRAILER;
      Buffers[2].pvBuffer     = pbIOBuffer + Sizes.cbHeader + lSize;
      Buffers[2].cbBuffer     = Sizes.cbTrailer;

      // Set up the buffer descriptor
      SecBufferDesc secBufDescIn;
      secBufDescIn.ulVersion       = SECBUFFER_VERSION;
      secBufDescIn.cBuffers        = 3;
      secBufDescIn.pBuffers        = Buffers;

      // Encrypt the data
      ss = EncryptMessage(phContext, 0, &secBufDescIn, 0);
      if (ss != SEC_E_OK){
         __leave;
      }

      // Send all three buffers in one chunk
      ULONG lOut = Buffers[0].cbBuffer + Buffers[1].cbBuffer +
         Buffers[2].cbBuffer;
      SendData(pbIOBuffer, lOut);

      fSuccess = TRUE;
   }__finally{}

   return (fSuccess);
}
```

Here is *GetEncryptedMessage*:

```
PVOID GetEncryptedMessage(
   CtxtHandle* phContext,
```

```
PULONG plSize,
PBYTE* ppbExtraData,
PULONG pcbExtraData,
ULONG  lSizeExtraDataBuf,
BOOL* pfReneg){

PVOID pDecrypMsg = NULL;
*pfReneg = FALSE;

__try
{
   // Declare buffer descriptor
   SecBufferDesc  SecBuffDesc = {0};
   // Declare buffers
   SecBuffer      Buffers[4] = {0};
   // Extra data coming in
   PBYTE pbData = *ppbExtraData;
   ULONG cbData = *pcbExtraData;

   SECURITY_STATUS ss = SEC_E_INCOMPLETE_MESSAGE;
   do
   {
      // Set up initial data buffer to point
      // to extra data
      Buffers[0].BufferType = SECBUFFER_DATA;
      Buffers[0].pvBuffer = pbData;
      Buffers[0].cbBuffer = cbData;

      // Set up three empty out buffers
      Buffers[1].BufferType = SECBUFFER_EMPTY;
      Buffers[2].BufferType = SECBUFFER_EMPTY;
      Buffers[3].BufferType = SECBUFFER_EMPTY;

      // Set up descriptor
      SecBuffDesc.ulVersion       = SECBUFFER_VERSION;
      SecBuffDesc.cBuffers        = 4;
      SecBuffDesc.pBuffers        = Buffers;

      // If we have any data at all, let's try
      // to decrypt it
      if (cbData)
         ss = DecryptMessage(phContext, &SecBuffDesc, 0, NULL);

      if (ss == SEC_E_INCOMPLETE_MESSAGE || cbData == 0){
         ULONG lReadSize;
         // Need to read in more data and try again
         lReadSize = lSizeExtraDataBuf - cbData;
         ReceiveData(pbData+cbData, &lReadSize);
         cbData += lReadSize;
      }

   }while (ss == SEC_E_INCOMPLETE_MESSAGE);

   // Was there actually a successful decryption?
   if (ss == SEC_E_OK){

      // Allocate a buffer for the caller
      // and copy decrypted message to it
      *plSize = Buffers[1].cbBuffer;
      pDecrypMsg = (PVOID)LocalAlloc(0,*plSize);
      if (pDecrypMsg == NULL){
         __leave;
      }
      CopyMemory(pDecrypMsg,Buffers[1].pvBuffer,*plSize);

      // If there is extra data, move it to the beginning of the
```

```
            // buffer, and then set the size of the extra data returned
            int nIndex = 3;
            while(Buffers[nIndex].BufferType
               != SECBUFFER_EXTRA && (nIndex-- != 0));

            if (nIndex != -1){
               // There is more data to handle. Move it to front of
               // the extra data buffer and the caller can handle
               // the decrypted message and then come back to finish.
               *pcbExtraData = Buffers[nIndex].cbBuffer;
               MoveMemory(pbData,pbData + (cbData - *pcbExtraData),
                  *pcbExtraData);
            } else *pcbExtraData = 0;
         }
         if (ss == SEC_I_RENEGOTIATE){
            *pfReneg = TRUE;
         }

      }__finally{}

      // Return decrypted message
      return (pDecrypMsg);
   }
```

As you can see from the code, the logic is similar to other sample functions in this chapter, except that these functions must pay attention to the possibility of extra data. Whereas the sending function has no concern for extra data, the receiving or decrypting function must take a buffer with potential extra data and be able to return extra data in this same buffer. This receive can overlap with receives before or after it, or both.

SSL merges the notions of encryption and message signing, so there is no need to implement *MakeSignature* and *VerifySignature* for SSL. The functions are not supported with the SChannel security package.

You might have noticed that the *GetEncryptedMessage* sample function tests for a return value of SEC_I_RENEGOTIATE and populates a supplied Boolean variable with TRUE if this value is returned by *DecryptMessage*. This return value indicates that the server has requested a renegotiation of certificates, and typically it indicates that it wants to upgrade an anonymous client authentication to a client authentication complete with certificate.

When *DecryptMessage* returns SEC_I_RENEGOTIATE, there is no message to decrypt, and there will not be any output buffers with decrypted data.

Handling Renegotiation

If a client requests a protected resource, your server might want to upgrade an anonymous client authentication to an authentication with a certificate. To do this, your server must initiate a renegotiation.

Typically in an SSL communication, the client and server communicate by sending and receiving encrypted buffers full of data. However, when a server wants to initiate a renegotiation, rather than encrypting and sending information, a server calls *AcceptSecurityContext*. In this case, the server passes no input buffers to the function, and the function returns a blob that should be sent to the client. The client detects that this blob indicates that it should enter into a renegotiation. The following sample function shows how a server might implement this code:

```
BOOL SSLServerInitReneg(
   CredHandle* phCredentials,
   CtxtHandle* phContext)
{
   BOOL fSuccess = FALSE;
```

```
    __try
    {

       // Declare out buffer
       SecBuffer secBufferOut;
       SecBufferDesc secBufDescriptorOut;


       // Set up out buffer (allocated by the SSPI)
       secBufferOut.BufferType = SECBUFFER_TOKEN;
       secBufferOut.cbBuffer = 0;
       secBufferOut.pvBuffer = NULL;

       // Set up out buffer descriptor
       secBufDescriptorOut.cBuffers = 1;
       secBufDescriptorOut.pBuffers = &secBufferOut;
       secBufDescriptorOut.ulVersion = SECBUFFER_VERSION;

       ULONG lAttrOut = 0;
       SECURITY_STATUS ss =
          AcceptSecurityContext(
             phCredentials,
             phContext,
             NULL,
             ASC_REQ_ALLOCATE_MEMORY|ASC_REQ_STREAM|ASC_REQ_MUTUAL_AUTH,
             SECURITY_NATIVE_DREP,
             phContext,
             &secBufDescriptorOut,
             &lAttrOut,
             NULL);
       if (ss != SEC_E_OK)
          __leave;

       // Was there data to be sent?
       if (secBufferOut.cbBuffer!=0){
          // Send it then
          ULONG lOut = secBufferOut.cbBuffer;
          SendData(secBufferOut.pvBuffer, lOut);
          // And free up that out buffer
          FreeContextBuffer( secBufferOut.pvBuffer );
       }
       fSuccess = TRUE;
    }__finally{}

    return (fSuccess);
 }
```

As I mentioned in my discussion of *DecryptMessage*, the client first knows of the server's desire to renegotiate when *DecryptMessage* returns SEC_I_ RENEGOTIATE. At this time, if the client is interested in doing a renegotiation, it should reenter the authentication loop with a credentials handle for an appropriate certificate.

The server should enter into its side of the authentication renegotiation after requesting a renegotiation as well. It is the server's responsibility to verify that the client has provided an appropriate certificate. It is entirely possible that the client has resent its anonymous credentials.

Using the sample functions listed in this chapter, the code fragment below would represent the server's role in a renegotiation.

```
 // Server needs to renegotiate with the client
 if(!SSLServerInitReneg(
    &hCredentials,
    &hContext)){
```

```
   // Error case
}

// Set up flags for our authentication handshake
dwSSPIFlags = ASC_REQ_SEQUENCE_DETECT   |
              ASC_REQ_REPLAY_DETECT     |
              ASC_REQ_CONFIDENTIALITY   |
              /* Indicates that client certificate is required */
              ASC_REQ_MUTUAL_AUTH       |
              ASC_RET_EXTENDED_ERROR;


cbExtra = 0;
// Reenter the authentication loop
ss = SSLServerHandshakeAuth(&hCredentials,
   &dwSSPIFlags, &hContext, pIOBuff, &cbExtra, lIOBuffSize);
if (ss == SEC_E_OK){
   // Renegotiation completed successfully
}
```

In a similar way, the following code fragment shows the client's role in a renegotiation (assuming that the client does have a certificate to renegotiate with):

```
// Get decrypted message as usual
cbMsg = 0;
cbExtra = 0;
pMsg =
   GetEncryptedMessage(
      &hContext,
      &cbMsg,
      &pIOBuff,
      &cbExtra,
      lIOBuffSize,
      &fReneg);
// If it failed, is this a renegotiate request?
if ((pMsg == NULL) && fReneg){

   // If so, then reenter the authentication loop with a
   // credentials handle associated with a certificate
   ULONG   lSSPIFlags = ISC_REQ_SEQUENCE_DETECT |
                        ISC_REQ_REPLAY_DETECT     |
                        ISC_REQ_CONFIDENTIALITY   |
                        ISC_RET_EXTENDED_ERROR;
   ss =
      SSLClientHandshakeAuth(
         &hCertCredentials,
         &hCertCredentials,
         &lSSPIFlags,
         &hContext,
         TEXT("Jason's Test Server Certificate on Davemm"),
         pIOBuff, &cbExtra, lIOBuffSize);
   if (ss == SEC_E_OK){
      // Renegotiation success
   }
}
```

NOTE

If your client software will be communicating with a server that might request a renegotiation, it is important that your client be able to gracefully handle a renegotiation request whenever it calls *DecryptMessage*.

SSL and Impersonation

If the server has authenticated the client by receiving a certificate from the client (as opposed to anonymous client authentication), it is possible for the server to then impersonate the client. In fact, the details of the impersonation do not differ from impersonation with other security protocols using the SSPI. You simply pass the completed context handle to *ImpersonateSecurityContext*, or you retrieve the token directly using *QuerySecurityContextToken*. (Both of these techniques were discussed earlier in this chapter.)

However, one burning question remains: How does Windows 2000 create a token for a user with nothing other than the certificate? The answer to this question is via a certificate mapping in Active Directory, or via Microsoft- specific information imbedded in the certificate itself. In all, there are three ways in which a certificate can be mapped to a token for a user trustee account in a Windows 2000 domain.

Before covering the ways in which you can map a certificate to a user in your domain, I would like to take a moment to point out the significance of this mapping ability. If your server running on Windows 2000 connects to a client with a known certificate, it is possible for your server to build a token and log this client on to your server machine, regardless of the operating system the client is using. This means you can impersonate a client who has connected to you and is running client software on a machine running UNIX, a machine running Windows, or any other machine capable of communicating via SSL.

Here are the three different approaches you can use to map a certificate to a user account in Windows 2000:

- **One-to-one mapping** Any certificate signed by an issuer trusted by the server can be associated with a user account. When this certificate is used in an SSL connection to initiate an impersonation, the system looks up the common name and issuer name on the certificate in Active Directory to find the user account for which a token should be built.

- **Many-to-one mapping** Any CA trusted by the server can be associated with a user account. When any certificate signed by this CA is used in an SSL connection to initiate an impersonation, the system looks up the issuer name in Active Directory to find the user account for which a token should be built.

- **User principal name (UPN) mapping** Unlike the other two approaches, UPN mapping uses information in the certificate to find the user account for which a token should be built. The UPN is the user principal name of the account that should be impersonated and will exist as a special field on the certificate. A UPN will look something like this: "jclark@subdomain.microsoft.com".

Although the impersonation of an SSL connection is simple, the administration of these certificate mappings is your responsibility. The simplest approach requiring the least administration is the UPN mapping. With UPN mapping, all you need is a CA running Microsoft Certificate Services as an enterprise CA. Then you request a "user" certificate from the CA. This user certificate will include the UPN attribute, which contains the user account name of the account that requested the certificate. If this certificate is used in a client connection to an SSL server, it can be impersonated as is.

Of course, you might find that you want to impersonate certificates generated without the Microsoft-specific extension, and doing so requires you to do an explicit mapping in Active Directory. Both one-to-one mappings and the many-to-one mappings use a similar technique to map a certificate to a user in Active Directory.

To start, you must export the certificate to a .CER file. This file type is a standard certificate export format that includes a signed copy of the certificate and the public key, but it does not include the matching private key for the certificate. The Certificates snap-in in the MMC allows you to export a certificate by right-clicking on the certificate and selecting All Tasks and then Export. This will start the Certificate Export Wizard.

Once you have a .CER file, you should follow these steps to create a mapping to a user account in Active Directory:

1. Open the Active Directory Users and Computers snap-in in the MMC.
2. Select Advanced Features from the View menu.
3. Select the Users folder in the left pane.
4. Right-click on the user to which you wish to map a certificate in the right pane, and select Name Mappings from the context menu.
5. In the X509 Certificates tab of the Security Identity Mapping dialog box, click the Add button.
6. Select the .CER file for the certificate that you wish to map and click Open. The Add Certificate dialog box appears.
7. If you check the Use Subject For Alternate Security Identity check box, then you will have a one-to-one mapping.
8. If you uncheck the Use Subject For Alternate Security Identity check box and leave only the Use Issuer For Alternate Security Identity check box checked, you will have a many-to-one mapping.

After these steps have been properly executed, your server software can impersonate a client.

NOTE

There are two common gotchas with certificate mapping that warrant mentioning. First, if the server does not trust the issuer of a certificate, the client connecting with that certificate cannot be impersonated. Second, if a certificate is a user certificate created by an enterprise CA running Microsoft Certificate Services, the UPN will be used for mapping before Active Directory is searched for an explicit mapping. This can cause unexpected results if you explicitly mapped such a certificate to a user in Active Directory.

Troubleshooting SSL

The SChannel security package, which implements SSL on Windows 2000, supports an error and tracking reporting mechanism that adds events to the event log. By default it is set to report errors to the system log. However, it can also be configured to report warning-level and tracking-level events to the log.

To enable these features, you must modify a value in the registry. The key is located in HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SecurityProviders\Schannel, and the value is named EventLogging.

The EventLogging value is the combination of three values where 1 indicates error reporting, 2 indicates warning reporting, and 4 indicates tracking event reporting. You can combine these values to achieve any combination of reporting you wish. For example, a value of 3 would indicate error and warning reporting, whereas a value of 7 would turn on all reporting levels. This feature can be helpful in understanding the underlying tasks that the security provider is performing on behalf of your client and server software using SSL.

Using the SSPI to implement SSL certainly involves some relatively difficult coding, but knowing what you need to do and having good samples to work with can be helpful. I strongly suggest that you spend some time

gaining an in-depth understanding of the samples in this book, as well as the samples in the Platform SDK documentation, before attempting to implement your own SSL server.

The SSLChat Sample Application

The SSLChat sample application ("12 SSLChat.exe") demonstrates how to use all the SSL-related technologies we have discussed so far, including client and server certificate authentication, message signing, and encryption. The source code and resource files for the sample application are in the 12-SSLChat directory on the companion CD. The SSLChat sample application has the user interface shown in Figure 12-7.

The SSLChat sample application is very similar to the SSPIChat sample application. However, in the SSLChat sample application, both the client and the server can select a certificate to use for authenticating to the remote principal.

It is important that both the client and the server machine recognize the root CAs of each other's certificates. A full discussion on certificate management from the user perspective is beyond the scope of this book, but you can find quite a bit of information on this topic in the Windows 2000 Help.

Figure 12-7. *User interface for the SSLChat sample application*

[Previous] [Next]

Applying Secure Communication

Deciding how to secure the communication of your service can be tough, so you should make this decision armed with as much information as you can get. Here are some questions you should ask yourself:

1. Is it important that I can authenticate my client to a Windows domain account?
2. Will my client be in my domain, outside of my domain (perhaps via the Internet), or both?
3. Will I be interoperating with non-Windows clients?

If your authentication needs will boil down to a Windows token (which is great for impersonation and access control), you will most likely want to use NTLM or Kerberos. However, you could use SSL and allow the system to maintain a mapping of certificates to domain accounts. This would require extra maintenance, as I have described earlier.

NTLM and Kerberos authenticate to a domain naturally. So if your answer to the first question is "yes," you need additional incentive to use SSL and certificates.

If your client is going to be in your domain, your options really open up. You can use named pipes, which automatically use Kerberos or NTLM authentication and ease this process greatly. Of course, pipes have scalability issues, and if you are connecting to thousands of clients, you will most likely want to use TCP/IP with sockets.

Even so, clients that are guaranteed to be in your domain make coding easy. You can set up your server to use the Negotiate security package, and your clients can use the most appropriate package for them. Windows 2000 clients will use Kerberos, and all others will most likely use NTLM. Using these options, you can design a clean, secure environment that is relatively easy to implement.

But what happens if your client is going to be outside of your domain? If you are not concerned with gaining a Windows token, clients outside of your domain might as well use SSL and certificates. If you do want a token for your user, on the server side you've got some issues to contend with where impersonation is concerned.

If the client is outside of your domain, you can assume that the client is able to communicate with your server. This means you can consider the NTLM protocol. The client will pass its username, password, and the server's domain into *AcquireCredentialsHandle*. This information is communicated to the server. Since the client communicates only with the server, it is not necessary for the client to have access to the DC, and so the client can be outside of the domain. Meanwhile, the server is inside your domain, has access to the DC, and can get a token for your client. This is pretty nifty.

With Kerberos, dealing with a client outside of your domain is not this easy. Much of the authentication process involves client communication with the KDC. This is great in terms of efficiency but can complicate communication with clients outside of your domain. So NTLM looks attractive once again for authentication because it's relatively easy to use outside of your domain.

SSL can be used to achieve a token on your server side, and it requires the client to communicate only with the server, but it carries with it the baggage of certificate management. Where do you get your certificates? Are you paying for them or issuing them yourself? If you are issuing them, will it be difficult to acquire trust on the client side? If not, do you and your client trust the same root CAs? Finally, if a token is your goal, you must either manage the certificate mapping in Active Directory or use certificates issued by an enterprise CA running Microsoft Certificate Services (which may or may not meet your needs in other respects). And what about the issue of interoperability with other operating systems? Well, SSL shines in this manner. The SSL and certificates scale well and are implemented on other operating systems. Our implementation interoperates great! But SSL comes with some limitations, which you know of.

You do have some other options though. The Kerberos SSP is interoperable with the UNIX Generic Security Service (GSS) library. You must handle some extra details when using *EncryptMessage* and *DecryptMessage*. However, this interoperability makes it reasonable for you to create a UNIX client that communicates using Kerberos with your Windows server. The GSS and interoperability is covered in detail in the Platform SDK documentation. The Platform SDK also includes a complete GSS sample.

Encryption Is Not Equal to Security

Once you have mastered the SSPI and other features provided by Windows, you might feel that your server is secure. But it might not necessarily be.

Kerberos and NTLM make tight security easier because they use symmetric encryption based on a user's password. Once your certificate is authenticated, you can be reasonably sure that you are communicating with an entity that knows the password, especially when you use Kerberos. (However, passwords are not "strong" security in that they can be "known" or derived more simply than a private key can. So password security is a major concern.)

SSL is nowhere near as simple as NTLM and Kerberos. Certificates do not naturally map to a user account in a domain, and they do not directly map to a known secret as simple as a password. So here is the question that you have to constantly consider: Now that I know I am communicating securely with the holder of this certificate's matching private key, *how do I know who the holder is?*

Public key encryption and decryption have given us powerful tools to ensure that we are communicating with one entity and one entity only. But the question of whether to trust that entity in the first place is still open. Here is the catch: all the information in a certificate is public. So what information are you checking to ensure the certificate in question belongs to the entity that you want to connect to?

- Are you relying on the common name in the certificate? Can someone else use another CA to create a certificate with the same common name?
- Are you relying on the common name matching a URL? Is it possible for someone to create another certificate that matches the same URL?

The answer to all of these questions is yes, but knowing this information doesn't make your authentication process any more secure. So it is necessary for you to know extra information about the CA that issued the certificate:

- What information does the CA guarantee to be unique?
- How much research does the CA do before granting a certificate?

Another consideration is how you should communicate these questions to the user. How can you do this in a way that is not so confusing as to cause the user to always give up and trust everything and everyone?

The challenges posed by these questions are serious but not insurmountable, though no single solution has arisen that solves these problems for every scenario. With proper consideration, you can solve these problems so that your specific needs are met. In short, public key infrastructure is powerful, and if you choose to use it, you can do it in a secure way. You just have to be able to answer all the questions I just posed for your software. Figuring out the answers to these questions is the challenge of secure programming and the reason that writing secure software is so incredibly rewarding.

[Previous] [Next]

Appendix A -- The Build Environment

To build the sample applications in this book, you must deal with compiler and linker switch settings. I tried to isolate these setting details from the sample applications by placing almost all of them in a single header file named CmnHdr.h. It is included in all the sample application source code files.

Since I wasn't able to place all the settings in CmnHdr.h, I made some changes to each sample application's project settings in the Project Settings dialog box:

- On the General tab, I set the Output Files directory so that all final .exe and .dll files go to a single directory.
- On the C/C++ tab, I selected the Code Generation category and selected Multithreaded DLL for the Use Run-Time Library field.

That's it. These are the only settings that I explicitly changed; I accepted the default settings for everything else. Note that I made the two changes for both the debug and release builds of each project. I was able to set all other compiler and linker settings in the source code, so these settings will be in effect if you use any of my source code modules in your projects.

[Previous] [Next]

The CmnHdr.h Header File

All the sample programs include the CmnHdr.h header file before any other header file. I wrote CmnHdr.h, shown in Listing A-1 at the end of this appendix, to make my life a little easier. The file contains macros, linker directives, and other code that is common across all the sample applications. When I want to try something different, all I do is modify CmnHdr.h and rebuild all the sample applications. CmnHdr.h is in the root directory on the companion CD.

The remainder of this appendix discusses each section of the CmnHdr.h header file. I'll explain the rationale for each section and describe how and why you might want to make changes before rebuilding the sample applications.

[Previous] [Next]

Forcing the Linker to Look for a (*w*)*WinMain* Entry-Point Function

After readers add my source code modules to a new Microsoft Visual C++ project and then try to build the project, they receive linker errors. The problem is that they created a Win32 Console Application project, causing the linker to look for a (*w*)*main* entry-point function. Since all the book's sample applications are GUI applications, my source code has a *_tWinMain* entry-point function instead, which causes the linker to complain.

My standard reply to readers is as follows: delete the project, create a new Win32 Application project (note that the word "Console" doesn't appear in this project type) with Visual C++, and then add my source code files to it. The linker looks for a (*w*)*WinMain* entry-point function, which I do supply in my code, and which allows the projects to build properly.

To reduce the amount of e-mail I get on this issue, I added a *pragma* to CmnHdr.h that forces the linker to look for the (*w*)*WinMain* entry-point function even if you create a Win32 Console Application project with Visual C++.

In Chapter 4 of *Programming Applications for Microsoft Windows, Fourth Edition* (Jeffrey Richter, Microsoft Press, 1999), I go into great detail about what the Visual C++ project types are all about, how the linker

chooses which entry-point function to look for, and how to override the linker's default behavior.

[Previous] [Next]

Windows Version Build Option

Because some of the sample applications call functions that are new in Microsoft Windows 2000, this section of CmnHdr.h defines the _WIN32_WINNT symbol as follows:

```
#define _WIN32_WINNT 0x0500
```

I have to define this symbol because the new Windows 2000 functions are prototyped in the Windows header files like this:

```
#if (_WIN32_WINNT >= 0x0500)

WINBASEAPI
BOOL
WINAPI
AssignProcessToJobObject(
   IN HANDLE hJob,
   IN HANDLE hProcess
   );

#endif /* _WIN32_WINNT >= 0x0500 */
```

Unless you specifically define _WIN32_WINNT as I have (before including Windows.h), the prototypes for the new functions will not be declared, and the compiler will generate errors when you attempt to call these functions. Microsoft has protected these functions with the _WIN32_WINNT symbol to help ensure that applications you develop can run on multiple versions of Microsoft Windows NT and Microsoft Windows 98.

[Previous] [Next]

Unicode Build Option

Most of the sample applications can be compiled as either ANSI or Unicode. This book is about writing services for Windows 2000, so Unicode is the default because it allows the application to use less memory and run faster. However, if you want to use the code in your own ANSI applications, you can do so.

> **NOTE**
>
> Over time, Microsoft has added more and more Unicode-only features to the operating system. For example, the Net functions for managing trustee accounts discussed in Chapter 9 support only Unicode strings. Because these functions appear in my code, some of the sample applications in this book will compile only for Unicode. For more information on Unicode, see Chapter 2 of *Programming Applications for Microsoft Windows, Fourth Edition*.

[Previous] [Next]

Windows Definitions and Warning Level 4

When I develop software, I always try to ensure that the code compiles free of errors and warnings. I also like to compile at the highest possible warning level so that the compiler does the most work for me, examining even the smallest details of my code. For the Microsoft C/C++ compilers, I built all the sample applications using warning level 4.

In this section of CmnHdr.h, I make sure that the warning level is set to 4 and that CmnHdr.h includes the standard Windows.h header file. At warning level 4, the compiler emits "warnings" for things that I don't consider problems, so I explicitly tell the compiler to ignore certain benign warnings by using the *#pragma warning* directive.

[Previous] [Next]

The *pragma* Message Helper Macro

When I work on code, I often like to get something working immediately and then make it bulletproof later. To remind myself that some code needs additional attention, I used to include a line like this:

```
#pragma message("Fix this later")
```

When the compiler compiled this line, it output a string reminding me that I had some more work to do; however, this message was not that helpful. So I decided to find a way to make the compiler output the name of the source code file and the line number that the *pragma* appears on. If I have that information, not only do I know that I have additional work to do, but I immediately know the location of the surrounding code.

To output the source code file and the line number, you have to trick the *pragma message* directive by using a series of macros. I named the resulting macro *chMSG*, and it is used like this:

```
#pragma chMSG(Fix this later)
```

When this line is compiled, the compiler produces a line that looks like this:

```
C:\CD\CmnHdr.h(82):Fix this later
```

Using Microsoft Visual Studio, you can double-click on this line in the output window and be automatically positioned at the place in the file that requires additional attention.

Conveniently, the *chMSG* macro does not require that you use quotes around the text string.

[Previous] [Next]

The *chINRANGE*, *chDIMOF*, and *chSIZEOFSTRING* Macros

I frequently use the handy *chINRANGE*, *chDIMOF*, and *chSIZEOFSTRING* macros in my applications. The first one, *chINRANGE*, checks to see whether a value is between two other values. The second macro, *chDIMOF*, simply returns the number of elements in an array. It does this by using the *sizeof* operator to first calculate the size of the entire array in bytes. It then divides this number by the number of bytes required for a single entry in the array. The third macro, *chSIZEOFSTRING*, returns the number of bytes occupied by a zero-terminated string.

[Previous] [Next]

The *chROUNDDOWN* and *chROUNDUP* Inline Template Functions

The *chROUNDDOWN* and *chROUNDUP* inline template functions simply round a value down or up to the nearest specified multiple.

[Previous] [Next]

The *chBEGINTHREADEX* Macro

All the multithreaded samples in this book use the *_beginthreadex* function, which is in Microsoft's C/C++ run-time library, instead of the operating system's *CreateThread* function. I use *_beginthreadex* because it prepares the new thread to use the C/C++ run-time library functions and ensures that the per-thread C/C++ run-time library information is destroyed when the thread returns. (See Chapter 6 of *Programming Applications for Microsoft Windows, Fourth Edition,* for more details.) Unfortunately, the *_beginthreadex* function is prototyped as follows:

```
unsigned long __cdecl _beginthreadex(
   void *,
   unsigned,
   unsigned (__stdcall *)(void *),
   void *,
   unsigned,
   unsigned *);
```

Although the parameter values for *_beginthreadex* are identical to the parameter values for the *CreateThread* function, the parameters' data types do not match. Here is the prototype for the *CreateThread* function:

```
typedef DWORD (WINAPI *PTHREAD_START_ROUTINE)(PVOID pvParam);

HANDLE CreateThread(
   PSECURITY_ATTRIBUTES  psa,
   DWORD                 cbStack,
   PTHREAD_START_ROUTINE pfnStartAddr,
   PVOID                 pvParam,
   DWORD                 fdwCreate,
   PDWORD                pdwThreadId);
```

Microsoft did not use the Windows data types when creating the *_beginthreadex* function's prototype because Microsoft's C/C++ run-time group does not want to have any dependencies on the operating system group. I commend this decision; however, this makes using the *_beginthreadex* function more difficult.

There are really two problems with the way Microsoft prototyped the *_beginthreadex* function. First, some of the data types used for the function do not match the primitive types used by the *CreateThread* function. For example, the Windows data type DWORD is defined as follows:

```
typedef unsigned long DWORD;
```

This data type is used for *CreateThread*'s *cbStack* parameter as well as for its *fdwCreate* parameter. The problem is that *_beginthreadex* prototypes these two parameters as *unsigned*, which really means *unsigned int*. The compiler considers an *unsigned int* to be different from an *unsigned long* and generates a warning. Because the *_beginthreadex* function is not a part of the standard C/C++ run-time library and exists only as an alternative to calling the *CreateThread* function, I believe that Microsoft should have prototyped *_beginthreadex* this way to avoid generating warnings:

```
unsigned long __cdecl _beginthreadex(
   void                    *psa,
   unsigned                long cbStack,
   unsigned (__stdcall *) (void *pvParam),
   void                    *pvParam,
   unsigned long           fdwCreate,
   unsigned long           *pdwThreadId);
```

The second problem is just a small variation of the first. The *_beginthreadex* function returns an *unsigned long* representing the handle of the newly created thread. An application typically wants to store this return value in a data variable of type HANDLE as follows:

```
HANDLE hThread = _beginthreadex(...);
```

This code causes the compiler to generate a warning. To avoid the compiler warning, you must rewrite the line, introducing a cast as follows:

```
HANDLE hThread = (HANDLE) _beginthreadex(...);
```

But this is inconvenient. To make life a little easier, I defined a *chBEGINTHREADEX* macro in CmnHdr.h to perform all of this casting for me:

```
typedef unsigned (__stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStack, pfnStartAddr, \
   pvParam, fdwCreate, pdwThreadId)                 \
      ((HANDLE)_beginthreadex(                      \
         (void *)         (psa),                    \
         (unsigned)       (cbStack),                \
         (PTHREAD_START)  (pfnStartAddr),           \
         (void *)         (pvParam),                \
         (unsigned)       (fdwCreate),              \
         (unsigned *)     (pdwThreadId)))
```

[Previous] [Next]

DebugBreak Improvement for *x*86 Platforms

I sometimes want to force a breakpoint in my code even when the process is not running under a debugger. You can do this in Windows by having a thread call the *DebugBreak* function. This function, which resides in Kernel32.dll, lets you attach a debugger to the process. Once the debugger is attached, the instruction pointer is positioned on the CPU instruction that caused the breakpoint. This instruction is contained in the *DebugBreak* function in Kernel32.dll, so to see my source code I must single-step out of the *DebugBreak* function.

On the *x*86 architecture, you perform a breakpoint by executing an "int 3" CPU instruction, so I redefine *DebugBreak* as this inline assembly language instruction. When my *DebugBreak* macro is executed, I do not call into Kernel32.dll; the breakpoint occurs right in my code, and the instruction pointer is positioned to the next C/C++ language statement. This improved *DebugBreak* macro just makes things a little more convenient.

[Previous] [Next]

Creating Software Exception Codes

When you work with software exceptions, you must create your own 32-bit exception codes. These codes follow a specific format (discussed in Chapter 24 of *Programming Applications for Microsoft Windows,*

Fourth Edition). To make creating these codes easier, I use the *MAKESOFTWAREEXCEPTION* macro.

[Previous] [Next]

The *chMB* Macro

The *chMB* macro simply displays a message box. The caption is the full pathname of the executable file for the calling process.

[Previous] [Next]

The *chASSERT* and *chVERIFY* Macros

To find potential problems as I developed the sample applications, I sprinkled *chASSERT* macros throughout the code. The *chASSERT* macro tests whether the expression identified by *x* is TRUE and, if it isn't, displays a message box indicating the file, line, and expression that failed. In release builds of the applications, this macro expands to nothing. The *chVERIFY* macro is almost identical to the *chASSERT* macro except that the expression is evaluated in release builds as well as in debug builds.

[Previous] [Next]

The *chHANDLE_DLGMSG* Macro

When you use message crackers with dialog boxes, you should not use the *HANDLE_MSG* macro from Microsoft's WindowsX.h header file because it doesn't return TRUE or FALSE to indicate whether a message was handled by the dialog box procedure. My *chHANDLE_DLGMSG* macro massages the window message's return value and handles it properly for use in a dialog box procedure.

[Previous] [Next]

The *chSETDLGICONS* Macro

Because most of the sample applications use a dialog box as their main window, you must change the dialog box icon manually so that it is displayed correctly on the taskbar, in the task switch window, and in the application's caption. The *chSETDLGICONS* macro is always called when dialog boxes receive a WM_INITDIALOG message so that the icons are set correctly.

[Previous] [Next]

Making Sure the Host System Supports Unicode

The CmnHdr.h file includes a Unicode check macro that was used in *Programming Applications for Microsoft Windows, Fourth Edition,* to ensure that the host system supports Unicode. Since this book's sample applications require Windows 2000, this macro is not necessary.

[Previous] [Next]

The OS Version Check Inline Functions

The CmnHdr.h file includes two inline functions that were used in *Programming Applications for Microsoft Windows, Fourth Edition,* to check the operating system version of the host system. Since this book's sample applications require Windows 2000, these functions are not necessary.

[Previous] [Next]

The CSystemInfo C++ Class

This very simple C++ class wraps the Windows SYSTEM_INFO structure and initializes this structure when an instance of the class is constructed. Many of the sample applications make use of members in this structure; wrapping the structure in a C++ class insures that the members are always initialized and simplifies some of the code.

Listing A-1 *The CmnHdr.h header file*

CmnHdr.h

```
/******************************************************************************
Module: CmnHdr.h Notices: Copyright (c) 2000 Jeffrey Richter Purpose: Common header file
containing handy macros and definitions used throughout all the applications in the book. See
Appendix A.
******************************************************************************/
#pragma once // Include this header file once per compilation unit //////////////////////////// Force Windows
subsystem //////////////////////////// #pragma comment(linker, "/subsystem:Windows,5") ////////////////////////////
Force Windows 2000 ////////////////////////////////// #pragma comment(linker, "/version:5") ////////////////////////
Windows Version Build Option ////////////////////////// #define _WIN32_WINNT 0x0500 #define
WINVER 0x0500 ////////////////////////////// Unicode Build Option ////////////////////////////// // If we are not
compiling for an x86 CPU, we always compile using Unicode. #ifndef _M_IX86 #define UNICODE
#endif // To compile without Unicode, comment out the line below. #define UNICODE // When using
Unicode Windows functions, use Unicode C-Runtime functions too. #ifdef UNICODE #define
_UNICODE #endif ////////////////////////// Include Windows Definitions ////////////////////////// #pragma
warning(push, 4) #include <Windows.h> #include <TChar.h> ////////////// Verify that the proper header
files are being used ////////////// #ifndef WT_EXECUTEINPERSISTENTIOTHREAD #pragma
message("You are not using the latest Platform SDK header/library ") #pragma message("files. This
may prevent the project from building correctly.") #pragma message("You may install the Platform
SDK from the book's CD-ROM or ") #pragma message("from
http://msdn.microsoft.com/downloads/") #endif ////////////// Allow code to compile cleanly at warning
level 4 /////////////// /* nonstandard extension 'single line comment' was used */ #pragma
warning(disable:4001) // unreferenced formal parameter #pragma warning(disable:4100) // Note:
Creating precompiled header #pragma warning(disable:4699) // function not inlined #pragma
warning(disable:4710) // unreferenced inline function has been removed #pragma
warning(disable:4514) // assignment operator could not be generated #pragma warning(disable:4512)
// cast truncates constant value #pragma warning(disable:4310) ////////////////////////// Pragma message
helper macro ////////////////////////// /* When the compiler sees a line like this: #pragma chMSG(Fix this
later) it outputs a line like this: c:\CD\CmnHdr.h(82):Fix this later You can easily jump directly to this
```

```
line and examine the surrounding code. */ #define chSTR2(x) #x #define chSTR(x) chSTR2(x)
#define chMSG(desc) message(__FILE__ "(" chSTR(__LINE__) "):" #desc) ////////////////////////////
chINRANGE Macro //////////////////////////////// // This macro returns TRUE if a number is between two
others #define chINRANGE(low, Num, High) (((low) <= (Num)) && ((Num) <= (High)))
//////////////////////////////// chDIMOF Macro ////////////////////////////////// // This macro evaluates to the number
of elements in an array. #define chDIMOF(Array) (sizeof(Array) / sizeof(Array[0]))
///////////////////////////// chSIZEOFSTRING Macro /////////////////////////// // This macro evaluates to the
number of bytes needed by a string. #define chSIZEOFSTRING(psz) ((lstrlen(psz) + 1) *
sizeof(TCHAR)) ////////////////// chROUNDDOWN & chROUNDUP inline functions ////////////////// //
This inline function rounds a value down to the nearest multiple template <class TV, class TM> inline
TV chROUNDDOWN(TV Value, TM Multiple) { return((Value / Multiple) * Multiple); } // This
inline function rounds a value down to the nearest multiple template <class TV, class TM> inline TV
chROUNDUP(TV Value, TM Multiple) { return(chROUNDDOWN(Value, Multiple) + (((Value %
Multiple) > 0) ? Multiple : 0)); } ////////////////////////////// chBEGINTHREADEX Macro
////////////////////////// // This macro function calls the C runtime's _beginthreadex function. // The C
runtime library doesn't want to have any reliance on Windows' data // types such as HANDLE. This
means that a Windows programmer needs to cast // values when using _beginthreadex. Since this is
terribly inconvenient, // I created this macro to perform the casting. typedef unsigned (__stdcall
*PTHREAD_START) (void *); #define chBEGINTHREADEX(psa, cbStack, pfnStartAddr, \
pvParam, fdwCreate, pdwThreadId) \ ((HANDLE)_beginthreadex( \ (void *) (psa), \ (unsigned)
(cbStack), \ (PTHREAD_START) (pfnStartAddr), \ (void *) (pvParam), \ (unsigned) (fdwCreate), \
(unsigned *) (pdwThreadId))) ////////////////// DebugBreak Improvement for x86 platforms //////////////////
#ifdef _X86_ #define DebugBreak() _asm { int 3 } #endif /////////////////////////// Software Exception
Macro /////////////////////////// // Useful macro for creating your own software exception codes #define
MAKESOFTWAREEXCEPTION(Severity, Facility, Exception) \ ((DWORD) ( \ /* Severity code */
(Severity ) | \ /* MS(0) or Cust(1) */ (1 << 29) | \ /* Reserved(0) */ (0 << 28) | \ /* Facility code */
(Facility << 16) | \ /* Exception code */ (Exception << 0))) /////////////////////////// Quick MessageBox
Macro ////////////////////////// inline void chMB(PCSTR s) { char szTMP[256];
GetModuleFileNameA(NULL, szTMP, chDIMOF(szTMP)); HWND hwnd = GetActiveWindow();
MessageBoxA(hwnd, s, szTMP, MB_OK | ((hwnd == NULL ) ? MB_SERVICE_NOTIFICATION :
0)); } /////////////////////////// Assert/Verify Macros ////////////////////////// inline void
chMBANDDEBUG(PSTR szMsg) { chMB(szMsg); DebugBreak(); } // Put up an assertion failure
message box. inline void chASSERTFAIL(LPCSTR file, int line, PCSTR expr) { char sz[256];
wsprintfA(sz, "File %s, line %d : %s", file, line, expr); chMBANDDEBUG(sz); } // Put up a message
box if an assertion fails in a debug build. #ifdef _DEBUG #define chASSERT(x) if (!(x))
chASSERTFAIL(__FILE__, __LINE__, #x) #else #define chASSERT(x) #endif // Put up a failure
message box in a debug build. #ifdef _DEBUG #define chFAIL() chASSERTFAIL(__FILE__,
__LINE__, "") #else #define chFAIL() #endif // Assert in debug builds, but don't remove the code in
retail builds. #ifdef _DEBUG #define chVERIFY(x) chASSERT(x) #else #define chVERIFY(x) (x)
#endif /////////////////////////// chHANDLE_DLGMSG Macro ////////////////////////// // The normal
HANDLE_MSG macro in WindowsX.h does not work properly for dialog // boxes because DlgProc
return a BOOL instead of an LRESULT (like // WndProcs). This chHANDLE_DLGMSG macro
corrects the problem: #define chHANDLE_DLGMSG(hwnd, message, fn) \ case (message): return
(SetDlgMsgResult(hwnd, uMsg, \ HANDLE_##message((hwnd), (wParam), (lParam), (fn))))
/////////////////////// Dialog Box Icon Setting Macro ////////////////////// // Sets the dialog box icons inline
void chSETDLGICONS(HWND hwnd, int idi) { SendMessage(hwnd, WM_SETICON, TRUE,
(LPARAM) LoadIcon((HINSTANCE) GetWindowLongPtr(hwnd, GWLP_HINSTANCE),
MAKEINTRESOURCE(idi))); SendMessage(hwnd, WM_SETICON, FALSE, (LPARAM)
LoadIcon((HINSTANCE) GetWindowLongPtr(hwnd, GWLP_HINSTANCE),
MAKEINTRESOURCE(idi))); } ////////////////////////// UNICODE Check Macro ////////////////////////// //
Since Windows 98 does not support Unicode, issue an error and terminate // the process if this is a
native Unicode build running on Windows 98 // This is accomplished by creating a global C++ object.
Its constructor is // executed before WinMain/main. #ifdef UNICODE class CUnicodeSupported {
public: CUnicodeSupported() { if (GetWindowsDirectoryW(NULL, 0) <= 0) { chMB("This
```

application requires an OS that supports Unicode."); ExitProcess(0); } } }; // "static" stops the linker from complaining that multiple instances of the // object exist when a single project contains multiple source files. static CUnicodeSupported g_UnicodeSupported; #endif //////////////////////////// OS Version Check Macros //////////////////////////// inline void chWindows9xNotAllowed() { OSVERSIONINFO vi = { sizeof(vi) }; GetVersionEx(&vi); if (vi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS) { chMB("This application requires features not present in Windows 9x."); ExitProcess(0); } } inline void chWindows2000Required() { OSVERSIONINFO vi = { sizeof(vi) }; GetVersionEx(&vi); if ((vi.dwPlatformId != VER_PLATFORM_WIN32_NT) && (vi.dwMajorVersion < 5)) { chMB("This application requires features present in Windows 2000."); ExitProcess(0); } } //////////////////////////// SYSTEM_INFO Wrapper Class //////////////////////////// class CSystemInfo : public SYSTEM_INFO { public: CSystemInfo() { GetSystemInfo(this); } }; ////////////////////////////////// End of File //////////////////////////////////

[Previous] [Next]

Appendix B -- The Class Library

While creating the sample applications in this book, I designed several C++ classes to make development easier. The source code for these classes is in the ClassLib directory on the companion CD. This appendix describes these classes.

[Previous] [Next]

The Ensure Cleanup Template C++ Class (EnsureCleanup.h)

Certainly, the code forgetting to close or release a resource is one of the most common programming bugs developers face. To guarantee that a resource is properly cleaned up when its use goes out of scope, I created the CEnsureCleanup template C++ class shown in Listing B-1. Because this class is a template class, it can help you to clean up almost any type of object you can imagine.

I already defined specific instances of the CEnsureCleanup class for the most common Microsoft Windows data types. Defining a type requires three pieces of information: the type of the object, the address of the function used to clean up the object, and a value that indicates that the object is invalid (usually NULL or 0). For example, to clean up a kernel object, the data type is a HANDLE, the function used to clean up the object is *CloseHandle*, and an invalid handle is identified with NULL. Similarly, to clean up a dynamically loaded DLL, the data type is an HMODULE, the function used to clean up the resources is *FreeLibrary*, and an invalid object is identified with NULL. You can see all the different clean-up classes at the end of the CEnsureCleanup.h file.

Using the CEnsureCleanup class could not be easier. All you have to do is use an instance of the class wherever you'd typically use the normal data type. For example, you wouldn't have to write this:

```
HANDLE hfile = CreateFile(...);

CloseHandle(hfile);
```

You'd now write this:

```
CEnsureCloseFile hfile = CreateFile(...);
```

With the C++ class data type, you never have to call *CloseHandle* to release the resource. When the C++ object goes out of scope, the destructor is called and *CloseHandle* will be called for you automatically! In fact, you're guaranteed that the object will be cleaned up even if you return from the middle of a function or if an exception is raised.

By the way, if you're worried about overhead, don't be. I checked the assembly language produced by the optimizing compiler when using these classes, and the code produced is just as efficient as if you had explicitly called the cleanup functions yourself. This is because the CEnsureCleanup class uses all inline functions. I simply love these classes, and I can't believe that I've been programming all these years without them. I'm now using them for almost everything I do.

Listing B-1. *The EnsureCleanup.h header file*

EnsureCleanup.h

```
/******************************************************************************
Module: EnsureCleanup.h Notices: Copyright (c) 2000 Jeffrey Richter Purpose: These classes ensure
object cleanup when an object goes out of scope. See Appendix B.
******************************************************************************/
#pragma once // Include this header file once per compilation unit
////////////////////////////////////////////////////////////////////////////// #include "..\CmnHdr.h" // See Appendix A.
////////////////////////////////////////////////////////////////////////////// // Data type representing the address of the
object's cleanup function. // I used UINT_PTR so that this class works properly in 64-bit Windows.
typedef VOID (WINAPI* PFNENSURECLEANUP)(UINT_PTR);
////////////////////////////////////////////////////////////////////////////// // Each template instantiation requires a data
type, address of cleanup // function, and a value that indicates an invalid value. template<class TYPE,
PFNENSURECLEANUP pfn, UINT_PTR tInvalid = NULL> class CEnsureCleanup { public: //
Default constructor assumes an invalid value (nothing to cleanup) CEnsureCleanup() { m_t = tInvalid;
} // This constructor sets the value to the specified value CEnsureCleanup(TYPE t) : m_t((UINT_PTR)
t) { } // The destructor performs the cleanup. ~CEnsureCleanup() { Cleanup(); } // Helper methods to
tell if the value represents a valid object or not.. BOOL IsValid() { return(m_t != tInvalid); } BOOL
IsInvalid() { return(!IsValid()); } // Re-assigning the object forces the current object to be cleaned-up.
TYPE operator=(TYPE t) { Cleanup(); m_t = (UINT_PTR) t; return(*this); } // Returns the value
(supports both 32-bit and 64-bit Windows). operator TYPE() { // If TYPE is a 32-bit value, cast m_t to
32-bit TYPE // If TYPE is a 64-bit value, case m_t to 64-bit TYPE return((sizeof(TYPE) == 4) ?
(TYPE) PtrToUint(m_t) : (TYPE) m_t); } // Cleanup the object if the value represents a valid object
void Cleanup() { if (IsValid()) { // In 64-bit Windows, all parameters are 64-bits, // so no casting is
required pfn(m_t); // Close the object. m_t = tInvalid; // We no longer represent a valid object. } }
private: UINT_PTR m_t; // The member representing the object };
////////////////////////////////////////////////////////////////////////////// // Macros to make it easier to declare instances
of the template // class for specific data types. #define MakeCleanupClass(className, tData,
pfnCleanup) \ typedef CEnsureCleanup<tData, (PFNENSURECLEANUP) pfnCleanup> className;
#define MakeCleanupClassX(className, tData, pfnCleanup, tInvalid) \ typedef
CEnsureCleanup<tData, (PFNENSURECLEANUP) pfnCleanup, \ (INT_PTR) tInvalid> className;
////////////////////////////////////////////////////////////////////////////// // Instances of the template C++ class for
common data types. MakeCleanupClass(CEnsureCloseHandle, HANDLE, CloseHandle);
MakeCleanupClassX(CEnsureCloseFile, HANDLE, CloseHandle, INVALID_HANDLE_VALUE);
MakeCleanupClass(CEnsureLocalFree, HLOCAL, LocalFree);
MakeCleanupClass(CEnsureGlobalFree, HGLOBAL, GlobalFree);
MakeCleanupClass(CEnsureRegCloseKey, HKEY, RegCloseKey);
MakeCleanupClass(CEnsureCloseServiceHandle, SC_HANDLE, CloseServiceHandle);
MakeCleanupClass(CEnsureCloseWindowStation, HWINSTA, CloseWindowStation);
MakeCleanupClass(CEnsureCloseDesktop, HDESK, CloseDesktop);
```

```
MakeCleanupClass(CEnsureUnmapViewOfFile, PVOID, UnmapViewOfFile);
MakeCleanupClass(CEnsureFreeLibrary, HMODULE, FreeLibrary);
////////////////////////////////////////////////////////////////////////////// // Special class for releasing a reserved region.
// Special class is required because VirtualFree requires 3 parameters class CEnsureReleaseRegion {
public: CEnsureReleaseRegion(PVOID pv = NULL) : m_pv(pv) { } ~CEnsureReleaseRegion() {
Cleanup(); } PVOID operator=(PVOID pv) { Cleanup(); m_pv = pv; return(m_pv); } operator
PVOID() { return(m_pv); } void Cleanup() { if (m_pv != NULL) { VirtualFree(m_pv, 0,
MEM_RELEASE); m_pv = NULL; } } private: PVOID m_pv; };
////////////////////////////////////////////////////////////////////////////// // Special class for freeing a block from a heap
// Special class is required because HeapFree requires 3 parameters class CEnsureHeapFree { public:
CEnsureHeapFree(PVOID pv = NULL, HANDLE hHeap = GetProcessHeap()) : m_pv(pv),
m_hHeap(hHeap) { } ~CEnsureHeapFree() { Cleanup(); } PVOID operator=(PVOID pv) { Cleanup();
m_pv = pv; return(m_pv); } operator PVOID() { return(m_pv); } void Cleanup() { if (m_pv != NULL)
{ HeapFree(m_hHeap, 0, m_pv); m_pv = NULL; } } private: HANDLE m_hHeap; PVOID m_pv; };
////////////////////////////////// End of File //////////////////////////////////
```

[Previous] [Next]

The Print Buffer C++ Class (PrintBuf.h)

Many of the book's sample applications construct strings of information. This information is then placed in a static control or read-only edit control or displayed in a message box. The CPrintBuf class, shown in Listing B-2, makes it easier to construct these strings.

Internally, a CPrintBuf object keeps track of a string buffer. Each time the *Print* or *PrintError* method is called, new string information is appended to the end of the string. As you can see, predicting how much storage is necessary for the string is difficult because the applications don't know ahead of time what will be placed into the string.

So that memory is used efficiently, CPrintBuf's string buffer is initialized by reserving a large region of memory. We default to 64 KB because it doesn't make sense to use a value smaller than the system's allocation granularity value (64 KB on all Windows platforms to date). After this region is reserved, one page of storage is committed to it. If the string data attempts to exceed this storage, an exception is raised and the CPrintBuf object commits more storage to the buffer, growing it as necessary.

The class overloads the PCTSTR cast operator so an instance of the class can be passed directly to functions that require a pointer to a buffer containing a zero-terminated string.

Listing B-2. *The PrintBuf.h header file*

PrintBuf.h

```
/******************************************************************************
Module: PrintBuf.h Notices: Copyright (c) 2000 Jeffrey Richter Purpose: This class wraps allows
sprintf-like operations while automatically growing the resulting data buffer. See Appendix B.
******************************************************************************/
#pragma once // Include this header file once per compilation unit
////////////////////////////////////////////////////////////////////////////// #include "..\CmnHdr.h" // See Appendix A.
#include <StdIO.h> // For _vstprintf ////////////////////////////////////////////////////////////////////////////// class
CPrintBuf { public: CPrintBuf(SIZE_T nMaxSizeInBytes = 64 * 1024); // 64KB is default virtual
~CPrintBuf(); BOOL Print(PCTSTR pszFmt, ...); BOOL PrintError(DWORD dwError =
```

```
GetLastError()); operator PCTSTR() { return(m_pszBuffer); } void Clear(); private: LONG
Filter(EXCEPTION_POINTERS* pep); private: int m_nMaxSizeInBytes; int m_nCurSize; PTSTR
m_pszBuffer; }; ////////////////////////////////////////////////////////////////////////////// #ifdef PRINTBUF_IMPL
////////////////////////////////////////////////////////////////////////////// CPrintBuf::CPrintBuf(SIZE_T
nMaxSizeInBytes) { // This constructor sets initial values of members, and reserves a block // of
addresses of size nMaxSizeInBytes and commits a single page. m_nMaxSizeInBytes =
nMaxSizeInBytes; m_nCurSize = 0; m_pszBuffer = (PTSTR) VirtualAlloc(NULL,
m_nMaxSizeInBytes, MEM_RESERVE, PAGE_READWRITE); chASSERT(m_pszBuffer !=
NULL); chVERIFY(VirtualAlloc(m_pszBuffer, 1, MEM_COMMIT, PAGE_READWRITE) !=
NULL); } ////////////////////////////////////////////////////////////////////////////// CPrintBuf::~CPrintBuf() {
VirtualFree(m_pszBuffer, 0, MEM_RELEASE); }
////////////////////////////////////////////////////////////////////////////// void CPrintBuf::Clear() {
VirtualFree(m_pszBuffer, m_nMaxSizeInBytes, MEM_DECOMMIT);
chVERIFY(VirtualAlloc(m_pszBuffer, 1, MEM_COMMIT, PAGE_READWRITE) != NULL);
m_nCurSize = 0; } ////////////////////////////////////////////////////////////////////////////// LONG
CPrintBuf::Filter(EXCEPTION_POINTERS* pep) { LONG lDisposition =
EXCEPTION_EXECUTE_HANDLER; EXCEPTION_RECORD* per = pep->ExceptionRecord;
__try { // Is exception is an access violation in the data buffer's region? if (per->ExceptionCode !=
EXCEPTION_ACCESS_VIOLATION) __leave; if (!chINRANGE(m_pszBuffer, (PVOID)
per->ExceptionInformation[1], ((PBYTE) m_pszBuffer) + m_nMaxSizeInBytes - 1)) { __leave; } //
Attempt to commit storage to the region if (VirtualAlloc((PVOID)
pep->ExceptionRecord->ExceptionInformation[1], 1, MEM_COMMIT, PAGE_READWRITE) ==
NULL) { __leave; } lDisposition = EXCEPTION_CONTINUE_EXECUTION; } __finally { }
return(lDisposition); } ////////////////////////////////////////////////////////////////////////////// int
CPrintBuf::Print(PCTSTR pszFmt , ...) { // This function appends text to the formatted print buffer. int
nLength = -1; // Assume failure va_list arglist; va_start(arglist, pszFmt); __try { // Append string to
end of buffer nLength = _vstprintf(m_pszBuffer + m_nCurSize, pszFmt, arglist); if (nLength > 0)
m_nCurSize += nLength; } __except (Filter(GetExceptionInformation())) { chMB("CPrintBuf
attempted to go over the maximum size."); DebugBreak(); } va_end(arglist); return(nLength); }
////////////////////////////////////////////////////////////////////////////// BOOL CPrintBuf::PrintError(DWORD dwErr)
{ // Append the last error string text to the buffer. PTSTR pszMsg = NULL; BOOL fOk = (0 !=
FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwErr, 0, (PTSTR) &pszMsg, 0, NULL)); fOk =
fOk && (Print(TEXT("Error %d: %s"), dwErr, pszMsg) >= 0); if (pszMsg != NULL)
LocalFree(pszMsg); return(fOk); } ////////////////////////////////////////////////////////////////////////////// #endif //
PRINTBUF_IMPL //////////////////////////////// End of File ////////////////////////////////
```

[Previous] [Next]

The Auto Buffer Template C++ Class (AutoBuf.h)

Many Windows functions fill buffers that you must allocate. Quite commonly, you have no idea what the proper buffer size is before you call a function. So for many of these Windows functions, you must call the function once to get the required buffer size, then allocate the buffer, and then call the function a second time to actually get the data into the buffer. This, of course, is quite inconvenient. Plus, the buffer requirement might change between calls to the function because Windows is a preemptive multithreading operating system. In such a case, your second call to the function would fail. The likelihood of such a failure is quite rare, but a well-written application should certainly take this into consideration and handle the situation with style and grace. For example, here is the code necessary to properly retrieve a registry value using *RegQueryValueEx*:

```
ULONG uValSize = 1;
PTSTR pszValue = NULL;
```

```
LONG lErr;
do {
   if (uValSize != 0) {
      if (pszValue != NULL) {
         HeapFree(GetProcessHeap(), 0, pszValue);
      }
      pszValue = (PTSTR) HeapAlloc(GetProcessHeap(), 0, uValSize);
      if (pszValue == NULL) {
         // Error: Proper handling not shown
      }
   }
   // Assume that hkey was initialized earlier
   lErr = RegQueryValueEx(hkey, TEXT("SomeValueName"), NULL,
      NULL, (PBYTE) pszValue, &uValSize);
} while (lErr == ERROR_MORE_DATA);
if (lErr != ERROR_SUCCESS){
   // Error: Proper handling not shown
}
```

The next code fragment performs the same functionality but makes use of the CAutoBuf class for allocating the buffer. A bonus feature is that the buffer is automatically freed when the object goes out of scope.

```
CAutoBuf<PTSTR, sizeof(TCHAR)> pszValue;
pszValue = 1;
LONG lErr;
do {
   lErr = RegQueryValueEx(hkey, TEXT("SomeValueName"), NULL, NULL,
      pszValue, pszValue);
} while (lErr == ERROR_MORE_DATA);
if (lErr != ERROR_SUCCESS){
   // Error: Proper handling not shown
}
```

The CAutoBuf C++ class, shown in Listing B-3, makes working with functions that require buffers a cinch. A CAutoBuf object allocates a data buffer whose size is determined automatically by whatever the Windows function tells it. In addition, a CAutoBuf object automatically frees its data buffer when the object goes out of scope.

The CAutoBuf C++ class is a template class, so it supports buffers holding data of any type. To use the class, you simply declare an instance by indicating the type of data you want it to hold. Here is an example of a CAutoBuf that should contain a QUERY_SERVICE_CONFIG structure (which is variable in length):

```
CAutoBuf<QUERY_SERVICE_CONFIG> pServiceConfig;
```

Now, to fill this buffer, you make a call to *QueryServiceConfig* as follows:

```
BOOL fOk;
fOk = QueryServiceConfig(
   hService,          // hService       (SC_HANDLE)
   pServiceConfig,    // pServiceConfig (QUERY_SERVICE_CONFIG*)
   pServiceConfig,    // cbBufferSize   (DWORD)
   pServiceConfig);   // pcbBytesNeeded (PDWORD)
```

You'll notice that I'm passing the pServiceConfig object for three of the parameters. This works because the CAutoBuf class exposes three cast methods on the objects: a cast method that returns the address of the data buffer, a DWORD cast method that returns the size of the data buffer in bytes, and a PDWORD cast method that returns the address of a DWORD member variable that will be filled in with the required size of the buffer.

If the size passed to *cbBufferSize* is too small, *QueryServiceConfig* returns FALSE and a subsequent call to *GetLastError* returns ERROR_ INSUFFICIENT_BUFFER. In this case, all you have to do is call *QueryServiceConfig* again and the buffer will automatically adjust its size to the value returned in the

pcbBytesNeeded parameter. This time, *QueryServiceConfig* should successfully fill the buffer and return nonzero.

However, *QueryServiceConfig* still might fail because another thread could have changed the service's status information. So a well-written application should call *QueryServiceConfig* repeatedly until it succeeds. To make this easy, the *GROWUNTIL* macro is included at the bottom of the AutoBuf.h file. This macro calls the desired function in a loop until the function succeeds or until it fails with an error other than ERROR_INSUFFICIENT_BUFFER or ERROR_ MORE_DATA. Here is an example of how to use this macro:

```
BOOL fOk;
GROWUNTIL(FALSE,
   fOk = QueryServiceConfig(
      hService,          // hService       (SC_HANDLE)
      pServiceConfig,    // pServiceConfig (QUERY_SERVICE_CONFIG*)
      pServiceConfig,    // cbBufferSize   (DWORD)
      pServiceConfig));  // pcbBytesNeeded (DWORD*)
```

Listing B-3. *The AutoBuf.h header file*

AutoBuf.h

/**
Module: AutoBuf.h Notices: Copyright (c) 2000 Jeffrey Richter Purpose: This class manages an auto-sizing data buffer. See Appendix B.
**/
#pragma once // Include this header file once per compilation unit
// #include "..\CmnHdr.h" // See Appendix A.
////////////////// CAutoBuf Template C++ Class Description ////////////////// /* The CAutoBuf template C++ class implements type safe buffers that automatically grow to meet the needs of your code. Memory is also automatically freed when the object is destroyed (typically when your code goes out of frame and it is popped off of the stack). Examples of use: // Create a buffer with no explicit data type, // the buffer grown in increments of a byte CAutoBuf<PVOID> buf; // Create a buffer of TCHARs, // the buffer grows in increments of sizeof(TCHAR) CAutoBuf<PTSTR, sizeof(TCHAR)> buf; // Force the buffer to be 10 bytes big buf = 10; */ // // This class is only ever used as a base class of the CAutoBuf template class. // The base class exists so that all instances of the template class share // a single instance of the common code. class CAutoBufBase { public: UINT Size() { return(* (PDWORD) PSize()); } UINT Size(UINT uSize); PUINT PSize() { AdjustBuffer(); m_uNewSize = m_uCurrentSize; return(&m_uNewSize); } void Free() { Reconstruct(); } protected: CAutoBufBase(PBYTE *ppbData, int nMult) { m_nMult = nMult; m_ppbBuffer = ppbData; // Derived class holds address of buffer to allow // debugger's Quick Watch to work with typed data. Reconstruct(TRUE); } virtual ~CAutoBufBase() { Free(); } void Reconstruct(BOOL fFirstTime = FALSE); PBYTE Buffer() { AdjustBuffer(); return(*m_ppbBuffer); } private: void AdjustBuffer(); private: PBYTE* m_ppbBuffer; // Address of address of data buffer int m_nMult; // Multiplier (in bytes) used for buffer growth UINT m_uNewSize; // Requested buffer size (in m_nMult units) UINT m_uCurrentSize; // Actual size (in m_nMult units) }; // template <class TYPE, int MULT = 1> class CAutoBuf : private CAutoBufBase { public: CAutoBuf() : CAutoBufBase((PBYTE*) &m_pData, MULT) {} void Free() { CAutoBufBase::Free(); } public: operator TYPE*() { return(Buffer()); } UINT operator=(UINT uSize) { return(CAutoBufBase::Size(uSize)); } operator UINT() { return(Size()); } operator ULONG() { return(Size()); } operator PUINT() { return(PSize()); } operator PLONG() { return((PLONG) PSize()); } operator PULONG() { return((PULONG) PSize()); } operator PBYTE() { return((PBYTE) Buffer()); } operator PVOID() { return((PVOID) Buffer()); } TYPE& operator[](int nIndex) { return(*(Buffer() + nIndex)); } private: TYPE* Buffer() {

return((TYPE*) CAutoBufBase::Buffer()); } private: TYPE* m_pData; }; // #define GROWUNTIL(fail, func) \ do { \ if ((func) != (fail)) \ break; \ } while ((GetLastError() == ERROR_MORE_DATA) || \ (GetLastError() == ERROR_INSUFFICIENT_BUFFER)); // #ifdef AUTOBUF_IMPL // void CAutoBufBase::Reconstruct(BOOL fFirstTime) { if (!fFirstTime) { if (*m_ppbBuffer != NULL) HeapFree(GetProcessHeap(), 0, *m_ppbBuffer); } *m_ppbBuffer = NULL; // Derived class doesn't point to a data buffer m_uNewSize = 0; // Initially, buffer has no bytes in it m_uCurrentSize = 0; // Initially, buffer has no bytes in it } // UINT CAutoBufBase::Size(UINT uSize) { // Set buffer to desired number of m_nMult bytes. if (uSize == 0) { Reconstruct(); } else { m_uNewSize = uSize; AdjustBuffer(); } return(m_uNewSize); } // void CAutoBufBase::AdjustBuffer() { if (m_uCurrentSize < m_uNewSize) { // We're growing the buffer HANDLE hHeap = GetProcessHeap(); if (*m_ppbBuffer != NULL) { // We already have a buffer, re-size it PBYTE pNew = (PBYTE) HeapReAlloc(hHeap, 0, *m_ppbBuffer, m_uNewSize * m_nMult); if (pNew != NULL) { m_uCurrentSize = m_uNewSize; *m_ppbBuffer = pNew; } } else { // We don't have a buffer, create new one. *m_ppbBuffer = (PBYTE) HeapAlloc(hHeap, 0, m_uNewSize * m_nMult); if (*m_ppbBuffer != NULL) m_uCurrentSize = m_uNewSize; } } } // #endif // AUTOBUF_IMPL ////////////////////////////////// End of File //////////////////////////////////

[Previous] [Next]

The UI Layout C++ Class (UILayout.h)

Many of the sample applications in this book allow the user to resize the main window. Resizing the main window, of course, causes all the child controls to be repositioned or resized. The CUILayout C++ class, shown in Listing B-4, encapsulates all the child control reposition and resize logic.

Each application that supports a resizable main window will create a CUILayout object. In that window's *Dlg_OnInitDialog* function, the CUILayout object's *Initialize* method is called to set the parent window handle and the minimum width and height of the window. Then the CUILayout object's *AnchorControl* or *AnchorControls* methods are called repeatedly to identify each child control window and how that control should be anchored relative to its parent window.

When the main window executes its *Dlg_OnSize* function, it just needs to call the CUILayout object's *AdjustControls* method, passing it the width and height of the main window's client area. The *AdjustControls* method will automatically reposition and resize all the child controls so that they keep their relative position to the parent window.

When the main window executes its *Dlg_OnGetMinMaxInfo* function, it needs to call the CUILayout object's *HandleMinMax* method. This method sets the members of the MINMAXINFO structure so that the user cannot shrink the window too small and obscure any of the child controls.

Listing B-4. *The UILayout.h header file*

UILayout.h

/**
Module: UILayout.h Notices: Copyright (c) 2000 Jeffrey Richter Purpose: This class manages child window positioning and sizing when a parent window is resized. See Appendix B.

```
******************************************************************************/
#pragma once // Include this header file once per compilation unit
////////////////////////////////////////////////////////////////////////////// #include "..\CmnHdr.h" // See Appendix A.
////////////////////////////////////////////////////////////////////////////// class CUILayout { public: enum
ANCHORPOINT { AP_TOPLEFT, AP_TOPMIDDLE, AP_TOPRIGHT, AP_MIDDLERIGHT,
AP_BOTTOMRIGHT, AP_BOTTOMMIDDLE, AP_BOTTOMLEFT, AP_MIDDLELEFT,
AP_CENTER }; public: void Initialize(HWND hwndParent, int nMinWidth = 0, int nMinHeight = 0);
BOOL AnchorControl(ANCHORPOINT apUpperLeft, ANCHORPOINT apLowerRight, int nID,
BOOL fRedraw = FALSE); BOOL AnchorControls(ANCHORPOINT apUpperLeft,
ANCHORPOINT apLowerRight, BOOL fRedraw, ...); BOOL AdjustControls(int cx, int cy); void
HandleMinMax(PMINMAXINFO pMinMax) { pMinMax->ptMinTrackSize = m_ptMinParentDims;
} private: struct CONTROL { int m_nID; BOOL m_fRedraw; ANCHORPOINT m_apUpperLeft;
ANCHORPOINT m_apLowerRight; POINT m_ptULDelta; POINT m_ptLRDelta; }; private: void
PixelFromAnchorPoint(ANCHORPOINT ap, int cxParent, int cyParent, PPOINT ppt); private:
CONTROL m_CtrlInfo[255]; // Max controls allowed in a dialog template int m_nNumControls;
HWND m_hwndParent; POINT m_ptMinParentDims; };
////////////////////////////////////////////////////////////////////////////// #ifdef UILAYOUT_IMPL
////////////////////////////////////////////////////////////////////////////// void CUILayout::Initialize(HWND
hwndParent, int nMinWidth, int nMinHeight) { m_hwndParent = hwndParent; m_nNumControls = 0;
if ((nMinWidth == 0) || (nMinHeight == 0)) { RECT rc; GetWindowRect(m_hwndParent, &rc);
m_ptMinParentDims.x = rc.right - rc.left; m_ptMinParentDims.y = rc.bottom - rc.top; } if (nMinWidth
!= 0) m_ptMinParentDims.x = nMinWidth; if (nMinHeight != 0) m_ptMinParentDims.y =
nMinHeight; } ////////////////////////////////////////////////////////////////////////////// BOOL
CUILayout::AnchorControl(ANCHORPOINT apUpperLeft, ANCHORPOINT apLowerRight, int
nID, BOOL fRedraw) { BOOL fOk = FALSE; try { { HWND hwndControl =
GetDlgItem(m_hwndParent, nID); if (hwndControl == NULL) goto leave; if (m_nNumControls >=
chDIMOF(m_CtrlInfo)) goto leave; m_CtrlInfo[m_nNumControls].m_nID = nID;
m_CtrlInfo[m_nNumControls].m_fRedraw = fRedraw;
m_CtrlInfo[m_nNumControls].m_apUpperLeft = apUpperLeft;
m_CtrlInfo[m_nNumControls].m_apLowerRight = apLowerRight; RECT rcControl;
GetWindowRect(hwndControl, &rcControl); // Screen coords of control // Convert coords to
parent-relative coordinates MapWindowPoints(HWND_DESKTOP, m_hwndParent, (PPOINT)
&rcControl, 2); RECT rcParent; GetClientRect(m_hwndParent, &rcParent); POINT pt;
PixelFromAnchorPoint(apUpperLeft, rcParent.right, rcParent.bottom, &pt);
m_CtrlInfo[m_nNumControls].m_ptULDelta.x = pt.x - rcControl.left;
m_CtrlInfo[m_nNumControls].m_ptULDelta.y = pt.y - rcControl.top;
PixelFromAnchorPoint(apLowerRight, rcParent.right, rcParent.bottom, &pt);
m_CtrlInfo[m_nNumControls].m_ptLRDelta.x = pt.x - rcControl.right;
m_CtrlInfo[m_nNumControls].m_ptLRDelta.y = pt.y - rcControl.bottom; m_nNumControls++; fOk =
TRUE; } leave:; } catch (...) { } chASSERT(fOk); return(fOk); }
////////////////////////////////////////////////////////////////////////////// BOOL
CUILayout::AnchorControls(ANCHORPOINT apUpperLeft, ANCHORPOINT apLowerRight,
BOOL fRedraw, ...) { BOOL fOk = TRUE; va_list arglist; va_start(arglist, fRedraw); int nID =
va_arg(arglist, int); while (fOk && (nID != -1)) { fOk = fOk && AnchorControl(apUpperLeft,
apLowerRight, nID, fRedraw); nID = va_arg(arglist, int); } va_end(arglist); return(fOk); }
////////////////////////////////////////////////////////////////////////////// BOOL CUILayout::AdjustControls(int cx, int
cy) { BOOL fOk = FALSE; // Create region consisting of all areas occupied by controls HRGN
hrgnPaint = CreateRectRgn(0, 0, 0, 0); for (int n = 0; n < m_nNumControls; n++) { HWND
hwndControl = GetDlgItem(m_hwndParent, m_CtrlInfo[n].m_nID); RECT rcControl;
GetWindowRect(hwndControl, &rcControl); // Screen coords of control // Convert coords to
parent-relative coordinates MapWindowPoints(HWND_DESKTOP, m_hwndParent, (PPOINT)
&rcControl, 2); HRGN hrgnTemp = CreateRectRgnIndirect(&rcControl); CombineRgn(hrgnPaint,
hrgnPaint, hrgnTemp, RGN_OR); DeleteObject(hrgnTemp); } for (n = 0; n < m_nNumControls; n++)
{ // Get control's upper/left position w/respect to parent's width/height RECT rcControl;
```

PixelFromAnchorPoint(m_CtrlInfo[n].m_apUpperLeft, cx, cy, (PPOINT) &rcControl); rcControl.left -= m_CtrlInfo[n].m_ptULDelta.x; rcControl.top -= m_CtrlInfo[n].m_ptULDelta.y; // Get control's lower/right position w/respect to parent's width/height PixelFromAnchorPoint(m_CtrlInfo[n].m_apLowerRight, cx, cy, (PPOINT) &rcControl.right); rcControl.right -= m_CtrlInfo[n].m_ptLRDelta.x; rcControl.bottom -= m_CtrlInfo[n].m_ptLRDelta.y; // Position/size the control HWND hwndControl = GetDlgItem(m_hwndParent, m_CtrlInfo[n].m_nID); MoveWindow(hwndControl, rcControl.left, rcControl.top, rcControl.right - rcControl.left, rcControl.bottom - rcControl.top, FALSE); if (m_CtrlInfo[n].m_fRedraw) { InvalidateRect(hwndControl, NULL, FALSE); } else { // Remove the regions occupied by the control's new position HRGN hrgnTemp = CreateRectRgnIndirect(&rcControl); CombineRgn(hrgnPaint, hrgnPaint, hrgnTemp, RGN_DIFF); DeleteObject(hrgnTemp); // Make the control repaint itself InvalidateRect(hwndControl, NULL, TRUE); SendMessage(hwndControl, WM_NCPAINT, 1, 0); UpdateWindow(hwndControl); } } // Paint the newly exposed portion of the dialog box's client area HDC hdc = GetDC(m_hwndParent); HBRUSH hbrColor = CreateSolidBrush(GetSysColor(COLOR_3DFACE)); FillRgn(hdc, hrgnPaint, hbrColor); DeleteObject(hbrColor); ReleaseDC(m_hwndParent, hdc); DeleteObject(hrgnPaint); return(fOk); } // void CUILayout::PixelFromAnchorPoint(ANCHORPOINT ap, int cxParent, int cyParent, PPOINT ppt) { ppt->x = ppt->y = 0; switch (ap) { case AP_TOPMIDDLE: case AP_CENTER: case AP_BOTTOMMIDDLE: ppt->x = cxParent / 2; break; case AP_TOPRIGHT: case AP_MIDDLERIGHT: case AP_BOTTOMRIGHT: ppt->x = cxParent; break; } switchl (ap) { case AP_MIDDLELEFT: case AP_CENTER: case AP_MIDDLERIGHT: ppt->y = cyParent / 2; break; case AP_BOTTOMLEFT: case AP_BOTTOMMIDDLE: case AP_BOTTOMRIGHT: ppt->y = cyParent; break; } } // #endif // UILAYOUT_IMPL ////////////////////////////////// End of File //////////////////////////////////

[Previous] [Next]

The I/O Completion Port C++ Class (IOCP.h)

The simple CIOCP C++ class, shown in Listing B-5, just wraps the Windows I/O completion port kernel object functions. All the methods are inline, so there is no performance penalty when you use the class. Wrapping the functions allows you to manipulate an I/O completion port object with a more logical interface.

Listing B-5. *The IOCP.h header file*

IOCP.h

/**
Module: IOCP.h Notices: Copyright (c) 2000 Jeffrey Richter Purpose: This class wraps an I/O Completion Port. See Appendix B.
**/
#pragma once // Include this header file once per compilation unit
// #include "..\CmnHdr.h" // See Appendix A.
// class CIOCP { public: CIOCP(int nMaxConcurrency = -1) { m_hIOCP = NULL; if (nMaxConcurrency != -1) (void) Create(nMaxConcurrency); } ~CIOCP() { if (m_hIOCP != NULL) chVERIFY(CloseHandle(m_hIOCP)); } BOOL Create(int nMaxConcurrency = 0) { m_hIOCP = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, nMaxConcurrency); chASSERT(m_hIOCP != NULL); return(m_hIOCP != NULL); } BOOL AssociateDevice(HANDLE hDevice, ULONG_PTR CompKey) { BOOL fOk = (CreateIoCompletionPort(hDevice, m_hIOCP,

```
CompKey, 0) == m_hIOCP); chASSERT(fOk); return(fOk); } BOOL AssociateSocket(SOCKET
hSocket, ULONG_PTR CompKey) { return(AssociateDevice((HANDLE) hSocket, CompKey)); }
BOOL PostStatus(ULONG_PTR CompKey, DWORD dwNumBytes = 0, OVERLAPPED* po =
NULL) { BOOL fOk = PostQueuedCompletionStatus(m_hIOCP, dwNumBytes, CompKey, po);
chASSERT(fOk); return(fOk); } BOOL GetStatus(ULONG_PTR* pCompKey, PDWORD
pdwNumBytes, OVERLAPPED** ppo, DWORD dwMilliseconds = INFINITE) {
return(GetQueuedCompletionStatus(m_hIOCP, pdwNumBytes, pCompKey, ppo, dwMilliseconds)); }
private: HANDLE m_hIOCP; }; ////////////////////////////////// End of File //////////////////////////////////
```

[Previous] [Next]

The Security Information C++ Class (SecInfo.h)

The CSecInfo class, shown in Listing B-6, is a very thin wrapper around the *ISecurityInformation* COM interface, which is used in calls to *EditSecurity* to produce the common dialog for editing the security for an object. The CSecInfo class simply implements the required members of the *IUnknown* interface so your code does not have to worry about the COM-related aspects of working with *EditSecurity*.

If you choose to use the CSecInfo class, you should derive your own class from it and implement the pure virtual functions of the base class.

Listing B-6. *The SecInfo.h header file*

SecInfo.h

```
/******************************************************************************
Module: SecInfo.h Notices: Copyright (c) 2000 Jeffrey Richter Purpose: This class wraps the
ISecurityInformation interface, which is used in calls to the EditSecurity function. See Appendix B.
******************************************************************************/
#pragma once // Include this header file once per compilation unit
//////////////////////////////////////////////////////////////////////////////// #include "..\CmnHdr.h" // See Appendix A.
#include <aclapi.h> #include <aclui.h> //////////////////////////////////////////////////////////////////////////////// class
CSecInfo: public ISecurityInformation { public: CSecInfo() { m_nRef = 1; m_fMod = FALSE; }
BOOL IsModified() { return(m_fMod); } protected: virtual ~CSecInfo() {} protected: void Modified()
{ m_fMod = TRUE; } static GUID m_guidNULL; static SI_ACCESS m_siAccessAllRights[]; private:
ULONG m_nRef; BOOL m_fMod; public: HRESULT WINAPI QueryInterface(REFIID riid,
PVOID* ppvObj); ULONG WINAPI AddRef(); ULONG WINAPI Release(); HRESULT
UseStandardAccessRights(const GUID* pguidObjectType, DWORD dwFlags, PSI_ACCESS*
ppAccess, ULONG* pcAccesses, ULONG* piDefaultAccess); protected: HRESULT WINAPI
GetObjectInformation(PSI_OBJECT_INFO pObjectInfo) = 0; HRESULT WINAPI
GetSecurity(SECURITY_INFORMATION RequestedInformation, PSECURITY_DESCRIPTOR*
ppSecurityDescriptor, BOOL fDefault) = 0; HRESULT WINAPI
SetSecurity(SECURITY_INFORMATION SecurityInformation, PSECURITY_DESCRIPTOR
pSecurityDescriptor) = 0; HRESULT WINAPI GetAccessRights(const GUID* pguidObjectType,
DWORD dwFlags, // SI_EDIT_AUDITS, SI_EDIT_PROPERTIES PSI_ACCESS *ppAccess,
ULONG *pcAccesses, ULONG *piDefaultAccess) = 0; HRESULT WINAPI MapGeneric(const
GUID *pguidObjectType, UCHAR *pAceFlags, ACCESS_MASK *pMask) = 0; HRESULT WINAPI
GetInheritTypes(PSI_INHERIT_TYPE* ppInheritTypes, ULONG *pcInheritTypes); HRESULT
WINAPI PropertySheetPageCallback(HWND hwnd, UINT uMsg, SI_PAGE_TYPE uPage);
PSECURITY_DESCRIPTOR LocalAllocSDCopy(PSECURITY_DESCRIPTOR psd); };
//////////////////////////////////////////////////////////////////////////////// #ifdef SECINFO_IMPL
```

```
////////////////////////////////////////////////////////////////////////////// GUID CSecInfo::m_guidNULL =
GUID_NULL; #define RIGHT(code, text, fGeneral, fSpecific) \ { &m_guidNULL, code, L ## text, \
(0 | (fGeneral ? SI_ACCESS_GENERAL : 0) | \ (fSpecific ? SI_ACCESS_SPECIFIC : 0)) } static
SI_ACCESS CSecInfo::m_siAccessAllRights[] = { RIGHT(DELETE, "DELETE", TRUE, FALSE),
RIGHT(READ_CONTROL, "READ_CONTROL", TRUE, FALSE), RIGHT(WRITE_DAC,
"WRITE_DAC", TRUE, FALSE), RIGHT(WRITE_OWNER, "WRITE_OWNER", TRUE, FALSE),
RIGHT(SYNCHRONIZE, "SYNCHRONIZE", TRUE, FALSE),
RIGHT(STANDARD_RIGHTS_REQUIRED, "STANDARD_RIGHTS_REQUIRED", TRUE,
FALSE), RIGHT(STANDARD_RIGHTS_READ, "STANDARD_RIGHTS_READ", TRUE,
FALSE), RIGHT(STANDARD_RIGHTS_WRITE, "STANDARD_RIGHTS_WRITE", TRUE,
FALSE), RIGHT(STANDARD_RIGHTS_EXECUTE, "STANDARD_RIGHTS_EXECUTE", TRUE,
FALSE), RIGHT(STANDARD_RIGHTS_ALL, "STANDARD_RIGHTS_ALL", TRUE, FALSE),
RIGHT(SPECIFIC_RIGHTS_ALL, "SPECIFIC_RIGHTS_ALL", TRUE, FALSE),
RIGHT(ACCESS_SYSTEM_SECURITY, "ACCESS_SYSTEM_SECURITY", TRUE, FALSE),
RIGHT(MAXIMUM_ALLOWED, "MAXIMUM_ALLOWED", TRUE, FALSE), };
////////////////////////////////////////////////////////////////////////////// PSECURITY_DESCRIPTOR
CSecInfo::LocalAllocSDCopy(PSECURITY_DESCRIPTOR pSD) { DWORD dwSize = 0;
SECURITY_DESCRIPTOR_CONTROL sdc; PSECURITY_DESCRIPTOR pSDNew = NULL;
DWORD dwVersion; __try { if (pSD == NULL) __leave; if (!GetSecurityDescriptorControl(pSD,
&sdc, &dwVersion)) __leave; if ((sdc & SE_SELF_RELATIVE) != 0) { dwSize =
GetSecurityDescriptorLength(pSD); if (dwSize == 0) __leave; pSDNew = LocalAlloc(LPTR,
dwSize); if (pSDNew == NULL) __leave; CopyMemory(pSDNew, pSD, dwSize); } else { if
(MakeSelfRelativeSD(pSD, NULL, &dwSize)) __leave; else if (GetLastError() !=
ERROR_INSUFFICIENT_BUFFER) __leave; pSDNew = LocalAlloc(LPTR, dwSize); if (pSDNew
== NULL) __leave; if (!MakeSelfRelativeSD(pSD, pSDNew, &dwSize)) { LocalFree(pSDNew);
pSDNew = NULL; } } } __finally { } return(pSDNew); }
////////////////////////////////////////////////////////////////////////////// HRESULT CSecInfo::QueryInterface(REFIID
riid, PVOID* ppvObj) { HRESULT hr = E_NOINTERFACE; if ((riid == IID_ISecurityInformation) ||
(riid == IID_IUnknown)) { *ppvObj = this; AddRef(); hr = S_OK; } return(hr); }
////////////////////////////////////////////////////////////////////////////// ULONG CSecInfo::AddRef() { m_nRef++;
return(m_nRef); } ////////////////////////////////////////////////////////////////////////////// ULONG
CSecInfo::Release() { ULONG nRef = --m_nRef; if (m_nRef == 0) delete this; return(nRef); }
////////////////////////////////////////////////////////////////////////////// HRESULT
CSecInfo::UseStandardAccessRights(const GUID* pguidObjectType, DWORD dwFlags,
PSI_ACCESS* ppAccess, ULONG* pcAccesses, ULONG* piDefaultAccess) { *ppAccess =
m_siAccessAllRights; *pcAccesses = chDIMOF(m_siAccessAllRights); *piDefaultAccess = 0;
return(S_OK); } ////////////////////////////////////////////////////////////////////////////// HRESULT
CSecInfo::GetInheritTypes(PSI_INHERIT_TYPE* ppInheritTypes, ULONG* pcInheritTypes) {
*ppInheritTypes = NULL; *pcInheritTypes = 0; return(S_OK); }
////////////////////////////////////////////////////////////////////////////// HRESULT
CSecInfo::PropertySheetPageCallback(HWND hwnd, UINT uMsg, SI_PAGE_TYPE uPage) {
return(S_OK); } ////////////////////////////////////////////////////////////////////////////// #pragma comment(lib,
"ACLUI.lib") // Force linking against this library
////////////////////////////////////////////////////////////////////////////// #endif // SECINFO_IMPL
////////////////////////////////// End of File //////////////////////////////////
```

[Previous] [Next]

Jeffrey Richter

Jeffrey Richter gives programming seminars to software developers and is available for consulting. His clients include such companies as Allen-Bradley, AT&T, Caterpillar, Digital, DreamWorks, GE Medical Systems, Hewlett-Packard, IBM, Intel, Intuit, Microsoft, Pitney Bowes, Sybase, Tandem, and Unisys.

Jeff also speaks regularly at industry conferences, including Software Development and COMDEX, Boston University's WinDev, Microsoft's Professional Developer's Conference, and TechEd. Jeff is also a contributing editor to *MSDN Magazine,* for which he authors the Win32 Q & A column and has written several feature articles.

Jeff lives in Bellevue, Washington. His hobbies include helicopter flying, magic, and drumming. He loves *The Simpsons* and has a few animation cells. He also has a passion for classic rock and jazz-fusion bands.

You may reach Jeff via his Web site: *http://www.JeffreyRichter.com*.

[Previous] [Next]

Jason D. Clark

Jason D. Clark is an accomplished Microsoft Windows developer and consultant. His specialties are base-services programming and developing software that takes full advantage of the security features in Windows. Jason frequently writes articles in publications such as *Microsoft System Journal, Dr. Dobb's Journal,* and *Windows Developer's Journal*. He makes regular contributions to the Dr. GUI column of MSDN as well as to the Microsoft Knowledge Base. In addition to publishing texts on software development for the Windows operating system, Jason occasionally speaks on topics in his area of expertise at developer conferences.

Jason has been writing software for Windows operating systems for nine years, seven of which he spent as a software developer/consultant in California. Some of his major clients included IBM, Seiko, Sony, and First American Mortgage. During those years, Jason's interest in Windows software development became a passion, and he accepted a job with Microsoft in 1997.

Jason now lives in Seattle where he enjoys the intriguing and unique culture of one of Washington's finest cities. You can frequently find him enjoying a sushi dinner with his wife, after which he may retire to one of Seattle's many microbreweries and enjoy a beer.

You can find up-to-date information about Jason (as well as this book) at *www.JasonDClark.com*. You can also e-mail him directly at jclark@microsoft.com.

[Previous] [Next]

About This Electronic Book

This electronic book was originally created—and still may be purchased—as a print book. For simplicity, the electronic version of this book has been modified as little as possible from its original form. For instance, there may be occasional references to sample files that come with the book. These files are available with the print version, but are not provided in this electronic edition.

“Expanding” Graphics

Many of the graphics shown in this book are quite large. To improve the readability of the book, reduced versions of these graphics are shown in the text. To see a full-size version, click on the reduced graphic.