Part of the five-volume
Networking Services Developer's Reference Library

Microsoft®

The essential reference set for developing with Microsoft® Windows® networking technologies

David Iseminger
Series Editor
www.iseminger.com

Network Protocols and Interfaces

BASED ON
msdn™ library

Microsoft®

David Iseminger
Series Editor

Network Protocols and Interfaces

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

Library of Congress Cataloging-in-Publication Data
Iseminger, David, 1969-
Networking Services Developer's Reference Library / David Iseminger.
p. cm.
ISBN 0-7356-0993-4
1. Application Software--Development. 2. Microsoft Windows (Computer file). 3. Computer networks. I. Title.
QA76.76.A65 I84 2000
005.4'4769--dc21 00-020241

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: Wendy Zucker

Part No. 097-0002784

Acknowledgements

First, thanks to **Ben Ryan** at Microsoft Press for continuing to share my enthusiasm about the series. Many thanks to Ben and **Steve Guty** for also managing the business details associated with publishing this series. We're just getting started!

Wendy Zucker again kept step with the difficult and tight schedule at Microsoft Press and orchestrated things in the way only project editors can endure. **John Pierce** was also instrumental in seeing the publishing process through completion, many thanks to both of them. The cool cover art that will continue through the series is directed by **Greg Hickman**—thanks for the excellent work. I'm a firm believer that artwork and packaging are integral to the success of a project.

Thanks also to the marketing team at Microsoft Press that handles this series: **Cora McLaughlin** and **Cheri Chapman** on the front lines and **Jocelyn Paul** each deserve recognition for their coordination efforts with MSDN, openness to my ideas and suggestions, creative marketing efforts, and other feats of marketing ingenuity.

On the Windows SDK side of things, thanks again to **Morgan Seeley** for introducing me to the editor at Microsoft Press, and thereby routing this series to the right place.

Thanks also to **Margot (Maley) Hutchison** for doing all those agent-ish things so well.

Author's Note In Part 2 you'll see some code blocks that have unusual margin settings, or code that wraps to a subsequent line. This is a result of physical page constraints of printed material; the original code in these places was indented too much to keep its printed form on one line. I've reviewed every line of code in this library in an effort to ensure it reads as well as possible (for example, modifying comments to keep them on one line, and to keep line-delimited comment integrity). In some places, however, the word wrap effect couldn't be avoided. As such, please ensure that you check closely if you use and compile these examples.

Contents

Part 2

CHAPTER 1

Getting Around in the Networking Services Library

Networking is pervasive in this digital age in which we live. Information at your fingertips, distributed computing, name resolution, and indeed the entire Internet—the advent of which will be ascribed to our generation for centuries to come—imply and require networking. Everything that has become the buzz of our business and personal lives, including e-mail, cell phones, and Web surfing, is enabled by the fact that networking has been brought to the masses (and we've barely scraped the beginning of the trend). You, the network-enabled Windows application developer, need to know how to lasso this all-important networking services capability and make it a part of your application. You've come to the right place.

Networking isn't magic, but it can seem that way to those who aren't accustomed to it (or to the programmer who isn't familiar with the technologies or doesn't know how to make networking part of his or her application). That's why the *Networking Services Developer's Reference Library* isn't just a collection of programmatic reference information; it would be only half-complete if it were. Instead, the Networking Services Library is a collection of explanatory and reference information that combine to provide you with the complete set that you need to create today's network-enabled Windows application.

The Networking Services Library is *the* comprehensive reference guide to network-enabled application development. This library, like all libraries in the Windows Programming Reference Series (WPRS), is designed to deliver the most complete, authoritative, and accessible reference information available on a given subject of Windows network programming—without sacrificing focus. Each book in each library is dedicated to a logical group of technologies or development concerns; this approach has been taken specifically to enable you to find the information you need quickly, efficiently, and intuitively.

In addition to its networking services development information, the Networking Services Library contains tips designed to make your programming life easier. For example, a thorough explanation and detailed tour of MSDN Online is included, as is a section that helps you get the most out of your MSDN subscription. Just in case you don't have an MSDN subscription, or don't know why you should, I've included information about that too, including the differences between the three levels of MSDN subscription, what each level offers, and why you'd want a subscription when MSDN Online is available over the Internet.

To ensure that you don't get lost in all the information provided in the Networking Services Library, each volume's appendixes provide an all-encompassing programming directory to help you easily find the particular programming element you're looking for. This directory suite, which covers all the functions, structures, enumerations, and other programming elements found in network-enabled application development, gets you quickly to the volume and page you need, saving you hours of time and bucketsful of frustration.

How the Networking Services Library Is Structured

The Networking Services Library consists of five volumes, each of which focuses on a particular aspect of network programming. These programming reference volumes have been divided into the following:

- Volume 1: Winsock and QOS
- Volume 2: Network Interfaces and Protocols
- Volume 3: RPC and WNet
- Volume 4: Remote Access Services
- Volume 5: Routing

Dividing the Networking Services Library into these categories enables you to quickly identify the Networking Services volume you need, based on your task, and facilitates your maintenance of focus for that task. This approach enables you to keep one reference book open and handy, or tucked under your arm while researching that aspect of Windows programming on sandy beaches, without risking back problems (from toting around all 3,000+ pages of the Networking Services Library) and without having to shuffle among multiple less-focused books.

Within the Networking Services Library—and in fact, in all WPRS Libraries—each volume has a deliberate structure. This per-volume structure has been created to further focus the reference material in a developer-friendly manner, to maintain consistency within each volume and each Library throughout the series, and to enable you to easily gather the information you need. To that end, each volume in the Networking Services Library contains the following parts:

- Part 1: Introduction and Overview
- Part 2: Guides, Examples, and Programmatic Reference
- Part 3: Intelligently Structured Indexes

Part 1 provides an introduction to the Networking Services Library and to the WPRS (what you're reading now), and a handful of chapters designed to help you get the most out of networking technologies, MSDN, and MSDN Online. MSDN and WPRS Libraries are your tools in the developer process; knowing how to use them to their fullest will enable you to be more efficient and effective (both of which are generally desirable traits). In certain volumes (where appropriate), I've also provided additional information that you'll need in your network-enabled development efforts, and included such information as concluding chapters in Part 1. For example, Volume 3 includes a chapter that explains terms used throughout the RPC development documentation; by putting it into Chapter 5 of that volume, you always know where to go when you have a question about an RPC term. Some of the other volumes in the Networking Services Library conclude their Part 1 with chapters that include information crucial to their volume's contents, but I've been very selective about including such information. Publishing constraints have limited the amount of information I can provide in each volume (and in the library as a whole), so I've focused on the priority: getting you the most useful information possible within the number of pages I have to work with.

Part 2 contains the networking reference material particular to its volume. You'll notice that each volume contains *much* more than simple collections of function and structure definitions. A comprehensive reference resource should include information about how to use a particular technology, as well as definitions of programming elements. Consequently, the information in Part 2 combines complete programming element definitions with instructional and explanatory material for each programming area.

Part 3 is a collection of intelligently arranged and created indexes. One of the biggest challenges of the IT professional is finding information in the sea of available resources and network programming is probably one of the most complex and involved of any development discipline. In order to help you get a handle on network programming references (and Microsoft technologies in general), Part 3 puts all such information into an understandable, manageable directory (in the form of indexes) that enables you to quickly find the information you need.

How the Networking Services Library Is Designed

The Networking Services Library (and all libraries in the WPRS) is designed to deliver the most pertinent information in the most accessible way possible. The Networking Services Library is also designed to integrate seamlessly with MSDN and MSDN Online by providing a look and feel consistent with their electronic means of disseminating Microsoft reference information. In other words, the way a given function reference appears on the pages of this book has been designed specifically to emulate the way that MSDN and MSDN Online present their function reference pages.

The reason for maintaining such integration is simple: to make it easy for you to use the tools and get the ongoing information you need to create quality programs. Providing a "common interface" among reference resources allows your familiarity with the Networking Services Library reference material to be immediately applied to MSDN or MSDN Online, and vice-versa. In a word, it means *consistency*.

You'll find this philosophy of consistency and simplicity applied throughout WPRS publications. I've designed the series to go hand-in-hand with MSDN and MSDN Online resources. Such consistency lets you leverage your familiarity with electronic reference material, then apply that familiarity to enable you to get away from your computer if you'd like, take a book with you, and—in the absence of keyboards and e-mail and upright chairs—get your programming reading and research done. Of course, each of the Networking Services Library volumes fits nicely right next to your mouse pad as well, even when opened to a particular reference page.

With any job, the simpler and more consistent your tools are, the more time you can spend doing work rather than figuring out how to use your tools. The structure and design of the Networking Services Library provide you with a comprehensive, presharpened toolset to build compelling Windows applications.

CHAPTER 2

What's In This Volume?

Volume 2 of the *Networking Services Developer's Reference Library* provides information about the vast array of interfaces and protocols (and even some services) that Windows networking makes available to applications developers. These interfaces, protocols, and services enable network programmers to get the most out of their network-enabled applications.

This volume also contains information about how you can use development resources such as MSDN, MSDN Online, and developer support resources. This helpful information is found in various chapters in Part 1, and those chapters are common to all WPRS volumes. By including this information in each library and in each volume, a few goals of the WPRS are achieved:

- I don't presume you have bought, or expect you to have to buy another WPRS Library to gain access to this information. Maybe your primary focus is network programming, and your budget doesn't allow for you to purchase the *Active Directory Developer's Reference Library*. Since I've included this information in this library, you don't have to.
- You can access this important and useful information regardless of which volume you have in your hand. You don't have to (nor *should* you have to) fumble with another physical book to access information about how to get the most out of MSDN, or where to get support for questions you have about a particular Windows development problem you're having.
- Each volume becomes more useful, more portable, and more complete in and of itself. This goal of the WPRS makes it easier for you to grab one of its libraries' volumes and take it with you, rather than feeling like you must bring multiple volumes with you to have access to the library's important overview and usability information.

These goals have guided choices about this library's content and included technologies; I hope you find its information useful, portable, a good value, and as accessible as it can be.

Part 2 of this volume addresses the interfaces, protocols, and services described in the following sections.

Domain Name System

Domain Name System (DNS), now an industry-standard protocol, locates computers on an IP-based network. IP networks, such as the Internet and Microsoft Windows 2000 networks, rely on number-based addresses to move information on the network. However, users are better at remembering friendly names than number-based addresses, so it is necessary to translate user-friendly names (*www.microsoft.com*) into addresses that the network can recognize (207.46.131.137). DNS is the locator service of choice in Windows 2000.

Dynamic Host Configuration Protocol

The Dynamic Host Configuration Protocol (DHCP) Application Programming Interface, also referred to as DHCP Client Options, enables Windows 2000 and Windows 98 clients to query specific options from DHCP servers. Such capability enables vendor-specific options exposed through DHCP servers to be queried by Windows 2000 or Windows 98 DHCP clients.

Multicast Address Dynamic Client Allocation Protocol

MADCAP, or Multicast Address Dynamic Client Allocation Protocol, is a technology aimed at making it easy for clients to renew and release multicast addresses, enabling clients to dynamically "connect" and "disconnect" from multicast network transmissions. The development of standards for MADCAP is ongoing, and falls under the Multicast Address Allocation (malloc) Working Group at the Internet Engineering Task Force (IETF).

Developers can use MADCAP to:

- Dynamically obtain a multicast address for a client, enabling that client to participate in network multicast transmission reception.
- Enumerate the available MADCAP transmissions available from a given server.
- Release multicast addresses when appropriate.

The Windows 2000 implementation of MADCAP adheres to the MADCAP recommendations published by the IETF, which are available on the IETF web site (*www.ietf.org*). Since MADCAP has not been ratified as a Request For Comments (RFC), and is rather in Internet Draft form, the technology is subject to continuing growth and evolution. Microsoft Corporation is actively involved with the standards process on an ongoing basis.

Internet Authentication Service

The Internet Authentication Service (IAS) API enables software developers to write their own extensions to IAS. IAS also allows developers to implement session control and accounting plug-ins, add authorizations, and use network authentication methods for remote access. IAS supports, as a client and server, the Remote Authentication Dial-In User Service (RADIUS) protocol.

IAS is applicable in any computing environment where it would improve efficiency to authenticate dial-in users through a remote server. This technology is especially useful for Internet Service Providers (ISPs).

NetBIOS

A Win32-based application can use the Network Basic Input/Output System (NetBIOS) interface to communicate with applications on other computers in a network. The NetBIOS interface provides commands and support for the following services:

- Network name registration and verification
- Session establishment and termination
- Reliable connection-oriented data transfer
- Unreliable connectionless data transfer (datagram)
- Protocol and adapter monitoring and management

The NetBIOS interface is provided primarily for existing applications that use IBM NetBIOS 3.0 and need to be ported to the Win32 API. New applications and applications not requiring compatibility with NetBIOS should use other interfaces, such as mailslots, named pipes, RPC, sockets, or distributed COM to accomplish tasks similar to those supported by NetBIOS. These interfaces are more flexible and portable than NetBIOS. In addition, you can use sockets over NetBIOS to communicate with NetBIOS applications.

However, there are plenty of NetBIOS-enabled applications in existence today, so I've included NetBIOS in this library to ensure that the reference information you need for such compatibility is available to you.

Synchronization Manager

The Synchronization Manager provides a centralized, standard technology for synchronizing files for offline use on either a mobile computer or a computer connected to a local area network that has latency issues. Developers can use the common interface to the Synchronization Manager in their applications to synchronize files between the user's local computer and network storage.

Files are synchronized independent of the protocol. For example, an e-mail program can transfer its messages using SMTP, NMTP, or POP3, a browser can use HTTP, and a database can use Remote Procedure Call (RPC).

The Synchronization Manager is intended for applications that run primarily on mobile computers. Applications that run on computers connected to high latency local area networks may also benefit from using the Synchronization Manager.

System Event Notification Service

Applications designed for mobile users require a unique set of connectivity functions and notifications. In the past, individual applications were required to implement these features internally. The System Event Notification Service (SENS) now provides these capabilities in the operating system, creating a uniform connectivity and notification interface for applications. Using SENS, developers can determine connection bandwidth and latency information from within their application and optimize the application's operation based on those conditions.

The SENS connectivity functions and notifications are useful for applications written for mobile computers or computers connected to high latency local area networks.

IP Helper

The Internet Protocol Helper (IP Helper) API enables a software developer to retrieve and modify network configuration settings for a local computer.

The IP Helper API is applicable in any computing environment where the TCP/IP network protocol is used and there is a need to programmatically manipulate the TCP/IP configuration. Typical applications include IP routing protocols and Simple Network Management Protocol (SNMP) agents.

Simple Network Management Protocol

The Simple Network Management Protocol (SNMP) is the Internet standard protocol for exchanging management information between management console applications such as HP Openview, Novell NMS, IBM NetView, or Sun Net Manager, and managed entities. The managed entities can include hosts, routers, bridges, and hubs.

WinSNMP

The Windows SNMP Application Programming Interface (the WinSNMP API) versions 1.1a and 2.0 allow you to develop SNMP-based network management applications that execute in the Windows 2000 operating environment. SNMP is a request-response protocol that transfers management information between protocol entities.

Network Management

Microsoft Windows NT, Windows 2000, Windows 95, and Windows 98 support a variety of networking APIs. The network management functions provide the ability to manage user accounts and network resources. Many of the capabilities provided by the network management functions are not provided by other networking functions.

CHAPTER 3

Using Microsoft Reference Resources

Keeping current with all the latest information on the latest networking technology is like trying to count the packets going through routers at the MAE-WEST Internet service exchange by watching their blinking activity lights: It's impossible. Often times, application developers feel like those routers might feel at a given day's peak activity; too much information is passing through them, none of which is being absorbed or passed along fast enough for their boss' liking.

For developers, sifting through all the *available* information to get to the *required* information is often a major undertaking, and can impose a significant amount of overhead upon a given project. What's needed is either a collection of information that has been sifted for you, shaking out the information you need the most and putting that pertinent information into a format that's useful and efficient, or direction on how to sift the information yourself. The *Networking Services Developer's Reference Library* does the former, and this chapter and the next provide you with the latter.

This veritable white noise of information hasn't always been a problem for network programmers. Not long ago, getting the information you needed was a challenge because there wasn't enough of it; you had to find out where such information might be located and then actually get access to that location, because it wasn't at your fingertips or on some globally available backbone, and such searching took time. In short, the availability of information was limited.

Today, the volume of information that surrounds us sometimes numbs us; we're overloaded with too much information, and if we don't take measures to filter out what we don't need to meet our goals, soon we become inundated and unable to discern what's "white noise" and what's information that we need to stay on top of our respective fields. In short, the overload of available information makes it more difficult for us to find what we *really* need, and wading through the deluge slows us down.

This fact applies equally to Microsoft's reference material, because there is so much information that finding what *you* need can be as challenging as figuring out what to do with it once you have it. Developers need a way to cut through what isn't pertinent to them and to get what they're looking for. One way to ensure you can get to the information you need is to understand the tools you use; carpenters know how to use nail-guns, and it makes them more efficient. Bankers know how to use ten-keys, and it makes them more adept. If you're a developer of Windows applications, two tools you should know are MSDN and MSDN Online. The third tool for developers—reference books from the WPRS—can help you get the most out of the first two.

Books in the WPRS, such as those found in the *Networking Services Developer's Reference Library*, provide reference material that focuses on a given area of Windows programming. MSDN and MSDN Online, in comparison, contain all of the reference material that all Microsoft programming technologies have amassed over the past few years, and create one large repository of information. Regardless of how well such information is organized, there's a lot of it, and if you don't know your way around, finding what you need (even though it's in there, somewhere) can be frustrating, time-consuming, and just an overall bad experience.

This chapter will give you the insight and tips you need to navigate MSDN and MSDN Online and enable you to use each of them to the fullest of their capabilities. Also, other Microsoft reference resources are investigated, and by the end of the chapter, you'll know where to go for the Microsoft reference information you need (and how to quickly and efficiently get there).

The Microsoft Developer Network

MSDN stands for Microsoft Developer Network, and its intent is to provide developers with a network of information to enable the development of Windows applications. Many people have either worked with MSDN or have heard of it, and quite a few have one of the three available subscription levels to MSDN, but there are many, many more who don't have subscriptions and could use some concise direction on what MSDN can do for a developer or development group. If you fall into any of these categories, this section is for you.

There is some clarification to be done with MSDN and its offerings; if you've heard of MSDN, or have had experience with MSDN Online, you may have asked yourself one of these questions during the process of getting up to speed with either resource:

- Why do I need a subscription to MSDN if resources such as MSDN Online are accessible for free over the Internet?
- What is the difference between the three levels of MSDN subscriptions?
- Is there a difference between MSDN and MSDN Online, other than the fact that one is on the Internet and the other is on a CD? Do their features overlap, separate, coincide, or what?

If you have asked any of these questions, then lurking somewhere in the back of your thoughts has probably been a sneaking suspicion that maybe you aren't getting the most out of MSDN. Maybe you're wondering whether you're paying too much for too little, or not enough to get the resources you need. Regardless, you want to be in the know and not in the dark. By the end of this chapter, you'll know the answers to all these questions and more, along with some effective tips and hints on how to make the most effective use of MSDN and MSDN Online.

Comparing MSDN with MSDN Online

Part of the challenge of differentiating between MSDN and MSDN Online comes with determining which has the features you need. Confounding this differentiation is the fact that both have some content in common, yet each offers content unavailable with the other. But can their difference be boiled down? Yes, if broad strokes and some generalities are used:

- MSDN provides reference content *and* the latest Microsoft product software, all shipped to its subscribers on CD or DVD.
- MSDN Online provides reference content *and* a development community forum, and is available only over the Internet.

Each delivery mechanism for the content that Microsoft is making available to Windows developers is appropriate for the medium, and each plays on the strength of the medium to provide its "customers" with the best possible presentation of material. These strengths and medium considerations enable MSDN and MSDN Online to provide developers with different feature sets, each of which has its advantages.

MSDN is perhaps less "immediate" than MSDN Online because it gets to its subscribers in the form of CDs or DVDs that come in the mail. However, MSDN can sit in your CD/DVD drive (or on your hard drive), and isn't subject to Internet speeds or failures. Also, MSDN has a software download feature that enables subscribers to automatically update their local MSDN content over the Internet, as soon as it becomes available, without having to wait for the update CD/DVD to come in the mail. The interface with which MSDN displays its material—which looks a whole lot like a specialized browser window—is also linked to the Internet as a browser-like window. To further coordinate MSDN with the immediacy of the Internet, MSDN Online has a section of the site dedicated to MSDN subscribers that enable subscription material to be updated (on their local machines) as soon as it's available.

MSDN Online has lots of editorial and technical columns that are published directly to the site, and are tailored (not surprisingly) to the issues and challenges faced by developers of Windows applications or Windows-based Web sites. MSDN Online also has a customizable interface (somewhat similar to *MSN.com*) that enables visitors to tailor the information that's presented upon visiting the site to the areas of Windows development in which they are most interested. However, MSDN Online, while full of up-to-date reference material and extensive online developer community content, doesn't come with Microsoft product software, and doesn't reside on your local machine.

Because it's easy to confuse the differences and similarities between MSDN and MSDN Online, it makes sense to figure out a way to quickly identify how and where they depart. Figure 3-1 puts the differences—and similarities—between MSDN and MSDN Online into a quickly identifiable format.

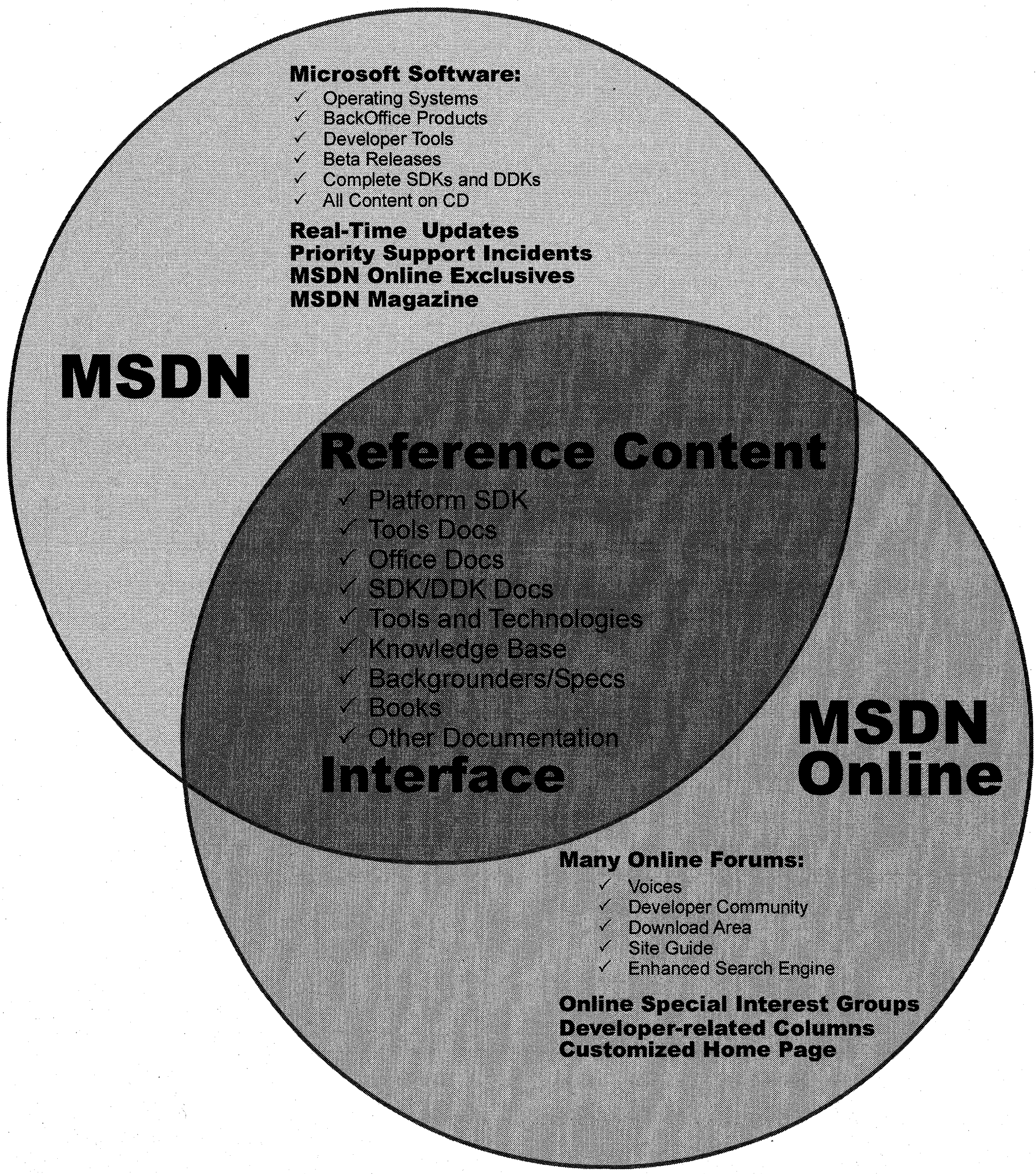

Figure 3-1: The similarities and differences in coverage between MSDN and MSDN Online.

One feature you'll notice is shared between MSDN and MSDN Online is the interface—they are very similar. That's almost certainly a result of attempting to ensure that developers' user experience with MSDN is easily associated with the experience had on MSDN Online, and vice-versa.

Remember, too, that if you are an MSDN subscriber, you can still use MSDN Online and its features. So it isn't an "either/or" question with regard to whether you need an MSDN subscription or whether you should use MSDN Online; if you have an MSDN subscription, you will probably continue to use MSDN Online and the additional features provided with your MSDN subscription.

MSDN Subscriptions

If you're wondering whether you might benefit from a subscription to MSDN, but you aren't quite sure what the differences between its subscription levels are, you aren't alone. This section aims to provide a quick guide to the differences in subscription levels, and even provides an estimate for what each subscription level costs.

The three subscription levels for MSDN are: Library, Professional, and Universal. Each has a different set of features. Each progressive level encompasses the lower level's features, and includes additional features. In other words, with the Professional subscription, you get everything provided in the Library subscription plus additional features; with the Universal subscription, you get everything provided in the Professional subscription plus even more features.

MSDN Library Subscription

The MSDN Library subscription is the basic MSDN subscription. While the Library subscription doesn't come with the Microsoft product software that the Professional and Universal subscriptions provide, it does come with other features that developers may find necessary in their development effort. With the Library subscription, you get the following:

- The Microsoft reference library, including SDK and DDK documentation, updated quarterly
- Lots of sample code, which you can cut-and-paste into your projects, royalty free
- The complete Microsoft Knowledge Base—*the* collection of bugs and workarounds
- Technology specifications for Microsoft technologies
- The complete set of product documentation, such as Microsoft Visual Studio, Microsoft Office, and others
- Complete (and in some cases, partial) electronic copies of selected books and magazines
- Conference and seminar papers—if you weren't there, you can use MSDN's notes

In addition to these items, you also get:

- Archives of MSDN Online columns
- Periodic e-mails from Microsoft chock full of development-related information
- A subscription to MSDN News, a bi-monthly newspaper from the MSDN folks
- Access to subscriber-exclusive areas and material on MSDN Online

MSDN Professional Subscription

The MSDN Professional subscription is a superset of the Library subscription. In addition to the features outlined in the previous section, MSDN Professional subscribers get the following:

- Complete set of Windows operating systems, including release versions of Windows 95, Windows 98, and Windows NT 4 Server and Workstation.
- Windows SDKs and DDKs in their entirety
- International versions of Windows operating systems (as chosen)
- Priority technical support for two incidents in a development and test environment

MSDN Universal Subscription

The MSDN Universal subscription is the all-encompassing version of the MSDN subscription. In addition to everything provided in the Professional subscription, Universal subscribers get the following:

- The latest version of Visual Studio, Enterprise Edition
- The Microsoft BackOffice test platform, which includes all sorts of Microsoft product software incorporated in the BackOffice family, each with a special 10-connection license for use in the development of your software products
- Additional development tools, such as Office Developer, Microsoft FrontPage, and Microsoft Project
- Priority technical support for two additional incidents in a development and test environment (for a total of four incidents)

Purchasing an MSDN Subscription

Of course, all the features that you get with MSDN subscriptions aren't free. MSDN subscriptions are one-year subscriptions, which are current as of this writing. Just as each MSDN subscription escalates in functionality of incorporation of features, so does each escalate in price. Please note that prices are subject to change.

The MSDN Library subscription has a retail price of $199, but if you're renewing an existing subscription you get a $100 rebate in the box. There are other perks for existing Microsoft customers, but those vary. Check out the Web site for more details.

The MSDN Professional subscription is a bit more expensive than the Library, with a retail price of $699. If you're an existing customer renewing your subscription, you again get a break in the box, this time in the amount of a $200 rebate. You also get that break if you're an existing Library subscriber who's upgrading to a Professional subscription.

The MSDN Universal subscription takes a big jump in price, sitting at $2,499. If you're upgrading from the Professional subscription, the price drops to $1,999, and if you're upgrading from the Library subscription level, there's an in-the-box rebate for $200.

As is often the case, there are academic and volume discounts available from various resellers, including Microsoft, so those who are in school or in the corporate environment can use their status (as learner or learned) to get a better deal—and in most cases, the deal is in fact much better. Also, if your organization is using lots of Microsoft products, whether or not MSDN is a part of that group, ask your purchasing department to look into the Microsoft Open License program; the Open License program gives purchasing breaks for customers who buy lots of products. Check out *www.microsoft.com/licensing* for more details. Who knows, if your organization qualifies you could end up getting an engraved pen from your purchasing department, or if you're really lucky maybe even a plaque of some sort for saving your company thousands of dollars on Microsoft products.

You can get MSDN subscriptions from a number of sources, including online sites specializing in computer-related information, such as *www.iseminger.com* (shameless self-promotion, I know), or from your favorite online software site. Note that not all software resellers carry MSDN subscriptions; you might have to hunt around to find one. Of course, if you have a local software reseller that you frequent, you can check out whether they carry MSDN subscriptions.

As an added bonus for owners of this *Networking Services Developer's Reference Library*, in the back of Volume 1, you'll find a $200 rebate good toward the purchase of an MSDN Universal subscription. For those of you doing the math, that means you actually *make* money when you purchase the *Networking Services Developer's Reference Library* and an MSDN Universal subscription. With this rebate, every developer in your organization can have the *Networking Services Developer's Refence Library* on their desk and the MSDN Universal subscription on thier desktop, and still come out $50 ahead. That's the kind of math even accountants can like.

Using MSDN

MSDN subscriptions come with an installable interface, and the Professional and Universal subscriptions also come with a bunch of Microsoft product software such as Windows platform versions and BackOffice applications. There's no need to tell you how to use Microsoft product software, but there's a lot to be said for providing some quick but useful guidance on getting the most out of the interface to present and navigate through the seemingly endless supply of reference material provided with any MSDN subscription.

To those who have used MSDN, the interface shown in Figure 3-2 is likely familiar; it's the navigational front-end to MSDN reference material.

The interface is familiar and straightforward enough, but if you don't have a grasp on its features and navigation tools, you can be left a little lost in its sea of information. With a few sentences of explanation and some tips for effective navigation, however, you can increase its effectiveness dramatically.

Navigating MSDN

One of the primary features of MSDN—and to many, its primary drawback—is the sheer volume of information it contains, over 1.1GB and growing. The creators of MSDN likely realized this, though, and have taken steps to assuage the problem. Most of those steps relate to enabling developers to selectively navigate through MSDN's content.

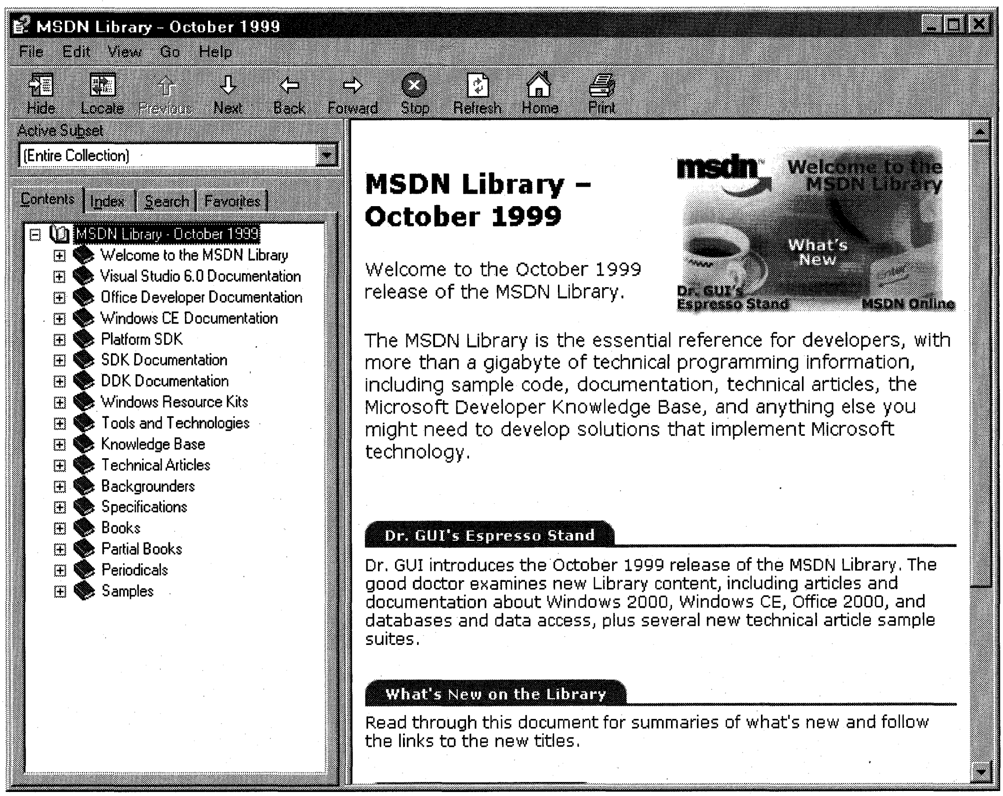

Figure 3-2: The MSDN interface.

Basic navigation through MSDN is simple and is a lot like navigating through Microsoft Windows Explorer and its folder structure. Instead of folders, MSDN has books into which it organizes its topics; expand a book by clicking the + box to its left, and its contents are displayed with its nested books or reference pages, as shown in Figure 3-3. If you don't see the left pane in your MSDN viewer, go to the View menu and select Navigation Tabs and they'll appear.

The four tabs in the left pane of MSDN—increasingly referred to as property sheets these days—are the primary means of navigating through MSDN content. These four tabs, in coordination with the Active Subset drop-down box above the four tabs, are the tools you use to search through MSDN content. When used to their full extent, these coordinated navigation tools greatly improve your MSDN experience.

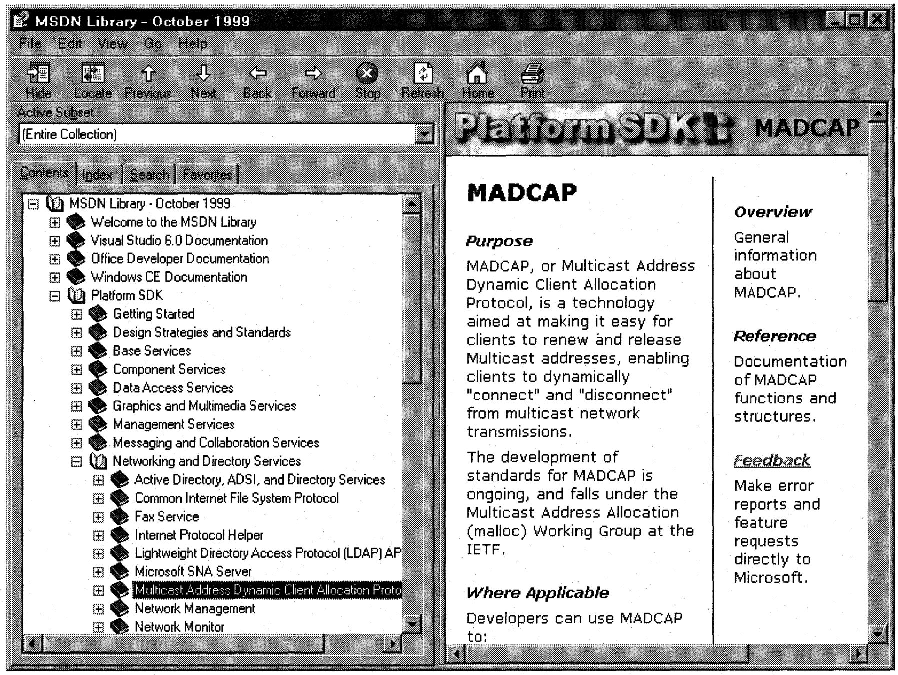

Figure 3-3: Basic navigation through MSDN.

The Active Subset drop-down box is a filter mechanism; choose the subset of MSDN information you're interested in working with from the drop-down box, and the information in each of the four Navigation Tabs (including the Contents tab) limits the information it displays to the information contained in the selected subset. This means that any searches you do in the Search tab, and in the index presented in the Index tab, are filtered by their results and/or matches to the subset you define, greatly narrowing the number of potential results for a given inquiry. This enables you to better find the information you're *really* looking for. In the Index tab, results that might match your inquiry but *aren't* in the subset you have chosen are grayed out (but still selectable). In the Search tab, they simply aren't displayed.

MSDN comes with the following predefined subsets (these subsets are subject to change, based on documentation updates and TOC reorganizations):

Entire Collection
MSDN, Books and Periodicals
MSDN, Content on Disk 2 only (CD only – not in DVD version)
MSDN, Content on Disk 3 only (CD only – not in DVD version)
MSDN, Knowledge Base
MSDN, Technical Articles and Backgrounders
Platform SDK, Networking Services
Platform SDK, Security
Platform SDK, Tools and Languages
Platform SDK, User Interface Services
Platform SDK, Web Services
Platform SDK, Win32 API
Repository 2.0 Documentation
Visual Basic Documentation
Visual C++ Documentation

Office Developer Documentation
Platform SDK, BackOffice
Platform SDK, Base Services
Platform SDK, Component Services
Platform SDK, Data Access Services
Platform SDK, Getting Started
Platform SDK, Graphics and Multimedia Services
Platform SDK, Management Services
Platform SDK, Messaging and Collaboration Services
Visual C++, Platform SDK and WinCE Docs
Visual C++, Platform SDK, and Enterprise Docs
Visual FoxPro Documentation
Visual InterDev Documentation
Visual J++ Documentation
Visual SourceSafe Documentation
Visual Studio Product Documentation
Windows CE Documentation

As you can see, these filtering options essentially mirror the structure of information delivery used by MSDN. But what if you are interested in viewing the information in a handful of these subsets? For example, what if you want to search on a certain keyword through the Platform SDK's ADSI, Networking Services, and Management Services subsets, as well as a little section that's nested way into the Base Services subset? Simple—you define your own subset by choosing the View menu, and then selecting the Define Subsets menu item. You're presented with the window shown in Figure 3-4.

Defining a subset is easy; just take the following steps:

1. Choose the information you want in the new subset; you can choose entire subsets or selected books/content within available subsets.
2. Add your selected information to the subset you're creating by clicking the Add button.
3. Name the newly created subset by typing in a name in the Save New Subset As box. Note that defined subsets (including any you create) are arranged in alphabetical order.

You can also delete entire subsets from the MSDN installation. Simply select the subset you want to delete from the Select Subset To Display drop-down box, and then click the nearby Delete button.

Once you have defined a subset, it becomes available in MSDN just like the predefined subsets, and filters the information available in the four Navigation Tabs, just like the predefined subsets do.

Quick Tips

Now that you know how to navigate MSDN, there are a handful of tips and tricks that you can use to make MSDN as effective as it can be.

Use the Locate button to get your bearings. Perhaps it's human nature to need to know where you are in the grand scheme of things, but regardless, it can be bothersome to have a reference page displayed in the right pane (perhaps jumped to from a search), without the Contents tab in the left pane being synchronized in terms of the reference page's location in the information tree. Even if you know the general technology in which your reference page resides, it's nice to find out where it is in the content structure.

This is easy to fix. Simply click the Locate button in the navigation toolbar and all will be synchronized.

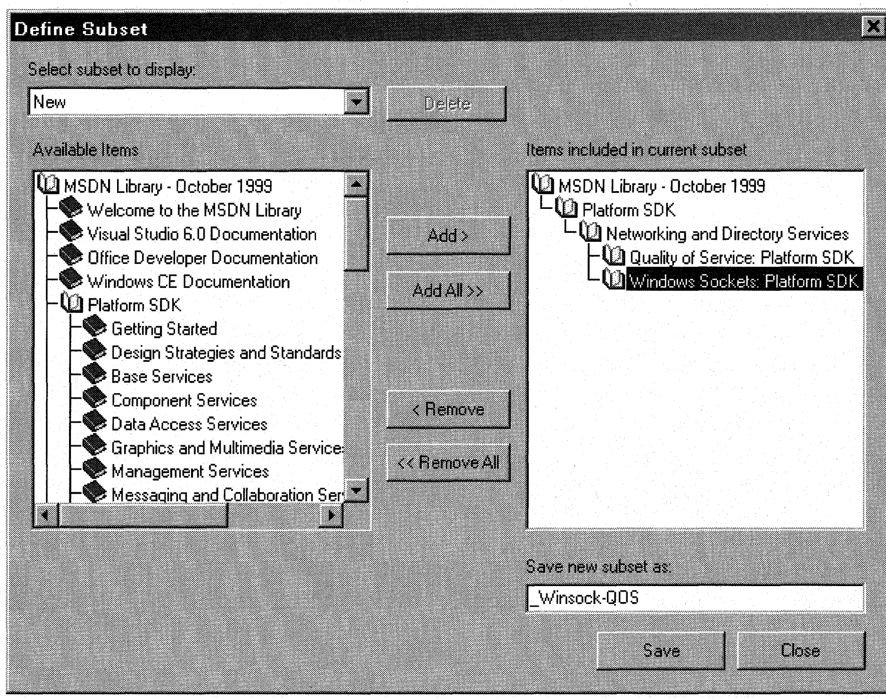

Figure 3-4: The Define Subsets window.

Use the Back button just like a browser. The Back button in the navigation toolbar functions just like a browser's Back button; if you need information on a reference page you viewed previously, you can use the Back button to get there, rather than going through the process of doing another search.

Define your own subsets, and use them. Like I said at the beginning of this chapter, the volume of information available these days can sometimes make it difficult to get our work done. By defining subsets of MSDN that are tailored to the work you do, you can become more efficient.

Use an underscore at the beginning of your named subsets. Subsets in the Active Subset drop-down box are arranged in alphabetical order, and the drop-down box shows only a few subsets at a time (making it difficult to get a grip on available subsets, I think). Underscores come before letters in alphabetical order, so if you use an underscore on all of your defined subsets, you get them placed at the front of the Active Subset listing of available subsets. Also, by using an underscore, you can immediately see which subsets you've defined, and which ones come with MSDN—it saves a few seconds at most, but those seconds can add up.

Using MSDN Online

MSDN underwent a redesign in December of 1999, aimed at streamlining the information provided, jazzing things up with more color, highlighting hot new technologies, and various other improvements. Despite its visual overhaul, MSDN Online still shares a lot of content and information delivery similarities with MSDN, and those similarities are by design; when you can go from one developer resource to another and immediately work with its content, your job is made easier. However, MSDN Online is different enough that it merits explaining in its own right—it's a different delivery medium, and can take advantage of the Internet in ways that MSDN simply cannot.

If you've used MSN's home page before (*www.msn.com*), you're familiar with the fact that you can customize the page to your liking; choose from an assortment of available national news, computer news, local news, local weather, stock quotes, and other collections of information or news that suit your tastes or interests. You can even insert a few Web links and have them readily accessible when you visit the site. The MSDN Online home page can be customized in a similar way, but its collection of headlines, information, and news sources are all about development. The information you choose specifies the information you see when you go to the MSDN Online home page, just like the MSN home page.

There are a couple of ways to get to the customization page; you can go to the MSDN Online home page (*msdn.microsoft.com*) and click the Personalize This Site button near the top of the page, or you can go there directly by pointing your browser to *msdn.microsoft.com/msdn-online/start/custom*. However you get there, the page you'll see is shown in Figure 3-5.

As you can see from Figure 3-5, there are lots of technologies to choose from (many more options can be found when you scroll down through available technologies). If you're interested in Web development, you can select the checkbox at the left of the page next to Standard Web Development, and a predefined subset of Web-centered technologies is selected. For technologies centered more on Network Services, you can go through and choose the appropriate technologies. If you want to choose all the technologies in a given technology group more quickly, click the Select All button in the technology's shaded title area.

You can also choose which tab is selected by default in the home page that MSDN Online presents to you, which is convenient for dropping you into the category of MSDN Online information that interests you most. All five of the tabs available on MSDN Online's home page are available for selection; those tabs are the following:

- Features
- News
- Columns
- Technical Articles
- Training & Events

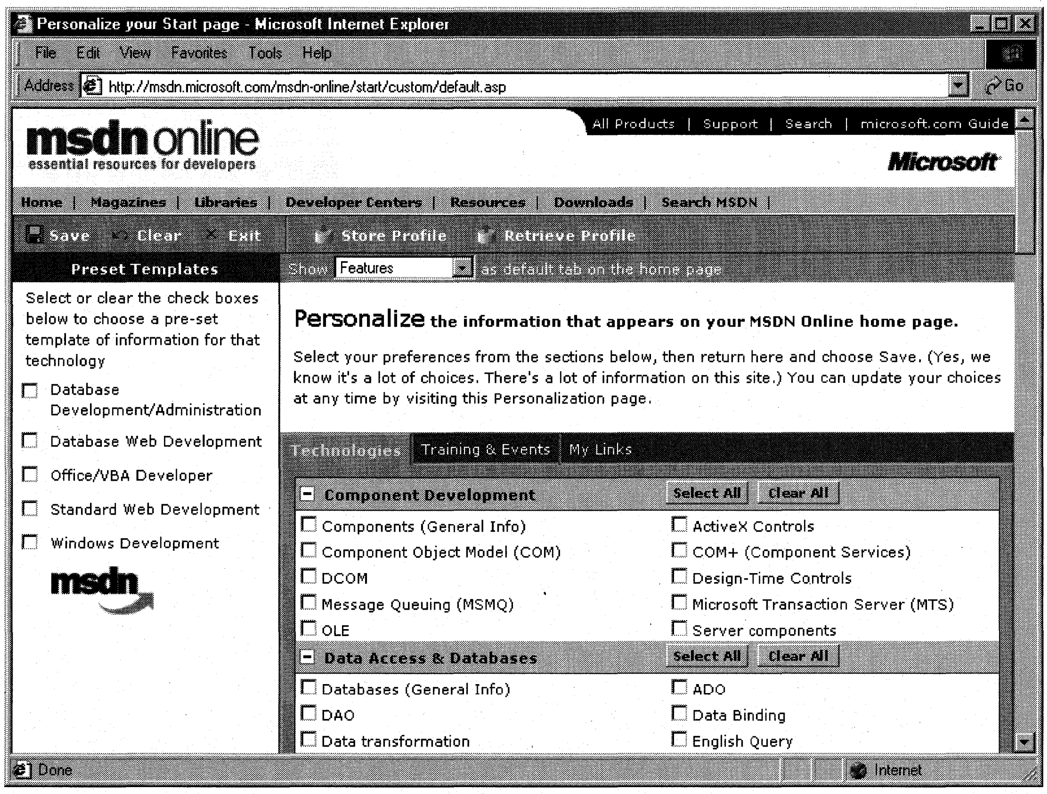

Figure 3-5: The MSDN Online Personalize Page.

Once you've defined your profile—that is, customized the MSDN Online content you want to see—MSDN Online shows you the most recent information pertinent to your profile when you go to MSDN Online's home page, with the default tab you've chosen displayed upon loading of the MSDN Online home page.

Finally, if you want your profile to be available to you regardless of which computer you're using, you can direct MSDN Online to store your profile. Storing a profile for MSDN Online results in your profile being stored on MSDN Online's server, much like roaming profiles in Windows 2000, and thereby makes your profile available to you regardless of the computer you're using. The option of storing your profile is available when you customize your MSDN Online home page (and can be done any time thereafter). The storing of a profile, however, requires that you become a registered member of MSDN Online. More information about becoming a registered MSDN Online user is provided in the section titled *MSDN Online Registered Users*.

Navigating MSDN Online

Once you're done customizing the MSDN Online home page to get the information you're most interested in, navigating through MSDN Online is easy. A banner that sits just below the MSDN Online logo functions as a navigation bar, with drop-down menus that can take you to the available areas on MSDN Online, as Figure 3-6 illustrates.

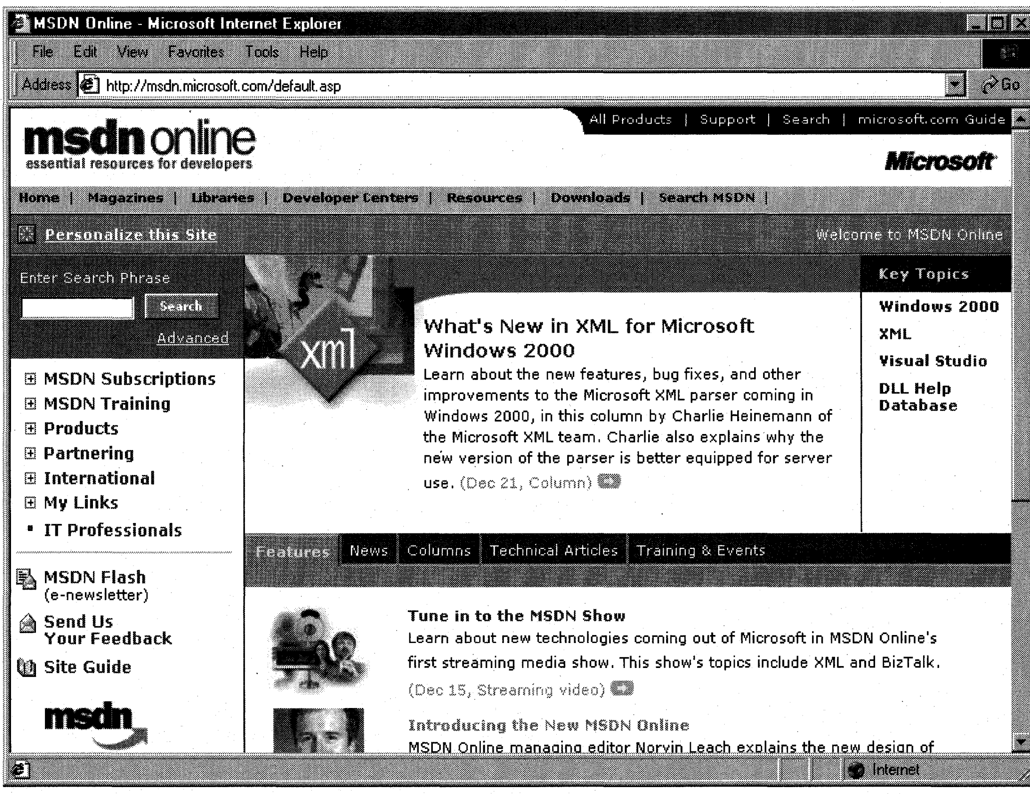

Figure 3-6: The MSDN Online Navigation Bar with Its Drop-Down Menus.

Following is a list of available menu categories, which groups the available sites and features within MSDN Online:

Home
Magazines
Libraries
Developer Centers
Resources
Downloads
Search MSDN

The navigation bar is available regardless of where you are in MSDN Online, so the capability to navigate the site from this familiar menu is always available, leaving you a click away from any area on MSDN Online. These menu categories create a functional and logical grouping of MSDN Online's feature offerings.

MSDN Online Features

Each of MSDN Online's seven feature categories contains various sites that comprise the features available to developers visiting MSDN Online.

Home is already familiar; clicking on Home in the navigation bar takes you to the MSDN Online home page that you've (perhaps) customized, showing you all the latest information about technologies that you've indicated you're interested in reading about.

Magazines is a collection of columns and articles that comprise MSDN Online's magazine section, as well as online versions of Microsoft's magazines such as MSJ, MIND, and the MSDN Show (a Webcast feature introduced with the December 1999 remodeling of MSDN Online). The Magazines feature of MSDN Online can be linked to directly at *msdn.microsoft.com/resources/magazines.asp*. The Magazines home page is shown in Figure 3-7.

Figure 3-7: The Magazines Home Page.

For those of you familiar with the **Voices** feature section that formerly found its home on the MSDN Online navigation banner, don't worry; all content formerly in the Voices section is included the Magazines section as a subsite (or menu item, if you prefer) of the Magazines site. For those of you who aren't familiar with the Voices subsite, you'll find a bunch of different articles or “voices” there, each of which adds its own particular twist on the issues that face developers. Both application and Web developers can get their fill of magazine-like articles from the sizable list of different articles available (and frequently refreshed) in the Voices subsite. With the combination of columns and online developer magazines offered in the Magazines section, you're sure to find plenty of interesting insights.

Libraries is where the reference material available on MSDN Online lives. The Libraries site is divided into two sections: Library and Web Workshop. This distinction divides the reference material between Windows application development and Web development. Choosing Library from the Libraries menu takes you to a page through which you can navigate in traditional MSDN fashion, and gain access to traditional MSDN reference material. The Library home page can be linked to directly at *msdn.microsoft.com/library*. Choosing Web Workshop takes you to a site that enables you to navigate the Web Workshop in a slightly different way, starting with a bulleted list of start points, as shown in Figure 3-8. The Web Workshop home page can be linked to directly at *msdn.microsoft.com/workshop*.

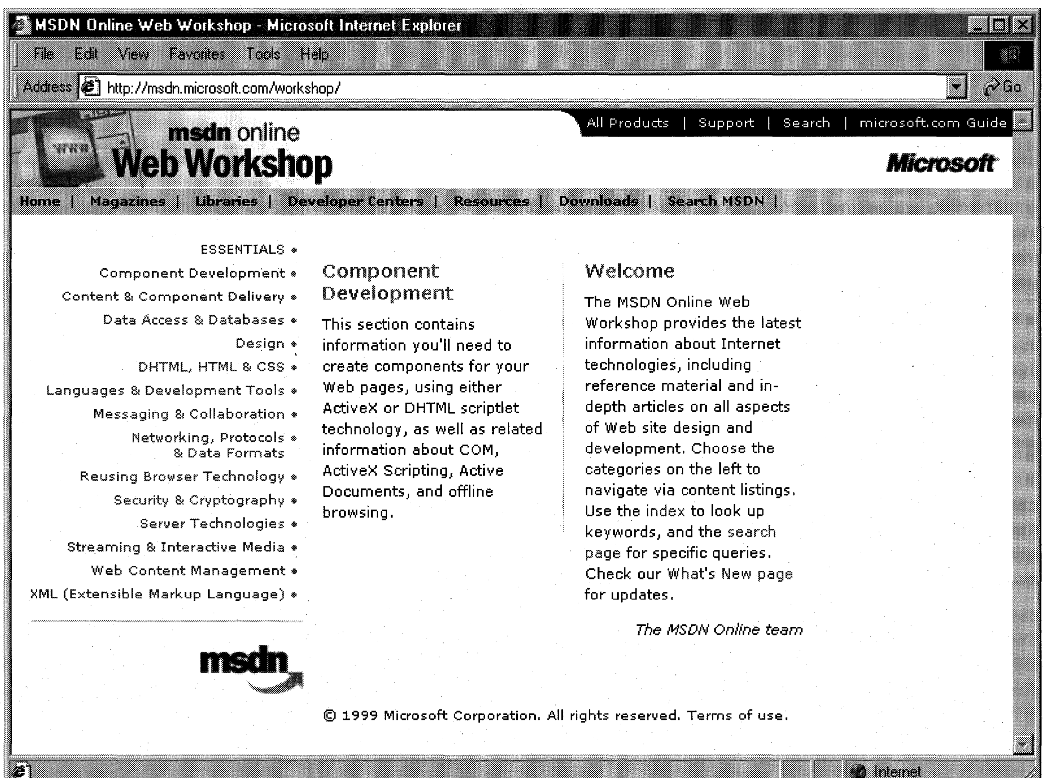

Figure 3-8: The Web Workshop Home Page.

Developer Centers is a hub from which developers who are interested in a particular area of development—such as Windows 2000, SQL Server, or XML—can go to find focused Web site centers within MSDN Online. Each developer center is dedicated to providing all sorts of information associated with its area of focus. For example, the Windows 2000 developer center has information about what's new with Windows 2000, including newsgroups, specifications, chats, knowledge base articles, and news, among others. At publication time, MSDN Online had the following developer centers:

- Microsoft Windows 2000
- Microsoft Exchange
- Microsoft SQL Server
- Microsoft Windows Media
- XML

In addition to these developer centers is a promise that new centers would be added to the site in the future. To get to the Developer Centers home page directly, link to *msdn.microsoft.com/resources/devcenters.asp*. Figure 3-9 shows the Developer Centers home page.

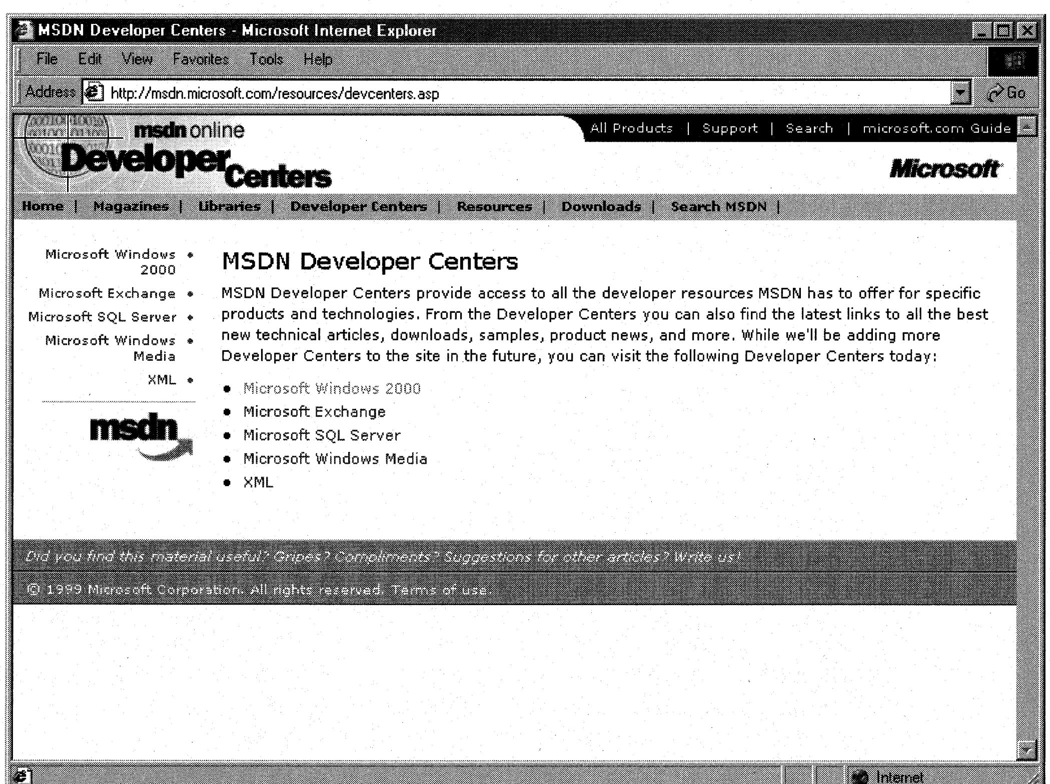

Figure 3-9: The Developer Centers Home Page.

Resources is a place where developers can go to take advantage of the online forum of Windows and Web developers, in which ideas or techniques can be shared, advice can be found or given (through MHM, or Members Helping Members), and the MSDN User Group Program can be joined or perused to find a forum to voice their opinions or chat with other developers. The Resources site is full of all sorts of useful stuff, including featured books, a DLL help database, online chats, case studies, and more. The Resources home page can be linked to directly at *msdn.microsoft.com/resources*. Figure 3-10 provides a look at the Resources home page.

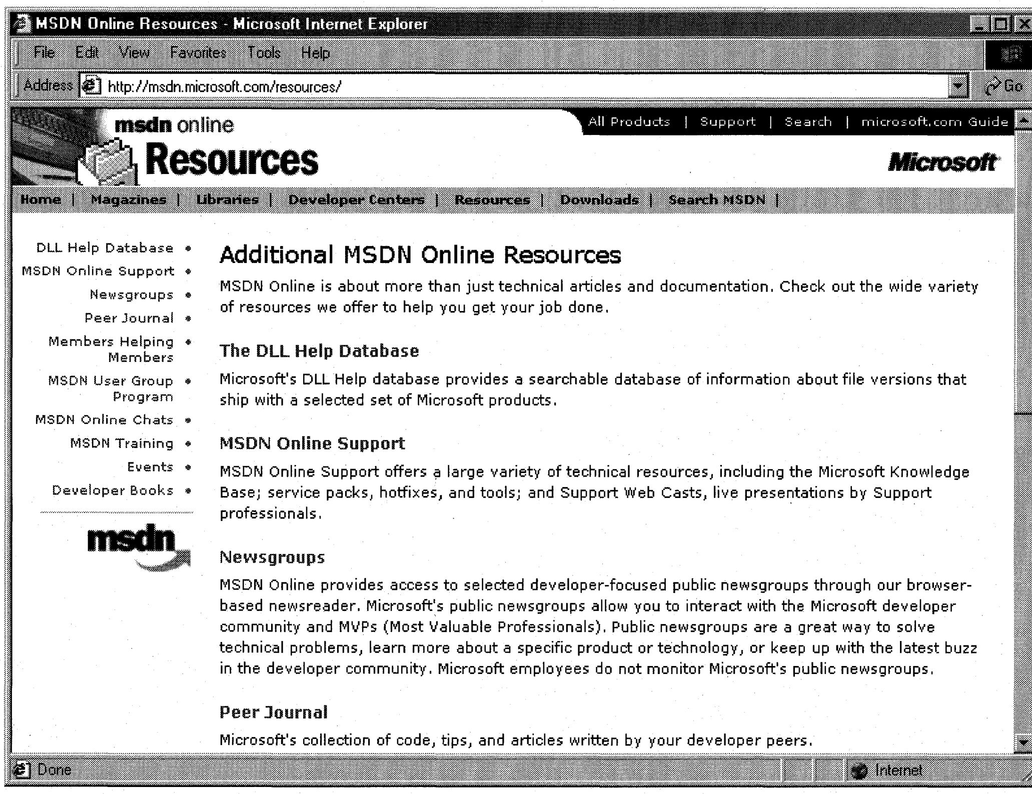

Figure 3-10: The Resources Home Page.

The **Downloads** site is where developers can find all sorts of useable items fit to be downloaded, such as tools, samples, images, and sounds. The Downloads site is also where MSDN subscribers go to get their subscription content updated over the Internet to the latest and greatest releases, as described previously in this chapter in the *Using MSDN* section. The Downloads home page can be linked to directly at *msdn.microsoft.com/downloads*. The Downloads home page is shown in Figure 3-11.

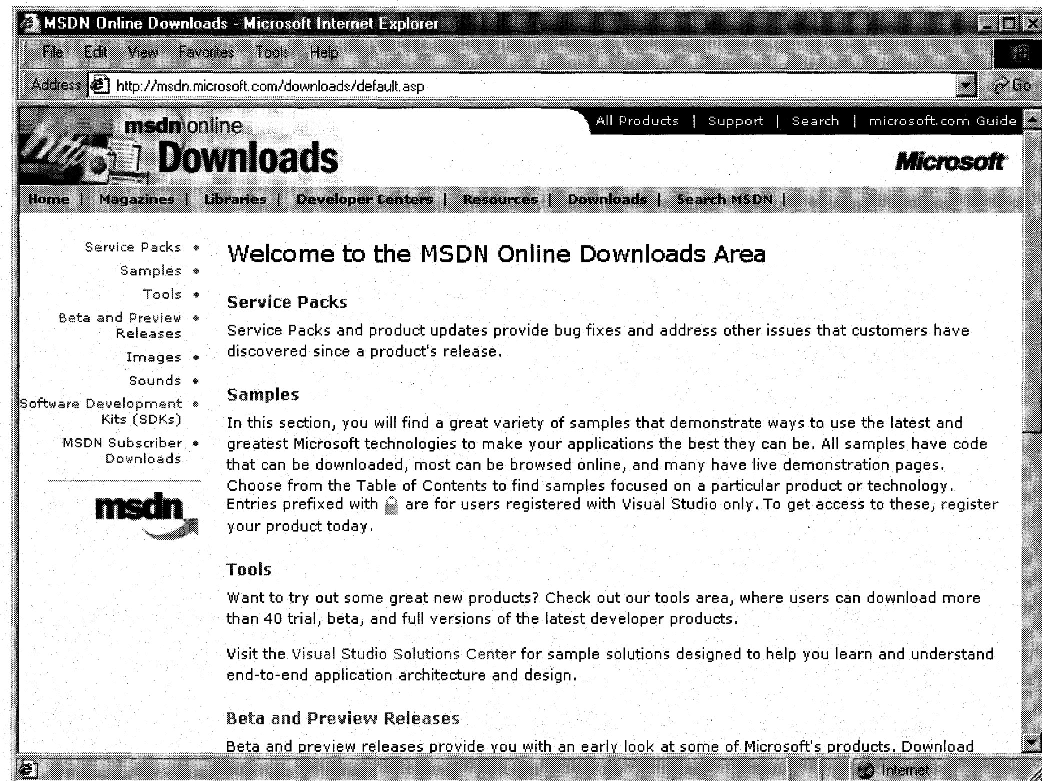

Figure 3-11: The Downloads Home Page.

The **Search MSDN** site on MSDN Online has been improved over previous versions, and includes the capability to restrict searches to either library (Library or Web Workshop), as well as other fine-tune search capabilities. The Search MSDN home page can be linked to directly at *msdn.microsoft.com/search*. The Search MSDN home page is shown in Figure 3-12.

There are two other destinations within MSDN Online of specific interest, neither of which is immediately reachable through the MSDN navigation bar. The first is the **MSDN Online Member Community** home page, and the other is the **Site Guide**.

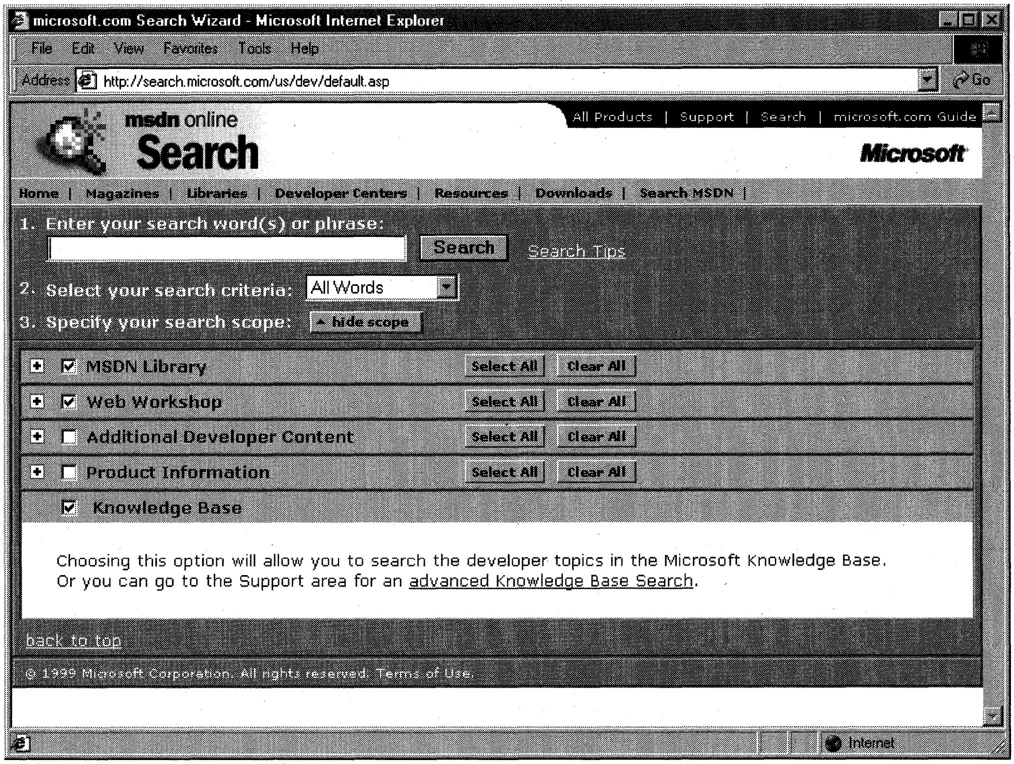

Figure 3-12: The Search MSDN Home Page.

The MSDN Online Member Community home page can be directly reached at *msdn.microsoft.com/community*. Many of the features found in the **Resources** navigation menu are actually subsites of the Community page. Of course, becoming a member of the MSDN Online member community requires that you register (see the next section for more details on joining), but doing so enables you to get access to Online Special Interest Groups (OSIGs) and other features reserved for registered members. The Community page is shown in Figure 3-13.

Another destination of interest on MSDN Online that isn't displayed on the navigation banner is the **Site Guide**. The Site Guide is just what its name suggests—a guide to the MSDN Online site that aims at helping developers find items of interest, and includes links to other pages on MSDN Online such as a recently posted files listing, site maps, glossaries, and other useful links. The Site Guide home page can be linked to directly at *msdn.microsoft.com/siteguide*.

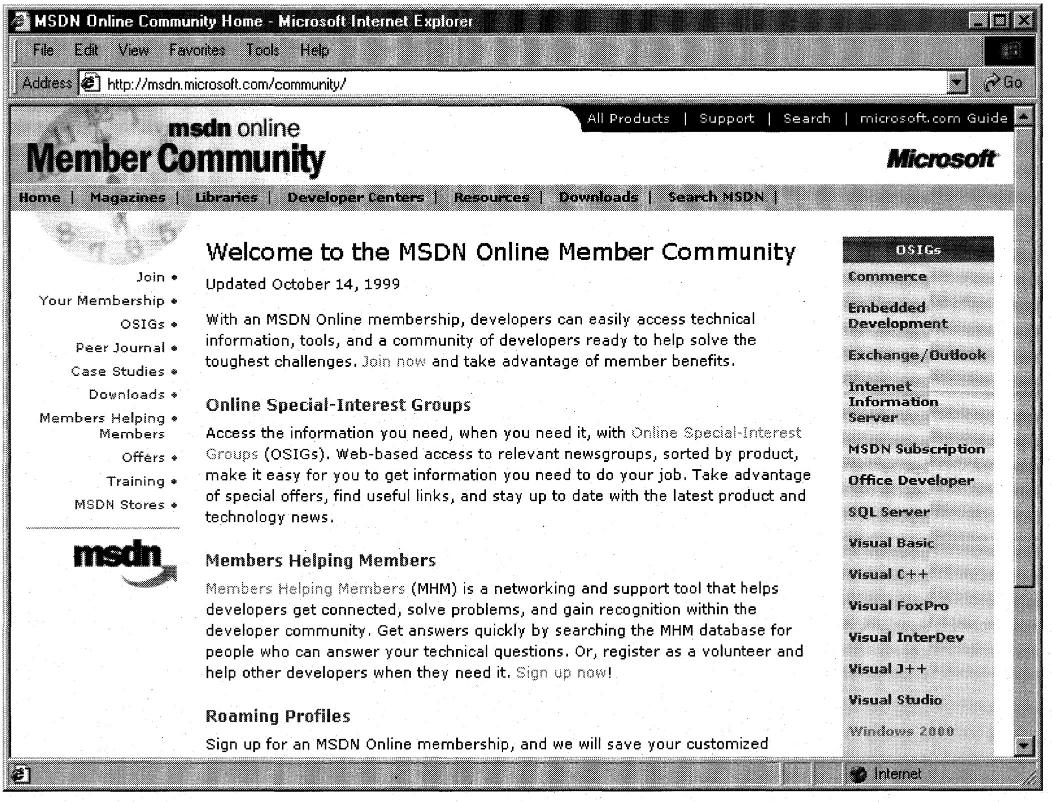

Figure 3-13: The MSDN Online Member Community Home Page.

MSDN Online Registered Users

You may have noticed that some features of MSDN Online—such as the capability to create a store profile of the entry ticket to some community features—require you to become a registered user. Unlike MSDN subscriptions, becoming a registered user of MSDN Online won't cost you anything more but a few minutes of registration time.

Some features of MSDN Online require registration before you can take advantage of their offerings. For example, becoming a member of an OSIG requires registration. That feature alone is enough to register; rather than attempting to call your developer buddy for an answer to a question (only to find out that she's on vacation for two days, and your deadline is in a few hours), you can go to MSDN Online's Community site and ferret through your OSIG to find the answer in a handful of clicks. Who knows; maybe your developer buddy will begin calling you with questions—you don't have to tell her where you're getting all your answers.

There are a number of advantages to being a registered user, such as the choice to receive newsletters right in your inbox if you want to. You can also get all sorts of other timely information, such as chat reminders that let you know when experts on a given subject will be chatting in the MSDN Online Community site. You can also sign up to get newsletters based on your membership in various OSIGs—again, only if you want to. It's easy for me to suggest that you become a registered user for MSDN Online—I'm a registered user, and it's a great resource.

The Windows Programming Reference Series

The WPRS provides developers with timely, concise, and focused material on a given topic, enabling developers to get their work done as efficiently as possible. In addition to providing reference material for Microsoft technologies, each Library in the WPRS also includes material that helps developers get the most out of its technologies, and provides insights that might otherwise be difficult to find.

The WPRS currently includes the following libraries:

- *Microsoft Win32 Developer's Reference Library*
- *Active Directory Developer's Reference Library*
- *Networking Services Developer's Reference Library*

In the near future (subject, of course, to technology release schedules, demand, and other forces that can impact publication decisions), you can look for these prospective WPRS Libraries that cover the following material:

- Web Technologies Library
- Web Reference Library
- MFC Developer's Reference Library
- Com Developer's Reference Library

What else might you find in the future? Planned topics such as a Security Library, Programming Languages Reference Library, BackOffice Developer's Reference Library, or other pertinent topics that developers using Microsoft products need in order to get the most out of their development efforts, are prime subjects for future membership in the WPRS. If you have feedback you want to provide on such libraries, or on the WPRS in general, you can send email to *winprs@microsoft.com*.

If you're sending mail about a particular library, make sure you put the name of the library in the subject line. For example, e-mail about the *Networking Services Developer's Reference Library* would have a subject line that reads "*Networking Services Developer's Reference Library*." There aren't any guarantees that you'll get a reply, but I'll read all of the mail and do what I can to ensure your comments, concerns, or (especially) compliments get to the right place.

CHAPTER 4

Finding the Developer Resources You Need

Networking is complex, and its resource information vast. With all the resources available for developers of network-enabled applications, and the answers they can provide to questions or problems that developers face every day, finding the developer information you need can be a challenge. To address that problem, this chapter is designed to be your one-stop resource to find the developer resources you need, making the job of actually developing your application just a little easier.

Microsoft provides plenty of resource material through MSDN and MSDN Online, and the WPRS provides a great filtered version of focused reference material and development knowledge. However, there is a lot more information to be had. Some of that information comes from Microsoft, some of it from the general development community, and yet more information comes from companies that specialize in such development services. Regardless of which resource you choose, in this chapter you can find out what your development resource options are, and be more informed about the resources that are available to you.

Microsoft provides developer resources through a number of different media, channels, and approaches. The extensiveness of Microsoft's resource offerings mirrors the fact that many are appropriate under various circumstances. For example, you wouldn't go to a conference to find the answer to a specific development problem in your programming project; instead, you might use one of the other Microsoft resources.

Developer Support

Microsoft's support sites cover a wide variety of support issues and approaches, including all of Microsoft's products, but most of those sites are not pertinent to developers. Some sites, however, *are* designed for developer support; the Product Services Support page for developers is a good central place to find the support information you need. Figure 4-1 shows the Product Services Support page for developers, which can be reached at *www.microsoft.com/support/customer/develop.htm*.

Note that there are a number of options for support from Microsoft, including everything from simple online searches of known bugs in the Knowledge Base to hands-on consulting support from Microsoft Consulting Services, and everything in between. The Web page displayed in Figure 4-1 is a good starting point from which you can find out more information about Microsoft's support services.

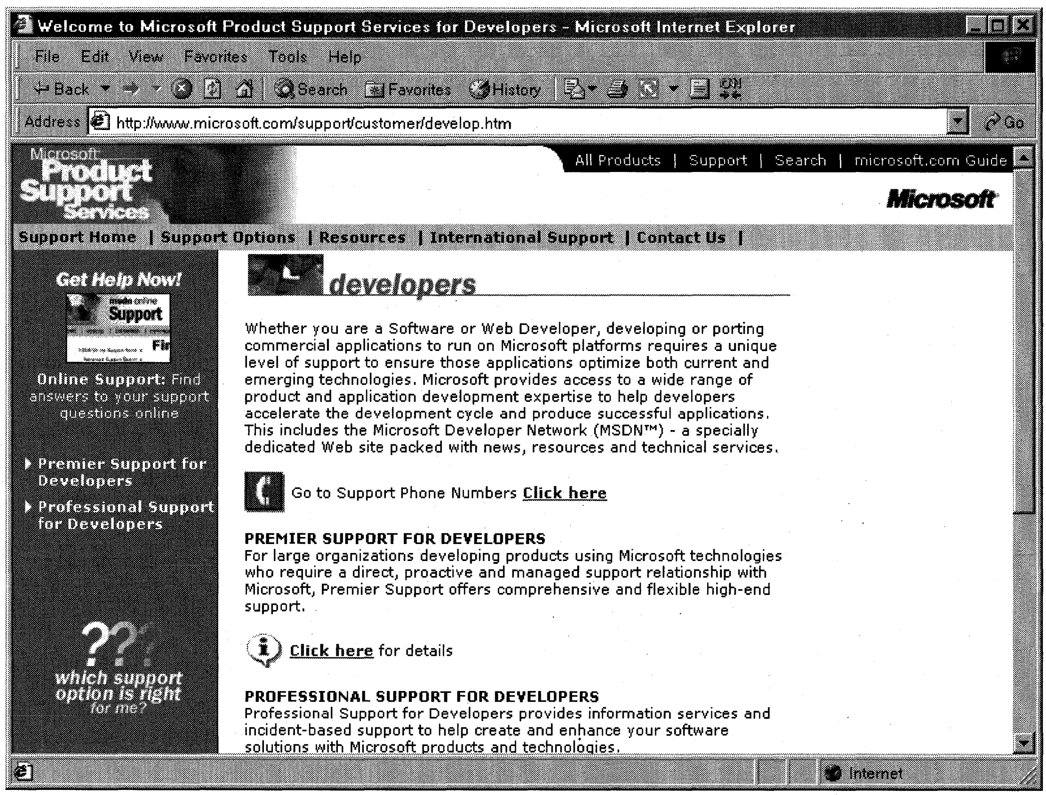

Figure 4-1: The Product Services Support page for developers.

Premier Support from Microsoft provides extensive support for developers, and includes different packages geared toward specific Microsoft customer needs. The packages of Premier Support that Microsoft provides are:

- Premier Support for Enterprises
- Premier Support for Developers
- Premier Support for Microsoft Certified Solution Providers
- Premier Support for OEMs

If you're a developer, you could fall into any of these categories. To find out more information about Microsoft's Premier Support, contact them at (800) 936-2000.

Priority Annual Support from Microsoft is geared toward developers or organizations that have more than an occasional need to call Microsoft with support questions and need priority handling of their support questions or issues. There are three packages of Priority Annual Support offered by Microsoft.

- Priority Comprehensive Support
- Priority Developer Support
- Priority Desktop Support

The best support option for you as a developer is the Priority Developer support. To obtain more information about Priority Developer Support, call Microsoft at (800) 936-3500.

Microsoft also offers a **Pay-Per-Incident Support** option so you can get help if there's just one question that you must have answered. With Pay-Per-Incident Support, you call a toll-free number and provide your Visa, MasterCard, or American Express account number, after which you receive support for your incident. In loose terms, an incident is a problem or issue that can't be broken down into subissues or subproblems (that is, it can't be broken down into smaller pieces). The number to call for Pay-Per-Incident Support is (800) 936-5800.

Note that Microsoft provides two priority technical support incidents as part of the MSDN Professional subscription, and provides four priority technical support incidents as part of the MSDN Universal subscription.

You can also **submit questions** to Microsoft engineers through Microsoft's support Web site, but if you're on a time line you might want to rethink this approach and consider going to MSDN Online and looking into the Community site for help with your development question. To submit a question to Microsoft engineers online, go to *support.microsoft.com/support/webresponse.asp*.

Online Resources

Microsoft also provides extensive developer support through its community of developers found on MSDN Online. At MSDN Online's Community site, you will find OSIGs that cover all sorts of issues in an online, ongoing fashion. To get to MSDN Online's Community site, simply go to *msdn.microsoft.com/community*.

Microsoft's MSDN Online also provides its **Knowledge Base** online, which is part of the Personal Support Center on Microsoft's corporate site. You can search the Knowledge Base online at *support.microsoft.com/support/search*.

Microsoft provides a number of **newsgroups** that developers can use to view information on newsgroup-specific topics, providing yet another developer resource for information about creating Windows applications. To find out which newsgroups are available and how to get to them, go to *support.microsoft.com/support/news*.

The following newsgroups will probably be of particular interest to readers of the *Microsoft Active Directory Developer's Reference Library*:

- *microsoft.public.win2000.**
- *microsoft.public.msdn.general*
- *microsoft.public.platformsdk.active.directory*
- *microsoft.public.platformsdk.adsi*

- *microsoft.public.platformsdk.dist_svcs*
- *microsoft.public.vb.**
- *microsoft.public.vc.**
- *microsoft.public.vstudio.*microsoft.public.cert.**
- *microsoft.public.certification.**

Of course, Microsoft isn't the only newsgroup provider on which newsgroups pertaining to developing on Windows are hosted. Usenet has all sorts of newsgroups—too many to list—that host ongoing discussions pertaining to developing applications on the Windows platform. You can access newsgroups on Windows development just as you access any other newsgroup; generally, you'll need to contact your ISP to find out the name of the mail server and then use a newsreader application to visit, read, or post to the Usenet groups.

For network developers with a taste for Winsock (and QOS) programming, another site of interest is *www.stardust.com*, which is chock full of up-to-date information about Winsock development and other network-related information. There's other information about network programming on the site, so it's worth a look.

Internet Standards

Many of the network protocols and services implemented in Windows platforms conform to one or more Internet standards recommendations that have gone through a process of review and comments. One especially useful source of information about such standards, recommendations, and ongoing comment periods is the Internet Engineering Task Force, or IETF. Rather than go into some long-winded (page-eating) explanation of what the IETF is, does, and stands for, let me simply say that this is the place where networking protocols and other various Internet-related services are often born, scrutinized, recast, commented upon, and although not standardized or implemented, recommended in a final form called a request for comment, or RFC, even though it's essentially a standard by the time it gets to RFC stage.

If you want to get a clear technical picture of a given technology or protocol, or if you're inclined to comment on the creation and subsequent scrutiny of such things, the place you should go is *www.ietf.org*. This site can tell you all you want to know about the goings on of the IETF, their (non-profit) mission, their Working Groups, and all the information you might ever want about almost anything that has to do with networking recommendations.

If you're curious about a given protocol or networking technology, and want to find an unadulterated (albeit technical) version of its explanation, this is a great place to go. It's a virtual hangout for the brightest people in networking, and it's worth a look or two, even just for the sake of satisfying curiosity.

Learning Products

Microsoft provides a number of products that enable developers to get versed in the particular tasks or tools that they need to achieve their goals (or to finish their tasks). One product line that is geared toward developers is called the Mastering series, and its products provide comprehensive, well-structured interactive teaching tools for a wide variety of development topics.

The Mastering Series from Microsoft contains interactive tools that group books and CDs together so that you can master the topic in question, and there are products available based on the type of application you're developing. To obtain more information about the Mastering series of products, or to find out what kind of offerings the Mastering series has, check out *msdn.microsoft.com/mastering*.

Other learning products are available from other vendors as well, such as other publishers, other application providers that create tutorial-type content and applications, and companies that issue videos (both taped and broadcast over the Internet) on specific technologies. For one example of a company that issues technology-based instructional or overview videos, take a look at *www.compchannel.com*.

Another way of learning about development in a particular language (such as C++, FoxPro, or Microsoft Visual Basic), for a particular operating system, or for a particular product (such as Microsoft SQL Server or Microsoft Commerce Server) is to read the preparation materials available for certification as a Microsoft Certified Solutions Developer (MCSD). Before you get defensive about not having enough time to get certified, or not having any interest in getting your certification (maybe you do—there *are* benefits, you know), let me just state that the point of the journey is not necessarily to arrive. In other words, you don't have to get your certification for the preparation materials to be useful; in fact, the materials might teach you things that you thought you knew well but actually didn't know as well as you thought you did. The fact of the matter is that the coursework and the requirements to get through the certification process are rigorous, difficult, and quite detail-oriented. If you have what it takes to get your certification, you have an extremely strong grasp of the fundamentals (and then some) of application programming and the developer-centric information about Windows platforms.

You are required to pass a set of core exams to get an MCSD certification, and then you must choose one topic from many available electives exams to complete your certification requirements. Core exams are chosen from among a group of available exams; you must pass a total of three exams to complete the core requirements. There are "tracks" that candidates generally choose which point their certification in a given direction, such as C++ development or Visual Basic development. The core exams and their exam numbers (at the time of publication) are as follows.

Desktop Applications Development (one required):

- Designing and Implementing Desktop Applications with Visual C++ 6.0 (70-016)
- Designing and Implementing Desktop Applications with Visual FoxPro 6.0 (70-156)
- Designing and Implementing Desktop Applications with Visual Basic 6.0 (70-176)

Distributed Applications Development (one required):

- Designing and Implementing Distributed Applications with Visual C++ 6.0 (70-015)
- Designing and Implementing Distributed Applications with Visual FoxPro 6.0 (70-155)
- Designing and Implementing Distributed Applications with Visual Basic 6.0 (70-175)

Solutions Architecture:

- Analyzing Requirements and Defining Solution Architectures (70-100)

Elective exams enable candidates to choose from a number of additional exams to complete their MCSD exam requirements. The following MCSD elective exams are available:

- Any Desktop or Distributed exam not used as a core requirement
- Designing and Implementing Data Warehouses with Microsoft SQL Server 7.0 (70-019)
- Developing Applications with C++ Using the Microsoft Foundation Class Library (70-024)
- Implementing OLE in Microsoft Foundation Class Applications (70-025)
- Implementing a Database Design on Microsoft SQL Server 6.5 (70-027)
- Designing and Implementing Databases with Microsoft SQL Server 7.0 (70-029)
- Designing and Implementing Web Sites with Microsoft FrontPage 98 (70-055)
- Designing and Implementing Commerce Solutions with Microsoft Site Server 3.0, Commerce Edition (70-057)
- Application Development with Microsoft Access for Windows 95 and the Microsoft Access Developer's Toolkit (70-069)
- Designing and Implementing Solutions with Microsoft Office 2000 and Microsoft Visual Basic for Applications (70-091)
- Designing and Implementing Database Applications with Microsoft Access 2000 (70-097)
- Designing and Implementing Collaborative Solutions with Microsoft Outlook 2000 and Microsoft Exchange Server 5.5 (70-105)
- Designing and Implementing Web Solutions with Microsoft Visual InterDev 6.0 (70-152)
- Developing Applications with Microsoft Visual Basic 5.0 (70-165)

The good news is that because there are exams you must pass to become certified, there are books and other material out there to teach you how to meet the knowledge level necessary to pass the exams. That means those resources are available to you—regardless of whether you care about becoming an MCSD.

The way to leverage this information is to get study materials for one or more of these exams and go through the exam preparation material (don't be fooled by believing that if the book is bigger, it must be better, because that certainly isn't always the case.) Exam preparation material is available from such publishers as Microsoft Press, IDG, Sybex, and others. Most exam preparation texts also have practice exams that let you assess your grasp on the material. You might be surprised how much you learn, even though you may have been in the field working on complex projects for some time.

Exam requirements, as well as the exams themselves, can change over time; more electives become available, exams based on previous versions of software are retired, and so on. You should check the status of individual exams (such as whether one of the exams listed has been retired) before moving forward with your certification plans. For more information about the certification process, or for more information about the exams, check out Microsoft's certification web site at *www.microsoft.com/train_cert/dev*.

Conferences

Like any industry, Microsoft and the development industry as a whole sponsor conferences on various topics throughout the year and around the world. There are probably more conferences available than any one human could possibly attend and still maintain his or her sanity, but often a given conference is geared toward a focused topic, so choosing to focus on a particular development topic enables developers to winnow the number of conferences that apply to their efforts and interests.

MSDN itself hosts or sponsors almost one hundred conferences a year (some of them are regional, and duplicated in different locations, so these could be considered one conference that happens multiple times). Other conferences are held in one central location, such as the big one—the Professional Developers Conference (PDC). Regardless of which conference you're looking for, Microsoft has provided a central site for event information, enabling users to search the site for conferences, based on many different criteria. To find out what conferences or other events are going on in your area of interest of development, go to *events.microsoft.com*.

Other Resources

Other resources are available for developers of Windows applications, some of which might be mainstays for one developer and unheard of for another. The list of developer resources in this chapter has been geared toward getting you more than started with finding the developer resources you need; it's geared toward getting you 100 percent of the way, but there are always exceptions.

Perhaps you're just getting started and you want more hands-on instruction than MSDN Online or MCSD preparation materials provide. Where can you go? One option is to check out your local college for instructor-led courses. Most community colleges offer night classes, and increasingly, community colleges are outfitted with pretty nice computer labs that enable you to get hands-on development instruction and experience without having to work on a 386/20.

There are undoubtedly other resources that some people know about that have been useful, or maybe invaluable. If you know of a resource that should be shared, send me e-mail at *winprs@microsoft.com*, and who knows—maybe someone else will benefit from your knowledge.

If you're sending mail about a particularly useful resource, simply put "Resources" in the subject line. There aren't any guarantees that you'll get a reply, but I'll read all of the mail and do what I can to ensure that your resource idea gets considered.

CHAPTER 5

Getting the Most Out of This Volume

DNS Resource Record (RR) Reference

When programming with the DNS API, it's useful to have a handy reference of the most commonly used DNS resource records; in the spirit of making this *Networking Services Developer's Reference Library* as useful as possible, I've included this reference information so that you don't have to search through other books to find it.

Let's begin with some overview information about DNS resource records. DNS resource records are the unit of information entries in DNS zone files (zone files are stored on DNS servers, and contain the resource records used locate computers in an IP network). DNS resource records are the basic building blocks of host name and IP information, and are used to resolve all DNS queries. While there are only a handful of commonly used resource record types, resource records actually come in a fairly wide variety of flavors in order to provide extended name-resolution services. This section provides reference for the most commonly used DNS resource records.

The various types of resource records come in different formats. In general however, many resource records share a common format, as the following A-type resource record example illustrates. Following the example are explanations of all of its fields.

```
Iseminger.com. 600 IN A 150.150.150.1
```

- The first field (Iseminger.com) denotes the owner.
- The second field (600) is the TTL parameter in seconds.
- The third field (IN) is the class field that represents the protocol family, which is almost always IN, for Internet class.
- The fourth field (A) is the type of resource the resource record is representing. I'll describe the commonly used types of resources in a moment.
- The fifth field (150.150.150.1) is the resource data, or RDATA. This field is a variable type that provides information appropriate for the type of resource; in this case, it's a 32-bit IP address.

There are a number of different resource record types, but there are only a handful that are commonly used in DNS. These types are the following:

- Start of authority (SOA)
- Name server (NS)
- Pointer record (PTR)
- Address (A)
- Mail exchange (MX)
- Canonical name (CNAME)
- Windows Internet Naming Service (WINS)
- WINS-reverse (WINS-R)
- Service (SRV)
- Load-sharing

These common resource record types are the subjects of this resource reference section. In the explanations I provide examples of these RR types taken from a private test deployment of Iseminger.com after the first Windows 2000 domain controller was brought online. Note that in these examples I use parentheses to identify the sample values of certain fields; remember that these values are the sample values and won't or shouldn't necessarily be the values in any other DNS deployment's resource records.

SOA Resource Records

The start of authority (SOA) record is the required first entry in all forward and reverse (in-addr.arpa) zone files and defines the zone for which the DNS server is authoritative, as well as the specific server that is authoritative for the domain. The following is an example of an SOA record:

```
@   IN  SOA     server4.iseminger.com.  dnsadmin.iseminger.com. (
                               1            ; serial number
                               3600         ; refresh    [1h]
                               600          ; retry      [10m]
                               86400        ; expire     [1d]
                               3600 )       ; min TTL    [1h]
```

The SOA RR has the following fields:

- Owner (@) specifies the owner of the record (the DNS server on which the zone file resides). The use of a freestanding @ specifies that the owner is the current origin (the server from which the file is taken).
- Class (**IN**) specifies the protocol family, in this case (and in most cases), the Internet protocol family.
- Type (**SOA**) indicates that this is an SOA RR.
- Authoritative server (**server4.iseminger.com**) specifies the DNS server that is authoritative for the zone.

- Responsible person (**dnsadmin.iseminger.com**) specifies the mailbox address of the person—presumably an administrator—who is responsible for the zone. Note that it uses a period instead of an @, as in dnsadmin.iseminger.com instead of dnsadmin@iseminger.com.
- Serial number (**1**) specifies the number of times the zone has been updated. When secondary servers contact the primary server to determine whether a zone transfer is necessary, the secondary servers compare their individual serial numbers with the primary server's serial number. If the primary server's serial number is higher, a zone transfer is necessary.
- Refresh number (**3600**) specifies the interval, in seconds, that secondary servers should wait between checks with the primary server for zone changes. The bracketed notation to its right denotes the time in common terms, such as [1h], which stands for one hour (which equates to 3600 seconds).
- Retry number (**600**) specifies the delay time, in seconds, between retries that secondary servers should use when contacting the primary server.
- Expire number (**86400**) specifies the time, in seconds, that secondary servers should wait for a response from the primary server before discarding their copies of the zone file as invalid.
- Minimum TTL (**3600**) is the default TTL value applied to resource records in the zone that do not specify their own TTL.

NS Resource Records

Name server (NS) records describe which servers are secondary servers for the zone specified in the SOA record and indicate which servers are primary servers for any delegated zones. The following are examples of NS RRs:

```
@       IN  NS      server4.iseminger.com.
@       IN  NS      dnsserver1.iseminger.com.
```

- Owner (**@**) specifies the owner of the record. As mentioned previously, the use of a freestanding @ specifies that the owner is the current origin.
- Class (**IN**) specifies the protocol family, in this case (and in most cases), the Internet protocol family.
- Type (**NS**) indicates that this is an NS RR.
- Authoritative server (**server4.iseminger.com** in the first record, **dnsserver1.iseminger.com** in the second) specifies the name of the server that houses information about the zone.

PTR Resource Records

The Pointer (PTR) record provides reverse address resolution (called reverse lookups); PTR RRs map an IP address to a host name, as the following example illustrates:

```
17.152.151.150.in-addr.arpa.    IN  PTR   filesrv1.iseminger.com.
```

Notice that the order of the IP address octets is reversed in this example:

- Class (**IN**) specifies the protocol family, in this case (and in most cases), the Internet protocol family.
- Type (**PTR**) indicates that this is a PTR RR.

A Resource Records

The Address (A) record is the most common; it simply maps an IP address to a host name, as the following example displays:

```
filesrv1        IN  A       150.151.152.17
```

- The first field (**filesrv1** in this example) is the owner (host) of the record.
- Class (**IN**) specifies the protocol family, in this case (and in most cases), the Internet protocol family.
- Type (**A**) indicates that this is an A RR.

MX Resource Records

The mail exchange (MX) record specifies where mail is to be routed for users in the given DNS domain. In addition to standard fields, the MX RR contains a field that enables administrators to weight multiple MX RRs based on whatever criteria seem appropriate. This field is called the preference field. Consider the following examples:

```
iseminger.com       IN  MX    4  mailsrv1.iseminger.com.
iseminger.com       IN  MX    9  mailsrv3.iseminger.com.
```

In these examples, the assignment of values in the preference fields (4 and 9) has the following effect:

A mail server that needs to send mail to the iseminger.com domain would contact a DNS server for iseminger.com and retrieve all of the MX records for the domain. This mail server would then attempt to contact the mail server with the lowest preference field value (mailsrv1.iseminger.com according to these sample MX entries). If contact with the host associated with the lowest preference value was not possible, the mail server would attempt to reach the MX-designated host that had the next-lowest value for its preference field (mailsrv3.iseminger.com in this example).

CNAME Resource Records

The canonical name (CNAME) record provides a mechanism by which you can assign an alias to a given host. CNAME RRs are useful for keeping the naming conventions of your network infrastructure hidden from the outside world (or the inside world, for that matter). When DNS resolves a CNAME RR, it uses the owner field (filesrv1.iseminger.com. in the following example) to subsequently find an A RR to resolve the name. An example of a CNAME RR is shown on the following page.

```
drawings        IN  CNAME       filesrv1.iseminger.com.
```

- The first field (**drawings** in this example) is the alias assigned to the host.
- Class (**IN**) specifies the protocol family, in this case (and in most cases), the Internet protocol family.
- Type (**CNAME**) indicates that this is a CNAME RR.

Note NS records must not point to a host that equates to a CNAME RR; that is, an NS record can't point to an alias. Also, NS records must have an A record in the same zone file as the NS record so that the name can be locally resolved.

WINS Resource Records

The Windows Internet Naming Service (WINS) record is implemented only by Microsoft DNS and is used when dynamically created host names registered with WINS are unavailable in a static DNS zone file. In essence, this resource record enables Microsoft DNS to make a request to a WINS server when DNS is unable to resolve a given host name. If the host name exists in the WINS database, WINS returns the query to DNS and DNS resolves the query. The following example illustrates a WINS RR:

```
@           IN  WINS        150.150.150.19
```

- Owner (@) specifies the owner of the record. As mentioned previously, the use of a freestanding @ specifies that the owner is the current origin.
- Class (**IN**) specifies the protocol family, in this case (and in most cases), the Internet protocol family.
- Type (**WINS**) indicates that this is a WINS RR.

Note The WINS and WINS-R RRs are specific to Microsoft DNS and won't work if you attempt to use them with other DNS server software.

WINS-R Resource Records

The WINS-reverse (WINS-R) record provides administrators the capability to perform reverse lookups through WINS. Consider the following WINS-R RR example:

```
17.152.151.150.in-addr.arpa.  0  IN  WINS-R  filesrv1 iseminger.com.
```

The WINS-R RR has a structure that is similar to that of the PTR RR, with the WINS-R RR containing additional information. WINS-R RRs have the following fields:

- The first field (**17.152.151.150.in-addr.arpa.**) is the reverse-lookup in-addr-arpa address.
- The time to live (TTL) value, which is specified in the second field (**0**), is usually set to zero to keep WINS-R records (which are often volatile) from being cached by DNS.

- Class (**IN**) specifies the protocol family, in this case (and in most cases), the Internet protocol family.
- Type (**WINS-R**) indicates that this is a WINS-R RR.
- The next field (**filesrv1**) indicates the NetBIOS name of the owner of the record.
- The domain name that should be appended to the host name for creation of the Fully Qualified Domain Name (FQDN) is specified in the final field (**iseminger.com.**).

SRV Resource Records

The service (SRV) record enables administrators to specify servers that service a specific service, protocol, and domain. SRV RRs have their own special syntax, as the following example illustrates:

```
http.tcp.iseminger.com. 600 IN SRV 0 100 80 web1.iseminger.com.
```

- The first field (**http.tcp.iseminger.com.**) follows a specific dot-delimited formatting convention, which can be defined as:

  [service].[protocol].[name].
- In this example, the service (**http**), protocol (**tcp**), and name (**Iseminger.com**) are dot-delimited and contain a trailing dot.
- The second field (**600**) specifies the TTL.
- The third and fourth fields (**IN** and **SRV**) specify class and type.
- The fifth field (**0**) specifies host priority. As with the MX RR preference field, clients give preference to SRV RRs with the lowest value in their priority fields.
- The sixth field (**100**) specifies weight and can be used for load balancing when SRV RRs have the same values in their priority fields. Clients should give preference to hosts with higher weight-field values.
- The seventh field (**80**) specifies the port number on which the server is listening for requests pertaining to the specified service.
- The last field (**web1.iseminger.com.**) is the FQDN for the host associated with the SRV RR.

Load Sharing Resource Records

This is less a resource type and more a means of incorporating load-sharing mechanisms into your DNS deployment. DNS can perform load sharing in a round-robin fashion. When multiple A RRs for a given host name exist in the zone file, DNS servers that are RFC 1794 compliant distribute the load across those entries by rotating which entry is returned when queries for the given host name are serviced. Take the following example:

```
www.iseminger.com.        IN  A       150.150.150.31
www.iseminger.com.        IN  A       150.150.150.32
www.iseminger.com.        IN  A       150.150.150.33
```

If *www.iseminger.com* were an internal site that was receiving lots of hits, I could mirror the site onto three (or more) servers, enter the sample RRs into DNS and viola! I get round-robin load balancing across all three servers. Windows 2000 DNS servers and versions of BIND 4.9.3 and later implement this kind of round-robin load balancing.

CHAPTER 6

Domain Name System (DNS)

DNS Overview

Domain Name System, more commonly referred to as DNS, is an industry-standard protocol used to locate computers on an IP-based network. Users are better at remembering friendly names, such as *www.microsoft.com* or *msdn.microsoft.com*, than they are at remembering number-based addresses such as 207.46.131.137.

IP networks, such as the Internet and Windows 2000 networks, rely on number-based addresses to ferry information across and throughout the network; therefore, it is necessary to translate user-friendly names (www.microsoft.com) into addresses that the network can recognize (207.46.131.137). DNS is the service of choice in Windows 2000 to locate resources and translate such resources into IP addresses.

DNS is the primary locator service for Active Directory, and therefore, DNS can be considered a base service for Windows 2000 and for Active Directory. Both Windows 2000 and Active Directory make heavy use of DNS;

Windows 2000 provides functions that enable application programmers to use DNS, such as programmatically making DNS queries, comparing records, and looking up names.

Many of the DNS functions are actually function types, in that there is a base name for the function, but its use depends on the character encoding used. For example, the **DnsQuery** function is listed in the function reference of the DNS Application Programmer's Interface (API) as **DnsQuery**, but its use in applications depends on whether the character encoding is ANSI (designated by appending _A to the function type name), Unicode (designated by appending _W to the function type name), or UTF-8 (designated by appending _UTF to the function type name). Therefore, the function call for the **DnsQuery** function would actually be one of the following:

DnsQuery_A (_A for ANSI encoding)
DnsQuery_W (_W for Unicode encoding)
DnsQuery_UTF8 (_UTF8 for UTF-8 encoding)

All functions that require this convention clearly state this requirement within the first few sentences of their function definition. You must use the proper function name; for example, you cannot simply call **DnsQuery** instead of **DnsQuery_A**.

DNS Standards Documents

The Domain Name System is an open protocol. As such, there have been many collaborative efforts from the industry as a whole to ensure that its implementation on various systems does not result in a lack of interoperability. The standards body overseeing such recommendations is the Internet Engineering Task Force (IETF). The following are IETF documents, some of them Requests for Comments (RFC) and some Internet Drafts, that are associated with DNS. For more information about any of these documents, visit *www.ietf.org.*

DNS-Related RFCs

RFC 1034: *Domain Names-Concepts and Facilities*

RFC 1035: *Domain Names-Implementation and Specification*

RFC 1123: *Requirements for Internet Hosts-Application and Support*

RFC 1886: *DNS Extensions to Support IP Version 6*

RFC 1995: *Incremental Zone Transfer in DNS*

RFC 1996: *A Mechanism for Prompt DNS Notification of Zone Changes*

RFC 2136: *Dynamic Updates in the Domain Name System* (DNS UPDATE)

RFC 2181: *Clarifications to the DNS Specification*

RFC 2308: *Negative Caching of DNS Queries* (DNS NCACHE)

DNS-Related Internet Drafts

Draft-ietf-dnsind-rfc2052bis-02.txt (*A DNS RR for Specifying the Location of Services* (DNS SRV))

Draft-skwan-utf8-dns-02.txt (*Using the UTF-8 Character Set in the Domain Name System*)

Draft-ietf-dhc-dhcp-dns-08.txt (**Interaction between DHCP and DNS**)

Draft-ietf-dnsind-tsig-11.txt (*Secret Key Transaction Signatures for DNS* (TSIG))

Draft-ietf-dnsind-tkey-00.txt (*Secret Key Establishment for DNS* (TKEY RR))

Draft-skwan-gss-tsig-04.txt (*GSS Algorithm for TSIG* (GSS-TSIG))

DNS Reference

This section defines the programmatic elements in the DNS API.

DnsAcquireContextHandle

The **DnsAcquireContextHandle** function type acquires a context handle to a set of credentials. Like many DNS functions, the **DnsAcquireContextHandle** function type is implemented in multiple forms to facilitate different character encoding. Based on the character encoding involved, use one of the following functions:

DnsAcquireContextHandle_A (_A for ANSI encoding)

DnsAcquireContextHandle_W (_W for Unicode encoding)

If the **DnsAcquireContextHandle** function type is called without its suffix (_A or _W), a compiler error will occur.

```
DNS_STATUS WINAPI DnsAcquireContextHandle(
  DWORD CredentialFlags,
  PVOID Credentials,
  HANDLE *ContextHandle
);
```

Parameters

CredentialFlags
[in] Flag indicating character encoding. Set to TRUE for Unicode, FALSE for ANSI.

Credentials
[in, optional] Pointer to the **SEC_WINNT_AUTH_IDENTITY_W** structure or the **SEC_WINNT_AUTH_IDENTITY_A** structure containing the name, domain, and password of the account to be used in a secure dynamic update. If not specified, the credentials of the calling service are used.

ContextHandle
[out] Pointer to a handle pointing to the credentials.

Return Values

Returns success confirmation upon successful completion. Otherwise, returns the appropriate DNS-specific error code as defined in Winerror.h.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

See Also

DnsQuery

DnsExtractRecordsFromMessage

The **DnsExtractRecordsFromMessage** function type extracts resource records from a DNS message, and stores those records in a **DNS_RECORD** structure. Like many DNS functions, the **DnsExtractRecordsFromMessage** function type is implemented in multiple forms to facilitate different character encoding. Based on the character encoding involved, use one of the following functions:

DnsExtractRecordsFromMessage_W (_W for Unicode encoding)

DnsExtractRecordsFromMessage_UTF8 (_UTF8 for UTF-8 encoding)

If the **DnsExtractRecordsFromMessage** function type is called without its suffix (either _W or _UTF8), a compiler error will occur.

```
DNS_STATUS WINAPI DnsExtractRecordsFromMessage (
  PDNS_MESSAGE_BUFFER pDnsBuffer,
  WORD wMessageLength,
  PDNS_RECORD *ppRecord
);
```

Parameters

pDnsBuffer
[in] Pointer to a DNS response message stored in a **DNS_MESSAGE_BUFFER** structure.

wMessageLength
[in] Size of the message stored in **DNS_MESSAGE_BUFFER**, in bytes.

ppRecord
[in, out] Pointer to a pointer to the list of extracted resource records.

Return Values

Returns success confirmation upon successful completion. Otherwise, returns the appropriate DNS-specific error code as defined in Winerror.h.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

See Also

DnsWriteQuestionToBuffer, **DnsQuery**

DnsFreeRecordList

The **DnsFreeRecordList** function frees memory allocated for DNS records obtained using the **DnsQuery** function.

```
VOID WINAPI DnsFreeRecordList (
  PDNS_RECORD pRecord,
);
```

Parameters

pRecord
[in, out] Pointer to the list of DNS records to be freed.

Remarks

The **DnsFreeRecordList** function can be used to free memory allocated from query results obtained using a **DnsQuery** function call; it cannot free memory allocated for DNS record lists created manually.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

+ See Also

DnsQuery

DnsModifyRecordsInSet

The **DnsModifyRecordsInSet** function alters an existing resource record set that was previously registered with DNS servers. Like many DNS functions, the **DnsModifyRecordsInSet** function type is implemented in multiple forms to facilitate different character encoding. Based on the character encoding involved, use one of the following functions:

DnsModifyRecordsInSet_A (_A for ANSI encoding)

DnsModifyRecordsInSet_W (_W for Unicode encoding)

DnsModifyRecordsInSet_UTF8 (_UTF8 for UTF 8 encoding)

If the **DnsModifyRecordsInSet** function type is called without its suffix (_A, _W, or _UTF8), a compiler error will occur.

```
DNS_STATUS WINAPI DnsModifyRecordsInSet(
  PDNS_RECORD pAddRecords,
  PDNS_RECORD pDeleteRecords,
  DWORD Options,
  HANDLE hContext,
  PIP_ARRAY pServerList,
  PVOID pReserved
);
```

Parameters

pAddRecords
[in] Pointer to the **DNS_RECORD** structure containing the resource records to be added to the resource record set.

pDeleteRecords
[in] Pointer to the **DNS_RECORD** structure containing the resource records to be deleted from the resource record set.

Options
[in] Options to apply to the operation. Options consist of the following, and may be combined.

Option	Meaning
DNS_UPDATE_SECURITY_USE_DEFAULT	Uses the default behavior, which is specified in the registry, for secure dynamic DNS updates.
DNS_UPDATE_SECURITY_OFF	Does not attempt secure dynamic updates.
DNS_UPDATE_SECURITY_ON	Attempts nonsecure dynamic update. If refused, then attempts secure dynamic update.
DNS_UPDATE_SECURITY_ONLY	Attempts secure dynamic updates only.
DNS_UPDATE_CACHE_SECURITY_CONTEXT	Caches the security context for use in future transactions.
DNS_UPDATE_TEST_USE_LOCAL_SYS_ACCT	Uses credentials of the local computer account.
DNS_UPDATE_FORCE_SECURITY_NEGO	Does not use cached security context
DNS_UPDATE_RESERVED	Reserved for future use.

hContextHandle
[in, optional] Handle to the credentials of a specific account. Used when secure dynamic update is required.

pServerList
[in] Array of DNS server IP addresses to which the Find Authoritative Zone (FAZ) request is sent.

pReserved
Reserved for future use.

Remarks

The **DnsModifyRecordsInSet** function type executes in the following steps.

1. Records specified in *pDeleteRecords* are deleted. If *pDeleteRecords* is empty or doesn't contain records that exist in the current set, the **DnsModifyRecordInSet** function goes to the next step.
2. Records specified in *pAddRecords* are added. If *pAddRecords* is empty, the operation completes without adding any records.

Return Values

Returns success confirmation upon successful completion. Otherwise, it returns the appropriate DNS-specific error code as defined in Winerror.h.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

+ See Also

DnsReplaceRecordSet, **DnsQuery**

DnsNameCompare

The **DnsNameCompare** function compares two DNS names. Like many DNS functions, the **DnsNameCompare** function type is implemented in multiple forms to facilitate different character encoding. Based on the character encoding involved, use one of the following functions:

DnsNameCompare_A (_A for ANSI encoding)

DnsNameCompare_W (_W for Unicode encoding)

If the **DnsNameCompare** function type is called without its suffix (_A or _W), a compiler error will occur.

```
BOOL DnsNameCompare(
  LPSTR pName1,
  LPSTR pName2
);
```

Parameters

pName1
[in] First DNS name of the comparison pair.

pName2
[in] Second DNS name of the comparison pair.

Remarks

Name comparisons are not case sensitive, and trailing dots are ignored.

As with other DNS comparison functions, the **DnsNameCompare** function deems different encoding as immediate indication of differing values, and as such, the same names with different characters encoding will not be reported identically.

Return Values

Returns TRUE if the compared names are equivalent, FALSE if they are not.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

See Also

DnsQuery, **DnsRecordCompare**, **DnsRecordSetCompare**

DnsReleaseContextHandle

The **DnsReleaseContextHandle** function releases memory used to store the credentials of a specific account.

```
VOID WINAPI DnsReleaseContextHandle(
  HANDLE ContextHandle
);
```

Parameters

ContextHandle
[in] Pointer to a handle pointing to the credentials of a specific account.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

See Also

DnsAcquireContextHandle

DnsRecordCompare

The **DnsRecordCompare** function compares two DNS resource records.

```
BOOL WINAPI DnsRecordCompare(
  PDNS_RECORD pRecord1,
  PDNS_RECORD pRecord2
);
```

Parameters

pRecord1
[in] Pointer to the first DNS resource record of the comparison pair.

pRecord2
[in] Pointer to the second DNS resource record of the comparison pair.

Remarks

When comparing records, DNS resource records that are stored using different character encoding are treated by the **DnsRecordCompare** function as different, even if the records are otherwise equivalent.

Return Values

Returns TRUE if the compared records are equivalent, FALSE if they are not.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

See Also

DnsRecordSetCompare

DnsRecordCopyEx

The **DnsRecordCopyEx** function creates a copy of a specified resource record. The **DnsRecordCopyEx** function is also capable of converting the character encoding during the **copy** operation.

```
PDNS_RECORD WINAPI DnsRecordCopyEx(
  PDNS_RECORD pRecord,
  DNS_CHARSET CharSetIn,
  DNS_CHARSET CharSetOut
);
```

Parameters

pRecord
[in] Pointer to the resource record to be copied.

CharSetIn
[in] Character encoding of the source resource record.

CharSetOut
[in] Character encoding required of the destination record.

Remarks

The *CharSetIn* parameter is used only if the character encoding of the source resource record is not specified in *pRecord*.

Return Values

Successful execution returns a pointer to the (newly created) destination record. Otherwise, returns NULL.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

See Also

DnsRecordSetCopyEx

DnsRecordSetCompare

The **DnsRecordSetCompare** function compares two resource record sets.

```
BOOL WINAPI DnsRecordSetCompare (
  PDNS_RECORD pRR1,
  PDNS_RECORD pRR2,
  PDNS_RECORD *ppDiff1,
  PDNS_RECORD *ppDiff2
);
```

Parameters

pRR1

[in, out] Pointer to the first DNS resource record set of the comparison pair.

pRR2

[in, out] Pointer to the second DNS resource record set of the comparison pair.

ppDiff1

[out] Pointer to a pointer to the list of resource records built as a result of the arithmetic performed on them: *pRRSet1* minus *pRRSet2*.

ppDiff2

[out] Pointer to a pointer to the list of resource records built as a result of the arithmetic performed on them: *pRRSet2* minus *pRRSet1*.

Remarks

When comparing records sets, DNS resource records that are stored using different character encoding are treated by the **DnsRecordSetCompare** function as *equivalent*. Contrast this to the **DnsRecordCompare** function, in which equivalent records with different encoding are *not* returned as equivalent records.

Return Values

Returns TRUE if the compared record sets are equivalent, FALSE if they are not.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

See Also

DnsRecordCompare

DnsRecordSetCopyEx

The **DnsRecordSetCopyEx** function creates a copy of a specified resource record set. The **DnsRecordSetCopyEx** function is also capable of converting the character encoding during the **copy** operation.

```
PDNS_RECORD WINAPI DnsRecordSetCopyEx(
  PDNS_RECORD pRecordSet,
  DNS_CHARSET CharSetIn,
  DNS_CHARSET CharSetOut
);
```

Parameters

pRecordSet
[in] Pointer to the resource record set to be copied.

CharSetIn
[in] Character encoding of the source resource record set.

CharSetOut
[in] Character encoding required of the destination record set.

Remarks

The *CharSetIn* parameter is used only if the character encoding of the source resource record set is not specified in *pRecordSet*.

Return Values

Successful execution returns a pointer to the (newly created) destination record set. Otherwise, it returns NULL.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

See Also

DnsRecordCopyEx

DnsRecordSetDetach

The **DnsRecordSet** function detaches the first record set from a specified list of DNS records.

```
PDNS_RECORD DnsRecordSetDetach(
  PDNS_RECORD pRR
);
```

Parameters

pRR
[in, out] On input, a pointer to the list prior to the detachment of the first DNS record in the list of DNS records. On output, a pointer to the list subsequent to the detachment of the DNS record.

Return Values

On return, the **DnsRecordSet** function points to the detached DNS record set.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

See Also

DnsQuery, **DnsRecordCompare**, **DnsRecordSetCompare**

DnsReplaceRecordSet

The **DnsReplaceRecordSet** function type replaces an existing record set. Like many DNS functions, the **DnsReplaceRecordSet** function type is implemented in multiple forms to facilitate different character encoding, which is indicated by a suffix. Based on the character encoding involved, use one of the following functions:

DnsReplaceRecordSetA (_A for ANSI encoding)

DnsReplaceRecordSetW (_W for Unicode encoding)

DnsReplaceRecordSetUTF8 (_UTF8 for UTF 8 encoding)

Notice the lack of an underscore between the function type name and its suffix. If the **DnsModifyRecordsInSet** function type is called without its suffix (A, W, or UTF8), a compiler error will occur.

```
DNS_STATUS WINAPI DnsReplaceRecordSet (
  PDNS_RECORD pNewSet,
  DWORD Options,
  HANDLE hContext,
  PIP_ARRAY pServerList,
  PVOID pReserved
);
```

Parameters

pNewSet

[in] Pointer to the **DNS_RECORD** structure holding the resource record set that replaces the existing set. The specified resource record set is replaced with the contents of *pNewSet*. To delete a resource record set, specify the set in *pNewSet* but set **RDATA** to NULL.

Options

[in] Options available for the function call, which may be combine, are shown in the table on the following page.

Option	Meaning
DNS_UPDATE_SECURITY_USE_DEFAULT	Uses the default behavior, which is specified in the registry, for secure dynamic DNS updates.
DNS_UPDATE_SECURITY_OFF	Does not attempt secure dynamic updates.
DNS_UPDATE_SECURITY_ON	Attempts nonsecure dynamic update. If refused, then attempts secure dynamic update.
DNS_UPDATE_SECURITY_ONLY	Attempts secure dynamic updates only.
DNS_UPDATE_CACHE_SECURITY_CONTEXT	Caches the security context for use in future transactions.
DNS_UPDATE_TEST_USE_LOCAL_SYS_ACCT	Uses credentials of the local computer account.
DNS_UPDATE_FORCE_SECURITY_NEGO	Does not use cached security context
DNS_UPDATE_RESERVED	Reserved for future use.

hContext
[in, optional] Handle to the credentials of a specific account. Used when secure dynamic update is required.

pServerList
[in] Array of DNS server IP addresses to which the Find Authoritative Zone (FAZ) request is sent.

pReserved
Reserved for future use.

Return Values

Returns success confirmation upon successful completion. Otherwise, returns the appropriate DNS-specific error code as defined in Winerror.h.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

DnsQuery

The **DnsQuery** function type is the generic query interface to the DNS name space, and provides application programmers with a DNS query resolution interface. Like many DNS functions, the **DnsQuery** function type is implemented in multiple forms to facilitate different character encoding. Based on the character encoding involved, use one of the following functions:

DnsQuery_A (for ANSI encoding)

DnsQuery_W (for Unicode encoding)

DnsQuery_UTF8 (for UTF-8 encoding)

If the **DnsQuery** function type is called without its suffix (_A, _W, or _UTF8), a compiler error will occur.

```
DNS_STATUS WINAPI DnsQuery (
  LPSTR lpstrName,
  WORD wType,
  DWORD fOptions,
  PIP_ARRAY aipServers,
  PDNS_RECORD *ppQueryResultsSet,
  PVOID *pReserved
);
```

Parameters

lpstrName

[in] Name of the owner of the record set being queried.

wType

[in] Numeric representation of the type of record set queried.

fOptions

[in] Query options. Options can be combined, and all options override DNS_QUERY_STANDARD. The following table lists the available query options.

Query	Meaning
DNS_QUERY_STANDARD	Standard query.
DNS_QUERY_ACCEPT_PARTIAL_UDP	Returns truncated results—does not retry under TCP.
DNS_QUERY_USE_TCP_ONLY	Uses TCP only for the query.
DNS_QUERY_NO_RECURSION	Directs the DNS server to perform an iterative query (specifically directs the DNS server not to perform recursive resolution to resolve the query).
DNS_QUERY_BYPASS_CACHE	Bypasses the resolver cache on the lookup.

(continued)

(continued)

Query	Meaning
DNS_QUERY_CACHE_ONLY	Attempts to resolve the query using locally cached data only.
DNS_QUERY_SOCKET_KEEPALIVE	Prevents the DNS query socket from closing after the response is received.
DNS_QUERY_TREAT_AS_FQDN	Prevents the DNS response from attaching suffixes to the submitted name in a name resolution process.
DNS_QUERY_ALLOW_EMPTY_ AUTH_RESP	Accepts the response with empty authority section.
DNS_QUERY_DONT_RESET_TTL_ VALUES	If set, and if the response contains multiple records, records are stored with the TTL corresponding to the minimum value TTL from among all records. When this option is set, "Do not change the TTL of individual records" in the returned record set is not modified.
DNS_QUERY_RESERVED	Reserved.

aipServers
[in, optional] Specifies DNS servers to which the query should be sent. If *aipServers* is NULL, default DNS servers for the local computer are used.

ppQueryResultsSet
[in, out, optional] Pointer to the pointer that points to the list of resource records comprising the response.

pReserved
[in, out, optional] Returns the response in original wire format.

Remarks

Callers of the **DnsQuery** function build a query using a fully-qualified DNS name and resource record type, and set query options depending on the type of service desired. When the DNS_QUERY_STANDARD option is set DNS uses the resolver cache, queries first with UDP then retries with TCP if the response is truncated, and asks the server to perform recursive resolution on behalf of the client to resolve the query.

Callers are responsible for freeing any returned resource record sets.

Note When calling one of the **DnsQuery** function types, it is important to realize that a DNS server may return multiple records in response to a query. A computer that is multihomed, for example, will receive multiple A records for the same IP address. It is the caller's responsibility to use as many of the returned records as necessary.

Consider the following scenario, in which multiple returned records requires additional activity on behalf of the application: A **DnsQuery_A** function call is made for a multihomed computer and the application finds that the address associated with first A record is not responding. The application should then attempt to use other IP addresses specified in the (additional) A records returned from the **DnsQuery_A** function call.

Return Values

Returns success confirmation upon successful completion. Otherwise, returns the appropriate DNS-specific error code as defined in Winerror.h.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

DnsQueryConfig

The **DnsQueryConfig** function enables application programmers to query for the configuration of the local computer or a specific adapter.

```
DNS_STATUS WINAPI DnsQueryConfig(
  DNS_CONFIG_TYPE Config,
  DWORD Flag,
  PWSTR pwsAdapterName,
  PVOID pReserved,
  PVOID pBuffer,
  PDWORD pBufferLength
);
```

Parameters

Config

[in] Structure specifying query requests. The following parameters can be queried:

DnsConfigPrimaryDomainName_W,
DnsConfigPrimaryDomainName_A,
DnsConfigPrimaryDomainName_UTF8,
DnsConfigAdapterDomainName_W,
DnsConfigAdapterDomainName_A,
DnsConfigAdapterDomainName_UTF8,
DnsConfigDnsServerList,
DnsConfigSearchList,

DnsConfigAdapterInfo,

DnsConfigPrimaryHostNameRegistrationEnabled,

DnsConfigAdapterHostNameRegistrationEnabled,

DnsConfigAddressRegistrationMaxCount

Flag
[in] Specifies whether the configuration should be associated with a **LocalAlloc** function call. Set *Flag* to TRUE to associate the query.

pwsAdapterName
[in] Specifies the adapter name against which the query is run.

pReserved
Reserved for future use.

pBuffer
[out] Pointer to the buffer storing the query response.

pBufferLength
[in, out] Length of the buffer, in bytes. If the buffer provided is not sufficient, an error is returned and *pBufferLength* contains the minimum necessary buffer size. Ignored on input if *Flag* is set to TRUE.

Return Values

Returns success confirmation upon successful completion. Otherwise, returns the appropriate DNS-specific error code as defined in Winerror.h.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

+ See Also

DnsQuery

DnsValidateName

The **DnsValidateName** function validates the status of a specified DNS name. Like many DNS functions, the **DnsValidateName** function type is implemented in multiple forms to facilitate different character encoding. Based on the character encoding involved, use one of the following functions:

DnsValidateName_A (_A for ANSI encoding)

DnsValidateName_W (_W for Unicode encoding)

DnsValidateName_UTF8 (_UTF8 for UTF-8 encoding)

If the **DnsValidateName** function type is called without its suffix (_A, _W, or _UTF8), a compiler error will occur.

```
DNS_STATUS DnsValidateName(
  LPCSTR pszName,
  DNS_NAME_FORMAT Format
);
```

Parameters

pszName
[in] DNS name to be examined.

Format
[in] Format of the name to be examined. The format may have the following values:

DnsNameDomain
DnsNameDomainLabel
DnsNameHostNameFull
DnsNameHostNameLabel
DnsNameWildcard
DnsNameSrvRecord

Remarks

To check the status of the Computer Host (single label), use the **DnsValidateName** function type with DnsNameHostNameLabel in *Format*.

Return Values

The **DnsValidateName** function type provides five possible return values.

- ERROR_SUCCESS
- ERROR_INVALID_NAME
- DNS_ERROR_INVALID_NAME_CHAR
- DNS_ERROR_NUMERIC_NAME
- DNS_ERROR_NON_RFC_NAME

The **DnsValidateName** function works in a progression when determining whether an error exists with a given DNS name, and returns upon finding its first error. Therefore, a DNS name that has multiple, different errors may be reported as having the first error, could be corrected and resubmitted, only then to find the second error.

The **DnsValidateName** function searches for the errors in the following progression:

Returns ERROR_INVALID_NAME if the DNS name:

- Is longer than 255 octets
- Contains a label longer than 63 octets
- Contains a space
- Contains two or more consecutive dots

- Begins with a dot
- Contains a dot if the name is submitted with *Format* set to DnsNameHostDomainLabel or DnsNameHostNameLabel.

Next, **DnsValidateName** returns DNS_ERROR_INVALID_NAME_CHAR if the DNS name:

- Contains any of the following invalid characters: {|}~[\]^':;<=>?@!"#$%^`()+/,
- Contains an asterisk (*), unless the asterisk is the first label in the multi-labeled name, submitted with *Format* set to DnsNameWildcard.

Next, **DnsValidateName** returns DNS_ERROR_NUMERIC_NAME if the DNS name:

- Consists of one or more labels build of only the numeric characters (0–9), Unless *Format* is DnsNameDomainLabel or DnsNameDomain, and one of the labels is not fully numeric.

Then, **DnsValidateName** returns DNS_ERROR_NON_RFC_NAME if the DNS name:

- Contains at least one extended or Unicode character
- Contains underscore (_), unless the underscore is a first character in a label, in the name, submitted with *Format* set to DnsNameSrvRecord.

Note that if **DnsValidateName** returns DNS_ERROR_NON_RFC_NAME, the error should be treated as a warning that not all DNS servers will accept the name. When this error is received, note that Windows 2000 DNS Server *does* accept the submitted name, if appropriately configured (default configuration does accept the name as submitted when DNS_ERROR_NON_RFC_NAME is returned), but other DNS server software may not.

If **DnsValidateName** returns any of the following, the error should be treated as an invalid host name:

- DNS_ERROR_NUMERIC_NAMEa
- DNS_ERROR_INVALID_NAME_CHAR
- ERROR_INVALID_NAME

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

+ See Also

DnsNameCompare, **DnsQuery**

DnsWriteQuestionToBuffer

The **DnsWriteQuestionToBuffer** function type creates a DNS query message and stores it in a **DNS_MESSAGE_BUFFER** structure. Like many DNS functions, the **DnsWriteQuestionToBuffer** function type is implemented in multiple forms to facilitate different character encoding. Based on the character encoding involved, use one of the following functions:

DnsWriteQuestionToBuffer_W (_W for Unicode encoding)

DnsWriteQuestionToBuffer_UTF8 (_UTF8 for UTF-8 encoding)

If the **DnsWriteQuestionToBuffer** function type is called without its suffix (either _W or _UTF8), a compiler error will occur.

```
BOOL WINAPI DnsWriteQuestionToBuffer (
  PDNS_MESSAGE_BUFFER pDnsBuffer,
  LPDWORD pdwBufferSize,
  LPWSTR pszName,
  WORD wType,
  WORD Xid,
  BOOL fRecursionDesired
);
```

Parameters

pDnsBuffer

[in, out] Pointer to a DNS query message stored in a buffer.

pdwBufferSize

[in, out] Size of the buffer allocated to store the message, in bytes. If the buffer size is insufficient to contain the message, an error is returned and *pdwBufferSize* contains the minimum required buffer size.

pszName

[in] Name of the owner of the record set being queried.

wType

[in] Numeric representation of the type of record set queried.

Xid

[in] Query identifier.

fRecursionDesired

[in] Flag indicating the desired type of DNS name resolution. Set to TRUE to request recursive name resolution, FALSE to request iterative name resolution.

Return Values

Returns TRUE upon successful execution, otherwise returns FALSE.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Windns.h.
Library: Use Dnsapi.lib.

See Also

DnsQuery

CHAPTER 7

Dynamic Host Configuration Protocol (DHCP)

DHCP Overview

The Dynamic Host Configuration Protocol (DHCP) Application Programming Interface, also referred to as DHCP Client Options, enables Windows 2000 and Windows 98 clients to query specific options from DHCP servers. Such capability enables vendor-specific options exposed through DHCP servers to be queried by Windows 2000 or Windows 98 DHCP clients. This documentation refers to the most recent version of the DHCP API, version 2.

Programmers using the DHCP API should note the following:

- The adapter name being passed in DHCP functions should be the adapter GUID for the routine.
- The class ID parameter in DHCP functions is the binary class ID information to pass in the DHCP INFORM packet through use of the USER CLASS option.
- The *AdapterName* parameter exists on Windows 98, but it refers to the adapter index (converted to a string) rather than the adapter name itself. This is necessary because Windows 98 does not have the Windows 2000 equivalent notion of adapter names.
- DHCP functions are exposed through Dhcpcsvc.dll.

DHCP Standards

Dynamic Host Configuration Protocol (DHCP) is a standardized protocol that enables clients to be dynamically assigned with various configuration parameters, such as an IP address, subnet mask, default gateway, and other critical network configuration information. DHCP servers centrally manage such configuration data, and are configured by network administrators with settings that are appropriate for a given network environment. DHCP servers in turn communicate with DHCP clients through the use of DHCP messages.

DHCP has many associated documents that standardize the protocol, and the messages DHCP clients and servers use to communicate their requests and data. These standardization documents can be found at the Internet Engineering Task Force (IETF) web site, located at *www.ietf.org*

The following are some relevant Request For Comments documents (RFCs) associated with DHCP, which include definitions for DHCP messages such as INFORM, and others:

- Dynamic Host Configuration Protocol (RFC 2131)
- Interoperation Between DHCP and BOOTP (RFC 1534)
- Clarifications and Extensions for the Bootstrap Protocol (RFC 1542)
- DHCP Options and BOOTP Vendor Extensions (RFC 2132)
- Procedure for Defining New DHCP Options (RFC 2489)
- DHCP Options for Service Location Protocol (RFC 2610)

Note that there are additional RFCs associated with DHCP available on the IETF web site, and that standards efforts and specifications are subject to change. If you are interested in tracking specific or new standards efforts, you should frequently consult the IETF web site.

DHCP Examples

The following examples illustrate two uses of the DHCP API:

- **Example 1** illustrates how to use the **DhcpRequestParams** function to retrieve a host name.
- **Example 2** shows how the **DhcpRegisterParamChange** function can be used to keep track of host name changes.

Example 1: Using the DhcpRequestParams function

The following example illustrates how to retrieve the host name using the **DhcpRequestParams** function call. The name of the adapter can be retrieved using the **GetInterfaceInfo** structure, which is part of the Internet Protocol Helper API:

```
BOOL
RetrieveHostName(
    IN LPCWSTR    pszAdapterName,
    IN OUT CHAR[] pszHostNameBuf, // must be large enough buffer
    IN DWORD      dwHostNameBufSize
)
/*++

Routine returns TRUE on success and FALSE on failure.

--*/
{
    DWORD dwError, dwSize;
    CHAR TmpBuffer[1000]; // host name won't be larger than this
```

```
DHCPCAPI_PARAMS DhcpApiHostNameParams = {
        0,                // Flags
        OPTION_HOST_NAME, // OptionId
        FALSE,            // vendor specific?
        NULL,             // data filled in on return
        0                 // nBytes
    };
DHCPCAPI_PARAMS_ARRAY DhcpApiParamsArray = {
        1,  // only one option to request
        &DhcpApiHostNameParams
    };

dwSize = sizeof(TmpBuffer);
dwError = DhcpRequestParams(
        DHCPCAPI_REQUEST_SYNCHRONOUS, // Flags
        NULL,                         // Reserved
        pszAdapterName,               // Adapter Name
        NULL,                         // not using class id
        NULL,                         // nothing to send
        &RequestParams,               // requesting params
        (PBYTE) TmpBuf,               // buffer
        &dwSize,                      // buffer size
        NULL                          // Request ID
    );

if( ERROR_MORE_DATA == dwError )
{
        //
        // dwSize is not large enough.
        //
}

if( NO_ERROR == dwError )
{

        // Check if the requested option was obtained.

        if( DhcpApiHostNameParams.nBytesData )
        {

            // Check size with dwHostNameBufSize.

            CopyMemory(
                pszHostNameBuf, DhcpApiHostNameParams.Data,
                DhcpApiHostNameParams.nBytesData
```

(continued)

(continued)

```
                    );
                pszHostNameBuf[DhcpApiHostNameParams.nBytesData] = '\0';
                return TRUE;
            }
    }

    return FALSE;
}
```

Example 2: Using the DhcpRegisterParamChange function

The following code illustrates how the **DhcpRegisterParamChange** function can be used to keep track of host name changes:

```
ULONG
UpdateHostNameLoop(
    IN LPCWSTR    pszAdapterName,
    IN CHAR       pszHostNameBuf[],
    IN ULONG      dwHostBufSize
)
{
    DWORD dwError;
    HANDLE hEvent;
    DHCPCAPI_PARAMS DhcpApiHostNameParams = {
            0,                  // Flags
            OPTION_HOST_NAME,   // OptionId
            FALSE,              // vendor specific?
            NULL,               // data filled in on return
            0                   // nBytes
        };
    DHCPCAPI_PARAMS_ARRAY DhcpApiParamsArray = {
        1,          // only one option to request
        &DhcpApiHostNameParams
        };

    dwError = DhcpRegisterParamChange(
        DHCPCAPI_REGISTER_HANDLE_EVENT, // Flags
        NULL,                           // Reserved
        pszAdapterName,                 // adapter name
        NULL,                           // no class ID
        &DhcpApiHostNameParams,         // params of interest
        (LPVOID)&hEvent                 // event handle
        );

    if( NO_ERROR != dwError ) return dwError;
```

```
    // Wait on event all the time.

    while( WAIT_OBJECT_0 == WaitForSingleObject(hEvent, INFINITE) )
    {

        // Get host name and update it.

        ResetEvent(hEvent);
        dwError = RetrieveHostName(pszAdapterName, pszHostNameBuf, dwHostBufSize
);

        // Ignore this error.

        break;
    }


    // Wait failed or retrieve failed? De-register the event handle.

    (void)DhcpDeRegisterParamChange(
        DHCPCAPI_REGISTER_HANDLE_EVENT, // Flags
        NULL,                           // Reserved
        (LPVOID) hEvent                 // event
        );

    return dwError;
}
```

Note The event handle obtained by this routine must *not* be closed with the **CloseHandle** function. It should be released using the **DhcpDeRegisterParamChange** function in order to avoid resources leaks; the **DhcpDeRegisterParamChange** function releases internal resources allocated for this notification.

DHCP Functions

DHCP provides the following functions that enable application programmers to initialize, request, and clean up DHCP-specific data for any given application:

- **DhcpCApiInitialize**
- **DhcpCApiCleanup**
- **DhcpRequestParams**
- **DhcpUndoRequestParams**
- **DhcpRegisterParamChange**
- **DhcpDeRegisterParamChange**

The **DhcpCApiInitialize** function should always be the first function called whenever this suite of DHCP functions are implemented.

DhcpCApiInitialize

The **DhcpCApiInitialize** function must be the first function call made by users of DHCP, it prepares the system for all other DHCP function calls. Other DHCP functions should only be called if the **DhcpCApiInitialize** function executes successfully.

```
DWORD
APIENTRY
DhcpCApiInitialize(
  OUT DWORD *pdwVersion
);
```

Parameters

pdwVersion
Pointer to the DHCP version implemented by the client.

Return Values

Returns ERROR_SUCCESS upon successful completion.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Dhcpcsdk.h.
Library: Use Dhcpcsvc.lib.

See Also

DHCP Overview, DHCP Functions, **DhcpCApiCleanup**

DhcpCApiCleanup

The **DhcpCApiCleanup** function enables DHCP to properly clean up resources allocated throughout the use of DHCP function calls. The **DhcpCApiCleanup** function must only be called if a previous call to **DhcpCApiInitialize** executed successfully.

```
VOID
APIENTRY
DhcpCApiCleanup(VOID);
```

Parameters

This function has no parameters.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Dhcpcsdk.h.
Library: Use Dhcpcsvc.lib.

See Also

DHCP Overview, DHCP Functions, **DhcpCApiInitialize**

DhcpRequestParams

The **DhcpRequestParams** function enables callers to synchronously, or synchronously and persistently obtain DHCP data from a DHCP Server.

```
DWORD
APIENTRY
DhcpRequestParams(
  DWORD dwFlags,
  LPVOID pReserved,
  LPWSTR pszAdapterName,
  LPDHCPCAPI_CLASSID pClassId,
  DHCPCAPI_PARAMS_ARRAY pSendParams,
  DHCPCAPI_PARAMS_ARRAY pRecdParams,
  LPBYTE pbBuffer,
  LPDWORD pdwSize,
  LPWSTR pszRequestIdStr
);
```

Parameters

dwFlags
[in] Flags that specify the data being requested. Must be set to DHCPAPI_REQUEST_SYNCHRONOUS, and may optionally be set with the additional DHCPAPI_REQUEST_PERSISTENT flag. This parameter is optional.

pReserved
[in] Reserved for future use. Must be set to NULL.

pszAdapterName
[in] Name of the adapter on which requested data is being made.

pClassId
[in] Class ID that should be used if DHCP INFORM messages are being transmitted onto the network. This parameter is optional.

pSendParams

[in] Optional data to be requested, in addition to the data requested in the *pRecdParams* array. The *pSendParams* parameter cannot contain any of the standard options that the DHCP client sends by default. This parameter is optional.

pRecdParams

[in, out] Array of DHCP data the caller is interested in receiving. This array must be empty prior to the **DhcpRequestParams** function call.

pbBuffer

[in] Buffer used for storing the data associated with requests made in *pRecdParams*.

pdwSize

[in] Size of *pbBuffer*.

[out] Required size of the buffer, if *pbBuffer* is insufficiently sized to hold the data, otherwise indicates size of the successfully filled *pbBuffer*.

pszRequestIdStr

[in] Application Identifier (ID) used to facilitate a persistent request. Must be a printable string with no special characters (for example, commas, backslashes, colons, or other illegal characters may not be used). The specified application ID is used in a subsequent **DhcpUndoRequestParams** function call to clear the persistent request, as necessary.

Remarks

DHCP clients store data obtained from a DHCP server in their local cache. If the DHCP client cache contains all data requested in the *pRecdParams* array of a **DhcpRequestParams** function call, the client returns data from its cache. If requested data is not available in the client cache, the client processes the **DhcpRequestParams** function call by submitting a DHCP-INFORM message to the DHCP server.

When the client submits a DHCP-INFORM message to the DHCP server, it includes any requests provided in the optional *pSendParams* parameter, and provides the Class ID specified in the *pClassId* parameter, if provided.

Clients can also specify that DHCP data be retrieved from the DHCP server each time the DHCP client boots, which is considered a *persistent* request. To enable persistent requests, the caller must specify the *pszRequestIdStr* parameter, and also specify the additional DHCPAPI_REQUEST_PERSISTENT flag in the *dwFlags* parameter. This persistent request capability is especially useful when clients need to automatically request application-critical information at each boot. To disable a persist request, clients must call the **DhcpUndoRequestParams** function.

For more information about DHCP INFORM messages, and other standards-based information about DHCP, consult ***DHCP Standards***.

To see the **DhcpRequestParams** function in use, see ***DHCP Examples***.

Return Values

Returns ERROR_SUCCESS upon successful completion.

Upon return, *pRecdParams* is filled with pointers to requested data, with corresponding data placed in *pbBuffer*. If *pdwSize* indicates that *pbBuffer* has insufficient space to store returned data, the **DhcpRequestParams** function returns ERROR_MORE_DATA, and returns the required buffer size in *pdwSize*. Note that the required size of *pbBuffer* may increase during the time that elapses between the initial function call's return and a subsequent call; therefore, the required size of *pbBuffer* (indicated in *pdwSize*) provides an indication of the *approximate* size required of *pbBuffer*, rather than guaranteeing that subsequent calls will return successfully if *pbBuffer* is set to the size indicated in *pdwSize*.

Other errors return appropriate Win32 error codes.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Dhcpcsdk.h.
Library: Use Dhcpcsvc.lib.

See Also

DHCP Overview, DHCP Functions, **DhcpCApiInitialize**, **DhcpUndoRequestParams**

DhcpUndoRequestParams

The **DhcpUndoRequestParams** function removes persistent requests previously made with a **DhcpRequestParams** function call.

```
DWORD
APIENTRY
DhcpUndoRequestParams(
  DWORD dwFlags,
  LPVOID pReserved,
  LPWSTR pszAdapterName,
  LPWSTR pszRequestIdStr
);
```

Parameters

dwFlags
[in] Must be zero.

pReserved
[in] Reserved for future use. Must be set to NULL.

pszAdapterName
[in] Name of the adapter for which information is no longer required.

pszRequestIdStr
[in] Application Identifier (ID) originally used to make a persistent request. This string must match the *pszRequestIdStr* parameter used in the **DhcpRequestParams** function call that obtained the corresponding persistent request. Note that this must match the previous application ID used, and must be a printable string with no special characters (for example, commas, backslashes, colons, or other illegal characters may not be used).

Remarks

Persistent requests are typically made by the setup or installer process associated with the application. When appropriate, the setup or installer process would likely make the **DhcpUndoRequestParams** function call to cancel its associated persistent request.

Return Values

Returns ERROR_SUCCESS upon successful completion. Otherwise, returns Win32 error codes.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Dhcpcsdk.h.
Library: Use Dhcpcsvc.lib.

See Also

DHCP Overview, DHCP Functions, **DhcpCApiInitialize**, **DhcpRequestParams**

DhcpRegisterParamChange

The **DhcpRegisterParamChange** function enables clients to register for notification of changes in DHCP configuration parameters.

```
DWORD
APIENTRY
DhcpRegisterParamChange(
  DWORD dwFlags,
  LPVOID pReserved,
  LPWSTR pszAdapterName,
  LPDHCPCAPI_CLASSID pClassId,
  DHCPCAPI_PARAMS_ARRAY pParams,
  LPVOID pHandle
);
```

Parameters

dwFlags

[in] Identifies the notification mechanism to be used. In version 2 of the Microsoft DHCP Application Programming Interface (API), only event-based notification is supported, and *dwFlags* must be set to DHCPAPI_REGISTER_HANDLE_EVENT.

pReserved

[in] Reserved for future use. Must be set to NULL.

pszAdapterName

[in] Name of the adapter for which event notification is being requested.

pClassId

[in] Class ID with which requested notification parameters are to be associated.

pParams

[in] Parameters for which the client is interested in registering for notification.

pHandle

[in, out] Attributes of *pHandle* are determined by the value of *dwFlags*. In version 2 of the DHCP API, *dwFlags* must be set to DHCPAPI_REGISTER_HANDLE_EVENT, and therefore, *pHandle* must be a pointer to a HANDLE variable that will hold the handle to a Windows event that gets signaled when parameters specified in *pParams* change. Note that *pHandle* is used in a subsequent call to the **DhcpDeRegisterParamChange** function to de-register event notifications associated with this particular call to the **DhcpRegisterParamChange** function.

Remarks

Version 2 of the DHCP API provides only event-based notification. With event-based notification in DHCP, clients enable notification by having *pHandle* point to a variable that, upon successful return, holds the EVENT handles that are signaled whenever changes occur to the parameters requested in *pParams*.

Return Values

Returns ERROR_SUCCESS upon successful completion. Otherwise, returns Win32 error codes.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Dhcpcsdk.h.
Library: Use Dhcpcsvc.lib.

+ See Also

DHCP Overview, DHCP Functions, **DhcpCApiInitialize**, **DhcpDeRegisterParamChange**

DhcpDeRegisterParamChange

The **DhcpDeRegisterParamChange** function releases resources associated with previously registered event notifications, and closes the associated event handle.

```
DWORD
APIENTRY
DhcpDeRegisterParamChange(
  DWORD dwFlags,
  LPVOID pReserved,
  LPVOID hEvent
);
```

Parameters

dwFlags
[in] Must be the same value as the *dwFlags* parameter in the **DhcpRegisterParamChange** function call associated with *hEvent*.

pReserved
[in] Reserved for future use. Must be set to NULL.

hEvent
[in] Must be the same value as the *hEvent* parameter in the **DhcpRegisterParamChange** function call for which the client is de-registering event notification.

Remarks

The **DhcpDeRegisterParamChange** function must be made subsequent to an associated **DhcpRegisterParamChange** function call, and the *dwFlags* and *hEvents* parameters of **DhcpDeRegisterParamChange** must match corresponding parameters of the previous and associated **DhcpRegisterParamChange** function call.

Return Values

Returns ERROR_SUCCESS upon successful completion. Otherwise, returns Win32 error codes.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Dhcpcsdk.h.
Library: Use Dhcpcsvc.lib.

See Also

DHCP Overview, DHCP Functions, **DhcpRegisterParamChange**, **DhcpCApiInitialize**

CHAPTER 8

Multicast Address Dynamic Client Allocation Protocol (MADCAP)

MADCAP Overview

Multicast Address Dynamic Client Allocation Protocol (MADCAP) enables clients to query and request multicast addresses from multicast (MADCAP) servers. By using this set of client APIs, MADCAP clients can lease, renew, and release multicast addresses from qualifying MADCAP servers across the network.

MADCAP is based on an Internet standard recommendation being developed and reviewed by the Multicast-Address Allocation (MALLOC) Internet E Task Force working group.

For more information about IETF and the MALLOC working group, visit *www.ietf.org/html.charters/malloc-charter.html*. For more information about MADCAP, review the IETF Internet-Draft titled draft-ietf-malloc-madcap-05.txt, available at *www.ietf.org/internet-drafts/draft-ietf-malloc-madcap-05.txt*.

MADCAP Functions

The following reference pages explain the functions that are available for MADCAP clients.

McastApiStartup
McastApiCleanup
McastEnumerateScopes
McastGenUID
McastRequestAddress
McastRenewAddress
McastReleaseAddress

McastApiStartup

The **McastApiStartup** function facilitates MADCAP-version negotiation between requesting clients and the version of MADCAP implemented on the system. Calling **McastApiStartup** allocates necessary resources; it must be called before any other MADCAP client functions are called.

```
DWORD APIENTRY McastApiStartup(
  PDWORD pVersion
);
```

Parameters

pVersion

[in] Pointer to the version of multicast (MCAST) that the client wishes to use.

[out] Pointer to the version of MCAST implemented on the system.

Remarks

Clients can specify which version they want to use in the *pVersion* parameter. If the system's implementation supports the requested MCAST version, the function call succeeds. If the system's implementation does not support the requested version, the function fails with MCAST_API_CURRENT_VERSION.

The client can automatically negotiate the first version of MCAST (MCAST_API_VERSION_1) by setting the *pVersion* parameter to zero.

The **McastApiStartup** function always returns the most recent version of MADCAP available on the system (MCAST_API_CURRENT_VERSION) in *pVersion*, enabling clients to discover the most recent version implemented on the system.

Return Values

If the client requests a version of MADCAP that is not supported by the system, the **McastApiStartup** function returns ERROR_NOT_SUPPORTED. If resources fail to be allocated for the function call, ERROR_NO_SYSTEM_RESOURCES is returned.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

McastApiCleanup

The **McastApiCleanup** function deallocates resources that are allocated with **McastApiStartup**. The **McastApiCleanup** function must only be called after a successful call to **McastApiStartup.**

```
VOID APIENTRY McastApiCleanup(VOID);
```

Parameters

The **McastApiCleanup** function has no parameters.

Return Values

The **McastApiCleanup** function does not return any values.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

McastEnumerateScopes

The **McastEnumerateScopes** function enumerates multicast scopes available on the network.

```
DWORD APIENTRY McastEnumerateScopes(
  IP_ADDR_FAMILY AddrFamily,
  BOOL ReQuery,
  PMCAST_SCOPE_ENTRY pScopeList,
  PDWORD pScopeLen,
  PDWORD pScopeCount
);
```

Parameters

AddrFamily

[in] Specifies the address family to be used in enumeration, in the form of an **IPNG_ADDRESS** structure. Use AF_INET for IPv4 addresses and AF_INET6 for IPv6 addresses.

ReQuery

[in] Enables a caller to requery a list. Set this parameter to TRUE if the list is to be requeried. Otherwise, set it to FALSE.

pScopeList

[in, out] Pointer to a buffer used for storing scope list information, in the form of an **MCAST_SCOPE_ENTRY** structure. The return value of *pScopeList* depends on its input value, and on the value of the buffer to which it points:

If *pScopeList* is a valid pointer on input, the scope list is returned.

If *pScopeList* is NULL on input, the length of the buffer required to hold the scope list is returned.

If the buffer pointed to in *pScopeList* is NULL on input, **McastEnumerateScopes** forces a requerying of scope lists from MCAST servers.

To determine the size of buffer required to hold scope list data, set *pScopeList* to NULL and *pScopeLen* to a non-NULL value. The **McastEnumerateScopes** function will then return ERROR_SUCCESS and store the size of the scope list data, in bytes, in *pScopeLen*.

pScopeLen

[in, out] Pointer to a value used to communicate the size of data or buffer space in *pScopeList*. On input, *pScopeLen* points to the size, in bytes, of the buffer pointed to by *pScopeList*. On return, *pScopeLen* points to the size of the data copied to *pScopeList*.

The *pScopeLen* parameter cannot be NULL. If the buffer pointed to by *pScopeList* is not large enough to hold the scope list data, **McastEnumerateScopes** returns ERROR_MORE_DATA and stores the required buffer size, in bytes, in *pScopeLen*.

To determine the size of buffer required to hold scope list data, set *pScopeList* to NULL and *pScopeLen* to a non-NULL value. The **McastEnumerateScopes** function will then return ERROR_SUCCESS and store the size of the scope list data, in bytes, in *pScopeLen*.

pScopeCount

[out] Pointer to the number of scopes returned in *pScopeList*.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the buffer pointed to by *pScopeList* is too small to hold the scope list, the **McastEnumerateScopes** function returns ERROR_MORE_DATA, and stores the required buffer size, in bytes, in *pScopeLen*.

If the **McastApiStartup** function has not been called (it must be called before any other MADCAP client functions may be called), the **McastEnumerateScopes** function returns ERROR_NOT_READY.

Remarks

The **McastEnumerateScopes** function queries multicast scopes for each network interface, and the interface on which the scope is retrieved is returned as part of the *pScopeList* parameter. Therefore, on multihomed computers it is possible that some scopes will get listed multiple times; once for each interface.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

McastGenUID

The **McastGenUID** function generates a unique identifier, subsequently used by clients to request and renew addresses.

```
DWORD APIENTRY McastGenUID(
  LPMCAST_CLIENT_UID pRequestID
);
```

Parameters

pRequestID

[in] Pointer to the **MCAST_CLIENT_UID** structure into which the unique identifier is stored. The size of the buffer to which *pRequestID* points must be at least MCAST_CLIENT_ID_LEN in size.

Return Values

The **McastGenUID** function returns the status of the operation.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

McastRequestAddress

The **McastRequestAddress** function requests one or more multicast addresses from a MADCAP server.

```
DWORD APIENTRY McastRequestAddress(
  IP_ADDR_FAMILY AddrFamily,
  LPMCAST_CLIENT_UID pRequestID,
  PMCAST_SCOPE_CTX pScopeCtx,
  PMCAST_LEASE_REQUEST pAddrRequest,
  PMCAST_LEASE_RESPONSE pAddrResponse
);
```

Parameters

AddrFamily

[in] Specifies the address family to be used in the request, in the form of an **IPNG_ADDRESS** structure. Use AF_INET for IPv4 addresses and AF_INET6 for IPv6 addresses.

pRequestID
[in] Pointer to a unique identifier for the request, in the form of an **MCAST_CLIENT_UID** structure. Clients are responsible for ensuring that each request contains a unique identifier; unique identifiers can be obtained by calling the **McastGenUID** function.

pScopeCtx
[in] Pointer to the context of the scope from which the address is to be allocated, in the form of an **MCAST_SCOPE_CTX** structure. The scope context must be retrieved by calling the **McastEnumerateScopes** function prior to calling the **McastRequestAddress** function.

pAddrRequest
[in] Pointer to the **MCAST_LEASE_REQUEST** structure containing multicast lease request parameters.

pAddrResponse
[in, out] Pointer to a buffer containing response parameters for the multicast address request, in the form of an **MCAST_LEASE_RESPONSE** structure. The caller is responsible for allocating sufficient buffer space for the *pAddrBuf* member of the **MCAST_LEASE_RESPONSE** structure to hold the requested number of addresses; the caller is also responsible for setting the pointer to that buffer.

Return Values

The **McastRequestAddress** function returns the status of the operation.

Remarks

Before the **McastRequestAddress** function is called, the scope context must be retrieved by calling the **McastEnumerateScopes** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

McastRenewAddress

The **McastRenewAddress** function renews one or more multicast addresses from a MADCAP server.

```
DWORD APIENTRY McastRenewAddress(
  IP_ADDR_FAMILY AddrFamily,
  LPMCAST_CLIENT_UID pRequestID,
  PMCAST_LEASE_REQUEST pRenewRequest,
  PMCAST_LEASE_RESPONSE pRenewResponse
);
```

Parameters

AddrFamily

[in] Designates the address family. Use AF_INET for Internet Protocol version 4 (IPv4), and AF_INET6 for Internet Protocol version 6 (IPv6).

pRequestID

[in] Unique identifier used when the address or addresses were initially obtained.

pRenewRequest

[in] Pointer to the **MCAST_LEASE_REQUEST** structure containing multicast renew–request parameters.

pRenewResponse

[in, out] Pointer to a buffer containing response parameters for the multicast address–renew request, in the form of an **MCAST_LEASE_RESPONSE** structure. The caller is responsible for allocating sufficient buffer space for the **pAddrBuf** member of the **MCAST_LEASE_RESPONSE** structure to hold the requested number of addresses; the caller is also responsible for setting the pointer to that buffer.

Return Values

The **McastRenewAddress** function returns the status of the operation.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

McastReleaseAddress

The **McastReleaseAddress** function releases leased multicast addresses from the MCAST server.

```
DWORD APIENTRY McastReleaseAddress(
  IP_ADDR_FAMILY AddrFamily,
  LPMCAST_CLIENT_UID pRequestID,
  PMCAST_LEASE_REQUEST pReleaseRequest
);
```

Parameters

AddrFamily

[in] Designates the address family. Use AF_INET for Internet Protocol version 4 (IPv4), and AF_INET6 for Internet Protocol version 6 (IPv6).

pRequestID

[in] Unique identifier used when the address or addresses were initially obtained.

pReleaseRequest
[in] Pointer to the **MCAST_LEASE_REQUEST** structure containing multicast parameters associated with the release request.

Return Values

The **McastReleaseAddress** function returns the status of the operation.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

MADCAP Structures

The following reference pages explain the union and structures that facilitate programming for Multicast Address Dynamic Client Allocation Protocol (MADCAP).

IPNG_ADDRESS
MCAST_CLIENT_UID
MCAST_SCOPE_CTX
MCAST_SCOPE_ENTRY
MCAST_LEASE_REQUEST
MCAST_LEASE_RESPONSE

IPNG_ADDRESS

The **IPNG_ADDRESS** union provides Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6) addresses.

```
typedef union _IPNG_ADDRESS {
  DWORD   IpAddrV4;
  BYTE    IpAddrV6[16];
} IPNG_ADDRESS, *PIPNG_ADDRESS;
```

Members

IpAddrV4
Internet Protocol (IP) address, in version 4 format (IPv4).

IpAddrV6
Internet Protocol (IP) address, in version 6 format (IPv6).

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

MCAST_CLIENT_UID

The **MCAST_CLIENT_UID** structure describes the unique client identifier for each multicast request

```
typedef struct      _MCAST_CLIENT_UID {
  LPBYTE              ClientUID;
  DWORD               ClientUIDLength;
} MCAST_CLIENT_UID, *LPMCAST_CLIENT_UID;
```

Members

ClientUID

Buffer containing the unique client identifier.

ClientUIDLength

Size of the **ClientUID** member, in bytes.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

MCAST_SCOPE_CTX

The **MCAST_SCOPE_CTX** structure defines the scope context for programmatic interaction with multicast addresses. The **MCAST_SCOPE_CTX** structure is used by various MADCAP functions as a handle for allocating, renewing, or releasing MADCAP addresses.

```
typedef struct _MCAST_SCOPE_CTX {
  IPNG_ADDRESS          ScopeID;
  IPNG_ADDRESS          Interface;
  IPNG_ADDRESS          ServerID;
} MCAST_SCOPE_CTX, *PMCAST_SCOPE_CTX;
```

Members

ScopeID

Identifier for the multicast scope, in the form of an **IPNG_ADDRESS** structure.

Interface

Interface on which the multicast scope is available, in the form of an **IPNG_ADDRESS** structure.

ServerID

Internet Protocol (IP) address of the MADCAP server, in the form of an **IPNG_ADDRESS** structure.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

MCAST_SCOPE_ENTRY

The **MCAST_SCOPE_ENTRY** structure provides a complete set of information about a given multicast scope.

```
typedef struct _MCAST_SCOPE_ENTRY {
  MCAST_SCOPE_CTX       ScopeCtx;
  IPNG_ADDRESS          LastAddr;
  DWORD                 TTL;
  UNICODE_STRING        ScopeDesc;
} MCAST_SCOPE_ENTRY, *PMCAST_SCOPE_ENTRY;
```

Members

ScopeCtx
Handle for the multicast scope, in the form of an **MCAST_SCOPE_CTX** structure.

LastAddr
Internet Protocol (IP) address of the last address in the scope, in the form of an **IPNG_ADDRESS** structure.

TTL
Time To Live (TTL) value of the scope.

ScopeDesc
Description of the scope, in user-friendly format.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

MCAST_LEASE_REQUEST

The **MCAST_LEASE_REQUEST** structure defines the request, renew, or release parameters for a given multicast scope. In the MCAST_API_VERSION_1 implementation, only one IP address may be allocated at a time.

```
typedef struct _MCAST_LEASE_REQUEST {
  time_t          LeaseStartTime;
  time_t          MaxLeaseStartTime;
  DWORD           LeaseDuration;
```

```
    DWORD            MinLeaseDuration;
    IPNG_ADDRESS     ServerAddress;
    WORD             MinAddrCount;
    WORD             AddrCount;
    PBYTE              pAddrBuf;
} MCAST_LEASE_REQUEST, *PMCAST_LEASE_REQUEST;
```

Members

LeaseStartTime

Requested start time, in seconds, for the multicast scope lease elapsed since midnight of January 1, 1970, coordinated universal time. To request the current time as the lease start time, set **LeaseStartTime** to zero.

MaxLeaseStartTime

Maximum start time, in seconds, elapsed since midnight of January 1, 1970, coordinated universal time, that the client is willing to accept.

LeaseDuration

Duration of the lease request, in seconds. To request the default lease duration, set **LeaseDuration** to zero.

MinLeaseDuration

Minimum lease duration, in seconds, that the client is willing to accept.

ServerAddress

Internet Protocol (IP) address of the server on which the lease is to be requested or renewed, in the form of an **IPNG_ADDRESS** structure. If the IP address of the server is unknown, such as when using this structure in an **McastRequestAddress** function call, set **ServerAddress** to zero.

MinAddrCount

Minimum number of IP addresses the client is willing to accept.

AddrCount

Number of requested IP addresses. Note that the value of this member dictates the size of **pAddrBuf**.

pAddrBuf

Pointer to a buffer containing the requested IP addresses. For IPv4 addresses, the **pAddrBuf** member points to 4-byte addresses; for IPv6 addresses, the **pAddrBuf** member points to 16-byte addresses. If no specific addresses are requested, set **pAddrBuf** to NULL.

Remarks

In MCAST_API_VERSION_1 version, **MaxLeaseStartTime**, **MinLeaseDuration**, and **MinAddrCount** members are ignored. Clients should still set appropriate values for these members, however, to take advantage of their implementation in future updates.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

MCAST_LEASE_RESPONSE

The **MCAST_LEASE_RESPONSE** structure is used to respond to multicast lease requests.

```
typedef struct _MCAST_LEASE_RESPONSE {
  time_t          LeaseStartTime;
  time_t          LeaseEndTime;
  IPNG_ADDRESS    ServerAddress;
  WORD            AddrCount;
  PBYTE           pAddrBuf;
} MCAST_LEASE_RESPONSE, *PMCAST_LEASE_RESPONSE;
```

Members

LeaseStartTime
Start time, in seconds, for the multicast scope lease elapsed since midnight of January 1, 1970, coordinated universal time.

LeaseEndTime
Expiration time, in seconds of the multicast scope lease elapsed since midnight of January 1, 1970, coordinated universal time.

ServerAddress
Internet Protocol (IP) address of the server on which the lease request has been granted or renewed, in the form of an **IPNG_ADDRESS** structure.

AddrCount
Number of IP addresses that are granted or renewed with the lease. Note that the value of this member dictates the size of **pAddrBuf**.

pAddrBuf
Pointer to a buffer containing the granted IP addresses. For IPv4 addresses, the **pAddrBuf** member points to 4-byte addresses; for IPv6 addresses, the **pAddrBuf** member points to 16-byte addresses.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Madcapcl.h.

CHAPTER 9

Internet Authentication Service (IAS)

IAS Overview

Internet Authentication Service (IAS) is a feature of Microsoft® Windows® that extends networking capabilities. You can use IAS to implement session control and accounting plug-ins, to add your own authorizations, and to use your own network authentication methods.

IAS is incorporated into Microsoft® Windows® 2000. IAS can also be installed into Microsoft® Windows NT® version 4.0 from the Windows NT® Option Pack or Microsoft® Commercial Internet Service (MCIS).

IAS requires Windows 2000, or Windows NT 4.0 with Service Pack 5 or higher.

The Windows 2000 version of IAS is more fully extensible than the Windows NT 4.0 version. Windows 2000 supports an additional function, **RadiusExtensionProcessEx**, and also supports authorization DLLs. These features are not available in the Windows NT 4.0 version of IAS.

This chapter describes the following topics:

- Scope
- Authentication and Accounting
- Implementing DLLs to Extend IAS
- RADIUS Accounting Packets
- Working with a State Server

Scope

You can use extensions to Internet Authentication Service (IAS) to implement the following capabilities:

- Control the number of end-user network sessions, using a state server.
- Extend the remote access authorizations currently provided by IAS (on Windows® 2000 and later versions only).

- Connect to Windows NT® Domain authentication databases and the Windows 2000 Active Directory.
- Create custom authentication methods for Windows NT version 4.0 SP5, and for Windows 2000.

Authentication and Accounting

Internet Authentication Service (IAS) fully supports, as a client and server, the Remote Authentication Dial-In User Service (RADIUS) protocol. The RADIUS protocol is the *de facto* standard for remote user authentication. (See Figure 9-1.)

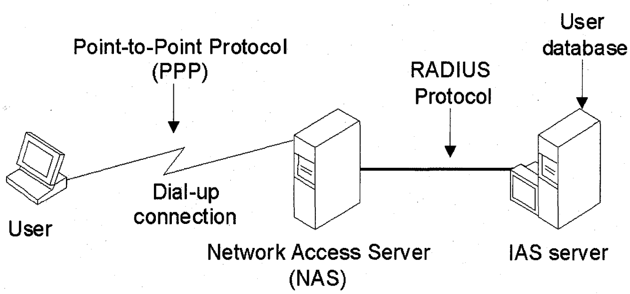

Figure 9-1: The RADIUS Protocol in Remote User Authentication.

The following paragraphs describe the roles played by the various elements of a RADIUS authentication solution.

1. A Network Access Server (NAS) operates as a client of the server that supports the RADIUS protocol. The server that supports the RADIUS protocol is generally referred to as the RADIUS server. The RADIUS client, that is, the NAS, passes user information to designated RADIUS servers, and then acts on the response that the servers return. The request sent by the client to the server in order to authenticate the user is generally called an “authentication request”.
2. The NAS also sends information to designated RADIUS servers when the user logs on and logs off. The requests sent by the client to the server to record logon/logoff and usage information are generally called “accounting requests”. The Internet Engineering Task Force (IETF) RADIUS Interim Accounting Draft also allows the NAS to send usage information on a periodic basis while the session is in progress.
3. RADIUS servers receive connection requests from remote users. For each user, the RADIUS server authenticates the user, and returns configuration information to the NAS so that it can provide network service to the user. This configuration information is composed of “authorizations”. The RADIUS server also collects a variety of information sent by the NAS that can be used for accounting and for reporting on network activity.
4. A RADIUS server can act as a proxy client to other RADIUS servers. In these cases, the RADIUS server contacted by the NAS passes the authentication request to another RADIUS server that actually performs the authentication.

While the RADIUS server is processing the authentication request, it can perform authorization functions such as verifying the user's telephone number and checking whether the user already has a session in progress. The RADIUS server can determine whether the user already has a session in progress by contacting a state server.

Implementing DLLs to Extend IAS

This section describes how to implement DLLs to extend the Internet Authentication Service (IAS). It describes the interaction between IAS and the DLLs, and presents some design considerations regarding the DLLs.

IAS provides two "plug-in" points, one for authentication and the other for authorization. Authentication refers to verifying the identity of the user. Authorization refers to determining what services the network should provide to the user. The two plug-in points correspond to Extension DLLs and Authorization DLLs. (Authorization DLLs are supported only on Windows 2000 and later systems.) Each plug-in point can support multiple DLLs.

IAS provides both authentication and authorization services. Extension DLLs are called by IAS prior to the built-in IAS authentication and authorization. Authorization DLLs are called after IAS authentication and authorization. (See Figure 9-2.)

Setting Up the Extension and Authorization DLLs

At startup, IAS checks the registry for a list of third-party DLLs to call.

To set up an Extension or Authorization DLL on an IAS server, list the paths to the DLLs in values below the following registry key:

HKLM\System\CurrentControlSet\Services\AuthSrv\Parameters

The value in which to list the Extension DLLs is:

ExtensionDLLs

The value in which to list the Authorization DLLs is:

AuthorizationDLLs

Both the ExtensionDLLs and AuthorizationDLLs values must be of type REG_MULTI_SZ. This type allows you to list multiple DLLs.

Authentication and Authorization Process

Both Extension DLLs and Authorization DLLs must export the function, **RadiusExtensionProcess**. The prototype for this function is in the Authif.h header file. IAS calls this function for each valid authentication or accounting packet that it receives from the Network Access Server (NAS). IAS calls **RadiusExtensionProcess** in each of the DLLs listed below the preceding registry key. The DLLs are called in the order in which they are listed.

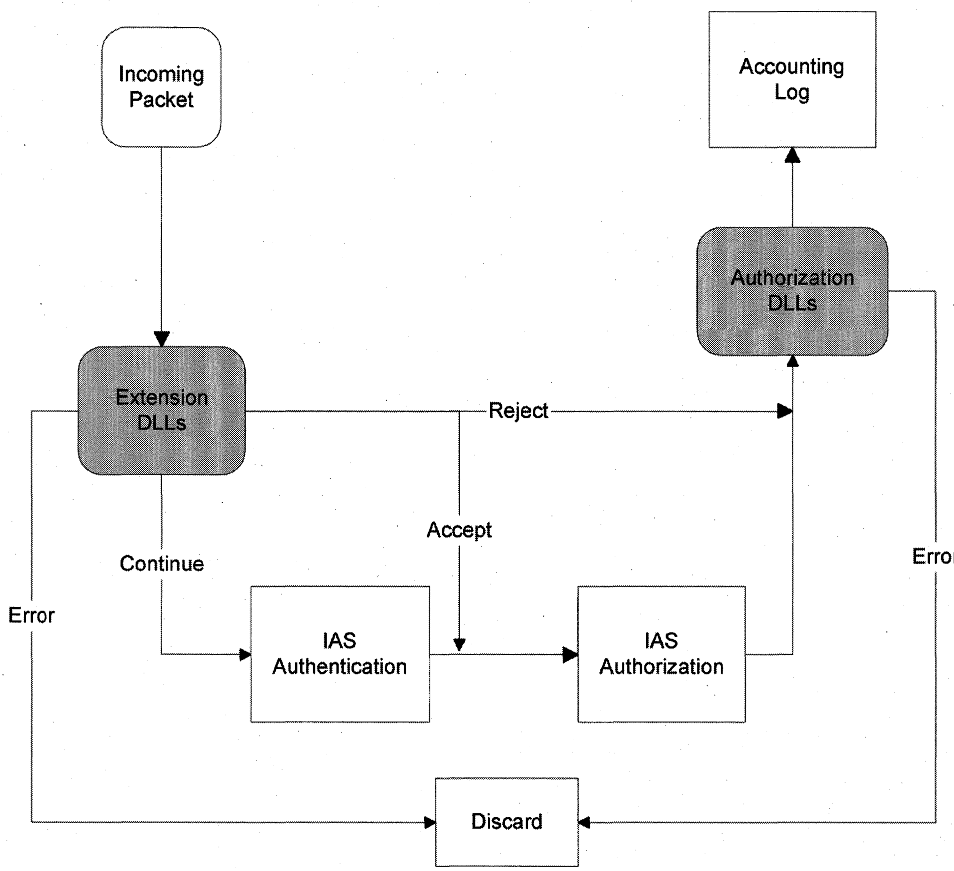

Figure 9-2: Authentication and Authorization Services in the IAS.

On Windows 2000 and later systems, the DLLs may export **RadiusExtensionProcessEx** instead of **RadiusExtensionProcess**. **RadiusExtensionProcessEx** enables the DLL to append additional authorization attributes to the authentication response. The DLL cannot, however, modify or remove any of the attributes that are already present.

If a scenario arises in which the DLL must modify or remove attributes, the only option is to use the IAS user interface to ensure that the attributes are not present. (By default, no authorization attributes will be present. Any that are present must have been added through the user interface.)

In other respects, **RadiusExtensionProcessEx** behaves the same as **RadiusExtensionProcess**. The following discussion of **RadiusExtensionProcess** is applicable to both functions.

The **RadiusExtensionProcess** function has access to all the attributes received in the authentication or accounting request. Using these attributes, the function can perform additional validations, validate the user's authorizations, or send accounting records to a central state server.

IAS will take various actions depending on the return value of **RadiusExtensionProcess**, and the value returned in the *pfAction* parameter of **RadiusExtensionProcess**:

pfAction	Extension DLL	Authorization DLL
Accept	Bypasses any further Extension DLLs and also bypasses the IAS authentication mechanism.	Accept not allowed.
Reject	Bypasses any further Extension DLLs and also bypasses the IAS authentication mechanism. Access-Reject packet is sent.	Bypasses any further Authorization DLLs.
Continue	The packet is sent to the next Extension DLL or to the IAS authentication mechanism if no more Extension DLLs are listed in the registry.	The packet is sent to the next Authorization DLL or to the IAS accounting log if no more Authorization DLLs are listed in the registry.

In the case of both Extension and Authorization DLLs, if **RadiusExtensionProcess**(**Ex**) returns an error, the packet is discarded. Packets that are discarded because of an error are not processed by the IAS accounting log.

If an error occurs, IAS posts a generic error event to the Event Log. It is recommended that the Extension or Authorization DLL provide additional error logging.

RadiusExtensionProcess(**Ex**) should return an error if it cannot reach a decision regarding the acceptance or rejection of the packet. Such a situation might arise if a network problem prevents **RadiusExtensionProcess**(**Ex**) from communicating with its user authentication database.

DLLs that process accounting packets should return either an error, or a *pfAction* of Continue.

Note Some authentication functions may also implement authorizations within them; omitting such an authentication function may cause authorizations to be omitted as well. For example, Windows NT Domain Authentication also checks some of the authorizations. If **RadiusExtensionProcess** returns an accept, it is important not to make any assumptions about the authorizations retrieved or evaluated by current or future versions of IAS. After receiving an accept, IAS does not call the remaining **RadiusExtensionProcess** DLLs in the sequence.

If a continue or accept is returned, the profile corresponding to the realm will be sent back in the Access-Accept packet.

Extension DLLs should be designed to coexist with the built-in IAS authentication providers and with other Extension DLLs. If an extension is only applicable to a certain user database (e.g., Windows NT Domain Authentication or Active Directory), then it should check the ratProvider attribute passed in through the *pAttrs* parameter, before processing the request. The ratProvider attribute would be one of a list of attributes pointed to by the *pAttrs* parameter.

Extension and Authorization DLLs should generally not reject requests simply because needed attributes are missing. For example, if an authentication extension requires the User-Password attribute (ratUserPassword), and the attribute is not present, the extension should return an action of raContinue to give other extensions and providers a chance to process the request.

IAS calls the **RadiusExtensionProcess** function after the decision to use a particular authentication database is made, but before the user is authenticated. Therefore, information about which authentication database to use is available to the function, so that the function can check for the user's authorizations in the appropriate authentication database. Windows IAS can support various authentication databases including: Windows NT Domain Authentication, and the Windows 2000 Active Directory.

If the DLL exports both **RadiusExtensionProcess** and **RadiusExtensionProcessEx**, IAS will call **RadiusExtensionProcess** Windows NT 4.0 and **RadiusExtensionProcessEx** on Windows 2000.

The DLL may also export **RadiusExtensionInit** and **RadiusExtensionTerm** functions. IAS will call these functions if they are present.

The declarations for **RadiusExtensionProcess** and other functions supported for RADIUS extension DLLs can be found in the header file Authif.h.

User Identification Attributes

The identity of the user requesting authentication is supplied to the Extension and Authorization DLLs in a number of different attributes:

- ratUserName
- ratStrippedUserName
- ratFQUserName

Each attribute provides the user identity in a different format. In general, developers should use ratStrippedUserName. The uses of the ratUserName and ratFQUserName attributes are more specialized.

ratUserName

The ratUserName attribute contains the name that was actually sent "over the wire". IAS has not in any way processed or validated the contents of this attribute. This attribute may not be available at all because the user may have been identified through a means such as caller ID. If this attribute is available, it is available only at the Extension DLL plug-in point. It is not available at the Authorization DLL plug-in point because Authorization DLLs see only the "outbound" attributes.

ratStrippedUserName

The ratStrippedUserName is the user's identity after "realm stripping". This attribute will always be present at both the Extension DLL plug-in point and the Authorization DLL plug-in point. The format of the contents of this attribute may differ between Windows NT 4.0 and Windows 2000. On Windows 2000, this attribute is guaranteed to have the format:

```
Domain\UserName
```

Where "Domain" is the NetBios domain name. On Windows NT 4.0, this attribute generally has the above format, but IAS does not guarantee it.

ratFQUserName

The ratFQUserName attribute is the "fully-qualified" user name. This name is available at both the Extension DLL plug-in point and the Authorization DLL plug-in point. However, the format of the name may differ between the two plug-in points. At the Extension DLL plug-in point, the user name will always be of the form:

```
Domain\UserName
```

The format of the name at the Authorization DLL plug-in point depends on whether the user is an Active Directory user. If the user is a local user, or a Windows NT 4.0 user, ratFQUserName will have the same format at the Authorization DLL plug-in point. If the user is an Active Directory user, ratFQUserName will contain the user's name in "canonical" format. Canonical format is the format used by the Active Directory to identify the user. It is the path from the root of the Active Directory tree, and includes the user's Organizational Unit (OU). The IAS server must be running Windows 2000 in order for ratFQUserName to be in canonical format.

RADIUS Accounting Packets

This section describes only the most important aspects of the RADIUS accounting packets. The RADIUS Accounting RFC (RFC 2139) provides detailed information on these packets.

RADIUS accounting packets can be divided into:

1. Accounting-Start packet contains userid, nas-identifier/ipaddress, plus other information received from the NAS.
2. Accounting-Stop record contains userid, nas-identifier/ipaddress, plus other information received from the NAS.
3. Accounting-On record contains nas-identifier/ipaddress record and indicates that a particular NAS has restarted.
4. Accounting-Off record contains nas-identifier/ipaddress record and indicates that a particular NAS has been shutdown.
5. Accounting-Interim record is an accounting record that could be received from the NAS. This record is sent periodically by the NAS for each user that is logged on at the NAS. This feature is generally supported in newer versions of the NASs.

The following issues are important to consider when collecting accounting information made available through RADIUS: In rare cases, records could be lost during transmission and may never reach the RADIUS server.

1. The RADIUS server is not notified if the NAS aborts.
2. If the authentication and accounting requests are received from a RADIUS Proxy, then the other ISP may not forward accounting-on, off records.
3. ISDN supports multiple sessions and each session generates an accounting start/stop pair of records. There is an accounting attribute called multi-session identifier that clearly identifies such multi-session records. Check for the multi-session identifier in addition to the session identifier to calculate the number of sessions.

Working With a State Server

Internet Authentication Service (IAS) performs authentications using a database that is configured at the IAS server site. This authentication database could be the user database for a Microsoft® Windows NT®/Windows® 2000 Domain or it could draw upon the user information obtained from the Windows 2000 Active Directory. Figure 9-3 illustrates a typical configuration that shows how IAS interacts with authentication databases such as a Windows Domain user database or Active Directory. The diagram also shows how IAS could interact with a state server that is provided by a third party.

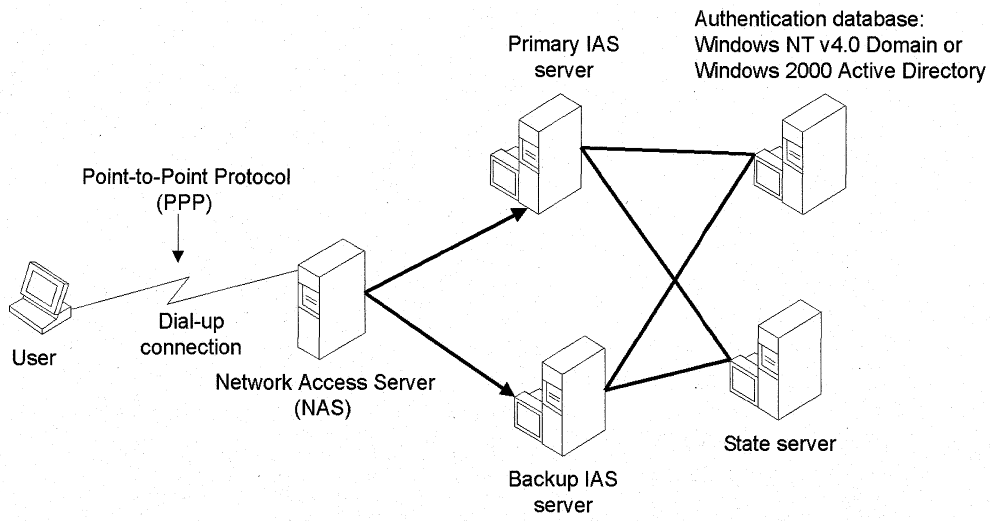

Figure 9-3: Authentications Performed Using a Database Configured at an IAS Server Site.

The primary purpose of a state server is to limit the number of simultaneous logon sessions a single user can run.

There are two points of interaction between IAS and the state server. One interaction takes place when IAS receives an authentication request from the NAS. The state server provides information from its database to determine whether to accept or deny the request. The other interaction takes place when IAS receives accounting records from the NAS. The state server uses these accounting records to update its database.

State Server Design Considerations

Depending on your design, you may need a server to track the users that are currently logged onto the network. The main challenge with the state server is maintaining accurate information about current users in the state server database. If the information in the state server is out of date, unauthorized users may succeed in having multiple sessions. Also, users who do not have multiple sessions could be inadvertently penalized. The following should be taken into consideration in implementing the state server.

1. The state server must make the decision online in a few seconds. For this reason the state server requires a scalable infrastructure that can support many updates and queries per second. Relational databases are not appropriate for such large queries with simultaneous updates. Relational databases are primarily built to keep data consistent and to provide a consistent view of the data to all consumers. They are not built for quick updates.
2. Transactional consistency on updates between multiple objects is not important. This is because the state server can tolerate a small window of opportunity. However, transactional consistency of a single update is important to reduce the chances of leaving the state server in an inconsistent state if one of the RADIUS servers is shut down in the middle of the update.
3. Persistence (saving the state of the network to persistent storage) is not important because the persistent information will quickly fall out of sync with the current state of the network.
4. If ISDN or other forms of multilink are supported on the network, the state server should be able to handle scenarios that use these features.

One possible design is to implement both an Extension DLL and an Authorization DLL. Each of these DLLs can communicate over the network with a database. The Authorization DLL can update the database with information about who is currently logged onto the network. The Extension DLL can query the database for this information to decide whether to accept or reject a particular user's authentication request; if the user is already logged on, the request is rejected.

The advantage of having the Authorization DLL update the state-server database is that the Authorization DLL has access to more information about the authenticated user. The Authorization DLL has access to all of the authorization attributes from the IAS Authorization mechanism. For example, some users may have authorizations that allow them to have multiple sessions. The state server should treat such users as a special case.

Using Internet Authentication Service

The following sample code implements the functions for a RADIUS extension DLL that checks the dial-in bit for the user.

```
///////////////////////////////////////////////////////////////////////////////
//
// Copyright (c) 1998, Microsoft Corp. All rights reserved.
//
// FILE
//
//    dialin.cpp
//
```

```
// SYNOPSIS
//
//    Implements the DLL exports for a RADIUS extension that checks the
//    dial-in bit.
//
// MODIFICATION HISTORY
//
//    09/29/1998    Original version.
//
///////////////////////////////////////////////////////////////////////

#include <windows.h>
#include <malloc.h>
#include <authif.h>
#include <rassapi.h>
#include <lmerr.h>

//////////
// The UNC name of the localhost.
//////////
WCHAR theLocalServer[UNCLEN + 1];

//////////
// Finds an attribute of the desired type.
//////////
CONST RADIUS_ATTRIBUTE*
WINAPI
FindAttribute(
    IN CONST RADIUS_ATTRIBUTE *pAttrs,
    IN DWORD dwAttrType
    ) throw ()
{
   for ( ; pAttrs->dwAttrType != ratMinimum; ++pAttrs)
   {
      if (pAttrs->dwAttrType == dwAttrType) { return pAttrs; }
   }

   return NULL;
}

//////////
// Initialize the extension.
//////////
extern "C"
```

(continued)

(continued)

```
DWORD
WINAPI
RadiusExtensionInit( VOID )
{
   // theLocalServer must be in UNC format.
   wcscpy(theLocalServer, L"\\\\");

   // Append the computer name.
   DWORD cbData = UNCLEN - 1;
   if (!GetComputerNameW(theLocalServer + 2, &cbData))
   { return GetLastError(); }

   return NO_ERROR;
}

//////////
// Process a request.
//////////
extern "C"
DWORD
WINAPI
RadiusExtensionProcess(
    IN CONST RADIUS_ATTRIBUTE *pAttrs,
    OUT OPTIONAL PRADIUS_ACTION pfAction
    )
{
   // If we can't abort the request, we're not interested.
   if (pfAction == NULL) { return NO_ERROR; }

   // If it's not destined for the Windows NT provider, we're not interested.
   CONST RADIUS_ATTRIBUTE* p = FindAttribute(pAttrs, ratProvider);
   if (p == NULL || p->dwValue != rapWindowsNT) { return NO_ERROR; }

   // Find the Stripped-User-Name attribute.
   p = FindAttribute(pAttrs, ratStrippedUserName);

   // Make sure we found an attribute and it's the right data type.
   if (p == NULL || p->fDataType != rdtString) { return ERROR_INVALID_DATA; }

   // Allocate a buffer for the UNICODE string.
   PWSTR domain = (PWSTR)_alloca((p->cbDataLength + 1) * sizeof(WCHAR));

   // Convert to UNICODE.
   int nChar = MultiByteToWideChar(
```

```
                 CP_ACP,
                 0,
                 p->lpValue,
                 p->cbDataLength,
                 domain,
                 p->cbDataLength + 1
                 );
if (nChar == 0) { return GetLastError(); }
domain[nChar] = L'\0';

// Look for the delimiter in "domain\username".
PWSTR username = wcschr(domain, L'\\');

DWORD error;
WCHAR buffer[UNCLEN + 1], *accountServer = theLocalServer;

if (username)
{
   // We found a delimiter, so null it out and advance.
   *username++ = L'\0';

   // Is this the local computer?
   if (_wcsicmp(domain, theLocalServer + 2))
   {
      // No, so we have to find a DC.
      error = RasAdminGetUserAccountServer(
                 domain,
                 NULL,
                 buffer
                 );
      if (error != NO_ERROR) { return error; }

      accountServer = buffer;
   }
}
else
{
   // No delimiter, so the whole thing is the user name.
   username = domain;
}

// Retrieve the dial-in information for this user.
RAS_USER_0 ru0;
error = RasAdminUserGetInfo(
```

(continued)

(continued)

```
                accountServer,
                username,
                &ru0
                );

    // If the user doesn't exist, it's not an error,
    // but we always reject.
    if (error == NERR_UserNotFound)
    {
       *pfAction = raReject;
       return NO_ERROR;
    }

    // Any other error and we bail.
    if (error != NO_ERROR) { return error; }

    // Is the dial-in bit set?
    if ((ru0.bfPrivilege & RASPRIV_DialinPrivilege) == 0)
    {
       *pfAction = raReject;
    }

    return NO_ERROR;
}
```

Internet Authentication Service Reference

The following section describes the functions, structures, enumeration types to use when implementing RADIUS extension DLLs for Internet Authentication Service (IAS).

- Internet Authentication Service Functions
- Internet Authentication Service Structures
- Internet Authentication Service Enumerated Types

Internet Authentication Service Functions

An architecture for RADIUS extension DLLs supports the following exported functions:

RadiusExtensionInit
RadiusExtensionTerm
RadiusExtensionProcess
RadiusExtensionProcessEx

The **RadiusExtensionInit** and **RadiusExtensionTerm** functions are optional. However, the extension DLL must export either **RadiusExtensionProcess** or **RadiusExtensionProcessEx**.

RadiusExtensionInit

The **RadiusExtensionInit** function is called by IAS while the service is starting up. Use **RadiusExtensionInit** to perform any initialization operations for the extension DLL.

```
typedef DWORD ( WINAPI * RadiusExtensionInit )( VOID );
```

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value should be an appropriate error code from Winerror.h.

Remarks

A return value other then NO_ERROR will cause IAS to fail to start.

RadiusExtensionInit is an optional function. The RADIUS extension DLL need not implement **RadiusExtensionInit**.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in AuthIf.h.

See Also

About Internet Authentication Service Overview, Internet Authentication Service Functions, **RadiusExtensionTerm**

RadiusExtensionTerm

The **RadiusExtensionTerm** function is called by IAS prior to unloading the extension DLL. Use **RadiusExtensionTerm** to perform any clean-up operations for the extension DLL.

```
typedef VOID ( WINAPI * RadiusExtensionTerm )( VOID );
```

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value should be an appropriate error code from Winerror.h.

Remarks

RadiusExtensionTerm is an optional function. The RADIUS extension DLL need not implement **RadiusExtensionTerm**.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in AuthIf.h.

See Also

About Internet Authentication Service Overview, Internet Authentication Service Functions, **RadiusExtensionInit**

RadiusExtensionProcess

The **RadiusExtensionProcess** function is called by IAS for each authentication or accounting packet that IAS receives from the NAS.

```
typedef DWORD (WINAPI * RadiusExtensionProcess )(
  RADIUS_ATTRIBUTE  * pAttrs,    // pointer to array of
                                 // attributes
  PRADIUS_ACTION      pfAction   // action that IAS should
                                 // take
  );
```

Parameters

pAttrs
: Pointer to an array of attributes from the request. The array is terminated by an attribute with **dwAttrType** set to ratMinimum. These attributes should be treated as read-only; they should not be modified by **RadiusExtensionProcess**. Also, these attributes should not be referenced in any way after **RadiusExtensionProcess** returns.

pfAction
: Pointer to a value of type **RADIUS_ACTION**. This parameter specifies the action that IAS should take in response to an Access-Request. If the request is not an access request, **RadiusExtensionProcess** should return NULL for this parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value should be an appropriate error code from Winerror.h.

Remarks

If the return value is anything other than NO_ERROR, IAS discards the request.

IAS supports multiple extension DLLs. IAS calls **RadiusExtensionProcess** for each of the DLLs listed in the registry.

On Windows 2000, the extension DLL may export **RadiusExtensionProcessEx** instead of **RadiusExtensionProcess**.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in AuthIf.h.

See Also

About Internet Authentication Service Overview, Internet Authentication Service Functions, **RADIUS_ACTION**, **RADIUS_ATTRIBUTE**, **RADIUS_ATTRIBUTE_TYPE**, **RadiusExtensionProcessEx**

RadiusExtensionProcessEx

The **RadiusExtensionProcessEx** function is called by IAS for each authentication or accounting packet that IAS receives from the NAS. This function is similar to **RadiusExtensionProcess**. However, **RadiusExtensionProcessEx** enables the extension DLL to append attributes to the authentication response.

```
typedef DWORD (WINAPI * RadiusExtensionProcessEx )(
  CONST RADIUS_ATTRIBUTE    * pInAttrs,
          // pointer to array of input attributes
  RADIUS_ATTRIBUTE          * pOutAttrs,
          // pointer to array of output attributes
  PRADIUS_ACTION              pfAction
          // action that IAS should take
  );
```

Parameters

pInAttrs
: Pointer to an array of attributes from the request. The array is terminated by an attribute with **dwAttrType** set to ratMinimum. These attributes should be treated as read-only; they should not be modified by **RadiusExtensionProcessEx**. Also, these attributes should not be referenced in any way after **RadiusExtensionProcessEx** returns.

pOutAttrs

Pointer to an array of attributes from the request. The array is terminated by an attribute with **dwAttrType** set to ratMinimum. Internet Authentication Service adds these attributes to the authentication response.

pfAction

Pointer to a value of type **RADIUS_ACTION**. This parameter specifies the action that IAS should take in response to an Access-Request. If the request is not an access request, **RadiusExtensionProcessEx** should return NULL for this parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value should be an appropriate error code from Winerror.h.

Remarks

If the return value is anything other than NO_ERROR, IAS discards the request.

IAS supports multiple extension DLLs. IAS calls **RadiusExtensionProcessEx** for each of the DLLs listed in the registry.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Authlf.h.

See Also

About Internet Authentication Service Overview, Internet Authentication Service Functions, **RADIUS_ACTION**, **RADIUS_ATTRIBUTE**, **RADIUS_ATTRIBUTE_TYPE**, **RadiusExtensionProcess**

Internet Authentication Service Structures

Use the **RADIUS_ATTRIBUTE** structure to represent a RADIUS attribute or an extended attribute when developing RADIUS extension DLLs.

RADIUS_ATTRIBUTE

The **RADIUS_ATTRIBUTE** structure represents a RADIUS attribute or an extended attribute.

```
typedef struct _RADIUS_ATTRIBUTE {
  DWORD             dwAttrType;    // attribute type
  RADIUS_DATA_TYPE  fDataType;     // type of value
```

```
  DWORD            cbDataLength;  // length of value
  union {
    DWORD   dwValue;  // for rdtAddress, rdtInteger,
                      // and rdtTime
    PCSTR   lpValue;  // for rdtUnknown, and rdtString
  };
} RADIUS_ATTRIBUTE, *PRADIUS_ATTRIBUTE;
```

Members

dwAttrType
: Stores a value from the **RADIUS_ATTRIBUTE_TYPE** enumeration. This value specifies the type of the attribute represented by the **RADIUS_ATTRIBUTE** structure.

fDataType
: Stores a value from the **RADIUS_DATA_TYPE** enumeration. This value specifies the type of the value stored in the union containing the **dwValue** and **lpValue** members.

cbDataLength
: Stores the length, in bytes, of the data. The **cbDataLength** member is used only if **lpValue** member is used.

dwValue
: Stores a value of type **DWORD**. The **dwValue** member is used if the **fDataType** member specifies rdtAddress, rdtInteger or rdtTime.

lpValue
: Stores a pointer to a multi-byte data value. The **lpValue** member is used if the **fDataType** member specifies rdtUnknown or rdtString.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in AuthIf.h.

See Also

About Internet Authentication Service Overview, Internet Authentication Service Structures, **RADIUS_ATTRIBUTE_TYPE**, **RADIUS_DATA_TYPE**

Internet Authentication Service Enumerated Types

Use the following enumerated types when developing RADIUS extension DLLs:

RADIUS_ACTION

RADIUS_ATTRIBUTE_TYPE

RADIUS_AUTHENTICATION_PROVIDER

RADIUS_DATA_TYPE

RADIUS_ACTION

The **RADIUS_ACTION** type enumerates the responses that a RADIUS extension DLL can generate in response to an Access-Request.

```
typedef enum _RADIUS_ACTION {
  raContinue,
  raReject,
  raAccept
} RADIUS_ACTION, *PRADIUS_ACTION;
```

Values

raContinue
IAS continues to process the request. IAS also continues to call **RadiusExtensionProcess** in other extension DLLs.

raReject
Return an Access-Reject packet. The Access-Request is declined. In this case, IAS does not call **RadiusExtensionProcess** in any other extension DLLs.

raAccept
IAS accepts the Access-Request. IAS does not continue to call **RadiusExtensionProcess** in this case. However, it does continue to obtain authorizations for the user requesting access.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in AuthIf.h.

See Also

About Internet Authentication Service Overview, Internet Authentication Service Enumerated Types, **RadiusExtensionProcess**

RADIUS_ATTRIBUTE_TYPE

The **RADIUS_ATTRIBUTE_TYPE** type enumerates the possible types for a RADIUS attribute.

```
typedef enum _RADIUS_ATTRIBUTE_TYPE {
  ratMinimum = 0,      // Used to terminate attribute arrays.
  // RADIUS standard attributes.
  ratUserName = 1,
  ratUserPassword = 2,
```

```
ratCHAPPassword = 3,
ratNASIPAddress = 4,
ratNASPort = 5,
ratServiceType = 6,
ratFramedProtocol = 7,
ratFramedIPAddress = 8,
ratFramedIPNetmask = 9,
ratFramedRouting = 10,
ratFilterId = 11,
ratFramedMTU = 12,
ratFramedCompression = 13,
ratLoginIPHost = 14,
ratLoginService = 15,
ratLoginPort = 16,
ratReplyMessage = 18,
ratCallbackNumber = 19,
ratCallbackId = 20,
ratFramedRoute = 22,
ratFramedIPXNetwork = 23,
ratState = 24,
ratClass = 25,
ratVendorSpecific = 26,
ratSessionTimeout = 27,
ratIdleTimeout = 28,
ratTerminationAction = 29,
ratCalledStationId = 30,
ratCallingStationId = 31,
ratNASIdentifier = 32,
ratProxyState = 33,
ratLoginLATService = 34,
ratLoginLATNode = 35,
ratLoginLATGroup = 36,
ratFramedAppleTalkLink = 37,
ratFramedAppleTalkNetwork = 38,
ratFramedAppleTalkZone = 39,
ratAcctStatusType = 40,
ratAcctDelayTime = 41,
ratAcctInputOctets = 42,
ratAcctOutputOctets = 43,
ratAcctSessionId = 44,
ratAcctAuthentic = 45,
ratAcctSessionTime = 46,
ratAcctInputPackets = 47,
ratAcctOutputPackets = 48,
```

(continued)

(continued)

```
    ratAcctTerminationCause = 49,
    ratCHAPChallenge = 60,
    ratNASPortType = 61,
    ratPortLimit = 62,
    // Extended attribute types used to
    // pass additional information.
    ratCode = 262,              /* Request type code. */
    ratIdentifier = 263,        /* Request identifier. */
    ratAuthenticator = 264,     /* Request authenticator. */
    ratSrcIPAddress = 265,      /* Source IP address. */
    ratSrcPort = 266,           /* Source IP port. */
    ratProvider = 267,          /* Authentication provider. */
    ratStrippedUserName = 268   /* User-Name with realm stripped. */
    ratFQUserName = 269,        /* Fully-Qualified-User-Name */
    ratPolicyName = 270         /* Remote Access Policy name */

} RADIUS_ATTRIBUTE_TYPE;
```

Values

ratMinimum

This value is equal to zero, and used as the NULL terminator in any array of **RADIUS_ATTRIBUTE** structures.

ratUserName

Specifies the name of the user to be authenticated. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information. Also see *User Identification Attributes*.

ratUserPassword

Specifies the password of the user to be authenticated. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratCHAPPassword

Specifies the password provided by the user in response to an Challenge Handshake Authentication Protocol (CHAP) challenge. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratNASIPAddress

Specifies the NAS IP address. An Access-Request should specify either an NAS IP address or an NAS identifier. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratNASPort

Identifies the physical or virtual private network (VPN) through which the user is connecting to the NAS. Note that this value is not a port number in the sense of TCP or UDP. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratServiceType

Specifies the type of service the user has requested or the type of service to be provided. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratFramedProtocol

Specifies the type of framed protocol to use for framed access, for example SLIP, PPP, or ARAP (AppleTalk Remote Access Protocol). The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratFramedIPAddress

Specifies the IP address that will be configured for the user requesting authentication. This attribute is typically returned by the authentication provider. However, the NAS may use it in an authentication request to specify a preferred IP address. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratFramedIPNetmask

Specifies the IP network mask for a user that is a router to a network. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratFramedRouting

Specifies the routing method for a user that is a router to a network. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratFilterId

Identifies the filter list for the user requesting authentication. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratFramedMTU

Specifies the Maximum Transmission Unit (MTU) for the user. This attribute is used in cases where the MTU is not negotiated through some other means, such as PPP. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratFramedCompression

Specifies a compression protocol to use for the connection. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. *RFC 2138*.

ratLoginIPHost

Specifies the system with which to connect the user. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratLoginService

Specifies the service to use to connect the user to the host specified by *ratLoginIPHost*. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratLoginPort
Specifies the port to which to connect the user. This attribute is present only if the *ratLoginService* attribute is present. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratReplyMessage
Specifies a message to display to the user. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratCallbackNumber
Specifies a callback number. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratCallbackId
Identifies a location to callback. The value of this attribute is interpreted by the NAS. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratFramedRoute
Provides routing information to configure on the NAS for the user. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratFramedIPXNetwork
Specifies the IPX network number to configure for the user. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratState
Please refer to *RFC 2138* for detailed information about this value. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer.

ratClass
Specifies a value that is provided to the NAS by the authentication provider. The NAS should use this value when communicating with the accounting provider. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratVendorSpecific
Allows vendors to provide their own extended attributes. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratSessionTimeout
Specifies the maximum number of seconds for which to provide service to the user. After this time, the session is terminated. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratIdleTimeout
Specifies the maximum number of consecutive seconds the session can be idle. If the idle time exceeds this value, the session is terminated. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratTerminationAction
Please refer to the above-referenced files at ds.internic.net for detailed information about this value. The value field in **RADIUS_ATTRIBUTE** for this type is 32-bit integral value. See *RFC 2138* for more information.

ratCalledStationId
Specifies the number that the user dialed to connect to the NAS. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratCallingStationId
Specifies the number from which the user is calling. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratNASIdentifier
Specifies the NAS identifier. An Access-Request should specify either an NAS identifier or an NAS IP address. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratProxyState
Specifies a value that a proxy server includes when forwarding an authentication request. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratLoginLATService
This attribute is not currently used for authentication on Windows 2000. See *RFC 2138* for more information.

ratLoginLATNode
This attribute is not currently used for authentication on Windows 2000. See *RFC 2138* for more information.

ratLoginLATGroup
This attribute is not currently used for authentication on Windows 2000. See *RFC 2138* for more information.

ratFramedAppleTalkLink
Specifies the AppleTalk network number for a user that is another router. The value field in **RADIUS_ATTRIBUTE** for this type is 32-bit integral value. See *RFC 2138* for more information.

ratFramedAppleTalkNetwork
Specifies the AppleTalk network number that the NAS should use to allocate an AppleTalk node for the user. This attribute is used only when the user is not another router. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratFramedAppleTalkZone
Specifies the AppleTalk default zone for the user. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratAcctStatusType
Specifies whether the accounting provider should start or stop accounting for the user. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2139* for more information.

ratAcctDelayTime
Specifies the length of time that the client has been attempting to send the current request. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2139* for more information.

ratAcctInputOctets
Specifies the number of octets that have been received during the current accounting session. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2139* for more information.

ratAcctOutputOctets
Specifies the number of octets sent during the current accounting session. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2139* for more information.

ratAcctSessionId
Specifies a value to enable the identification of matching start and stop records within a log file. The start and stop records are sent in the *ratAcctStatusType* attribute. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2139* for more information.

ratAcctAuthentic
Specifies, to the accounting provider, how the user was authenticated; for example by Windows 2000 Directory Services, RADIUS, or some other authentication provider. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2139* for more information.

ratAcctSessionTime
Specifies the number of seconds that have elapsed in the current accounting session. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2139* for more information.

ratAcctInputPackets
Specifies the number of packets that have been received during the current accounting session. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2139* for more information.

ratAcctOutputPackets
Specifies the number of packets that have been sent during the current accounting session. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2139* for more information.

ratAcctTerminationCause
Specifies how the current accounting session was terminated. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2139* for more information.

ratCHAPChallenge
Specifies the CHAP challenge sent by the NAS to a CHAP user. The value field in **RADIUS_ATTRIBUTE** for this type is a pointer. See *RFC 2138* for more information.

ratNASPortType
Specifies the type of the port through which the user is connecting, for example, asynchronous, ISDN, virtual. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratPortLimit
Specifies the number of ports the NAS should make available to the user for multilink sessions. The value field in **RADIUS_ATTRIBUTE** for this type is a 32-bit integral value. See *RFC 2138* for more information.

ratCode
Specifies the request type code. This is an extended attribute.

ratIdentifier
Specifies the request identifier. This is an extended attribute.

ratAuthenticator
Specifies the request authenticator. This is an extended attribute.

ratSrcIPAddress
Specifies the source IP address. This is an extended attribute.

ratSrcPort
Specifies the source IP port. This is an extended attribute.

ratProvider
Specifies the authentication provider. The value for this attribute is taken from the **RADIUS_AUTHENTICATION_PROVIDER** enumerated type. This is an extended attribute.

ratStrippedUserName
Specifies the user name with the realm removed. See *User Identification Attributes* for more information. This is an extended attribute.

ratFQUserName
Specifies the fully-qualified user name. See *User Identification Attributes* for more information. This is an extended attribute.

ratPolicyName
Specifies the policy name. This is an extended attribute.

Remarks

The value for an attribute of type **ratProvider** is taken from the **RADIUS_AUTHENTICATION_PROVIDER** enumerated type.

! Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in AuthIf.h.

See Also

About Internet Authentication Service Overview, Internet Authentication Service Enumerated Types, **RADIUS_ATTRIBUTE**, **RADIUS_AUTHENTICATION_PROVIDER**

RADIUS_AUTHENTICATION_PROVIDER

The **RADIUS_AUTHENTICATION_PROVIDER** type enumerates the possible authentication providers that Internet Authentication Service can use.

```
typedef enum _RADIUS_AUTHENTICATION_PROVIDER {
  rapUnknown,
  rapUsersFile,
  rapProxy,
  rapWindowsNT,
  rapMCIS,
  rapODBC
} RADIUS_AUTHENTICATION_PROVIDER;
```

Values

rapUnknown
The authentication provider is unknown.

rapUsersFile
A users' file is providing the authentication information.

rapProxy
Authentication is provided by a RADIUS proxy server.

rapWindowsNT
Authentication is provided by Window 2000 Domain Authentication.

rapMCIS
Authentication is provided by a Microsoft Commercial Internet System (MCIS) database.

rapODBC
Authentication is provided by an Open Database Connectivity (ODBC) compliant database.

Remarks

The ratProvider extended attribute in **RADIUS_ATTRIBUTE_TYPE** uses values from the **RADIUS_AUTHENTICATION_PROVIDER** enumeration type.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in AuthIf.h.

See Also

About Internet Authentication Service Overview, Internet Authentication Service Enumerated Types, **RADIUS_ATTRIBUTE**, **RADIUS_ATTRIBUTE_TYPE**

RADIUS_DATA_TYPE

The **RADIUS_DATA_TYPE** type enumerates the possible data type for a RADIUS attribute or extended attribute.

```
typedef enum _RADIUS_DATA_TYPE {
  rdtUnknown,
  rdtString,
  rdtAddress,
  rdtInteger,
  rdtTime
} RADIUS_DATA_TYPE;
```

Values

rdtUnknown
The value is a pointer. However, the attribute is not recognized by the dictionary.

rdtString
The value of attribute is a pointer to a character string.

rdtAddress
The value of the attribute is a 32-bit **DWORD** value representing address.

rdtInteger
The value of the attribute is a 32-bit **DWORD** value representing an integer.

rdtTime
The value of the attribute is a 32-bit **DWORD** value representing a time.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in AuthIf.h.

See Also

About Internet Authentication Service Overview, Internet Authentication Service Enumerated Types, **RADIUS_ATTRIBUTE**, **RADIUS_ATTRIBUTE_TYPE**

CHAPTER 10

The NetBIOS Interface

A Win32-based application can use the Network Basic Input/Output System (NetBIOS) interface to communicate with applications on other computers in a network. The NetBIOS interface provides commands and support for the following services:

- Network name registration and verification
- Session establishment and termination
- Reliable connection-oriented data transfer
- Unreliable connectionless data transfer (datagram)
- Protocol and adapter monitoring and management

The NetBIOS interface exposes an explicit set of commands that are submitted through a structure known as the *Network Control Block* (NCB). An application can issue NetBIOS commands over any protocol that supports the NetBIOS interface.

NetBIOS Interface Overview

The NetBIOS interface is provided primarily for existing applications that use IBM NetBIOS 3.0 and need to be ported to the Win32 API. New applications and applications not requiring compatibility with NetBIOS should use other interfaces, such as mailslots, named pipes, RPC, sockets, or distributed COM to accomplish tasks similar to those supported by NetBIOS. These interfaces are more flexible and portable than NetBIOS. In addition, you can use sockets over NetBIOS to communicate with NetBIOS applications.

The **Netbios** function takes one parameter, a pointer to a structure describing the NCB. The **NCB** structure contains information about the command to perform, an optional post routine, an optional event handle, and a pointer to a buffer that is used for messages or other data.

This overview discusses the following topics:

- NetBIOS Operation
- NetBIOS LANA Numbers
- NetBIOS Name Table
- NetBIOS Session
- NetBIOS Enhancements
- NetBIOS Commands

The NetBIOS Requests for Comments (RFC) are 1001, 1002, and 1088. For more information on NetBIOS 3.0, contact IBM and order the *IBM Local Area Network Technical Reference: IEEE 802.2 and NETBIOS Application Programming Interfaces.*

NetBIOS Operation

Protocol drivers expose the NetBIOS interface and map NetBIOS commands to their own native commands. The NetBIOS Frames protocol (NBFP) can be implemented by the underlying protocol software to perform the network I/O required by the NetBIOS interface.

The NetBIOS emulator accepts NetBIOS commands, translates them to Transport Driver Interface (TDI) calls, and forwards them to the transport driver using the TDI interface. NetBIOS emulation requires some functionality that is not required for all TDI drivers. A TDI driver that provides this functionality is called a NetBIOS-compatible TDI driver.

Figure 10-1 shows how NetBIOS works on Windows NT/Windows 2000.

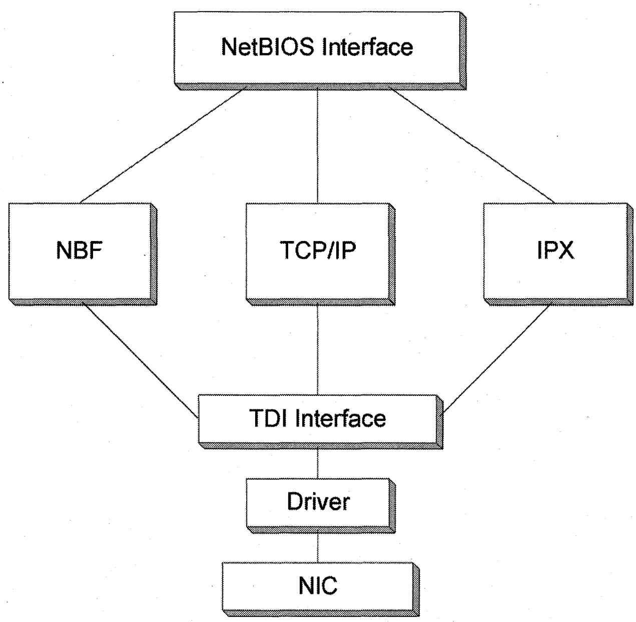

Figure 10-1: NetBIOS in Windows 2000/Windows NT.

The NetBIOS over TCP/IP (NetBT or NBT) protocol provides NetBIOS support for the TCP/IP protocol. It is defined by RFCs 1001 and 1002. The NetBIOS over NetBEUI protocol provides NetBIOS support for the NetBEUI protocol. This protocol is also called NetBIOS Frames (NBF). The NetBIOS over IPX (NBIPX) protocol provides IPX support. This protocol is also called NetBIOS on NetWare (NWNBLINK). It is based on Novell's NetBIOS.

For more information about the protocol and driver layers and the TDI interface, see the *Device Development Kit* (DDK) documentation.

NetBIOS LANA Numbers

The NetBIOS *LANA number* identifies the transport driver, network interface card (NIC) driver, and adapter that will be used to send and receive NetBIOS packets. This is known as the *network route*. The following example shows the network route that will be used when you specify a LANA value of 1:

001 NetBT -> IEEPRO -> IEEPRO1

Specify the LANA number in the **ncb_lana_num** member of the **NCB** structure when you issue a NetBIOS command.

The IBM NetBIOS 3.0 specification supports only two LANA numbers, because NetBEUI was originally the only protocol that supported NetBIOS, and a computer could contain only two network adapters at that time. Specifying LANA 0 directed a command to the first adapter, and specifying LANA 1 directed a command to the second adapter. Because many computers had only one network adapter, many MS-DOS-based applications sent all their requests to LANA 0. If a second network adapter was installed, some applications allowed the user to specify the use of LANA 1 instead. As a result, LANA 0 became the default setting, though it was never intended as such.

Windows NT/Windows 2000 enables NetBIOS to use transport protocols other than NetBEUI. Therefore, Microsoft has extended the meaning of a LANA number to indicate a specific transport protocol on a specific adapter. For example, if you have two network adapters, and have three transport protocols installed, you have six LANA numbers. The LANA numbers are not necessarily sequential.

In addition to extending the meaning of a LANA number, Microsoft also added the NCBENUM command to enumerate the available LANA numbers. As an example, the **LANA_ENUM** structure filled by NCBENUM might hold an array with values 0, 3, 5, and 6. Zero might map to IPX/SPX on the first adapter, three might map to NETBEUI on a second adapter, and so on.

Windows NT/2000: You can associate specific LANA numbers with specific network routes using the following steps.

▶ To associate a LANA number with a network route

1. Start the Network Control Panel application.
2. Click the **Services** tab.
3. Double-click **NetBIOS Interface**.
4. Click the LANA number you want to change.
5. Enter the new LANA number to be associated with the network route.

Windows 95/98: You cannot configure LANA numbers because of the way plug and play was designed. LANA numbers can change as users install plug and play devices. You may set only LANA 0, which is the default protocol. The next protocol is LANA 7, then LANA 6, and so on. If no protocol is set as the default, there may not be a LANA 0.

You can set the default protocol in the control panel using the following steps:

▶ To set LANA 0 on Windows 95/98

1. Start the Network Control Panel application.
2. Choose the protocol you want as the default.
3. Click **Properties**.
4. Click the **Advanced** tab.
5. Click **Set this protocol to be the default protocol**.

The best way to write a NetBIOS application is to support all LANA numbers, and establish connections over any LANA number. This allows your application to transparently support any transport protocol that supports NetBIOS, as well as dynamic LANA numbers associated with dial-up adapters or plug-and-play hardware. A good approach is outlined in the following steps.

▶ To support connections over any LANA

1. Enumerate the LANA numbers by issuing an NCBENUM command.
2. Reset each LANA by issuing one NCBRESET command per LANA number.
3. Add your local NetBIOS name to each LANA. The name may be the same on each LANA.
4. Connect using any LANA number:
   - For servers, issue an NCBLISTEN command on each LANA. If necessary, cancel any outstanding listen operation after the first listen operation has been completed.
   - For clients, issue an NCBFINDNAME (Windows NT/Windows 2000 only) or an NCBCALL (Windows NT/Windows 2000 or Windows 95/98) command on each LANA. The first successful NCBFINDNAME or NCBCALL operation will indicate which LANA to use. When using NCBCALL instead of NCBFINDNAME, you must cancel any pending NCBCALL commands and hang up any extra completed calls.

Though this is the best technique for writing a NetBIOS application, it generates several datagrams, making the NetBIOS interface less desirable than other networking interfaces.

NetBIOS Name Table

NetBIOS *names* are used as the basis for communication between applications. NetBIOS maintains a *name table* for each LANA for each process that contains the names by which the process is known on the network. These names are used when forming connections between processes.

Names are provided to NetBIOS by the application through the **ncb_name** member of the **NCB** structure. A name can be a *unique name* or a *group name*. NetBIOS checks the network to verify that a unique name is not already in use by another adapter. A group name can be used by several adapters.

NetBIOS names can be up to 16 characters long. This is large enough to accommodate a text version of the media access control (MAC) address, plus a few other characters. Using the MAC address in this way results in a name that is guaranteed unique on any network. The name NAME_NUMBER_1 is always present and has the value 1.

For more information, see *Name Support*.

NetBIOS Session

A NetBIOS *session* is a logical connection between any two processes on the network. The NetBIOS *local session number* identifies the virtual circuit established between two processes. Local session numbers are provided to applications by NetBIOS through the **ncb_lsn** member of the **NCB** structure.

To establish a session, have one process issue an NCBLISTEN command, and have the other process issue an NCBCALL command. After a session is established, the computers can exchange data using NetBIOS commands.

Each process can establish 254 sessions per LANA number. Because names are maintained on a per-process basis, the NetBIOS emulator can identify two separate sessions even if they have the same local session number in their respective processes. The emulator maps each local session number to a unique TDI connection endpoint handle.

For more information, see *Session Support*.

NetBIOS Enhancements

The Win32 implementation of NetBIOS is based on the NetBIOS 3.0 specification. However, the Win32 implementation includes the following enhancements that are not part of the NetBIOS 3.0 specification:

- NetBIOS emulator manages resources separately per process. For example, the network name numbers are assigned on a per-process basis. Therefore, if a process issues the NCBRESET command, the names, sessions, and outstanding NCBs allocated for that process are cleared, but those of other processes are not affected. Also, requests for the status of the local adapter retrieve only the names that were added by the process making the request.
- The NetBIOS emulator supports 254 sessions per process, per LANA number.
- You can supply an event in the **ncb_event** member of the **NCB** structure. This is not a standard NetBIOS 3.0 NCB member. The event is set to the signaled state when NetBIOS completes an asynchronous command. This method of issuing asynchronous commands is faster than using post routines and it uses fewer system resources. The system creates an event and a worker thread for each command that uses a post routine.

- A process can enable extensions to the transport interface by using the NCBACTION command in the **ncb_command** member of the **NCB** structure. This is not a standard NetBIOS 3.0 command.
- A process can enumerate all available LAN adapters by using the NCBENUM command in the **ncb_command** member of the **NCB** structure. This is not a standard NetBIOS 3.0 command.
- A process is required to issue the NCBRESET command on each LANA number before it can issue any other NetBIOS command on the LANA number, with the exception of NCBENUM.
- The NetBIOS 3.0 specification allows LANA numbers 0 and 1. Win32 allows additional LANA numbers.
- The value 1 for the **ncb_num** member of the **NCB** structure is not exclusive when the NCBRESET command is issued. All MS-DOS and 16-bit Windows-based applications also share access to NAME_NUMBER_1.

NetBIOS Commands

A NetBIOS application issues commands to an underlying transport driver using the **Netbios** function. You provide information in the **NCB** structure. The appropriate transport driver receives the information, performs the command, and reports status by filling in selected **NCB** members.

When a command is issued synchronously, **Netbios** does not return until the protocol driver completes the command. Both the **ncb_retcode** and **ncb_cmd_cplt** members contain the return value.

When a command is issued asynchronously, **Netbios** does not return until the protocol driver checks the command syntax, checks whether the session is valid, and checks whether there are sufficient available resources. Both the **ncb_retcode** and **ncb_cmd_cplt** members contain the return value. The return value is an error code if the command was not successfully queued. A return value of NRC_PENDING indicates that the protocol driver has successfully queued the command. When the protocol driver completes the command, it places the final return value in both the **ncb_retcode** and **ncb_cmd_cplt** members. If the **ncb_post** member specifies a post routine, the protocol driver calls the post routine.

Post routines are called in the context of the calling process. The post routine in a Win32-based application typically posts a message to an appropriate window and then exits. The thread that receives the message uses it as an indication that the asynchronous command has been completed. The system creates an event and a worker thread for each command that uses a post routine. As a result, it is faster to use the **ncb_event** member instead of a post routine for asynchronous commands.

The NetBIOS commands can be divided into the following categories:

- Name Support
- Session Support
- Data-Transfer Support
- Datagram Support
- General Support
- Extension Support

Name Support

The following are the name support commands:

- NCBADDGRNAME (add group name)
- NCBADDNAME (add name)
- NCBDELNAME (delete name)
- NCBFINDNAME (find name)

The NCBADDGRNAME and NCBADDNAME commands register names. The transport driver verifies the name, registers it in the name table, and returns a corresponding *name number* in the **ncb_num** member of the **NCB** structure. The registration process is shown in Figure 10-2. The names "station1" and "station2" will be used in subsequent examples.

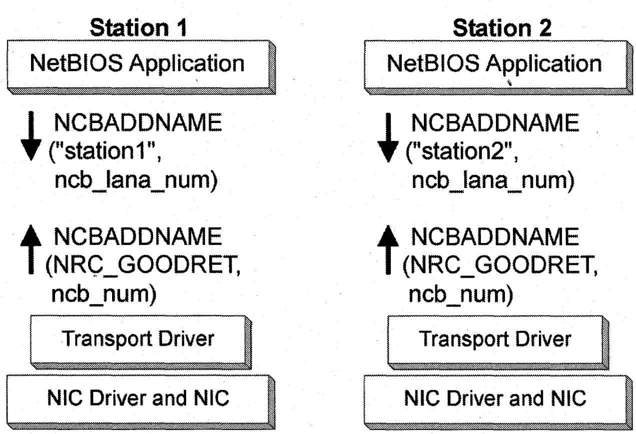

Figure 10-2: Registering Names with the Name Support Commands.

The NCBFINDNAME command obtains the MAC header information for the computer that has registered the specified name. The protocol driver queries the network for the specified name. If any computers have registered the name, they respond with an indication of whether the name is a unique name or a group name. Multiple computers may respond. The **ncb_buffer** member receives a **FIND_NAME_HEADER** structure, followed by one or more **FIND_NAME_BUFFER** structures. The length of the buffer is returned in the **ncb_length** member. This process is shown in Figure 10-3.

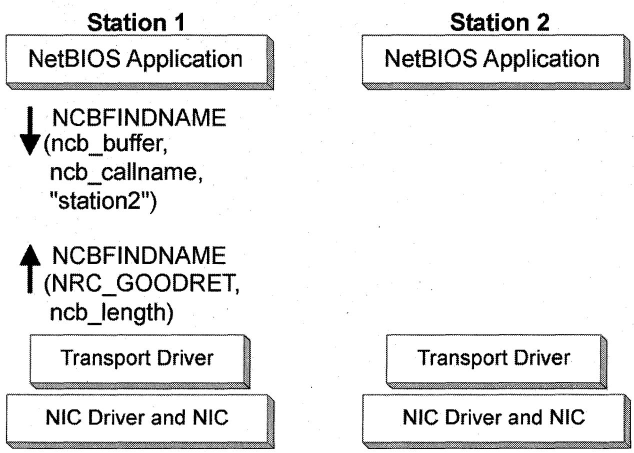

Figure 10-3: Name Registering Process.

The NCBDELNAME command deletes a name from its name table. Names are stored with a reference to the process that registered the name and a corresponding LANA number. A process cannot delete a name that was added by another process, even if it has the name number. The deletion process is shown in Figure 10-4.

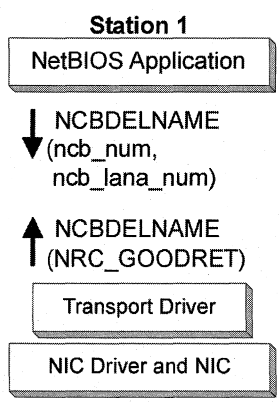

Figure 10-4: Name Deletion Process.

Session Support

The following are the session support commands:

- NCBCALL (call)
- NCBHANGUP (hang up)
- NCBLISTEN (listen)
- NCBSSTAT (session status)

The NCBCALL and NCBLISTEN commands establish a session between processes. One process issues the NCBLISTEN command to prepare to open a session. The other process issues the NCBCALL command to open the session. The remote computer must have an NCBLISTEN command pending. When the session is established, the NCBLISTEN command is completed and the calling process receives a local session number and the name of the remote session partner. In addition, the NCBCALL command is completed and the calling process receives a local session number in the **ncb_lsn** member of the **NCB** structure. The process of establishing a session is shown in Figure 10-5.

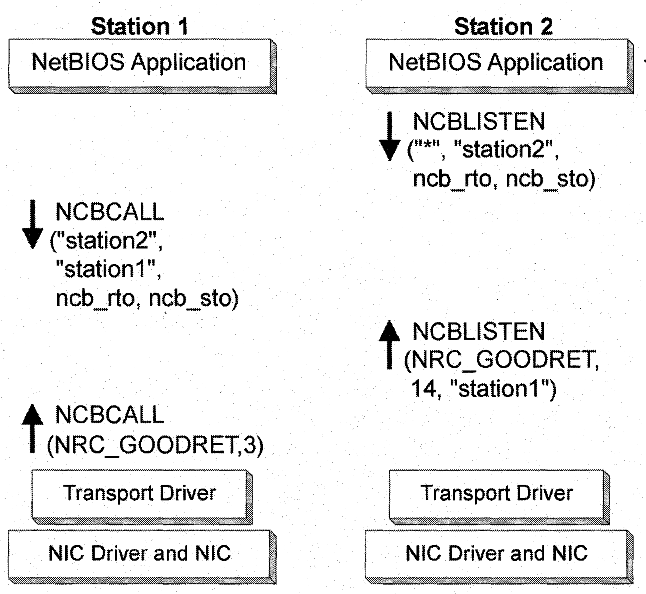

Figure 10-5: Establishing a Session.

The NCBSSTAT command obtains the status of any sessions that were opened using the specified name. The **ncb_buffer** member receives a **SESSION_HEADER** structure, followed by one or more **SESSION_BUFFER** structures. The process of obtaining the session status is shown in Figure 10-6.

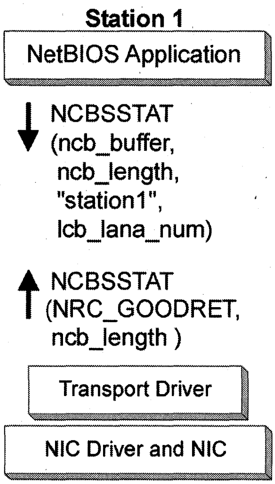

Figure 10-6: Obtaining Session Status.

The NCBHANGUP command closes the session identified by the specified local session number, as shown in Figure 10-7.

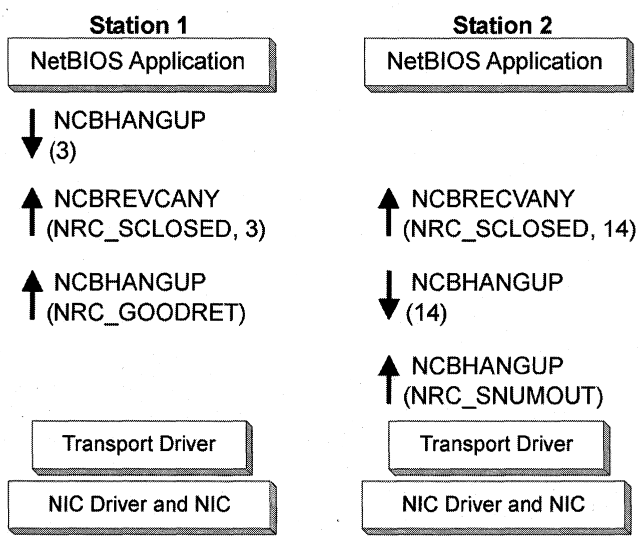

Figure 10-7: Closing the Session.

This example assumes that there are NCBRECVANY commands pending on both computers.

Data-Transfer Support

The following page shows the data-transfer support commands.

- NCBCHAINSEND (chain send)
- NCBCHAINSENDNA (chain send noack)
- NCBRECV (receive)
- NCBRECVANY (receive any)0
- NCBSEND (send)
- NCBSENDNA (send noack)

These commands provide reliable connection-oriented data transfer between session partners. Each data block is sent as a single message and received as a single message. The buffer supplied on a receive request must be large enough to hold an entire incoming message. If the receiving transport driver does not have enough space to store the message in the client-supplied buffer, it returns an error indicating that the buffer does not contain the entire message. The client must issue a subsequent receive command to obtain the remaining portion of the message.

The NCBSEND and NCBRECV commands transfer a single data buffer to the specified session partner, as shown in Figure 10-8.

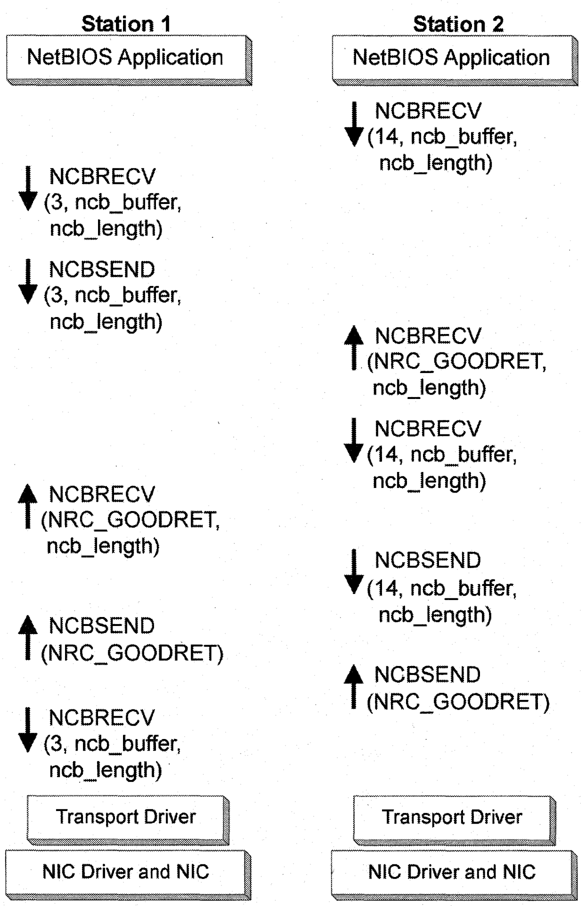

Figure 10-8: Transfering a Single Data Buffer to a Session Partner.

The NCBRECVANY command receives data from any session that was opened with the specified name, as shown in Figure 10-9.

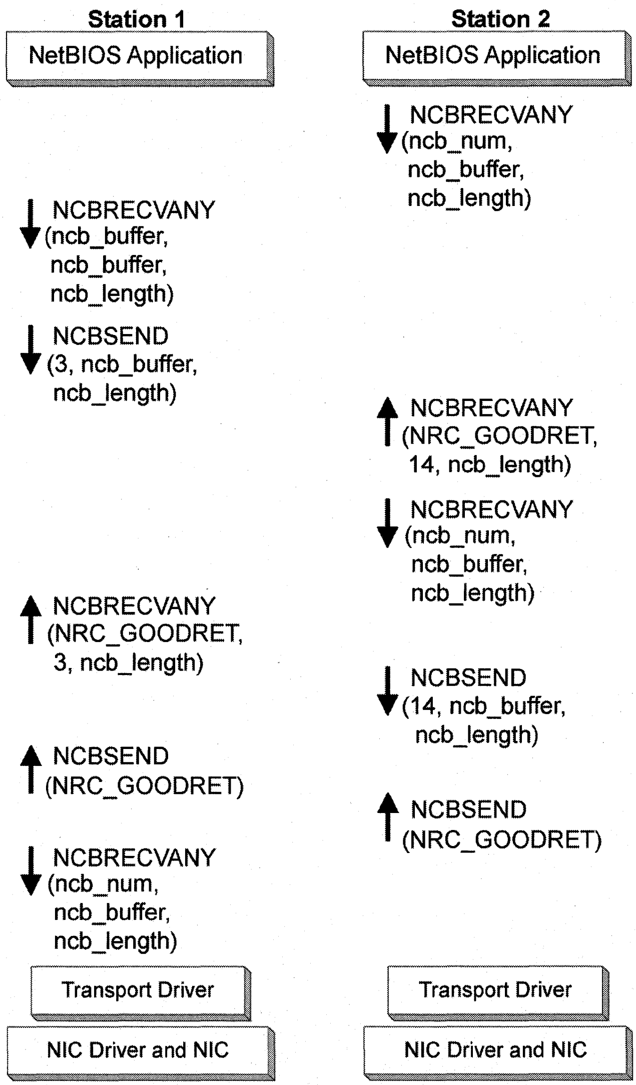

Figure 10-9: Receiving Data from a Specified Session.

The NCBCHAINSEND command sends two data buffers to the specified session partner as one message. The NCBSENDNA and NCBCHAINSENDNA commands are similar to the NCBSEND and NCBCHAINSEND commands, respectively. However, no acknowledgment is required.

Datagram Support

The following are the datagram support commands:

- NCBDGRECV (receive datagram)
- NCBDGRECVBC (receive broadcast datagram)
- NCBDGSEND (send datagram)
- NCBDGSENDBC (send broadcast datagram)

Datagram support provides unreliable connectionless data transfer. The message is a single data frame whose size is limited to a MAC frame minus any headers. The protocol driver ensures that the message is transmitted to the network medium. The receiver does not generate a response to the sender to indicate that the datagram was received. Therefore, unreliable connectionless data transfer requires fewer system resources than reliable connection-oriented data transfer.

The NCBDGRECV and NCBDGSEND commands transfer a datagram to a specified NetBIOS name. If the specified name is a unique name, the datagram is received by the single process that registered the name. If the specified name is a group name, the datagram is received by all processes that registered the name. This process is shown in Figure 10-10.

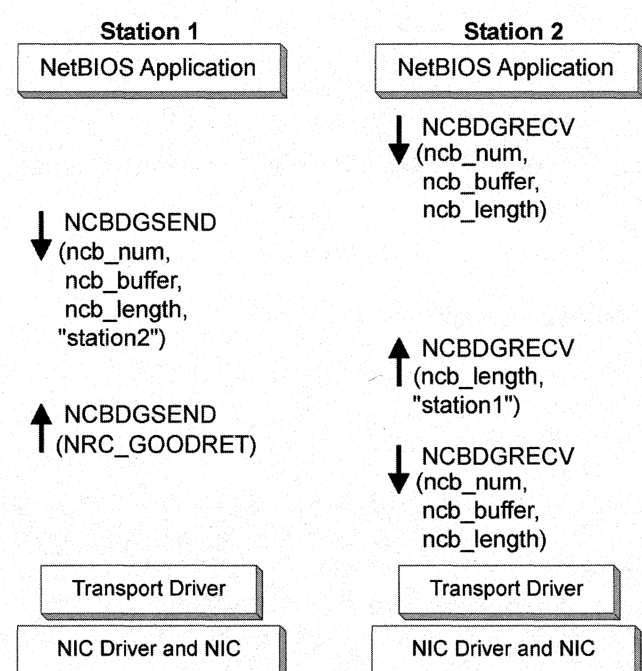

Figure 10-10: A Datagram Received by All Processes That Registered the Name.

The NCBDGSENDBC command broadcasts a datagram to all computers on the network. The datagram is received by all processes that have issued the NCBDGRECVBC command, as shown in Figure 10-11.

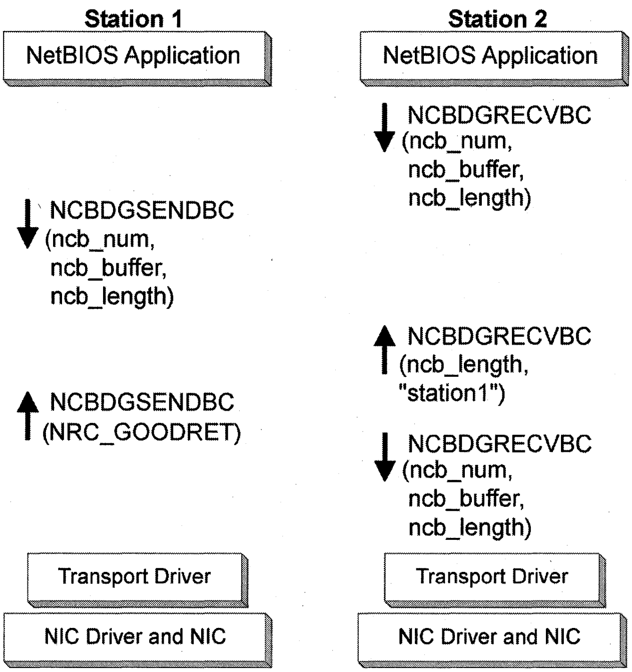

Figure 10-11: A Datagram Received by All Processes That Have Issued the NCBDGRECVBC command.

General Support

The following are the general support commands:

- NCBASTAT (adapter status)
- NCBCANCEL (cancel)
- NCBRESET (reset)

The NCBRESET command clears the name and session tables. It also ends all pending commands and sessions for the network route specified by the **ncb_lana_num** member of the **NCB** structure. A process is required to issue the NCBRESET command on each LANA number before it can issue any other NetBIOS command on the LANA number, with the exception of NCBENUM. This process is shown in Figure 10-12.

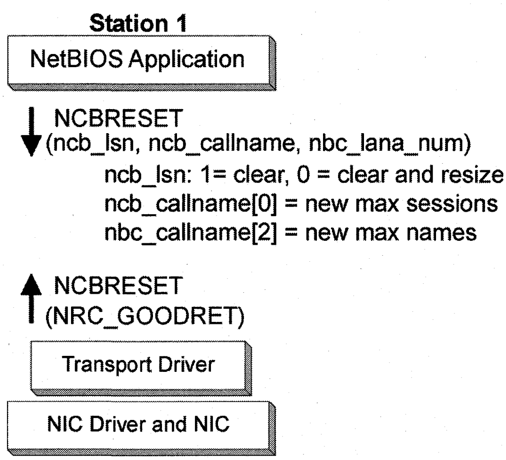

Figure 10-12: Issuing the NCBRESET Command on Each LANA Number.

The NCBASTAT command returns the current status and operation information about the network route specified in the **ncb_lana_num** member of the **NCB** structure. The **ncb_buffer** member receives an **ADAPTER_STATUS** structure, followed by an array of **NAME_BUFFER** structures.

The NCBASTAT command can be issued for the local computer or a remote computer. If the **ncb_callname** member contains an asterisk (*), information is returned for the local computer. If **ncb_callname** contains a NetBIOS name, the transport provider requests information from the remote computer where the name is registered, as shown in Figure 10-13.

The NCBCANCEL command cancels the command listed in the **ncb_command** member of the **NCB** structure passed to the **Netbios** function. NCBCANCEL closes the associated session when canceling the following commands:

- NCBCALL
- NCBCHAINSEND
- NCBHANGUP
- NCBLISTEN
- NCBSEND

The emulator simply returns NRC_CMDCAN if the following commands are successfully canceled or it returns NRC_CANOCCR if they finish before they could be canceled:

- NCBDGRECV
- NCBDGRECVBC
- NCBLANSTALERT
- NCBRECV
- NCBRECVANY

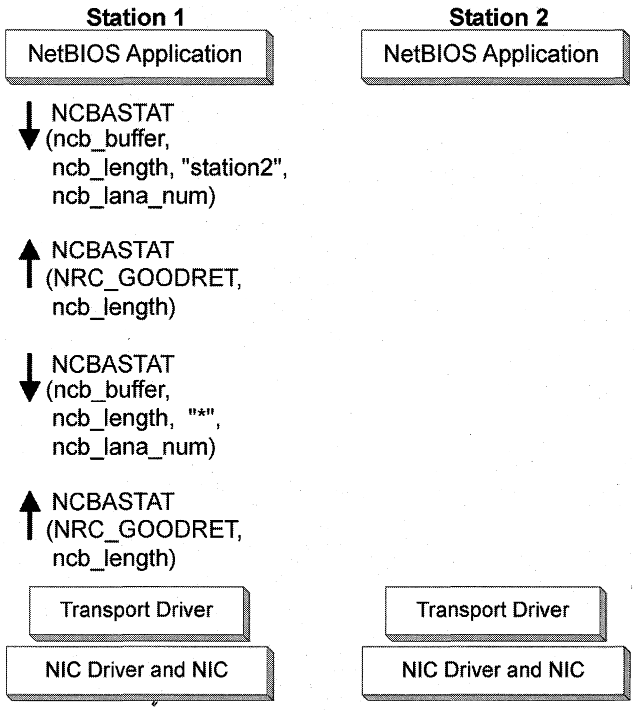

Figure 10-13: Transport Provider Requesting Information from the Remote Computer Where the Name Is Registered.

The following commands cannot be canceled:

- NCBADDGRNAME
- NCBADDNAME
- NCBCANCEL
- NCBDELNAME
- NCBDGSEND
- NCBDGSENDBC
- NCBRESET
- NCBSSTAT

An asynchronous NCBACTION command cannot be canceled using NCBCANCEL. To cancel a synchronous NCBACTION command, the transport provider must support an action code that cancels other action codes. Alternatively, you can cancel the command by hanging up the session, deleting the name, or resetting the LANA number.

Extension Support

The following commands are Windows NT/Windows 2000 extensions to the NetBIOS interface:

- NCBACTION (action)
- NCBENUM (enumerate)
- NCBLANSTALERT (LAN status alert)

The NCBACTION command enables extensions to the NetBIOS interface. The **ncb_buffer** member of the **NCB** structure points to an **ACTION_HEADER** structure, which specifies the transport provider and the provider-defined action code. The process of issuing the NCBACTION command is shown in Figure 10-14.

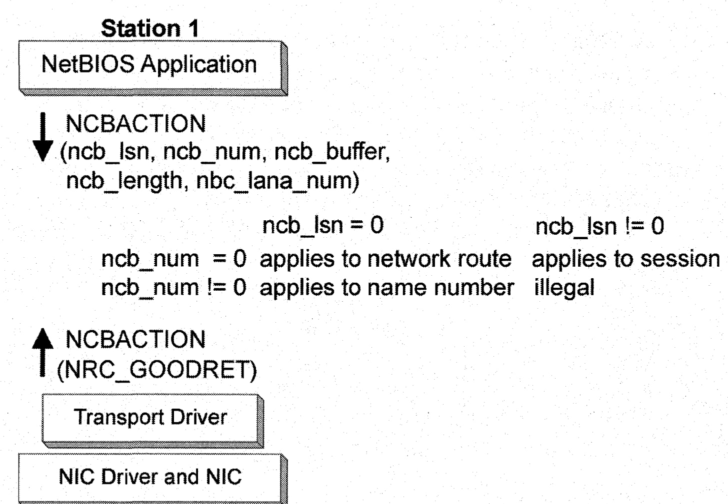

Figure 10-14: Issuing the NCBACTION Command.

The NCBENUM command enables enumeration of all LANA numbers. The **ncb_buffer** member of the **NCB** structure points to a **LANA_ENUM** structure, which specifies how many valid LANA numbers were returned, and an array of LANA numbers. The process of issuing the NCBENUM command is shown in Figure 10-15.

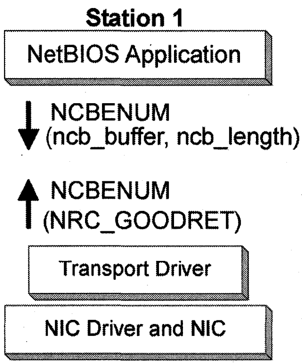

Figure 10-15: Issuing the NCBENUM Command.

The NCBLANSTALERT command notifies the user of catastrophic network failures, which can occur, for example, when there are duplicate names on the network. NCBLANSTALERT is typically issued as an asynchronous command. The command returns if an error occurs. At that time, the application should cease to use the network route. When the command returns, **ncb_retcode** does not indicate the error status associated with the network error condition. It indicates the success or failure of the NCBLANSTALERT command. The process of issuing the NCBLANSTALERT command is shown in Figure 10-16.

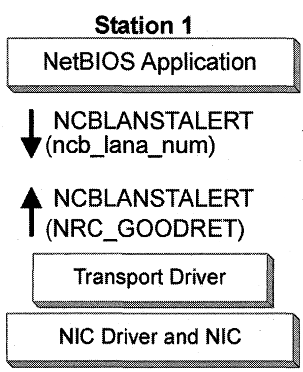

Figure 10-16: Issuing the NCBLANSTALERT Command.

Using the NetBIOS Interface

Listing All NetBIOS Names on a LANA

You can use the **Netbios** function to list all the NetBIOS names on a LANA. The following example uses a unique name as the name in the **ncb_callname** member of the **NCB** structure. This causes the adapter status to be treated as a remote call, which enables you to retrieve names added by other processes.

```
// Set LANANUM and LOCALNAME as appropriate for your system
#define LANANUM     0
#define LOCALNAME   "MAKEUNIQUE"
#define NBCheck(x)  if (NRC_GOODRET != x.ncb_retcode) { \
                        printf("Line %d: Got 0x%x from NetBios()\n", \
                               __LINE__, x.ncb_retcode); \
                    }

void MakeNetbiosName (char *, LPCSTR);
BOOL NBAddName (int, LPCSTR);
BOOL NBReset (int, int, int);
BOOL NBListNames (int, LPCSTR);
BOOL NBAdapterStatus (int, PVOID, int, LPCSTR);

void main ()
{
    if (!NBReset (LANANUM, 20, 30)) return;

    if (!NBAddName (LANANUM, LOCALNAME)) return;

    if (!NBListNames (LANANUM, LOCALNAME)) return;

    printf ("Succeeded.\n");
}

BOOL NBReset (int nLana, int nSessions, int nNames)
{
    NCB ncb;

    memset (&ncb, 0, sizeof (ncb));
    ncb.ncb_command = NCBRESET;
    ncb.ncb_lsn = 0;                // Allocate new lana_num resources
    ncb.ncb_lana_num = nLana;
    ncb.ncb_callname[0] = nSessions;  // maximum sessions
    ncb.ncb_callname[2] = nNames;   // maximum names
```

(continued)

(continued)

```
    Netbios (&ncb);
    NBCheck (ncb);

    return (NRC_GOODRET == ncb.ncb_retcode);
}

BOOL NBAddName (int nLana, LPCSTR szName)
{
    NCB ncb;

    memset (&ncb, 0, sizeof (ncb));
    ncb.ncb_command = NCBADDNAME;
    ncb.ncb_lana_num = nLana;

    MakeNetbiosName (ncb.ncb_name, szName);

    Netbios (&ncb);
    NBCheck (ncb);

    return (NRC_GOODRET == ncb.ncb_retcode);
}

// Build a name of length NCBNAMSZ, padding with spaces.
void MakeNetbiosName (char *achDest, LPCSTR szSrc)
{
    int cchSrc;

    cchSrc = lstrlen (szSrc);
    if (cchSrc > NCBNAMSZ)
        cchSrc = NCBNAMSZ;

    memset (achDest, ' ', NCBNAMSZ);
    memcpy (achDest, szSrc, cchSrc);
}

BOOL NBListNames (int nLana, LPCSTR szName)
{
    int cbBuffer;
    ADAPTER_STATUS *pStatus;
    NAME_BUFFER *pNames;
    int i;
    HANDLE hHeap;

    hHeap = GetProcessHeap();
```

```
    // Allocate the largest buffer that might be needed.
    cbBuffer = sizeof (ADAPTER_STATUS) + 255 * sizeof (NAME_BUFFER);
    pStatus = (ADAPTER_STATUS *) HeapAlloc (hHeap, 0, cbBuffer);
    if (NULL == pStatus)
        return FALSE;

    if (!NBAdapterStatus (nLana, (PVOID) pStatus, cbBuffer, szName))
    {
        HeapFree (hHeap, 0, pStatus);
        return FALSE;
    }

    // The list of names follows the adapter status structure.
    pNames = (NAME_BUFFER *) (pStatus + 1);

    for (i = 0; i < pStatus->name_count; i++)
        printf ("\t%.*s\n", NCBNAMSZ, pNames[i].name);

    HeapFree (hHeap, 0, pStatus);

    return TRUE;
}
BOOL NBAdapterStatus (int nLana, PVOID pBuffer, int cbBuffer, LPCSTR szName)
{
    NCB ncb;

    memset (&ncb, 0, sizeof (ncb));
    ncb.ncb_command = NCBASTAT;
    ncb.ncb_lana_num = nLana;

    ncb.ncb_buffer = (PUCHAR) pBuffer;
    ncb.ncb_length = cbBuffer;

    MakeNetbiosName (ncb.ncb_callname, szName);

    Netbios (&ncb);
    NBCheck (ncb);

    return (NRC_GOODRET == ncb.ncb_retcode);
}
```

Getting the MAC Address for an Ethernet Adapter

You can use the **Netbios** function to get the Media Access Control (MAC) address for an ethernet adapter if your card is bound to NetBIOS. The following example uses the NCBASTAT command, providing an asterisk (*) as the name in the **ncb_callname** member of the **NCB** structure.

Note The following code does not work reliably on Windows 95 or Windows 98.

```
#include <windows.h>
#include <wincon.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

typedef struct _ASTAT_
{
    ADAPTER_STATUS adapt;
    NAME_BUFFER    NameBuff [30];
}ASTAT, * PASTAT;

ASTAT Adapter;

void main (void)
{
    NCB ncb;
    UCHAR uRetCode;
    char NetName[50];

    memset( &ncb, 0, sizeof(ncb) );
    ncb.ncb_command = NCBRESET;
    ncb.ncb_lana_num = 0;

    uRetCode = Netbios( &ncb );
    printf( "The NCBRESET return code is: 0x%x \n", uRetCode );

    memset( &ncb, 0, sizeof(ncb) );
    ncb.ncb_command = NCBASTAT;
    ncb.ncb_lana_num = 0;

    strcpy( ncb.ncb_callname,  "*               " );
    ncb.ncb_buffer = (char *) &Adapter;
    ncb.ncb_length = sizeof(Adapter);

    uRetCode = Netbios( &ncb );
```

```
    printf( "The NCBASTAT return code is: 0x%x \n", uRetCode );
    if ( uRetCode == 0 )
    {
        printf( "The Ethernet Number is: %02x%02x%02x%02x%02x%02x\n",
                Adapter.adapt.adapter_address[0],
                Adapter.adapt.adapter_address[1],
                Adapter.adapt.adapter_address[2],
                Adapter.adapt.adapter_address[3],
                Adapter.adapt.adapter_address[4],
                Adapter.adapt.adapter_address[5] );
    }
}
```

NetBIOS Reference

NetBIOS Functions

Netbios

The **Netbios** function interprets and executes the specified Network Control Block (NCB).

The **Netbios** function is provided primarily for applications that were written for the NetBIOS interface and need to be ported to Win32. Applications not requiring compatibility with NetBIOS should use other interfaces, such as mailslots, named pipes, RPC, or distributed COM to accomplish tasks similar to those supported by NetBIOS. These other interfaces are more flexible and portable.

```
UCHAR Netbios(
  PNCB pncb
);
```

Parameters

pncb
: [in] Pointer to an **NCB** structure that describes the network control block.

Return Values

For synchronous requests, the return value is the return code of the **NCB** structure. That value is also returned in the **ncb_retcode** member of the **NCB** structure.

For asynchronous requests, there are the following possibilities:

- If the asynchronous command has already completed when **Netbios** returns to its caller, the return value is the return code of the **NCB** structure, just as if it were a synchronous **NCB** structure.

- If the asynchronous command is still pending when **Netbios** returns to its caller, the return value is zero.

If the address specified by the *pncb* parameter is invalid, the return value is NRC_BADNCB.

If the buffer length specified in the **ncb_length** member of the **NCB** structure is incorrect, or if the buffer specified by the **ncb_buffer** member is protected from write operations, the return value is NRC_BUFLEN.

Remarks

When an asynchronous network control block finishes and the **ncb_post** member is nonzero, the routine specified in **ncb_post** is called with a single parameter of type **PNCB**. This parameter contains a pointer to the Network Control Block.

The **NCB** structure also contains a handle of an event (the **ncb_event** member). The system sets the event to the nonsignaled state when an asynchronous NetBIOS command is accepted, and sets the event to the signaled state when the asynchronous NetBIOS command is completed. Only manual reset events should be used for synchronization. A specified event should not be associated with more than one active asynchronous NetBIOS command.

Using **ncb_event** to submit asynchronous requests requires fewer system resources than using **ncb_post**. Also, when **ncb_event** is nonzero, the pending request is canceled if the thread terminates before the request is processed. This is not true for requests sent by using **ncb_post**.

Win32s: This function does not support features that conflict with the non-preemptive, shared-memory design of Windows 3.1. Because the system does not implement events, this function ignores the **ncb_event** member of the **NCB** structure. Also, the system maintains one system-wide name table rather the a per-process name table.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Nb30.h.
Library: Use Netapi32.lib.

+ See Also

The NetBIOS Interface Overview, NetBIOS Functions, **NCB**

NetBIOS Structures

The following structures are used in NetBIOS.

ACTION_HEADER
ADAPTER_STATUS
FIND_NAME_BUFFER
FIND_NAME_HEADER
LANA_ENUM
NAME_BUFFER
NCB
SESSION_BUFFER
SESSION_HEADER

ACTION_HEADER

The **ACTION_HEADER** structure contains information about an action. This action is an extension to the standard transport interface.

```
typedef struct _ACTION_HEADER {
    ULONG   transport_id;
    USHORT  action_code;
    USHORT  reserved;
} ACTION_HEADER, *PACTION_HEADER;
```

Members

transport_id

Specifies the transport provider. This member can be used to check the validity of the request by the transport.

This member is always a four-character string. All strings starting with the letter M are reserved, as shown in the following example.

String	Meaning
M000	All transports
MNBF	NBF
MABF	AsyBEUI
MXNS	XNS

action_code

Specifies the action.

reserved

Reserved.

Remarks

The scope of the action is determined by the **ncb_lsn** and **ncb_num** members of the **NCB** structure, as shown on the following page.

	ncb_lsn = 0	ncb_lsn != 0
ncb_num = 0	Action applies to control channel associated with the valid LAN adapter.	Action applies to connection identifier associated with the valid local session number.
ncb_num != 0	Action applies to address associated with the valid LAN adapter.	Illegal combination.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Nb30.h.

See Also

The NetBIOS Interface Overview, NetBIOS Structures, **NCB**

ADAPTER_STATUS

The **ADAPTER_STATUS** structure contains information about a network adapter. This structure is pointed to by the **ncb_buffer** member of the **NCB** structure. **ADAPTER_STATUS** is followed by as many **NAME_BUFFER** structures as required to describe the network adapters on the system.

```
typedef struct _ADAPTER_STATUS {
    UCHAR   adapter_address[6];
    UCHAR   rev_major;
    UCHAR   reserved0;
    UCHAR   adapter_type;
    UCHAR   rev_minor;
    WORD    duration;
    WORD    frmr_recv;
    WORD    frmr_xmit;
    WORD    iframe_recv_err;
    WORD    xmit_aborts;
    DWORD   xmit_success;
    DWORD   recv_success;
    WORD    iframe_xmit_err;
    WORD    recv_buff_unavail;
    WORD    t1_timeouts;
    WORD    ti_timeouts;
    DWORD   reserved1;
    WORD    free_ncbs;
    WORD    max_cfg_ncbs;
```

```
    WORD    max_ncbs;
    WORD    xmit_buf_unavail;
    WORD    max_dgram_size;
    WORD    pending_sess;
    WORD    max_cfg_sess;
    WORD    max_sess;
    WORD    max_sess_pkt_size;
    WORD    name_count;
} ADAPTER_STATUS, *PADAPTER_STATUS;
```

Members

adapter_address
Specifies encoded address of the adapter.

rev_major
Specifies the major software-release level. This value is 3 for IBM NetBIOS 3.*x*.

reserved0
Reserved. This value is always zero.

adapter_type
Specifies the adapter type. This value is 0xFF for a Token Ring adapter or 0xFE for an Ethernet adapter.

rev_minor
Specifies the minor software-release level. This value is zero for IBM NetBIOS *x*.0.

duration
Specifies the duration of the reporting period, in minutes.

frmr_recv
Specifies the number of FRMR frames received.

frmr_xmit
Specifies the number of FRMR frames transmitted.

iframe_recv_err
Specifies the number of I frames received in error.

xmit_aborts
Specifies the number of aborted transmissions.

xmit_success
Specifies the number of successfully transmitted packets.

recv_success
Specifies the number of successfully received packets.

iframe_xmit_err
Specifies the number of I frames transmitted in error.

recv_buff_unavail
Specifies the number of times a buffer was not available to service a request from a remote computer.

t1_timeouts
Specifies the number of times that the DLC T1 timer timed out.

ti_timeouts
Specifies the number of times that the ti inactivity timer timed out. The ti timer is used to detect links that have been broken.

reserved1
Reserved. This value is always zero.

free_ncbs
Specifies the current number of free NCBs.

max_cfg_ncbs
Undefined for IBM NetBIOS 3.0.

max_ncbs
Undefined for IBM NetBIOS 3.0.

xmit_buf_unavail
Undefined for IBM NetBIOS 3.0.

max_dgram_size
Specifies the maximum size of a datagram packet. This value is always at least 512 bytes.

pending_sess
Specifies the number of pending sessions.

max_cfg_sess
Specifies the configured maximum pending sessions.

max_sess
Undefined for IBM NetBIOS 3.0.

max_sess_pkt_size
Specifies the maximum size of a session data packet.

name_count
Specifies the number of names in the local names table.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Nb30.h.

See Also

The NetBIOS Interface Overview, NetBIOS Structures, **NAME_BUFFER**, **NCB**

FIND_NAME_BUFFER

The **FIND_NAME_BUFFER** structure contains information about a local network session. One or more **FIND_NAME_BUFFER** structures follows a **FIND_NAME_HEADER** structure when an application specifies the NCBFINDNAME command in the **ncb_command** member of the **NCB** structure.

```
typedef struct _FIND_NAME_BUFFER {
    UCHAR length;
    UCHAR access_control;
    UCHAR frame_control;
    UCHAR destination_addr[6];
    UCHAR source_addr[6];
    UCHAR routing_info[18];
} FIND_NAME_BUFFER, *PFIND_NAME_BUFFER;
```

Members

length
Specifies the length, in bytes, of the **FIND_NAME_BUFFER** structure. Although this structure always occupies 33 bytes, not all of the structure is necessarily valid.

access_control
Specifies the access control for the LAN header.

frame_control
Specifies the frame control for the LAN header.

destination_addr
Specifies the destination address of the remote node where the name was found.

source_addr
Specifies the source address for the remote node where the name was found.

routing_info
Specifies additional routing information.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Nb30.h.

See Also

The NetBIOS Interface Overview, NetBIOS Structures, **FIND_NAME_HEADER**, **NCB**

FIND_NAME_HEADER

The **FIND_NAME_HEADER** structure contains information about a network name. This structure is followed by as many **FIND_NAME_BUFFER** structures as are required to describe the name.

```
typedef struct _FIND_NAME_HEADER {
    WORD  node_count;
    UCHAR reserved;
    UCHAR unique_group;
} FIND_NAME_HEADER, *PFIND_NAME_HEADER;
```

Members

node_count

Specifies the number of nodes on which the specified name was found. This structure is followed by the number of **FIND_NAME_BUFFER** structures specified by the **node_count** member.

reserved

Reserved.

unique_group

Specifies whether the name is unique. This value is 0 to specify a unique name or 1 to specify a group.

Remarks

The **FIND_NAME_HEADER** structure is pointed to by the **ncb_buffer** member of the **NCB** structure when an application issues an NCBFINDNAME command.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Nb30.h.

See Also

The NetBIOS Interface Overview, NetBIOS Structures, **FIND_NAME_BUFFER**, **NCB**

LANA_ENUM

The **LANA_ENUM** structure contains the numbers for the current LAN adapters.

```
typedef struct _LANA_ENUM {
    UCHAR length;
    UCHAR lana[MAX_LANA];
} LANA_ENUM, *PLANA_ENUM;
```

Members

length

Specifies the number of valid entries in the array of LAN adapter numbers.

lana

Specifies an array of LAN adapter numbers.

Remarks

The **LANA_ENUM** structure is pointed to by the **ncb_buffer** member of the **NCB** structure when an application issues the NCBENUM command. The NCBENUM command is not part of the IBM NetBIOS 3.0 specification.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Nb30.h.

See Also

The NetBIOS Interface Overview, NetBIOS Structures, **NCB**

NAME_BUFFER

The **NAME_BUFFER** structure contains information about a local network name. One or more **NAME_BUFFER** structures follows an **ADAPTER_STATUS** structure when an application specifies the NCBASTAT command in the **ncb_command** member of the **NCB** structure.

```
typedef struct _NAME_BUFFER {
    UCHAR name[NCBNAMSZ];
    UCHAR name_num;
    UCHAR name_flags;
} NAME_BUFFER, *PNAME_BUFFER;
```

Members

name

Specifies the local network name. This value is in the **ncb_name** member of the **NCB** structure.

name_num

Specifies the number for the local network name. This value is in the **ncb_num** member of the **NCB** structure.

name_flags

Specifies the current state of the name table entry. This member can be one of the following values.

Value	Meaning
REGISTERING	The name specified by the **name** member is being added to the network.
REGISTERED	The name specified by the **name** member has been successfully added to the network.
DEREGISTERED	The name specified by the **name** member has an active session when an NCBDELNAME command is issued. The name will be removed from the name table when the session is closed.
DUPLICATE	A duplicate name was detected during registration.
DUPLICATE_DEREG	A duplicate name was detected with a pending deregistration.
GROUP_NAME	The name specified by the **name** member was created by using the NCBADDGRNAME command.
UNIQUE_NAME	The name specified by the **name** member was created by using the NCBADDNAME command.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Nb30.h.

See Also

The NetBIOS Interface Overview, NetBIOS Structures, **ADAPTER_STATUS**, **NCB**

NCB

The **NCB** structure represents a network control block. It contains information about the command to perform, an optional post routine, an optional event handle, and a pointer to a buffer that is used for messages or other data. A pointer to this structure is passed to the **Netbios** function.

```
typedef struct _NCB {
    UCHAR  ncb_command;
    UCHAR  ncb_retcode;
    UCHAR  ncb_lsn;
    UCHAR  ncb_num;
    PUCHAR ncb_buffer;
    WORD   ncb_length;
    UCHAR  ncb_callname[NCBNAMSZ];
```

```
    UCHAR  ncb_name[NCBNAMSZ];
    UCHAR  ncb_rto;
    UCHAR  ncb_sto;
    void (CALLBACK *ncb_post) (struct _NCB *);
    UCHAR  ncb_lana_num;
    UCHAR  ncb_cmd_cplt;
#ifdef _WIN64
    UCHAR  ncb_reserve[18];
#else
    UCHAR  ncb_reserve[10];
#endif
    HANDLE ncb_event;
} NCB, *PNCB;
```

Members

ncb_command

Specifies the command code and a flag that indicates whether the **NCB** structure is processed asynchronously. The most significant bit contains the flag. If the ASYNCH constant is combined with a command code (by using the OR operator), the **NCB** structure is processed asynchronously. The following command codes are supported.

Code	Meaning
NCBACTION	**Windows NT/2000:** Enables extensions to the transport interface. NCBACTION is mapped to **TdiAction**. When this code is specified, the **ncb_buffer** member points to a buffer to be filled with an **ACTION_HEADER** structure, which is optionally followed by data. NCBACTION commands cannot be canceled by using NCBCANCEL. NCBACTION is not a standard NetBIOS 3.0 command.
NCBADDGRNAME	Adds a group name to the local name table. This name cannot be used by another process on the network as a unique name, but it can be added by anyone as a group name.
NCBADDNAME	Adds a unique name to the local name table. The TDI driver ensures that the name is unique across the network.
NCBASTAT	Retrieves the status of either a local or remote adapter. When this code is specified, the **ncb_buffer** member points to a buffer to be filled with an **ADAPTER_STATUS** structure, followed by an array of **NAME_BUFFER** structures.
NCBCALL	Opens a session with another name.
NCBCANCEL	Cancels a previous pending command.
NCBCHAINSEND	Sends the contents of two data buffers to the specified session partner.
NCBCHAINSENDNA	Sends the contents of two data buffers to the specified session partner and does not wait for acknowledgment.

(continued)

(continued)

Code	Meaning
NCBDELNAME	Deletes a name from the local name table.
NCBDGRECV	Receives a datagram from any name.
NCBDGRECVBC	Receives a broadcast datagram from any name.
NCBDGSEND	Sends datagram to a specified name.
NCBDGSENDBC	Sends a broadcast datagram to every host on the Local Area Network (LAN).
NCBENUM	**Windows NT/2000:** Enumerates LAN adapter (LANA) numbers. When this code is specified, the **ncb_buffer** member points to a buffer to be filled with a **LANA_ENUM** structure. NCBENUM is not a standard NetBIOS 3.0 command.
NCBFINDNAME	Determines the location of a name on the network. When this code is specified, the **ncb_buffer** member points to a buffer to be filled with a **FIND_NAME_HEADER** structure followed by one or more **FIND_NAME_BUFFER** structures.
NCBHANGUP	Closes a specified session.
NCBLANSTALERT	**Windows NT/2000:** Notifies the user of LAN failures that last for more than one minute.
NCBLISTEN	Enables a session to be opened with another name (local or remote).
NCBRECV	Receives data from the specified session partner.
NCBRECVANY	Receives data from any session corresponding to a specified name.
NCBRESET	Resets a LAN adapter. An adapter must be reset before it can accept any other NCB command that specifies the same number in the **ncb_lana_num** member. Use the following values to specify how resources are to be freed: • If **ncb_lsn** is not 0x00, all resources associated with **ncb_lana_num** are to be freed. • If **ncb_lsn** is 0x00, all resources associated with **ncb_lana_num** are to be freed, and new resources are to be allocated. The **ncb_callname**[0] byte specifies the maximum number of sessions, and the **ncb_callname**[2] byte specifies the maximum number of names. A nonzero value for the **ncb_callname**[3] byte requests that the application use NAME_NUMBER_1.
NCBSEND	Sends data to the specified session partner.
NCBSENDNA	Sends data to specified session partner and does not wait for acknowledgment.
NCBSSTAT	Retrieves the status of the session. When this value is specified, the **ncb_buffer** member points to a buffer to be filled with a **SESSION_HEADER** structure, followed by one or more **SESSION_BUFFER** structures.

Code	Meaning
NCBTRACE	Activates or deactivates NCB tracing. This command is not supported.
NCBUNLINK	Unlinks the adapter. This command is provided for compatibility with earlier versions of NetBIOS. It has no effect in Win32.

ncb_retcode

Specifies the return code. This value is set to NRC_PENDING while an asynchronous operation is in progress. The system returns one of the following values:

Value	Meaning
NRC_GOODRET	The operation succeeded.
NRC_BUFLEN	An illegal buffer length was supplied.
NRC_ILLCMD	An illegal command was supplied.
NRC_CMDTMO	The command was timed out.
NRC_INCOMP	The message was incomplete. The application is to issue another command.
NRC_BADDR	The buffer address was illegal.
NRC_SNUMOUT	The session number was out of range.
NRC_NORES	No resource was available.
NRC_SCLOSED	The session was closed.
NRC_CMDCAN	The command was canceled.
NRC_DUPNAME	A duplicate name existed in the local name table.
NRC_NAMTFUL	The name table was full.
NRC_ACTSES	The command finished; the name has active sessions and is no longer registered.
NRC_LOCTFUL	The local session table was full.
NRC_REMTFUL	The remote session table was full. The request to open a session was rejected.
NRC_ILLNN	An illegal name number was specified.
NRC_NOCALL	The system did not find the name that was called.
NRC_NOWILD	Wildcards are not permitted in the **ncb_name** member.
NRC_INUSE	The name was already in use on the remote adapter.
NRC_NAMERR	The name was deleted.
NRC_SABORT	The session ended abnormally.
NRC_NAMCONF	A name conflict was detected.
NRC_IFBUSY	The interface was busy.

(continued)

(continued)

Value	Meaning
NRC_TOOMANY	Too many commands were outstanding; the application can retry the command later.
NRC_BRIDGE	The **ncb_lana_num** member did not specify a valid network number.
NRC_CANOCCR	The command finished while a cancel operation was occurring.
NRC_CANCEL	The NCBCANCEL command was not valid; the command was not canceled.
NRC_DUPENV	The name was defined by another local process.
NRC_ENVNOTDEF	The environment was not defined. A reset command must be issued.
NRC_OSRESNOTAV	Operating system resources were exhausted. The application can retry the command later.
NRC_MAXAPPS	The maximum number of applications was exceeded.
NRC_NOSAPS	No service access points (SAPs) were available for NetBIOS.
NRC_NORESOURCES	The requested resources were not available.
NRC_INVADDRESS	The NCB address was not valid. This return code is not part of the IBM NetBIOS 3.0 specification and is not returned in the **NCB** structure. Instead, it is returned by **Netbios**.
NRC_INVDDID	The NCB DDID was invalid.
NRC_LOCKFAIL	The attempt to lock the user area failed.
NRC_OPENERR	An error occurred during an open operation being performed by the device driver. This error code is not part of the NetBIOS 3.0 specification.
NRC_SYSTEM	A system error occurred.
NRC_PENDING	An asynchronous operation is not yet finished.

ncb_lsn

Identifies the local session number. This number uniquely identifies a session within an environment. This number is returned by **Netbios** after a successful NCBCALL command.

ncb_num

Specifies the number for the local network name. This number is returned by **Netbios** after a successful NCBADDNAME or NCBADDGRNAME command. This number, not the name, must be used with all datagram commands and for NCBRECVANY commands.

The number for NAME_NUMBER_1 is always 0x01. **Netbios** assigns values in the range 0x02 to 0xFE for the remaining names.

ncb_buffer

Pointer to the message buffer. The buffer must have write access. Its uses are as follows:

Command	Purpose
NCBSEND	Contains the message to be sent.
NCBRECV	Receives the message.
NCBSSTAT	Receives the requested status information.

ncb_length

Specifies the size, in bytes, of the message buffer. For receive commands, this member is set by the **Netbios** function to indicate the number of bytes received.

If the buffer length is incorrect, the **Netbios** function returns the NRC_BUFLEN error code.

ncb_callname

Specifies the name of the remote application. Trailing-space characters should be supplied to make the length of the string equal to NCBNAMSZ.

ncb_name

Specifies the name by which the application is known. Trailing-space characters should be supplied to make the length of the string equal to NCBNAMSZ.

ncb_rto

Specifies the time-out period for receive operations, in 500-millisecond units, for the session. A value of zero implies no time-out. Specify with the NCBCALL or NCBLISTEN command. Affects subsequent NCBRECV commands.

ncb_sto

Specifies the time-out period for send operations, in 500-millisecond units, for the session. A value of zero implies no time-out. Specify with the NCBCALL or NCBLISTEN command. Affects subsequent NCBSEND and NCBCHAINSEND commands.

ncb_post

Specifies the address of the post routine to call when the asynchronous command is completed. The post routine is defined as:

NCB_POST PostRoutine(PNCB *pncb*);

where the *pncb* parameter is a pointer to the completed network control block.

ncb_lana_num

Specifies the LAN adapter number. This zero-based number corresponds to a particular transport provider using a particular LAN adapter board.

ncb_cmd_cplt

Specifies the command complete flag. This value is the same as the **ncb_retcode** member.

ncb_reserve

Reserved; must be zero.

ncb_event

Specifies a handle to an event object that is set to the nonsignaled state when an asynchronous command is accepted, and it is set to the signaled state when the asynchronous command is completed. The event is signaled if the **Netbios** function returns a nonzero value. Only manual reset events should be used for synchronization. A specified event should not be associated with more than one active asynchronous command.

The **ncb_event** member must be zero if the **ncb_command** member does not have the ASYNCH flag set or if **ncb_post** is nonzero. Otherwise, **Netbios** returns the NRC_ILLCMD error code.

Remarks

Using **ncb_event** to issue asynchronous requests requires fewer system resources than using **ncb_post**. In addition, when **ncb_event** is nonzero, the pending request is canceled if the thread terminates before the request is processed. This is not true for asynchronous requests sent using **ncb_post**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Nb30.h.

See Also

The NetBIOS Interface Overview, NetBIOS Structures, **ACTION_HEADER**, **ADAPTER_STATUS**, **FIND_NAME_BUFFER**, **FIND_NAME_HEADER**, **LANA_ENUM**, **NAME_BUFFER**, **Netbios**, **SESSION_BUFFER**, **SESSION_HEADER**

SESSION_BUFFER

The **SESSION_BUFFER** structure contains information about a local network session. One or more **SESSION_BUFFER** structures follows a **SESSION_HEADER** structure when an application specifies the NCBSSTAT command in the **ncb_command** member of the **NCB** structure.

```
typedef struct _SESSION_BUFFER {
    UCHAR lsn;
    UCHAR state;
    UCHAR local_name[NCBNAMSZ];
    UCHAR remote_name[NCBNAMSZ];
    UCHAR rcvs_outstanding;
    UCHAR sends_outstanding;
} SESSION_BUFFER, *PSESSION_BUFFER;
```

Members

lsn

Specifies the local session number.

state

Specifies the state of the session. This member can be one of the following values.

Value	Meaning
LISTEN_OUTSTANDING	The session is waiting for a call from a remote computer.
CALL_PENDING	The session is attempting to connect to a remote computer.
SESSION_ESTABLISHED	The session connected and is able to transfer data.
HANGUP_PENDING	The session is being deleted due to a command by the local user.
HANGUP_COMPLETE	The session was deleted due to a command by the local user.
SESSION_ABORTED	The session was abandoned due to a network or user problem.

local_name

Specifies the 16-byte NetBIOS name on the local computer used for this session.

remote_name

Specifies the 16-byte NetBIOS name on the remote computer used for this session.

rcvs_outstanding

Specifies the number of pending NCBRECV commands.

sends_outstanding

Specifies the number of pending NCBSEND and NCBCHAINSEND commands.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Nb30.h.

+ See Also

The NetBIOS Interface Overview, NetBIOS Structures, **NCB**, **SESSION_HEADER**

SESSION_HEADER

The **SESSION_HEADER** structure contains information about a network session. This structure is pointed to by the **ncb_buffer** member of the **NCB** structure.

SESSION_HEADER is followed by as many **SESSION_BUFFER** structures as are required to describe the current network sessions.

```
typedef struct _SESSION_HEADER {
    UCHAR sess_name;
    UCHAR num_sess;
    UCHAR rcv_dg_outstanding;
    UCHAR rcv_any_outstanding;
} SESSION_HEADER, *PSESSION_HEADER;
```

Members

sess_name
Specifies the name number of the session. This value corresponds to the **ncb_num** member of the **NCB** structure.

num_sess
Specifies the number of sessions that have the name specified by the **sess_name** member.

rcv_dg_outstanding
Specifies the number of outstanding NCBDGRECV and NCBDGRECVBC commands.

rcv_any_outstanding
Specifies the number of outstanding NCBRECVANY commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Nb30.h.

See Also

The NetBIOS Interface Overview, NetBIOS Structures, **NCB**, **SESSION_BUFFER**

CHAPTER 11

Synchronization Manager

To enable applications for mobile computing, the operating system provides a synchronization manager (SyncMgr). Together with the connectivity functions, notifications (System Event Notification Service), and client side caching, the SyncMgr provides an infrastructure to support mobile computing. Instead of each application implementing its own technology to cache and synchronize network resources for local use, the operating system provides an integrated model that all applications can use.

To help you find the information you need, the following list describes each section of this Synchronization Manager chapter:

- Synchronization Manager Overview offers a broad overview of SyncMgr technology, including a discussion of mobile computing configurations where SyncMgr would be useful.
- SyncMgr Reference lists methods and interfaces for Synchronization Manager.

Synchronization Manager Overview

The Synchronization Manager provides a centralized, standard technology for synchronizing files for offline use on a local computer. The user can prepare a computer for offline use by updating the needed files from the network storage. Then, when the computer is brought back online, any offline changes can be copied back to the network. SyncMgr provides a common user interface that all applications can share for synchronizing their files.

Files are transferred independent of the protocol. For example, an e-mail program can transfer its messages using SMTP, NMTP, or POP3, while a browser can use HTTP and a database can use RPC. The protocol is transparent to the SyncMgr.

SyncMgr is also storage independent and can transfer files, e-mail messages, HTML pages, or database information transparently.

Mobile Computing Configurations

The Synchronization Manager is useful for computers configured as follows:

- Mobile computer used in a docking station on a high bandwidth network but occasionally used via a dial-in connection.
- Mobile computer used mostly via a dial-in connection

- Desktop computer used strictly via a dial-in connection, such as a computer used by a telecommuter
- Desktop computer used strictly via a high bandwidth network.

Although the last case is not a typical mobile scenario, latency issues with network access may make it convenient to cache resources locally. In all these configurations, the application can use the SyncMgr to keep files and other resources cached locally and synchronized between online and offline use.

Application Scenarios

Applications and services that can use the Synchronization Manager include the following:

- Microsoft® Office applications that need to prepare files for offline use
- Client side caching that let you cache files locally
- Browsers that can cache HTML pages locally
- Mail programs that can cache e-mail messages locally
- Databases that can store information locally

Synchronization Manager Architecture

The Synchronization Manager includes user interface components, such as the tabbed dialogs in the SyncMgr service, error dialogs, and progress dialogs. With the user interface components the end user can schedule applications for synchronization and set up automatic synchronization to occur in conjunction with specified system events. For example, the SyncMgr can be invoked at user logon or system shutdown. The user can also invoke a manual synchronization.

The user selects an application and specifies the items within the application to be synchronized and sets a trigger event. For example, within an e-mail application, the Inbox, the Outbox, or some other folder containing messages can be a separate item for the application.

SyncMgr also includes a programming interface so that applications can register to use synchronization features, can process errors, and can receive progress information and notifications during the synchronization process.

Using Synchronization Manager from a Program

To enable your application to work with the Sychronization Manager you must implement a COM object to handle synchronization notifications that you receive from SyncMgr. Your application's handler performs the synchronization for the items that you handle. Included in your handler, you must implement the **ISyncMgrSynchronize** interface. Also, you must provide an enumerator object and **ISyncMgrEnumItems** for any separate items that your application can synchronize.

SyncMgr implements **ISyncMgrSynchronizeCallback** and **ISyncMgrSynchronizeInvoke**.

The SyncMgr calls methods in your **ISyncMgrSynchronize** to get information on the items that your application handles and information on the handler that you provide for synchronizing these items.

At runtime, the synchronization process follows these steps:

1. SyncMgr notifies your application that it is time for synchronization to occur for one of the items that your application handles by calling your **ISyncMgrSynchronize::Initialize** method.
2. SyncMgr calls **ISyncMgrSynchronize::EnumSyncMgrItems** to obtain the **ISyncMgrEnumItems** interface for the items handled by your application.
3. SyncMgr calls **ISyncMgrSynchronize::SetProgressCallback** to provide your handler with the interface pointer for the **ISyncMgrSynchronizeCallback** interface. Your handler uses this interface to call back to the SyncMgr during synchronization.
4. SyncMgr then calls your **ISyncMgrSynchronize::PrepareForSync** method to give your handler a chance to display any user interface element that is necessary before synchronization begins. For example, an e-mail application may display a user logon dialog.
5. Your handler calls **ISyncMgrSynchronizeCallback::EnableModeless** before and after displaying any user interface elements. Your handler calls **ISyncMgrSynchronizeCallback::PrepareForSyncCompleted** when you are done.
6. SyncMgr calls your **ISyncMgrSynchronize::Synchronize** method to start the synchronization.

During the synchronization process, SyncMgr continues to call methods in your **ISyncMgrSynchronize** interface. It can send your handler errors, progress, and notifications. It can also enumerate through the items that your application handles or allow your application to show properties for the items.

Your handler calls methods in **ISyncMgrSynchronizeCallback** to determine if an item should be skipped, to log errors, and to post progress information during the synchronization process.

For further information, see the related reference pages for the interfaces involved.

When the synchronization is completed, your handler calls **ISyncMgrSynchronizeCallback::SynchronizeCompleted**.

SyncMgr Reference

This section lists the following methods and interfaces for Synchronization Manager:

- **ISyncMgrEnumItems**
- **ISyncMgrSynchronize**
- **ISyncMgrSynchronizeCallback**
- **ISyncMgrSynchronizeInvoke**
- **ISyncMgrRegister**

ISyncMgrEnumItems

The **ISyncMgrEnumItems** interface is used to enumerate through an array of **SYNCMGRITEM** structures. Each of these structures provides information on an item that can be synchronized. **ISyncMgrEnumItems** has the same methods as all standard enumerator interfaces: Next, Skip, Reset, and Clone. For general information on these methods, refer to **IEnum*XXXX***.

When to Implement

If the registered application works with the Synchronization Manager to synchronize items, it must implement an enumerator object with this interface to enumerate through the items.

When to Use

The SyncMgr obtains a pointer to this interface and calls each method during the synchronization process.

The prototypes of the methods are as follows:

```
HRESULT Next(
  ULONG celt,            // [in] Number of items in array
  LPSYNCMGRITEM rgelt,   // [out] Address of array
                         // containing items
  ULONG* pceltFetched    // [out] Address of variable
                         // containing actual number of items
);

HRESULT Skip(
  ULONG celt             // [in] Number of items to skip
);

HRESULT Reset(void);
```

```
HRESULT Clone(
  ISyncMgrEnumItems **ppenum      // [out] Address of
                                  // variable that receives
                                  // the ISyncMgrEnumItems
                                  // interface pointer
);
```

Remarks

ISyncMgrEnumItems::Next

Enumerates the next *celt* elements in the enumerator's list, returning them in *rgelt* along with the actual number of enumerated elements in *pceltFetched*.

E_NOTIMPL is not allowed as a return value. If an error value is returned, no entries in the *rgelt* array are valid on exit and require no release.

ISyncMgrEnumItems::Skip

Instructs the enumerator to skip the next *celt* elements in the enumeration so the next call to **ISyncMgrEnumItems::Next** does not return those elements.

ISyncMgrEnumItems::Reset

Instructs the enumerator to position itself at the beginning of the list of elements.

There is no guarantee that the same set of elements are enumerated on each pass through the list, nor are the elements necessarily be enumerated in the same order. The exact behavior depends on the collection being enumerated. It is too expensive in terms of memory for some collections, such as files in a directory, to maintain a specific state.

ISyncMgrEnumItems::Clone

Creates another items enumerator with the same state as the current enumerator to iterate over the same list. This method makes it possible to record a point in the enumeration sequence in order to return to that point at a later time.

The caller must release this new enumerator separately from the first enumerator.

! Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

+ See Also

SYNCMGRITEM

ISyncMgrSynchronize

The **ISyncMgrSynchronize** interface enables the registered application or service to receive notifications from the Synchronization Manager.

When to Implement

This interface should be implemented on the registered application's handler to receive notifications from the SyncMgr and to participate in the synchronization process.

When to Use

The SyncMgr calls the methods of this interface to send notifications to the registered application or service during synchronization.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

ISyncMgrSynchronize methods	Description
Initialize	Determines whether the registered application handles the synchronization event.
GetHandlerInfo	Called by SyncMgr to obtain handler information.
EnumSyncMgrItems	Called by SyncMgr to obtain the enumeration interface for the items maintained by the registered application.
GetItemObject	Called by SyncMgr to obtain an interface on the requested server's Items.
ShowProperties	Called by SyncMgr when the item is selected in the Choice dialog box and the user clicks the Properties button.
SetProgressCallback	Called by SyncMgr to set the **ISyncMgrSynchronizeCallback** interface.
PrepareForSync	Called by SyncMgr to give the registered application a chance to show any user interface and do any necessary initialization before calling the **Synchronize** method.
Synchronize	Called by SyncMgr once for each selected group after the user has selected applications for synchronization.
SetItemStatus	Called by SyncMgr to set the status of an item being synchronized.
ShowError	Called by SyncMgr to set an error status for the item being synchronized.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrEnumItems, **ISyncMgrSynchronizeCallback**, **ISyncMgrSynchronizeInvoke**

ISyncMgrSynchronize::Initialize

The Synchronization Manager calls the **ISyncMgrSynchronize::Initialize** method in the registered application's handler to determine whether the handler will process the synchronization event.

```
HRESULT Initialize(
  DWORD dwReserved,          // Reserved
  DWORD dwSyncMgrFlags,      // SYNCMGRFLAG enumeration
  DWORD cbCookie,            // Size of lpCookie
  BYTE const *lpCookie       // Token identifying application
);
```

Parameters

dwReserved
[in] Reserved; must be zero.

dwSyncMgrFlags
[in] Specifies the SYNCMGRFLAG enumeration values that describe how the synchronization event was initiated.

cbCookie
[in] Specifies the size in bytes of the *lpCookie* data.

lpCookie
[in] Points to the token identifying the application. This token was passed when the application programmatically invoked SyncMgr.

Return Values

This method supports the standard return values, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
Initialization was successful.

S_FALSE
Application handler will not process the synchronize event.

Remarks

The SYNCMGRFLAG enumeration values apply through the lifetime of the **ISyncMgrSynchronize** interface and are used by the other **ISyncMgrSynchronize** methods.

If the application does not recognize the SYNCMGRFLAG event, the application should treat the event as a manual synchronization.

The registered application's handler cannot display a user interface within this call unless it is the very first time the initialize method is called. The application is allowed to display any one-time initialization it needs to set up items and introduce the user to the application's feature. If you need to display a user interface for any other reason as part of the synchronization process, you can do so in the **ISyncMgrSynchronize::PrepareForSync** method.

lpCookie is NULL unless the handling application invoked the Synchronization Manager programmatically using **ISyncMgrSynchronizeInvoke::UpdateItems**. In this case the CLSID is that of the handling application and the value of **lpCookie** is passed in by the handling application and is passed back by SyncMgr during synchronization for context. **lpCookie** is only meaningful if SYNCMGRFLAG_INVOKE is set.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronize::PrepareForSync, **ISyncMgrSynchronizeInvoke::UpdateItems**, SYNCMGRFLAG

ISyncMgrSynchronize::GetHandlerInfo

The **ISyncMgrSynchronize::GetHandlerInfo** method obtains handler information.

```
HRESULT GetHandlerInfo(
  LPSYNCMGRHANDLERINFO *ppSyncMgrHandlerInfo
             // SYNCMGRHANDLERINFO structure
);
```

Parameters

ppSyncMgrHandlerInfo
[out] Returns a pointer to a SYNCMGRHANDLERINFO structure.

Return Values

This method supports the standard return values, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
: Handler information was returned successfully.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

SYNCMGRHANDLERINFO

ISyncMgrSynchronize::EnumSyncMgrItems

This method obtains the **ISyncMgrEnumItems** interface for the items handled by the registered application.

```
HRESULT EnumSyncMgrItems(
  ISyncMgrEnumItems **ppSyncMgrEnumItems
        // Address of ISyncMgrEnumItems pointer
);
```

Parameters

ppSyncMgrEnumItems
: [out] Address of variable that receives a pointer to a valid **ISyncMgrEnumItems** interface.

Return Values

This method supports the standard return values, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following.

S_OK
: The enumeration interface was successfully returned.

S_SYNCMGR_MISSINGITEMS
: The enumeration interface object is successfully returned but some items are missing. When this success code is returned SyncMgr does not remove any stored preferences for ItemIds that were not returned in the enumerator.

Remarks

The enumeration object created by this method implements the **ISyncMgrEnumItems** interface, a standard enumeration interfaces that contains the Next, Reset, Clone, and Skip methods.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrEnumItems

ISyncMgrSynchronize::GetItemObject

The **ISyncMgrSynchronize::GetItemObject** method obtains an interface on a specified item handled by the registered application.

```
HRESULT GetItemObject(
  REFSYNCMGRITEMID ItemID,  // Identifies the requested item
  REFIID riid,              // Identifies the requested
                            // interface
  void **ppv                // Address of variable
                            // containing the requested
                            // interface pointer
);
```

Parameters

ItemID
[in] Identifier for the requested item.

riid
[in] Identifier for the requested interface.

ppv
[out] Address of a variable that receives a pointer to the requested interface.

Return Values

E_NOTIMPL
The requested interface was not found.

Remarks

This method is for future use. There are currently no interfaces defined on an Item. Application implementers must return E_NOTIMPL from this method.

! Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

ISyncMgrSynchronize::ShowProperties

The **ISyncMgrSynchronize::ShowProperties** method is called by the Synchronization Manager when the user selects an item in the choice dialog box and clicks the properties button.

```
HRESULT ShowProperties(
  HWND hWndParent,    // Parent hwnd for any dialog displayed
  REFSYNCMGRITEMID ItemID    // Specifies item requested
);
```

Parameters

hWndParent
[in] Specifies the parent hWnd for any user interface the registered application displays to set properties. This value may be NULL.

ItemID
[in] Identifies the item for which properties are requested.

Return Values

This method supports the standard return values, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
Properties dialog for this item was handled properly.

Remarks

If a registered application provides a properties dialog box for an item, it must set the SYNCMGRITEM_HASPROPERTIES bit in the *dwFlags* member of the **SYNCMGRITEM** structure.

If *ItemID* is GUID_NULL the handler should show the Properties dialog for the overall handler.

The appearance of the displayed dialog box should be consistent with a standard Property Page dialog box.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

SYNCMGRITEM, **SYNCMGRITEMFLAGS**

ISyncMgrSynchronize::SetProgressCallback

The **ISyncMgrSynchronize::SetProgressCallback** method sets the **ISyncMgrSynchronizeCallback** interface. Registered applications use this callback interface to give status information from within the **ISyncMgrSynchronize::PrepareForSync** and **ISyncMgrSynchronize::Synchronize** methods.

```
HRESULT SetProgressCallback(
  ISyncMgrSynchronizeCallback *pSyncCallBack
            // Pointer to callback interface
);
```

Parameters

pSyncCallBack
[in] Pointer to **ISyncMgrSynchronizeCallback** interface the registered application uses to provide feedback to SyncMgr about the synchronization status and to notify SyncMgr when the synchronization is complete.

Return Values

This method supports the standard return values, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
Synchronization callback interface was successfully set.

Remarks

Registered applications must call the **AddRef** method in the **ISyncMgrSynchronizeCallback** interface and use it when calling SyncMgr to provide status text and progress indicator feedback.

If the registered application already has an **ISyncMgrSynchronizeCallback** interface when the method is called, the old interface must be released and the **AddRef** method of the new interface must be called. The new interface must be maintained by the registered application.

Before the **ISyncMgrSynchronize** interface is released, SyncMgr calls this method with *pSyncCallBack* parameter set to NULL. The registered application should then release the *pSyncCallback* interface previously passed.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronizeCallback

ISyncMgrSynchronize::PrepareForSync

The **ISyncMgrSynchronize::PrepareForSync** method allows the registered application to display any user interface and perform any necessary initialization before the **ISyncMgrSynchronize::Synchronize** method is called. For example, an application such as the Microsoft® Outlook® e-mail client may need to display the password dialog box to enable the user to log onto the mail server.

```
HRESULT PrepareForSync(
  ULONG cbNumItems,        // Number of items in ItemId array
  SYNCMGRITEMID *pItemIDs,    // Array of ItemIds
  HWND hWndParent,         // Parent hWnd for user interface
  DWORD dwReserved         // Reserved
);
```

Parameters

cbNumItems
[in] Number of items in the *ItemId* array pointed to by *pItemIDs*.

pItemIDs
[in] Array of *ItemID*s the user has chosen to synchronize.

hWndParent
[in] Handle to the parent *hWnd* that the registered application should use for any user interface element displayed. This value may be NULL.

dwReserved
[in] Reserved. Registered applications should ignore this value.

Return Values

This method supports the standard return values, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
: Preparation was successful.

Remarks

The registered application's handler should return from this method as soon as possible and then call the **ISyncMgrSynchronizeCallback::PrepareForSyncCompleted** method. It is possible for the registered application's handler to call the **ISyncMgrSynchronizeCallback::PrepareForSyncCompleted** method before returning from this method.

Registered applications should only show a user interface if the SYNCMGRFLAG_MAYBOTHERUSER flag was set in the *dwSyncFlags* parameter of the **ISyncMgrSynchronize::Initialize** method. If a registered application cannot prepare for synchronization without showing a user interface when the SYNCMGRFLAG_MAYBOTHERUSER flag is not set it should return S_FALSE from this method.

The array of *ItemIDs* that are passed into this method are relevant to the **ISyncMgrSynchronize::Synchronize** method also.

The **ISyncMgrSynchronizeCallback** methods can be called on any thread in the registered application.

! Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

+ See Also

ISyncMgrSynchronize::Synchronize, ISyncMgrSynchronize::Initialize, ISyncMgrSynchronizeCallback::PrepareForSyncCompleted, ISyncMgrSynchronizeCallback, SYNCMGRFLAG

ISyncMgrSynchronize::Synchronize

The Synchronization Manager calls the **ISyncMgrSynchronize::Synchronize** method once for each selected group after the user has chosen the registered applications to be synchronized.

```
HRESULT Synchronize(
  HWND hWndParent      // Parent hwnd for user interface
);
```

Parameters

hWndParent

[in] Handle to the parent *hWnd* the registered application should use for any user interface elements that it displays. This value may be NULL.

Return Values

This method supports the standard return values, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
Synchronization was successful.

E_FAIL
Synchronization failed.

Remarks

If the user does not select any item choices for the registered application, the **ISyncMgrSynchronize::Synchronize** method is not called and the interface is released. If this method is called, the application must synchronize the items that were specified in the **ISyncMgrSynchronize::PrepareForSync** method.

The registered application's handler should return from the **ISyncMgrSynchronize::Synchronize** method as soon as possible and then call the **ISyncMgrSynchronizeCallback::SynchronizeCompleted** method. It is acceptable for the handler to call the **ISyncMgrSynchronizeCallback::SynchronizeCompleted** call before returning from the **ISyncMgrSynchronize::Synchronize** method.

The application must give progress feedback and check whether the synchronization should be canceled by using the *pSyncCallBack* interface pointer that was setup in the **ISyncMgrSynchronize::SetProgressCallback** method.

Applications must provide progress information even if the SYNCMGRFLAG_MAYBOTHERUSER was not specified in **ISyncMgrSynchronize::Initialize.**

Applications should attempt not to show user interface elements from within the **ISyncMgrSynchronize::Synchronize** method. Any user interface elements should be shown in the **ISyncMgrSynchronize::PrepareForSync** and **ISyncMgrSynchronize::ShowError** methods so the end user experiences a consistent user interface which is limited to logon and to specifying shares to be synchronized. Subsequently, the synchronization can be performed without any user intervention. After the synchronization is complete, conflicts or other error messages can be shown.

The **ISyncMgrSynchronizeCallback** methods can be called on any thread in your application.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronize::Initialize, **ISyncMgrSynchronize::PrepareForSync**, **ISyncMgrSynchronize::SetProgressCallback**, **ISyncMgrSynchronize::ShowError**, **ISyncMgrSynchronizeCallback::SynchronizeCompleted**

ISyncMgrSynchronize::SetItemStatus

The Synchronization Manager calls the **ISyncMgrSynchronize::SetItemStatus** method in a registered application's handler to change the status of an item either between the time the handler has returned from the **ISyncMgrSynchronize::PrepareForSync** method and called the **ISyncMgrSynchronizeCallback::PrepareForSyncCompleted** callback method or after the handler has returned from the **ISyncMgrSynchronize::Synchronize** method but has not yet called the **ISyncMgrSynchronizeCallback::SynchronizeCompleted** callback method.

```
HRESULT SetItemStatus(
  REFSYNCMGRITEMID pItemID,     // Identifies the item with
                                // changed status
  DWORD dwSyncMgrStatus         // New status for item from
                                // SYNCMGRSTATUS
);
```

Parameters

pItemID
[in] Identifies the item with changed status.

dwSyncMgrStatus
[in] New status for the specified item taken from the SYNCMGRSTATUS enumeration.

Return Values

This method supports the standard return values, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the values on the next page.

S_OK
: Status was set.

Remarks

Currently, the only SYNCMGRSTATUS status value supported by the SyncMgr is SYNCMGRSTATUS_SKIPPED. The registered application's handler should skip the item specified in *pItemID* when it receives this status value.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronize::PrepareForSync, **ISyncMgrSynchronize::SetProgressCallback**, **ISyncMgrSynchronize::Synchronize**, **ISyncMgrSynchronizeCallback::PrepareForSyncCompleted**, **ISyncMgrSynchronizeCallback::SynchronizeCompleted**, SYNCMGRSTATUS

ISyncMgrSynchronize::ShowError

The Synchronization Manager calls the **ISyncMgrSynchronize::ShowError** method in a registered application's handler when the user double-clicks on the associated message in the error tab.

```
HRESULT ShowError(
  HWND hWndParent,        // Parent hWnd for user interface
  DWORD dwErrorID         // Error identifier
);
```

Parameters

hWndParent
: [in] Handle to the parent *hWndt* the registered application should use to display the user interface. This value may be NULL.

dwErrorID
: [in] Error identifier associated with this error message. The *ErrorID* value is passed in the **ISyncMgrSynchronizeCallback::LogError** method.

Return Values

This method supports the standard return values, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
  Call completed successfully.

Remarks

Handlers should return as soon as possible from this method and call the **ISyncMgrSynchronizeCallback::ShowErrorCompleted** method. It is acceptable for the handler to make a call to **ISyncMgrSynchronizeCallback::ShowErrorCompleted** before returning from this method. If a handler returns a failure code from this method, it should not call the **ISyncMgrSynchronizeCallback::ShowErrorCompleted** method.

Applications may display user interface elements in this method even if the **SYNCMGRFLAG_MAYBOTHERUSER** was not set in the *dwSyncFlags* parameter of the **ISyncMgrSynchronize::Initialize** method. Applications must still call **ISyncMgrSynchronizeCallback::EnableModeless** and check the return code before showing user interface.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronizeCallback::EnableModeless, **ISyncMgrSynchronizeCallback::LogError**, **ISyncMgrSynchronizeCallback::ShowErrorCompleted**, **ISyncMgrSynchronize::Initialize**, **SYNCMGRFLAG**

ISyncMgrSynchronizeCallback

The **ISyncMgrSynchronizeCallback** interface manages the synchronization process.

When to Implement

This interface is implemented by the Synchronization Manager to receive status, error, and progress information and display the user interface during the synchronization process.

When to Use

Applications should call the methods of this interface as often as possible and must call it before starting each new ItemID to check whether the user has canceled an individual item.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

ISyncMgrSynchronizeCallback methods	Description
Progress	Updates the progress information and determines if the operation should continue.
ShowPropertiesCompleted	Must by called by the handler before or after its **ShowProperties** method is complete.
PrepareForSyncCompleted	Must be called by the handler when its **PrepareForSync** method is complete.
SynchronizeCompleted	Must be called by the application when its **Synchronize** method is complete.
ShowErrorCompleted	Must be called by the handler before or after its **PrepareForSync** method is complete.
EnableModeless	Must be called by the application before and after any dialog boxes are displayed from within the **PrepareForSync** and **Synchronize** methods.
LogError	Called by the application to log an information, warning, or error message into the Error tab on the SyncMgr status dialog.
DeleteLogError	Called by the handler to delete a previously logged ErrorInformation, warning, or error message in the error tab on the SyncMgr status dialog box.
EstablishConnection	Called by the handler when it requires the system to establish a network a network connection.

! Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronize

ISyncMgrSynchronizeCallback::Progress

The registered application calls the **ISyncMgrSynchronizeCallback::Progress** method to update the progress information and determine whether the operation should continue.

```
HRESULT Progress(
  REFSYNCMGRITEMID pItemID,
            // Item identifier
  LPSYNCMGRPROGRESSITEM lpSyncProgressItem
            // SYNCMGRPROGRESSITEM structure
);
```

Parameters

pItemID
: [in] Item identifier for the item being updated.

lpSyncProgressItem
: [in] Pointer to a **SYNCMGRPROGRESSITEM** structure containing the updated progress information.

Return Values

S_OK
: Continues the synchronization.

S_SYNCMGR_CANCELITEM
: Cancel the synchronization on the specified *ItemID* as soon as possible.

S_SYNCMGR_CANCELALL
: Cancel the synchronization on all items associated with this application as soon as possible.

Remarks

Registered applications should call this to provide normal feedback even when the SYNCMGRFLAG_MAYBOTHERUSER flag has been set.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

SYNCMGRFLAG, **SYNCMGRPROGRESSITEM**

ISyncMgrSynchronizeCallback::ShowPropertiesCompleted

The registered application's handler must call the **ISyncMgrSynchronizeCallback::ShowPropertiesCompleted** method before or after its ShowProperties operation is completed.

```
HRESULT ShowPropertiesCompleted(
  HRESULT hrResult
);
```

Parameters

hrResult
[in] indicates whether the **ISyncMgrSynchronize::ShowProperties** was successful.

Return Values

S_OK
Call was completed successfully.

Remarks

It is acceptable for the registered application's handler to call this method before returning from the **ISyncMgrSynchronize::ShowProperties** method.

This method should not be called if the registered application's handler does not return a success code from the **ISyncMgrSynchronize::ShowProperties** method.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronize::ShowProperties

ISyncMgrSynchronizeCallback::PrepareForSyncCompleted

The registered application's handler must call the **ISyncMgrSynchronizeCallback::PrepareForSyncCompleted** method after the **ISyncMgrSynchronize::PrepareForSync** method has completed execution.

```
HRESULT PrepareForSyncCompleted(
  HRESULT hr      // HRESULT for the PrepareForSync method.
);
```

Parameters

hr

[in] Return value of the **ISyncMgrSynchronize::PrepareForSync** method. If S_OK is returned, SyncMgr calls **ISyncMgrSynchronize::Synchronize** for the item. If the HRESULT is set to anything other than S_OK, SyncMgr releases the handler without calling the **ISyncMgrSynchronize::Synchronize** method.

Return Values

S_OK

Call was completed successfully.

Remarks

A registered application's handler should return as soon as possible from the **ISyncMgrSynchronize::PrepareForSync** method and then call this method to notify the SyncMgr that the registered application is preparing for synchronization.

It is acceptable for the registered application's handler to call this method before returning from the **ISyncMgrSynchronize::PrepareForSync** method.

The registered application's handler should not call this method if the **ISyncMgrSynchronize::PrepareForSync** method returns any value other than S_OK.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronize::PrepareForSync, **ISyncMgrSynchronize::Synchronize**

ISyncMgrSynchronizeCallback::SynchronizeCompleted

The application must call the **ISyncMgrSynchronizeCallback::SynchronizeCompleted** method when its **ISyncMgrSynchronize::Synchronize** method has completed execution.

```
HRESULT SynchronizeCompleted(
  HRESULT hr
 // HRESULT from the ISyncMgrSynchronize::Synchronize method
);
```

Parameters

hr
[in] Returned result from the **ISyncMgrSynchronize::Synchronize** method.

Return Values

S_OK
Call was completed successfully.

Remarks

A registered application's handler should return from the **ISyncMgrSynchronize::Synchronize** method as soon as possible and then call this method to notify the SyncMgr that the synchronization process has completed.

It is acceptable for the registered application's handler to call this method before returning from the **ISyncMgrSynchronize::Synchronize** method.

The registered application's handler should not call this method if the **ISyncMgrSynchronize::Synchronize** method returns any value other than S_OK.

! Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

+ See Also

ISyncMgrSynchronize::Synchronize

ISyncMgrSynchronizeCallback::EnableModeless

The registered application must call the **ISyncMgrSynchronizeCallback::EnableModeless** method before and after any dialog boxes are displayed from within the **ISyncMgrSynchronize::PrepareForSync** and **ISyncMgrSynchronize::Synchronize** methods.

```
HRESULT EnableModeless(
  BOOL fEnable
);
```

Parameters

fEnable
[in] TRUE if the registered application is requesting permission to display a dialog box or FALSE if the registered application has finished displaying a dialog box.

Return Values

S_OK
Continue the synchronization.

S_FALSE
The dialog box should not be displayed.

Remarks

To request permission to display a dialog box, the registered application first calls **ISyncMgrSynchronize::EnableModeless** with *fEnable* set to TRUE. If S_OK is returned, the registered application may display the dialog box. Once the dialog box has been displayed, the registered application must call **ISyncMgrSynchronize::EnableModeless** with *fEnable* set to FALSE to notify SyncMgr that other items may now display user interface elements.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronize::PrepareForSync, **ISyncMgrSynchronize::Synchronize**

ISyncMgrSynchronizeCallback::LogError

The registered application calls the **ISyncMgrSynchronizeCallback::LogError** method by the to log an information, warning, or error message into the error tab on the Synchronization Manager status dialog box.

```
HRESULT LogError(
  [in] DWORD dwErrorLevel,   // Indicates the error level
  [in] const WCHAR *lpcErrorText,   // Error text
  [in] LPSYNCMGRLOGERRORINFO lpSyncLogError
                             // SYNCMGRLOGERRORINFO structure
);
```

Parameters

dwErrorLevel
[in] Indicates the error level. Values are taken from the SYNCMGRLOGLEVEL enumeration.

lpcErrorText
[in] Pointer to error text to be displayed in the Error tab.

lpSyncLogError
[in] Pointer to the **SYNCMGRLOGERRORINFO** structure containing additional error information. Registered applications that do not provide this data can pass NULL.

Return Values

This method supports the standard return values, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
The error information was successfully logged.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

SYNCMGRLOGLEVEL, **SYNCMGRLOGERRORINFO**

ISyncMgrSynchronizeCallback::ShowErrorCompleted

The registered application's handler must call the **ISyncMgrSynchronizeCallback::ShowErrorCompleted** method before or after its **ISyncMgrSynchronize::PrepareForSync** operation has been completed.

```
HRESULT ShowErrorCompleted(
  [in] HRESULT hrResult,
  [in] ULONG cbNumItems,
  [in] SYNCMGRITEMID *pItemIDs
);
```

Parameters

hrResult
: [in] indicates whether **ISyncMgrSynchronize::ShowError** was successful. This value is S_SYNCMGR_RETRYSYNC if the registered application's handler requires SyncMgr to retry the Synchronization. When this value is returned to SyncMgr both the **ISyncMgrSynchronize::PrepareForSync** and **ISyncMgrSynchronize::Synchronize** methods are called again.

cbNumItems
: [in] Indicates the number of *ItemIds* in the *pItemIDs* parameter. This parameter is ignored unless *hrResult* is S_SYNCMGR_RETRYSYNC.

pItemIDs
: [in] pointer to array of *ItemIds* to pass to **ISyncMgrSynchronize::PrepareForSync** in the event of a retry. This parameter is ignored unless *hrResult* is S_SYNCMGR_RETRYSYNC.

Return Values

S_OK
: The operation completed successfully.

Remarks

pItemIDs is an [in] parameter and the calling function owns the memory pointed to by *pItemIDs*. SyncMgr makes a copy of the array before returning.

The registered application's handler should return from the **ISyncMgrSynchronize::ShowError** method as soon as possible and then call this method to notify SyncMgr that it has completed processing the **ISyncMgrSynchronize::ShowError** call.

It is acceptable for the registered application's handler to call this method before returning from the **ISyncMgrSynchronize::ShowError** method.

The registered application's handler should not call this method unless a success code is returned from the **ISyncMgrSynchronize::Showerror** method.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronize::ShowError

ISyncMgrSynchronizeCallback::DeleteLogError

The registered application's handler calls the **ISyncMgrSynchronizeCallback::DeleteLogError** method to delete a previously logged ErrorInformation, warning, or error message in the error tab on the Synchronization Manager status dialog box.

```
HRESULT DeleteLogError(
  [in] REFSYNCMGRERRORID ErrorID,
  [in] DWORD dwReserved
);
```

Parameters

ErrorID
[in] Identifies LogError to be deleted. If *ErrorID* is GUID_NULL all errors logged by the instance of the registered application's handler will be deleted.

dwReserved
[in] Reserved for future use. Must be set to zero.

Return Values

This method supports the standard return values E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
The item was successfully deleted from the log.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

ISyncMgrSynchronizeCallback::EstablishConnection

The registered application's handler calls the **ISyncMgrSynchronizeCallback::EstablishConnection** method when a network connection is required.

```
HRESULT EstablishConnection(
  [in] WCHAR const *lpwszConnection,
  [in] DWORD dwReserved
);
```

Parameters

lpwszConnection
  [in] Identifies the name of the connection to dial.

dwReserved
  [in] Reserved for future use. Must be set to zero.

Return Values

This method supports the standard return values E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
  The connection was successfully established.

Remarks

SyncMgr should use the default autodial connection if *lpwszConnection* is NULL.

When an instance of **EstablishConnection** is called by a handler then SyncMgr tries to establish the connection If a subsequent **EstablishConnection** is called then SyncMgr attempts the new connection without hanging up the previous connection. All connections remain until all handlers have finished synchronizing. After all handlers have synchronized, then any opened connections are closed by SyncMgr.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

ISyncMgrSynchronizeInvoke

The **ISyncMgrSynchronizeInvoke** interface enables a registered application to invoke the Synchronization Manager to update items.

When to Implement

This interface is implemented by SyncMgr.

When to Use

A registered application calls the methods of this interface to update all items or to update specific items.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

ISyncMgrSynchronizeInvoke methods	Description
UpdateItems	Updates the specified items
UpdateAll	Updates all items

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

ISyncMgrSynchronizeInvoke::UpdateItems

The **ISyncMgrSynchronizeInvoke::UpdateItems** method programmatically starts an update for the specified items.

```
HRESULT UpdateItems(
  DWORD dwInvokeFlags,    // Indicates how to invoke the item
  REFCLSID rclsid,        // CLSID of application that
                          // handles the update
  DWORD cbCookie,         // Size in bytes of lpCookie
  const BYTE *lpCookie    // Token of calling application
);
```

Parameters

dwInvokeFlags
[in] Specifies how item should be invoked using the SYNCMGRINVOKEFLAGS enumeration values.

rclsid
[in] CLSID of the registered application that should be invoked to handle the Update.

cbCookie
[in] Size in bytes of *lpCookie* data.

lpCookie
[in] Pointer to the private token that SyncMgr uses to identify the application. This token is passed in the **ISyncMgrSynchronize::Initialize** method.

Return Values

This method supports the standard return values, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
The synchronization was successfully updated.

E_FAIL
Errors occurred during the synchronization update.

! Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

+ See Also

ISyncMgrSynchronize::Initialize, SYNCMGRINVOKEFLAGS

ISyncMgrSynchronizeInvoke::UpdateAll

The **ISyncMgrSynchronizeInvoke::UpdateAll** method programmatically starts an update for all items.

```
HRESULT UpdateAll( );
```

Return Values

S_OK
Call was completed successfully.

Remarks

This method returns immediately and the Synchronization Manager performs the synchronizations in a separate process from the calling application.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

ISyncMgrRegister

An application can register with the Synchronization Manager either through the **ISyncMgrRegister** interface or by registering directly in the registry.

The handler must be a standard OLE server. It must register the standard OLE keys for an InProc OLE server in addition to the SyncMgr key.

IUnknown methods	Description
QueryInterface	Returns pointer to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

ISyncMgrRegister methods	Description
RegisterSyncMgrHandler	A handler calls this to register with SyncMgr when it has items to synchronize
UnregisterSyncMgrHandler	A handler calls this to indicate it no longer has an items to synchronize
GetHandlerRegistrationInfo	Called by the handler to obtain registration information.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

ISyncMgrRegister::RegisterSyncMgrHandler

A handler should call the **ISyncMgrRegister::RegisterSyncMgrHandler** method to register with the Synchronization Manager when it has items to synchronize.

```
HRESULT RegisterSyncMgrHandler(
  [in] REFCLSID rclsid,
  [in] WCHAR const *pwsDescription,
  [in] DWORD dwReserved
);
```

Parameters

rclsid
[in] CLSID of the handler that should be registered to do synchronizations.

pwsDescription
[in] Text identifying the handler. This parameter may be NULL.

dwReserved
[in] Reserved for future use. Must be zero.

Return Values

This method supports the standard return values E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
The handler was successfully registered.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

ISyncMgrRegister::UnregisterSyncMgrHandler

The handler calls the **ISyncMgrRegister::UnregisterSyncMgrHandler** method to remove its CLSID from the registration. A handler should call this when it no longer has any items to synchronize.

```
HRESULT UnregisterSyncMgrHandler(
  [in] REFCLSID rclsidHandler,
  [in] DWORD dwReserved
);
```

Parameters

rclsidHandler
[in] CLSID of the handler that should be unregistered

dwReserved
[in] Reserved for future use. Must be zero.

Return Values

This method supports the standard return values E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
The handler was successfully removed from the registry with SyncMgr.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

ISyncMgrRegister::GetHandlerRegistrationInfo

The registered application's handler calls the **ISyncMgrRegister::GetHandlerRegistrationInfo** method to get current registration information.

```
HRESULT GetHandlerRegistrationInfo(
  [in] REFCLSID rclsidHandler,
  [in,out] LPDWORD pdwSyncMgrRegisterFlags
);
```

Parameters

rclsidHandler
[in] CLSID of the handler.

pdwSyncMgrRegisterFlags
[in,out] Returns registration flags.

Return Values

This method supports the standard return values E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK
Call was successful, the handler is registered.

S_FALSE
Call was not successful, the handler is not registered.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

SYNCMGRFLAG

The SYNCMGRFLAG enumeration values are used in the **ISyncMgrSynchronize::Initialize** method to indicate how the synchronization event was initiated.

```
typedef enum _tagSYNCMGRFLAG
   SYNCMGRFLAG_CONNECT                  = 0x0001,
   SYNCMGRFLAG_PENDINGDISCONNECT        = 0x0002,
   SYNCMGRFLAG_MANUAL                   = 0x0003,
   SYNCMGRFLAG_IDLE                     = 0x0004,
   SYNCMGRFLAG_INVOKE                   = 0x0005,
   SYNCMGRFLAG_SCHEDULED                = 0x0006,
   SYNCMGRFLAG_EVENTMASK                = 0x00FF,
   SYNCMGRFLAG_SETTINGS                 = 0x0100,
   SYNCMGRFLAG_MAYBOTHERUSER            = 0x0200,
} SYNCMGRFLAG;
```

Elements

SYNCMGRFLAG_CONNECT
Synchronization was initiated by a network connect event.

SYNCMGRFLAG_PENDINGDISCONNECT
Synchronization was initiated by a pending network disconnect event.

SYNCMGRFLAG_MANUAL
Synchronization was initiated manually by the end user.

SYNCMGRFLAG_IDLE
Synchronization was programmatically invoked.

SYNCMGRFLAG_INVOKE
Synchronization was programmatically invoked.

SYNCMGRFLAG_SCHEDULED
Synchronization was initiated by a scheduled update event.

SYNCMGRFLAG_EVENTMASK
Synchronization mask value.

SYNCMGRFLAG_SETTINGS
Synchronization was initiated for configuration purposes only in the System Properties dialog box.

SYNCMGRFLAG_MAYBOTHERUSER
Interaction with the user is permitted. The application is allowed to show user interface elements and interact with the user. If this flag is not set, the application must not display any user interface elements other than using the **ISynchronizeCallback** interface. If an application cannot complete the synchronization without displaying user interface elements and this flag is not set, the application fails the synchronization.

! Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

+ See Also

ISyncMgrSynchronize::Initialize

SYNCMGRHANDLERFLAGS

The **SYNCMGRHANDLERFLAGS** enumeration values are used in the **SYNCMGRHANDLERINFO** structure as flags that apply to the current handler.

```
typedef enum _tagSYNCMGRHANDLERFLAGS{
   SYNCMGRHANDLER_HASPROPERTIES    = 0x01,
} SYNCMGRHANDLERFLAGS;
```

Elements

SYNCMGRHANDLER_HASPROPERTIES
The current handler provides a Property Sheet dialog.

! Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

+ See Also

SYNCMGRHANDLERINFO

SYNCMGRSTATUS

The **SYNCMGRSTATUS** enumeration values are used in the **ISyncMgrSynchronize::SetItemStatus** method to specify the updated status for the item.

```
typedef enum _tagSYNCMGRSTATUS {
    SYNCMGRSTATUS_STOPPED       = 0x0000,
    SYNCMGRSTATUS_SKIPPED       = 0x0001,
    SYNCMGRSTATUS_PENDING       = 0x0002,
    SYNCMGRSTATUS_UPDATING      = 0x0003,
    SYNCMGRSTATUS_SUCCEEDED     = 0x0004,
    SYNCMGRSTATUS_FAILED        = 0x0005,
    SYNCMGRSTATUS_PAUSED        = 0x0006,
    SYNCMGRSTATUS_RESUMING      = 0x0007,
} SYNCMGRSTATUS;
```

Elements

SYNCMGRSTATUS_STOPPED
: Synchronization has been stopped.

SYNCMGRSTATUS_SKIPPED
: This item should be skipped.

SYNCMGRSTATUS_PENDING
: Synchronization for the item is pending.

SYNCMGRSTATUS_UPDATING
: The item is currently being synchronized.

SYNCMGRSTATUS_SUCCEEDED
: The synchronization for the item succeeded.

SYNCMGRSTATUS_FAILED
: The synchronization for the item failed.

SYNCMGRSTATUS_PAUSED
: The synchronization for the item paused.

SYNCMGRSTATUS_RESUMING
: The synchronization for the item is resuming.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronize::SetItemStatus

SYNCMGRLOGLEVEL

The **SYNCMGRLOGLEVEL** enumeration values specifies an error level for use in the **ISyncMgrSynchronizeCallback::LogError** method.

```
typedef enum _tagSYNCMGRLOGLEVEL {
    SYNCMGRLOGLEVEL_INFORMATION   = 0x0001,
    SYNCMGRLOGLEVEL_WARNING       = 0x0002,
    SYNCMGRLOGLEVEL_ERROR         = 0x0003,
} SYNCMGRLOGLEVEL;
```

Elements

SYNCMGRLOGLEVEL_INFORMATION
An information message was logged.

SYNCMGRLOGLEVEL_WARNING
A warning message was logged.

SYNCMGRLOGLEVEL_ERROR
An error message was logged.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronizeCallback::LogError

SYNCMGRITEMFLAGS

The **SYNCMGRITEMFLAGS** enumeration value specifies information for the current item in the **SYNCMGRITEM** structure.

```
typedef enum _tagSYNCMGRITEMFLAGS {
    SYNCMGRITEM_HASPROPERTIES  = 0x01,
    SYNCMGRITEM_TEMPORARY      = 0x02,
    SYNCMGRITEM_ROAMINGUSER    = 0x04,
} SYNCMGRITEMFLAGS;
```

Elements

SYNCMGRITEM_HASPROPERTIES
: The item has a properties dialog.

SYNCMGRITEM_TEMPORARY
: The item is temporary and any stored preferences can be removed.

SYNCMGRITEM_ROAMINGUSER
: The item roams with the user and is not specific to a machine.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

SYNCMGRITEM

SYNCMGRINVOKEFLAGS

The **SYNCMGRINVOKEFLAGS** enumeration value specifies how the SyncMgr is to be invoked in the **ISyncMgrSynchronizeInvoke::UpdateItems** method.

```
typedef enum _tagSYNCMGRINVOKEFLAGS {
    SYNCMGRINVOKE_STARTSYNC     = 0x02,
    SYNCMGRINVOKE_MINIMIZED     = 0x04,
} SYNCMGRINVOKEFLAGS;
```

Elements

SYNCMGRINVOKE_STARTSYNC
: Immediately start the synchronization without displaying the Choice dialog box.

SYNCMGRINVOKE_MINIMIZED
: The Choice dialog should be minimized by default.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronizeInvoke::UpdateItems

SYNCMGRHANDLERINFO

The **SYNCMGRHANDLERINFO** structure provides information about the handler for use in the **ISyncMgrSynchronize::GetHandlerInfo** method.

```
typedef struct _tagSYNCMGRHANDLERINFO {
    DWORD    cbSize;
    HICON    hIcon;
    DWORD    SyncMgrHandlerFlags;
    WCHAR    wszHandlerName[MAX_SYNCMGRHANDLERNAME];
} SYNCMGRHANDLERINFO, *LPSYNCMGRHANDLERINFO;
```

Members

cbSize
Size of the structure in bytes.

hIcon
Icon for the handler

SyncMgrHandlerFlags
Value of the **SYNCMGRHANDLERFLAGS** enumeration.

wszHandlerName[MAX_SYNCMGRHANDLERNAME]
Name to use for the handler.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronize::GetHandlerInfo, **SYNCMGRHANDLERFLAGS**

SYNCMGRPROGRESSITEM

The **SYNCMGRPROGRESSITEM** structure provides status information for the progress dialog for use in the **ISyncMgrSynchronizeCallback::Progress** method.

```
typedef struct _tagSYNCMGRPROGRESSITEM {
    DWORD          cbSize;
    UINT           mask;
    const WCHAR*   lpcStatusText;
    DWORD          dwStatusType;
    INT            iProgValue;
```

(continued)

(continued)

```
    INT             iMaxValue;
} SYNCMGRPROGRESSITEM, *LPSYNCMGRPROGRESSITEM;
```

Members

cbSize
Size of the structure in bytes.

mask
Mask value.

lpcStatusText
Status text.

dwStatusType
Status type.

iProgValue
Integer indicating the progress value.

iMaxValue
Integer indicating the maximum progress value.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronizeCallback::Progress

SYNCMGRLOGERRORINFO

The **SYNCMGRLOGERRORINFO** structure provides error information for use in the **ISyncMgrSynchronizeCallback::LogError** method.

```
typedef struct _tagSYNCMGRLOGERRORINFO {
    DWORD           cbSize;
    DWORD           mask;
    DWORD           dwErrorID;
    SYNCMGRITEMID   ItemID;
} SYNCMGRLOGERRORINFO, *LPSYNCMGRLOGERRORINFO;
```

Members

cbSize
Size of the structure.

mask
Mask value.

dwErrorID
Error identifier

ItemID
Item in which the error occurred.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

ISyncMgrSynchronizeCallback::LogError

SYNCMGRITEM

The **SYNCMGRITEM** structure provides information on the next item being enumerated in the **ISyncMgrEnumItems** interface.

```
typedef struct _tagSYNCMGRITEM {
    DWORD          cbSize;
    DWORD          dwFlags;
    SYNCMGRITEMID  ItemID;
    DWORD          dwItemState;
    HICON          hIcon;
    WCHAR          wszItemName[MAX_SYNCMGRITEMNAME];
    WCHAR          wszStatus[MAX_SYNCMGRITEMSTATUS];
} SYNCMGRITEM, *LPSYNCMGRITEM;
```

Members

cbSize
Size of the structure.

dwFlags
Value of the **SYNCMGRITEMFLAGS** enumeration.

ItemID
Identifier of the next item.

dwItemState
State of the next item.

hIcon
Icon for the handler for the next item.

wszItemName[MAX_SYNCMGRITEMNAME]
Item name.

wszStatus[MAX_SYNCMGRITEMSTATUS]
Item status.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Mobsync.h.

See Also

SYNCMGRITEMFLAGS

CHAPTER 12

System Event Notification Service

The System Event Notification Service (SENS) creates a uniform connectivity and notification interface for applications designed for mobile use.

For more information, System Event Notification Service Overview provides an overview, and SENS Reference lists methods and interfaces used for SENS.

System Event Notification Service Overview

Applications designed for use by mobile users require a unique set of connectivity functions and notifications, and in the past individual applications were required to implement these features internally. The System Event Notification Service (SENS) now provides these capabilities in the operating system, creating a uniform connectivity and notification interface for applications. SENS, the Synchronization Manager, and Client Side Caching combine to provide the infrastructure to fully support mobile computing.

The topics in this section offer an overview about SENS:

- Mobile Computing Configurations for SENS

  Lists several mobile computer configurations that can benefit from the use of SENS-enabled applications.
- Application Scenarios for SENS

  Lists several types of applications that can benefit by utilizing SENS functionality.
- Notifications

  Lists the system events that SENS monitors and dispatches.
- SENS Architecture

  A description of the internal architecture of SENS.

Mobile Computing Configurations for SENS

Connectivity functions and notifications are useful for computers configured as follows:

- A mobile computer used in a docking station on a high bandwidth network which may occasionally use a dial-in connection.
- A mobile computer using a dial-in connection exclusively.
- A desktop computer using a dial-in connection exclusively.
- A desktop computer connected to a high bandwidth network with latency issues.

In each of these configurations, the connection bandwidth and latency information can be used by an application to dynamically optimize its operations for network availability.

Application Scenarios for SENS

Several types of applications can utilize the connectivity functions and notification services that the System Event Notification Service offers:

- An application that requires network connectivity status, such as an application that utilizes directory services.
- An application that adapts its operations depending on the level of connectivity and the quality of network services, such as an Internet browser application that functions at a reduced level on a low bandwidth connection.
- An application that can perform deferred operations such as an e-mail program that can queue messages while offline and send them when a connection is established.

Notifications

The System Event Notification Service enables mobile-aware applications to receive notifications from system events that SENS monitors. When the requested event occurs, SENS notifies the application.

SENS can notify applications about three classes of system events:

- TCP/IP network events, such as the status of a TCP/IP network connection or the quality of the connection.
- User logon events.
- Battery and AC power events.

For example, an application can subscribe to any of the following system events:

- Establishment of network connectivity
- Notification when a specified destination can be reached within specified Quality of Connection (QOC) parameters
- The computer has switched to battery power
- The percentage of remaining battery power is within a specified parameter
- Scheduled events using Synchronization Manager occur

SENS Architecture

The System Event Notification Service works with the COM+ Event System. SENS is an event publisher for the classes of events that it monitors: network, logon, and power/battery events. The application receiving a notification is called an event subscriber.

When an application subscribes to receive notifications, it can also specify filters associated with the subscribed events. SENS and COM+ Events use the filters to further determine when the application should be notified.

Notifications are asynchronous, so the application receiving the notification does not have to be active when the notification is sent. When an application subscribes to receive notifications, it can specify whether it should be activated when the event occurs or notified later when it is active.

The subscription can be transient and valid only until the application stops running, or it can be persistent and valid until the application is removed from the system.

A COM+ Events data store contains information about the event publisher (SENS), event subscribers, and filters. During setup when you install or upgrade to Microsoft® Windows® 2000, SENS adds itself to the COM+ Events data store and provides information on the classes of events that it monitors using a GUID for each class of events. SENS also predefines an outgoing interface for each event class in a type library.

Event Class	GUID	Interface
Network events	SENSGUID_EVENTCLASS_NETWORK	ISensNetwork
Logon events	SENSGUID_EVENTCLASS_LOGON	ISensLogon
Power events	SENSGUID_EVENTCLASS_ONNOW	ISensOnNow

To receive notifications for any of these events, your application must do two things:

- Subscribe to the SENS events that interest you. To subscribe to an event, use the **IEventSubscription** and **IEventSystem** interfaces in COM+ Events. You need to supply an identifier for the event classes and the SENS publisher identifier, SENSGUID_PUBLISHER. Subscriptions are on a per event level so the subscribing application must also specify which events within the class are of interest. Each event corresponds to a method in the interface corresponding to its event class.

  Note Programmers using SENS on the Internet Explorer 5 platform should use only the **IEventSubscription** and **IEventSystem** COM+ Events interfaces.

- Create a sink object with an implementation for each interface that you handle. See ISensNetwork, ISensLogon, and ISensOnNow for more information about these interfaces and the events supported in each one.

When one of the monitored events occurs, SENS processes each subscription with any associated filters and notifies the subscribers through the COM+ Event system.

SENS Reference

This section lists the following methods for the System Event Notification Service (SENS):

- **IsDestinationReachable**
- **IsNetworkAlive**

The following interfaces are supported by the SENS object:

- **ISensLogon**
- **ISensNetwork**
- **ISensOnNow**

IsDestinationReachable

Determines if the specified destination can be reached and provides Quality of Connection (QOC) information for the destination.

```
Bool IsDestinationReachable(
  LPCSTR lpszDestination,     // Pointer to string
                              // specifying destination
  LPQOCINFO lpQOCInfo         // Pointer to Quality of
                              // Connection information
);
```

Parameters

lpszDestination
Pointer to a string that specifies the destination. The destination can be an IP address, a UNC name, or an URL.

lpQOCInfo
Pointer to the **QOCINFO** structure that receives the Quality of Connection (QOC) information. You can supply a NULL pointer if the QOC information is not desired.

Return Values

TRUE
The destination can be reached.

FALSE
Call **GetLastError** to determine the reason why the destination cannot be reached.

Remarks

This function is used by client applications to determine the QOC information before proceeding with network operations. For standalone computers that are directly connected to the network through a network card or Remote Access Server (RAS), this function generates minimal network traffic with RPC calls to the nearest router. For computers that are part of a network where the destination can be reached using RAS or a network gateway, this function pings to the destination to generate accurate QOC information.

Note This function is only available for TCP/IP connections.

The caller supplies the buffer for the **QOCINFO** structure and must release this memory when it is no longer needed.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Sensapi.h.
Library: Use Sensapi.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

IsNetworkAlive, **QOCINFO**, About System Event Notification Service

QOCINFO

The **QOCINFO** structure is returned by the **IsDestinationReachable** function and provides Quality of Connection information to the caller.

```
typedef struct tagQOCINFO {
    DWORD     dwSize;
    DWORD     dwFlags;
    DWORD     dwInSpeed;
    DWORD     dwOutSpeed;
} QOCINFO, *LPQOCINFO;
```

Members

dwSize

Upon calling the **IsDestinationReachable** function, the caller supplies the size of the **QOC** structure in this member. On return from the function, this member contains the actual size of the structure that was filled in.

dwFlags

Speed of connection. Flag bits indicate whether the connection is slow, medium, fast.

dwInSpeed

Speed of data coming in from the destination in bytes per second.

dwOutSpeed

Speed of data sent to the destination in bytes per second.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).

See Also

IsDestinationReachable

IsNetworkAlive

Determines whether the local system is connected to a network and the type of network connection, for example, LAN, WAN, or both.

```
Bool IsNetworkAlive(
  LPDWORD lpdwFlags     // Specifies the type of
                        // network connection
);
```

Parameters

lpdwFlags

Provides information on the type of network connection available when the return value is TRUE. The flags can be:

NETWORK_ALIVE_LAN

The computer has one or more LAN cards that are active.

NETWORK_ALIVE_WAN

The computer has one or more active RAS connections.

NETWORK_ALIVE_AOL

This flag is only valid in Windows 95 and Windows 98. Indicates the computer is connected to the America Online network.

Return Values

TRUE

The local system is connected to a network.

FALSE

Call **GetLastError** to determine the reason for no connectivity.

Remarks

This function can be used by applications to determine whether there is network connectivity before proceeding with network operations. Applications such as directory service applications, e-mail clients, or Internet browsers can adapt to various types of network connectivity. For example, a printing operation can be deferred until the network connection is available.

Note This function is only available for TCP/IP connections.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Sensapi.h.
Library: Use Sensapi.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

IsDestinationReachable, About System Event Notification Service

SENS Object

The System Event Notification Service (SENS) defines the SENS coclass as part of the SENS type library.

Implementation

The SENS object implementation is provided by the operating system.

Creation/Access Functions

Function	Description
CoCreateInstance	Creates an instance of the SENS object using its CLSID.

Interfaces

Interface	Description
IsensNetwork	Default. Outgoing interface implemented by sink object in subscriber application.
IsensOnNow	Outgoing interface implemented by sink object in subscriber application.
IsensLogon	Outgoing interface implemented by sink object in subscriber application.

See Also

ISensLogon, **ISensNetwork**, **ISensOnNow**, About System Event Notification Service

ISensLogon

The **ISensLogon** interface handles logon events fired by SENS.

When to Implement

Implement this interface on your sink object if you subscribe to any of the SENS logon events. Each event corresponds to a method in this interface. This interface is an outgoing interface defined by SENS and implemented by the subscriber application as a dispatch interface.

When to Use

SENS and the COM Event System call the **ISensLogon** methods on your sink object to fire the corresponding event.

Methods in Vtable Order

Iunknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

Idispatch methods	Description
GetTypeInfoCount	Retrieves the number of type descriptions.
GetTypeInfo	Retrieves a description of the object's programmable interface.
GetIDsOfNames	Maps name of method or property to DISPID.
Invoke	Calls one of the object's methods, or gets/sets one of its properties.

ISensILogon methods	Description
Logon	A user has logged on.
Logoff	A user has logged off.
StartShell	Shell has been started.
DisplayLock	Screen display has been locked.
DisplayUnlock	Screen display has been unlocked.

ISensILogon methods	Description
StartScreenSaver	Screen saver has been started.
StopScreenSaver	Screen saver has been stopped.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensNetwork, **ISensOnNow**, About System Event Notification Service

ISensLogon::Logon

A user has logged on.

```
HRESULT Logon(
  BSTR bstrUserName    // User name
);
```

Parameters

bstrUserName
[in] Name of the user who logged on.

Dispatch Identifier

[id(0x00000001)]

Return Values

S_OK
Method returned successfully.

Remarks

SENS calls this method to notify your application that a user has now logged on.

Filtering

Filtering is not currently supported for this event.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensLogon::Logoff, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensLogon::Logoff

A user has logged off.

```
HRESULT Logoff(
  BSTR bstrUserName    // User name
);
```

Parameters

bstrUserName
[in] Name of the user who logged off.

Dispatch Identifier

[id(0x00000002)]

Return Values

S_OK
Method returned successfully.

Remarks

SENS calls this method to notify your application that a user has logged off.

Filtering

Filtering is not currently supported for this event.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensLogon::Logon, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensLogon::StartShell

Shell has been started.

```
HRESULT StartShell(
  BSTR bstrUserName    // User name
);
```

Parameters

bstrUserName

[in] Name of the current user.

Dispatch Identifier

[id(0x00000004)]

Return Values

S_OK

Method returned successfully.

Remarks

SENS calls this method to notify your application that the shell has been started.

Filtering

Filtering is not currently supported for this event.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensLogon::DisplayLock

Screen display has been locked.

```
HRESULT DisplayLock(
  BSTR bstrUserName    // User name
);
```

Parameters

bstrUserName
[in] Name of the current user.

Dispatch Identifier

[id(0x00000006)]

Return Values

S_OK
Method returned successfully.

Remarks

SENS calls this method to notify your application that the screen display has been locked.

Filtering

Filtering is not currently supported for this event.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensLogon::DisplayUnLock, **ISensLogon::StartScreenSaver**, **ISensLogon::StopScreenSaver**, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensLogon::DisplayUnLock

Screen display has been unlocked.

```
HRESULT DisplayUnLock(
  BSTR bstrUserName    // User name
);
```

Parameters

bstrUserName
[in] Name of the current user.

Dispatch Identifier

[id(0x00000007)]

Return Values

S_OK
Method returned successfully.

Remarks

SENS calls this method to notify your application that the screen display has been unlocked.

Filtering

Filtering is not currently supported for this event.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensLogon::DisplayLock, **ISensLogon::StartScreenSaver**, **ISensLogon::StopScreenSaver**, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensLogon::StartScreenSaver

Screen saver has been started.

```
HRESULT StartScreenSaver(
  BSTR bstrUserName    // User name
);
```

Parameters

bstrUserName
[in] Name of the current user.

Dispatch Identifier

[id(0x00000008)]

Return Values

S_OK
Method returned successfully.

Remarks

SENS calls this method to notify your application that the screen saver has been started.

Filtering

Filtering is not currently supported for this event.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensLogon::DisplayLock, **ISensLogon::DisplayUnLock**, **ISensLogon::StopScreenSaver**, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensLogon::StopScreenSaver

Screen saver has been stopped.

```
HRESULT StopScreenSaver(
  BSTR bstrUserName    // User name
);
```

Parameters

bstrUserName
[in] Name of the current user.

Dispatch Identifier

[id(0x00000009)]

Return Values

S_OK
Method returned successfully.

Remarks

SENS calls this method to notify your application that the screen saver has been stopped.

Filtering

Filtering is not currently supported for this event.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensLogon::DisplayLock, **ISensLogon::DisplayUnLock**, **ISensLogon::StartScreenSaver**, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensNetwork

The **ISensNetwork** interface handles network events fired by the System Event Notification Service (SENS).

When to Implement

Implement this interface on your sink object if you subscribe to any of the SENS network events. Each event corresponds to a method in this interface. This interface is an outgoing interface defined by SENS and implemented by the subscriber application as a dispatch interface.

When to Use

SENS and the COM Event System call the **ISensNetwork** methods on your sink object to fire the corresponding event.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IDispatch methods	Description
GetTypeInfoCount	Retrieves the number of type descriptions.
GetTypeInfo	Retrieves a description of the object's programmable interface.
GetIDsOfNames	Maps name of method or property to DISPID.
Invoke	Calls one of the object's methods, or gets/sets one of its properties.

ISensNetwork methods	Description
ConnectionMade	Specified connection has been established.
ConnectionMadeNoQOCInfo	Specified connection has been established with no Quality of Connection information available.
ConnectionLost	Specified connection has been dropped.
DestinationReachable	Specified connection can be reached.
DestinationReachableNoQOCInfo	Specified connection can be reached with no Quality of Connection information.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensLogon, **ISensOnNow**, About System Event Notification Service

ISensNetwork::ConnectionMade

Specified connection has been established.

```
HRESULT ConnectionMade(
  BSTR bstrConnection,        // Connection name
  ULONG ulType,               // Connection type
  LPSENS_QOCINFO lpQOCInfo    // Pointer to Quality of
                              // Connection information
);
```

Parameters

bstrConnection
[in] Name of the connection. For WAN connections, the connection name is the name of the phone book entry; for LAN connections, it is the name of the network card.

ulType
[in] Connection type. This value can be CONNECTION_LAN or CONNECTION_WAN.

lpQOCInfo
[out] Pointer to the **SENS_QOCINFO** structure which contains Quality of Connection information.

Dispatch Identifier

[id(0x00000001)]

Return Values

S_OK
Method returned successfully.

Remarks

SENS calls this method to notify your application that the specified connection has been established. SENS also provides a pointer to a structure containing Quality of Connection information.

type

Connection type. Use 0 for LAN or 1 for WAN.

Note This function is only available for TCP/IP connections.

Filtering

Filtering can be performed on the publisher property ulConnectionMadeType by setting it to either CONNECTION_LAN or CONNECTION_WAN or both.
Use **IEventSubscription::PutPublisherProperty** to set the publisher property.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensNetwork::ConnectionMadeNoQOCInfo, **SENS_QOCINFO**, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensNetwork::ConnectionMadeNoQOCInfo

Specified connection has been established with no Quality of Connection information available.

```
HRESULT ConnectionMadeNoQOCInfo(
  BSTR bstrConnection,     // Connection name
  ULONG ulType             // Connection type
);
```

Parameters

bstrConnection

[in] Name of the connection. For WAN connections, the connection name is the name of the phone book entry; for LAN connections, it is the name of the network card.

ulType

[in] Connection type. This value can be CONNECTION_LAN or CONNECTION_WAN.

Dispatch Identifier

[id(0x00000002)]

Return Values

S_OK
  Method returned successfully.

Remarks

SENS calls this method to notify your application that the specified connection has been established when Quality of Connection information is not available.

type
  Connection type. Use 0 for LAN or 1 for WAN.

Note This function is only available for TCP/IP connections.

Filtering

Filtering can be performed on the publisher property ulConnectionMadeTypeNoQOC by setting it to either CONNECTION_LAN or CONNECTION_WAN or both.
Use **IEventSubscription::PutPublisherProperty** to set the publisher property.

! Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

+ See Also

ISensNetwork::ConnectionMade, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensNetwork::ConnectionLost

Specified connection has been dropped.

```
HRESULT ConnectionLost(
  BSTR bstrConnection,    // Connection name
  ULONG ulType            // Connection type
);
```

Parameters

bstrConnection

[in] Name of the connection. For WAN connections, the connection name is the name of the phone book entry; for LAN connections, it is the name of the network card.

ulType

[in] Connection type. This value can be CONNECTION_LAN or CONNECTION_WAN.

Dispatch Identifier

[id(0x00000003)]

Return Values

S_OK

Method returned successfully.

Remarks

SENS calls this method to notify your application that the specified connection has been dropped.

type

Connection type. Use 0 for LAN or 1 for WAN.

Note This function is only available for TCP/IP connections.

Filtering

Filtering can be performed on the publisher property ulConnectionLostType by setting it to either CONNECTION_LAN or CONNECTION_WAN or both.
Use **IEventSubscription::PutPublisherProperty** to set the publisher property.

! Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

+ See Also

About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensNetwork::DestinationReachable

Specified connection can be reached.

```
HRESULT DestinationReachable(
  BSTR bstrDestination,          // Destination name
  BSTR bstrConnection,           // Connection name
  ULONG ulType,                  // Connection type
  LPSENS_QOCINFO lpQOCInfo       // Quality of Connection
                                 // information
);
```

Parameters

bstrDestination
[in] Name of the destination. Can be an IP address, a URL, a UNC, or a NetBIOS name.

bstrConnection
[in] Name of the connection. For WAN connections, the connection name is the name of the phone book entry; for LAN connections, it is the name of the network card.

ulType
[in] Connection type. This value can be CONNECTION_LAN or CONNECTION_WAN.

lpQOCInfo
[out] Pointer to the **SENS_QOCINFO** structure which contains Quality of Connection information.

Dispatch Identifier

[id(0x00000004)]

Return Values

S_OK
Method returned successfully.

Remarks

SENS calls this method to notify your application that the specified destination can be reached. A pointer to a structure containing Quality of Connection information is also provided.

Note This function is only available for TCP/IP connections.

Filtering

Filtering can be performed on the publisher property *bstrDestination*. To determine reachability, set *bstrDestination* to the name of desired destination. Filtering can also be performed on the property *ulType* by setting it to either CONNECTION_LAN or CONNECTION_WAN, or both. Use **IEventSubscription::PutPublisherProperty** to set the publisher property. Note: if a *bstrDestination* property is not specified, the DestinationReachable event is returned for all destinations.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensNetwork::DestinationReachableNoQOCInfo, **SENS_QOCINFO**, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensNetwork::DestinationReachableNoQOCInfo

Specified connection can be reached with no Quality of Connection information.

```
HRESULT DestinationReachableNoQOCInfo(
  BSTR bstrDestination,    // Destination name
  BSTR bstrConnection,     // Connection name
  ULONG ulType             // Connection type
);
```

Parameters

bstrDestination
[in] Name of the destination. Can be an IP address, a URL, a UNC, or a NetBIOS name.

bstrConnection
[in] Name of the connection. For WAN connections, the connection name is the name of the phone book entry; for LAN connections, it is the name of the network card.

ulType
[in] Connection type. This value can be CONNECTION_LAN or CONNECTION_WAN.

Dispatch Identifier

[id(0x00000005)]

Return Values

S_OK
  Method returned successfully.

Remarks

SENS calls this method to notify your application that the specified destination can be reached when Quality of Connection information is not available.

Note This function is only available for TCP/IP connections.

Filtering

Filtering can be performed on the publisher property *bstrDestinationNoQOC*. To determine reachability, set *bstrDestinationNoQOC* to the name of desired destination. Filtering can also be performed on the property *ulDestinationTypeNoQOC* by setting it to either CONNECTION_LAN or CONNECTION_WAN, or both. Use **IEventSubscription::PutPublisherProperty** to set the publisher property. Note: if a *bstrDestinationNoQOC* property is not specified, the DestinationReachableNoQOC() event is returned for all destinations.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensNetwork::DestinationReachable, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

SENS_QOCINFO

The **SENS_QOCINFO** structure is provided by the **ISensNetwork::ConnectionMade** method and the **ISensNetwork::DestinationReachable** method. This structure contains Quality of Connection information to the sink object in an application that subscribes to SENS.

```
typedef struct _SENS_QOCINFO {
  DWORD dwSize;
  DWORD dwFlags;
```

(continued)

(continued)

```
    DWORD dwOutSpeed;
    DWORD dwInSpeed;
} SENS_QOCINFO;
```

Members

dwSize

This member contains the actual size of the structure that was filled in.

dwFlags

Speed of connection. Flag bits indicate whether the connection is slow, medium, fast.

dwOutSpeed

Speed of data sent to the destination in bits per second.

dwInSpeed

Speed of data coming in from the destination in bits per second.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 5 or later).
Windows 95/98: Requires Windows 95 or later (with Internet Explorer 5 or later).

See Also

ISensNetwork::ConnectionMade, **ISensNetwork::DestinationReachable**, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensOnNow

The **ISensOnNow** interface handles AC and battery power events fired by the System Event Notification Service (SENS).

When to Implement

Implement this interface on your sink object if you subscribe to any of the SENS power events. Each event corresponds to a method in this interface. This interface is an outgoing interface defined by SENS and implemented by the subscriber application as a dispatch interface.

When to Use

SENS and the COM Event System call the **ISensOnNow** methods on your sink object to fire the corresponding event.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IDispatch methods	Description
GetTypeInfoCount	Retrieves the number of type descriptions.
GetTypeInfo	Retrieves a description of the object's programmable interface.
GetIDsOfNames	Maps name of method or property to DISPID.
Invoke	Calls one of the object's methods, or gets/sets one of its properties.

ISensOnNow methods	Description
OnACPower	Switched to AC power.
OnBatteryPower	Switched to Battery power.
BatteryLow	Battery power is low.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

+ See Also

ISensLogon, **ISensNetwork**, About System Event Notification Service

ISensOnNow::OnACPower

SENS calls this method to notify your application that the computer is using AC power.

```
HRESULT OnACPower(void);
```

Dispatch Identifier

[id(0x00000001)]

Return Values

S_OK
  Method returned successfully.

Remarks

SENS calls this method to notify your application that AC power has been activated.

Filtering

Filtering is not currently supported for this event.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensOnNow::OnBatteryPower, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensOnNow::OnBatteryPower

SENS calls this method to notify your application that the computer is using battery power.

```
HRESULT OnBatteryPower(
  DWORD dwBatteryLifePercent    // Percent of battery power
                                // remaining
);
```

Parameters

dwBatteryLifePercent
  [in] Specifies the percent of battery power remaining.

Dispatch Identifier

[id(0x00000002)]

Return Values

S_OK
  Method returned successfully.

Remarks

SENS calls this method to notify your application that the computer is using battery power. The remaining percentage of battery power is specified.

Filtering

Filtering is not currently supported for this event.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensOnNow::BatteryLow, **ISensOnNow::OnACPower**, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

ISensOnNow::BatteryLow

Battery power is low.

```
HRESULT BatteryLow(
  DWORD dwBatteryLifePercent    // Percent of battery power
                                // remaining
);
```

Parameters

dwBatteryLifePercent
[in] Specifies the percent of battery power remaining.

Dispatch Identifier

[id(0x00000003)]

Return Values

S_OK
Method returned successfully.

Remarks

SENS calls this method to notify your application that the computer is using battery power. The remaining percentage of battery power is specified.

Filtering

Filtering is not currently supported for this event.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Sensevts.h.
Library: Use Sensevts.tlb.

See Also

ISensOnNow::OnBatteryPower, About System Event Notification Service, **IEventSubscription**, **IEventSubscription::PutPublisherProperty**

CHAPTER 13

IP Helper

IP Helper Overview

Internet Protocol Helper (IP Helper) is an API that assists in the network administration of the local computer. You can use IP Helper to programmatically retrieve information about the network configuration of the local computer, and to modify that configuration. IP Helper also provides notification mechanisms to ensure that an application is signaled when certain aspects of the network configuration change on the local computer.

Many of the IP Helper functions pass structure parameters that represent data types seen in the *Management Information Base* technology. These data types are also used by the MIB API, and are described in the *Management Information Base Reference*.

IP Helper provides capabilities in the following areas:

- Retrieving Information About Network Configuration
- Managing Network Adapters
- Managing Interfaces
- Managing IP Addresses
- Using the Address Resolution Protocol
- Retrieving Information on the Internet Protocol and the Internet Control Message Protocol
- Managing Routing
- Receiving Notification of Network Events
- Retrieving Information About the Transmission Control Protocol and the User Datagram Protocol

Retrieving Information About Network Configuration

IP Helper provides information about the network configuration of the local computer. To retrieve general configuration information, use the **GetNetworkParams** function. This function returns information that is not specific to a particular adapter or interface. For example, **GetNetworkParams** returns a list of the DNS servers that are used by the local computer.

Managing Network Adapters

IP Helper provides capabilities for managing network adapters. The functions described following are used to retrieve information about the network adapters in the local computer.

The **GetAdaptersInfo** function returns an array of **IP_ADAPTER_INFO** structures, one for each adapter in the local computer. The **GetPerAdapterInfo** function returns additional information about a specific adapter. The **GetPerAdapterInfo** function requires the caller to specify the index of the adapter. To obtain the adapter index from the adapter name, use the **GetAdapterIndex** function.

Some applications use adapters that can receive datagrams, but cannot transmit them. To obtain information about such adapters, use the **GetUniDirectionalAdapterInfo** function.

Managing Interfaces

IP Helper extends your abilities to manage network interfaces. Use the functions described following to manage interfaces on the local computer.

Interfaces are related to adapters in that there is a one-to-one correspondence between the interfaces and adapters on a given computer. An interface is an IP-level abstraction, whereas an adapter is a datalink-level abstraction.

The **GetNumberOfInterfaces** function returns the number of interfaces on the local computer.

The **GetInterfaceInfo** function returns a table that contains the names and corresponding indexes for the interfaces on the local computer.

The **GetFriendlyIfIndex** function takes an interface index and returns a backward-compatible interface index, that is, one that uses only the lower 24 bits. This type of index is sometimes referred to as a "friendly" interface index.

The **GetIfEntry** function returns a **MIB_IFROW** structure that contains information about a particular interface on the local computer. This function requires the caller to supply the index of the interface.

The **GetIfTable** function returns a table of **MIB_IFROW** entries, one for each interface on the computer.

Use the **SetIfEntry** function to modify the configuration of a particular interface.

Managing IP Addresses

IP Helper can assist you in managing IP addresses that are associated with interfaces on the local computer. Use the functions described following for IP address management.

The **GetIpAddrTable** function retrieves a table that contains the mapping of IP addresses to interfaces. More than one IP address may to associated with the same interface.

Use the **AddIPAddress** function to add an IP address to a particular interface. To remove IP addresses that were previously added using **AddIPAddress**, use the **DeleteIPAddress** function.

The **IpReleaseAddress** and **IpRenewAddress** functions require the local computer to be using Dynamic Host Configuration Protocol (DHCP). The **IpReleaseAddress** function releases an IP address that was previously obtained from DHCP. The **IpRenewAddress** function renews a DHCP lease on a particular IP address.

Using the Address Resolution Protocol

You can use IP Helper to perform Address Resolution Protocol (ARP) operations for the local computer. Use the following functions to retrieve and modify the ARP table.

The **GetIpNetTable** retrieves the ARP table. The ARP table contains the mapping of IP addresses to physical addresses. Physical addresses are sometimes referred to as Media Access Controller (MAC) addresses.

Use the **CreateIpNetEntry** and **DeleteIpNetEntry** functions to add or remove particular ARP entries to or from the table. The **FlushIpNetTable** function deletes all entries from the table.

To create or delete proxy ARP entries, use the **CreateProxyArpEntry** and **DeleteProxyArpEntry** functions.

The **SendARP** function sends an ARP request to the local network.

Retrieving Information on the Internet Protocol and the Internet Control Message Protocol

IP Helper provides information retrieval capabilities that are useful for the network administration of the local computer. The following functions retrieve statistics for the Internet Protocol (IP) and the Internet Control Message Protocol (ICMP). You can also use these functions to set certain configuration parameters for IP.

The **GetIpStatistics** function retrieves the current IP statistics for the local machine. The **GetIcmpStatistics** function retrieves the current ICMP statistics.

Use the **SetIpStatistics** function to enable or disable IP forwarding. This function also makes it possible for you to set the default Time-To-Live (TTL) for IP datagrams. Alternatively, you can set the TTL by using the **SetIpTTL** function.

Managing Routing

IP Helper provides features to manage network routing. Use the following functions to manage the IP routing table, and to obtain other routing information.

You can manipulate specific entries in the IP routing table. Use the **CreateIpForwardEntry** function to add a new routing table entry. Use the **DeleteIpForwardEntry** function to remove an existing entry. The **SetIpForwardEntry** function modifies an existing entry. You can retrieve the contents of the IP routing table by making a call to the **GetIpForwardTable** function.

You can also use the router management capabilities of IP Helper to retrieve information about how datagrams are routed over the network. The **GetBestRoute** function retrieves the best route to a specified destination address. The **GetBestInterface** function retrieves the index of the interface used by the best route to a specified destination address. Lastly, the **GetRTTAndHopCount** function retrieves the Round-Trip Time (RTT) and number of hops to a specified destination address.

Receiving Notification of Network Events

Use the following functions to ensure that an application receives notification of certain network events.

The **NotifyAddrChange** function enables an application to request notification of any change that occurs in the table that maps IP addresses to interfaces on the local computer.

Similarly, the **NotifyRouteChange** function enables an application to request notification of any change that occurs in the IP routing table.

The notifications provided by these functions do not specify what changed. They simply specify that something changed. Use other IP Helper functions to determine the exact nature of the change.

Retrieving Information About the Transmission Control Protocol and the User Datagram Protocol

IP Helper makes it possible to access information about network protocols that are used on the local computer. Use the functions described following to retrieve information about the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) on the local computer.

The **GetTcpStatistics** function retrieves the current statistics for TCP. Similarly, the **GetUdpStatistics** function retrieves the current statistics for UDP.

The **GetTcpTable** function retrieves the TCP connection table. The **GetUdpTable** retrieves the UDP listener table.

The **SetTcpEntry** function enables a developer to set the state of a specified TCP connection to MIB_TCP_STATE_DELETE_TCB.

IP Helper Function Reference

Use the following functions and structures to retrieve and modify configuration settings for the Transmission Control Protocol/Internet Protocol (TCP/IP) transport suite on the local computer:

- IP Helper Functions
- IP Helper Structures
- IPX Service Table Management

IP Helper Functions

Use the following functions to retrieve and modify configuration settings for the TCP/IP transport suite on the local computer. These functions are declared in IpHlpApi.h.

Alphabetical Listing

AddIPAddress
CreateIpForwardEntry
CreateIpNetEntry
CreateProxyArpEntry
DeleteIPAddress
DeleteIpForwardEntry
DeleteIpNetEntry
DeleteProxyArpEntry
FlushIpNetTable
GetAdapterIndex
GetAdaptersInfo
GetBestInterface
GetBestRoute
GetFriendlyIfIndex
GetIcmpStatistics
GetIfEntry
GetIfTable
GetInterfaceInfo
GetIpAddrTable
GetIpForwardTable
GetIpNetTable
GetIpStatistics
GetNetworkParams
GetNumberOfInterfaces
GetPerAdapterInfo
GetRTTAndHopCount
GetTcpStatistics
GetTcpTable
GetUdpStatistics
GetUdpTable
GetUniDirectionalAdapterInfo
IpReleaseAddress
IpRenewAddress
NotifyAddrChange
NotifyRouteChange
SendARP
SetIfEntry
SetIpForwardEntry
SetIpNetEntry
SetIpStatistics
SetIpTTL
SetTcpEntry

Categorical Listing

Adapter Management

GetAdapterIndex
GetAdaptersInfo
GetPerAdapterInfo
GetUniDirectionalAdapterInfo

Address Resolution Protocol

CreateIpNetEntry
CreateProxyArpEntry
DeleteIpNetEntry
DeleteProxyArpEntry
FlushIpNetTable
GetIpNetTable
SendARP
SetIpNetEntry

Interface Management

GetFriendlyIfIndex
GetIfEntry
GetIfTable
GetInterfaceInfo
GetNumberOfInterfaces
SetIfEntry

Internet Protocol and Internet Control Message Protocol

GetIcmpStatistics
GetIpStatistics
SetIpStatistics
SetIpTTL

IP Address Management

AddIPAddress
DeleteIPAddress
GetIpAddrTable
IpReleaseAddress
IpRenewAddress

Network Configuration

GetNetworkParams

Notification

NotifyAddrChange

NotifyRouteChange

Routing

CreateIpForwardEntry

DeleteIpForwardEntry

GetBestInterface

GetBestRoute

GetIpForwardTable

GetRTTAndHopCount

SetIpForwardEntry

Transmission Control Protocol and User Datagram Protocol

GetTcpStatistics

GetTcpTable

SetTcpEntry

GetUdpStatistics

GetUdpTable

AddIPAddress

The **AddIPAddress** function adds the specified IP address to the specified adapter.

```
DWORD AddIPAddress(
  IPAddr Address,          // IP address to add
  IPMask IpMask,           // subnet mask for IP address
  DWORD IfIndex,           // index of adapter
  PULONG NTEContext,       // Net Table Entry context
  PULONG NTEInstance       // Net Table Entry Instance
);
```

Parameters

Address

Specifies the IP address to add to the adapter.

IpMask

Specifies the subnet mask for the IP address.

IfIndex
Specifies the adapter to which to add the address.

NTEContext
Pointer to a **ULONG** variable that, on successful return, points to the Net Table Entry (NTE) context for this IP address. The caller can later use this context in a call to **DeleteIPAddress**.

NTEInstance
Pointer to a **ULONG** variable that, on successful return, points to the NTE instance for this IP address.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Remarks

For information about the **IPAddr** and **IPMask** data types, see *Win32 Simple Data Types*. To convert an IP address between dotted decimal notation and **IPAddr** format, use the **inet_addr** and **inet_ntoa** functions.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

DeleteIPAddress, **GetAdapterIndex**

CreateIpForwardEntry

The **CreateIpForwardEntry** function creates a route in the local computer's IP routing table.

```
DWORD CreateIpForwardEntry(
  PMIB_IPFORWARDROW pRoute    // pointer to route information
);
```

Parameters

pRoute

Pointer to a **MIB_IPFORWARDROW** structure that specifies the information for the new route. The caller must specify values for all members of this structure. The caller must specify PROTO_IP_NETMGMT for the **dwForwardProto** member of **MIB_IPFORWARDROW**.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes.

Value	Meaning
ERROR_INVALID_PARAMETER	The *pRoute* parameter is NULL, or **SetIpFowardEntry** is unable to read from the memory pointed to by *pRoute*, or one of the members of the **MIB_IPFORWARDROW** structure is invalid.
ERROR_NOT_SUPPORTED	The IP transport is not configured on the local computer.
Other	Use **FormatMessage** to obtain the message string for the returned error.

Remarks

To modify an existing route in the IP routing table, use the **SetIpForwardEntry** function.

The caller should not specify a routing protocol—for example, PROTO_IP_OSPF—for the **dwForwardProto** member of the **MIB_IPFORWARDROW** structure. Routing protocol identifiers are used only to identify route information received through the specified routing protocol. For example, PROTO_IP_OSPF is used only to identify route information received through the OSPF routing protocol.

The **dwForwardPolicy** member of the **MIB_IPFORWARDROW** structure is currently unused. The caller should specify zero for this member.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

DeleteIpForwardEntry, **MIB_IPFORWARDROW**, **SetIpForwardEntry**

CreateIpNetEntry

The **CreateIpNetEntry** function creates an Address Resolution Protocol (ARP) entry in the ARP table on the local computer.

```
DWORD CreateIpNetEntry(
  PMIB_IPNETROW pArpEntry   // pointer to info for new entry
);
```

Parameters

pArpEntry

Pointer to a **MIB_IPNETROW** structure that specifies information for the new entry. The caller must specify values for all members of this structure.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

DeleteIpNetEntry, **MIB_IPNETROW**, **SetIpNetEntry**

CreateProxyArpEntry

The **CreateProxyArpEnry** function creates a Proxy Address Resolution Protocol (PARP) entry on the local computer for the specified IP address.

```
DWORD CreateProxyArpEntry(
  DWORD dwAddress,    // IP address for which to act as proxy
  DWORD dwMask,       // subnet mask for IP address
  DWORD dwIfIndex     // interface on which to proxy
);
```

Parameters

dwAddress

Specifies the IP address for which this computer acts as a proxy.

dwMask
Specifies the subnet mask for the IP address specified by the *dwAddress* parameter.

dwIfIndex
Specifies the index of the interface on which to proxy ARP for the IP address specified by the *dwAddress* parameter. In other words, when an ARP request for *dwAddress* is received on this interface, the local computer responds with the physical address of this interface. If this interface is of a type that does not support ARP, such as PPP, then the call fails.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

DeleteProxyArpEntry, **MIB_PROXYARP**

DeleteIPAddress

Use the **DeleteIPAddress** to delete an IP address that was previously added using **AddIPAddress**.

```
DWORD DeleteIPAddress(
  ULONG NTEContext     // net table entry context
);
```

Parameters

NTEContext
Specifies the Net Table Entry (NTE) context for the IP address. This context was returned by the previous call to **AddIPAddress**.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

AddIPAddress

DeleteIpForwardEntry

The **DeleteIpForwardEntry** function deletes an existing route in the local computer's IP routing table.

```
DWORD DeleteIpForwardEntry(
  PMIB_IPFORWARDROW pRoute   // pointer to route information
);
```

Parameters

pRoute

Pointer to a **MIB_IPFORWARDROW** structure. This structure specifies information that identifies the route to delete. The caller must specify values for the **dwForwardIfIndex**, **dwForwardDest**, **dwForwardMask**, **dwForwardNextHop**, and **dwForwardPolicy** members of the structure.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Remarks

The **dwForwardPolicy** member of the **MIB_IPFORWARDROW** structure is currently unused. The caller should specify zero for this member.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

CreateIpForwardEntry, **MIB_IPFORWARDROW**, **SetIpForwardEntry**

DeleteIpNetEntry

The **DeleteIpNetEntry** function deletes an ARP entry from the ARP table on the local computer.

```
DWORD DeleteIpNetEntry(
  PMIB_IPNETROW pArpEntry    // info identifying entry
                             // to delete
);
```

Parameters

pArpEntry

Pointer to a **MIB_IPNETROW** structure. The information in this structure identifies the entry to delete. The caller must specify values for at least the **dwIndex** and **dwAddr** members of this structure.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

CreateIpNetEntry, **MIB_IPNETROW**, **SetIpNetEntry**

DeleteProxyArpEntry

The **DeleteProxyArpEntry** function deletes the PARP entry on the local computer specified by the *dwAddress* and *dwIfIndex* parameters.

```
DWORD DeleteProxyArpEntry(
  DWORD dwAddress,    // IP address for which to act as proxy
```

```
  DWORD dwMask,       // subnet mask for IP address
  DWORD dwIfIndex     // interface on which to proxy
);
```

Parameters

dwAddress
Specifies the IP address for which this computer is acting as a proxy.

dwMask
Specifies the subnet mask for the IP address specified by the *dwAddress* parameter.

dwIfIndex
Specifies the index of the interface on which this computer is supporting proxy ARP for the IP address specified by *dwAddress*.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

CreateProxyArpEntry, **MIB_PROXYARP**

FlushIpNetTable

The **FlushIpNetTable** function deletes all ARP entries for the specified interface from the ARP table on the local computer.

```
DWORD FlushIpNetTable(
  DWORD dwIfIndex   // delete ARP entries for this interface
);
```

Parameters

dwIfIndex
Specifies the index of the interface for which to delete all ARP entries.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetIfTable, **GetIpNetTable**

GetAdapterIndex

The **GetAdapterIndex** function obtains the index of an adapter, given its name.

```
DWORD GetAdapterIndex(
  LPWSTR AdapterName,     // name of the adapter
  PULONG IfIndex          // index of the adapter
);
```

Parameters

AdapterName
Pointer to a Unicode string that contains the name of the adapter.

IfIndex
Pointer to a **ULONG** variable that, on successful return, points to the index of the adapter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

MprConfigGetFriendlyName, **MprConfigGetGuidName**

GetAdaptersInfo

The **GetAdaptersInfo** function retrieves adapter information for the local computer.

```
DWORD GetAdaptersInfo(
  PIP_ADAPTER_INFO pAdapterInfo,    // buffer to receive data
  PULONG pOutBufLen                 // size of data returned
);
```

Parameters

pAdapterInfo

Pointer to a buffer that, on successful return, receives a linked list of **IP_ADAPTER_INFO** structures.

pOutBufLen

Pointer to a **ULONG** variable that contains the size of the buffer pointed to by the *pAdapterInfo* parameter. If this size is insufficient to hold the adapter information, **GetAdaptersInfo** fills in this variable with the required size, and returns an error code of ERROR_BUFFER_OVERFLOW.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is one of the following error codes.

Value	Meaning
ERROR_BUFFER_OVERFLOW	The buffer size indicated by the *pOutBufLen* parameter is too small to hold the adapter information. The *pOutBufLen* parameter points to the required size.
ERROR_INVALID_PARAMETER	The *pOutBufLen* parameter is NULL, or the calling process does not have read/write access to the memory pointed to by *pOutBufLen*, or the calling process does not have write access to the memory pointed to by the *pAdapterInfo* parameter.
ERROR_NO_DATA	No adapter information exists for the local computer.
ERROR_NOT_SUPPORTED	**GetAdaptersInfo** is not supported by the operating system running on the local computer.
Other	If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

IP_ADAPTER_INFO

GetBestInterface

The **GetBestInterface** function retrieves the index of the interface that has the best route to the specified IP address.

```
DWORD GetBestInterface(
  IPAddr dwDestAddr,       // destination IP address
  PDWORD pdwBestIfIndex    // index of interface with
                           // the best route
);
```

Parameters

dwDestAddr
Specifies the destination IP address for which to retrieve the interface that has the best route.

pdwBestIfIndex
Pointer to a **DWORD** variable. On successful return, this variable contains the index of the interface that has the best route to the address specified by the *dwDestAddr* parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Remarks

For information about the **IPAddr** data type, see *Win32 Simple Data Types*. To convert an IP address between dotted decimal notation and **IPAddr** format, use the **inet_addr** and **inet_ntoa** functions.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetBestRoute, **MIB_BEST_IF**

GetBestRoute

The **GetBestRoute** function retrieves the best route to the specified destination IP address.

```
DWORD GetBestRoute(
  DWORD dwDestAddr,              // destination IP address
  DWORD dwSourceAddr,            // local source IP address
  PMIB_IPFORWARDROW pBestRoute // best route for dest. addr.
);
```

Parameters

dwDestAddr
Specifies the destination IP address for which to obtain the best route.

dwSourceAddr
Specifies a source IP address. This IP address corresponds to an interface on the local computer. If multiple best routes to the destination address exist, the function selects the route that uses this interface.

This parameter is optional. The caller may specify zero for this parameter.

pBestRoute
Pointer to a **MIB_IPFORWARDROW** structure. On successful return, this structure contains the best route for the IP address specified by *dwDestAddr*.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetBestInterface, **MIB_IPFORWARDROW**

GetFriendlyIfIndex

The **GetFriendlyIfIndex** function takes an interface index and returns a backward-compatible interface index, that is, one that uses only the lower 24 bits.

```
DWORD GetFriendlyIfIndex(
  DWORD IfIndex     // interface index
);
```

Parameters

IfIndex

Specifies an interface index from which the backward-compatible or "friendly" interface index is derived.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetIfEntry, **MIB_IFROW**

GetIcmpStatistics

The **GetIcmpStatistics** function retrieves the Internet Control Message Protocol (ICMP) statistics for the local computer.

```
DWORD GetIcmpStatistics(
  PMIB_ICMP pStats    // pointer to ICMP stats
);
```

Parameters

pStats

Pointer to a **MIB_ICMP** structure that, on successful return, contains the ICMP statistics for the local computer.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetIpStatistics, **GetTcpStatistics**, **GetUdpStatistics**, **MIB_ICMP**

GetIfEntry

The **GetIfEntry** function retrieves information for the specified interface on the local computer.

```
DWORD GetIfEntry(
  PMIB_IFROW pIfRow    // pointer to interface entry
);
```

Parameters

pIfRow

Pointer to a **MIB_IFROW** structure that, on successful return, contains information for an interface on the local computer. Set the **dwIndex** member of **MIB_IFROW** to the index of the interface for which to retrieve information.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetNumberOfInterfaces, **MIB_IFROW**

GetIfTable

The **GetIfTable** function retrieves the MIB-II interface table.

```
DWORD GetIfTable(
  PMIB_IFTABLE pIfTable,    // buffer for interface table
  PULONG pdwSize,           // size of buffer
  BOOL bOrder               // sort the table by index?
);
```

Parameters

pIfTable
Pointer to a buffer that, on successful return, contains the interface table as a **MIB_IFTABLE** structure.

pdwSize
Specifies the size of the buffer pointed to by the *pIfTable* parameter. If the buffer is not large enough to hold the returned interface table, the function sets this parameter equal to the required buffer size.

bOrder
Specifies whether the returned interface table should be sorted in ascending order by interface index. If this parameter is TRUE, the table is sorted.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetIfEntry, **GetNumberOfInterfaces**, **MIB_IFTABLE**

GetInterfaceInfo

The **GetInterfaceInfo** function obtains a list of the network interface adapters on the local system.

```
DWORD GetInterfaceInfo(
  PIP_INTERFACE_INFO pIfTable,      // buffer to receive info
  PULONG dwOutBufLen                // size of buffer
);
```

Parameters

pIfTable

Pointer to a buffer that receives an **IP_INTERFACE_INFO** structure that contains the list of adapters. This buffer should be allocated by the caller.

dwOutBufLen

Pointer to a **DWORD** variable. If the buffer pointed to by the *pIfTable* parameter is NULL, or is not large enough to contain the list of adapters, **GetInterfaceInfo** returns the required size in this **DWORD** variable.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes.

Value	Meaning
ERROR_INVALID_PARAMETER	The *dwOutBufLen* parameter is NULL, or **GetInterfaceInterface** is unable to write to the memory pointed to by the *dwOutBufLen* parameter.
ERROR_INSUFFICIENT_BUFFER	The buffer pointed to by the pIfTable parameter is not large enough. The required size is returned in the **DWORD** variable pointed to by the *dwOutBufLen* parameter.

Value	Meaning
ERROR_NOT_SUPPORTED	This function is not supported on the operating system in use on the local system.
Other	Use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

IP_INTERFACE_INFO

GetIpAddrTable

The **GetIpAddrTable** function retrieves the interface-to-IP address mapping table.

```
DWORD GetIpAddrTable(
  PMIB_IPADDRTABLE pIpAddrTable, // buffer for mapping table
  PULONG pdwSize,                // size of buffer
  BOOL bOrder                    // sort the table
);
```

Parameters

pIpAddrTable

Pointer to a buffer that, on successful return, contains the interface-to-IP address mapping table as a **MIB_IPADDRTABLE** structure.

pdwSize

Specifies the size of the buffer pointed to by the *pIpAddrTable* parameter. If the buffer is not large enough to hold the returned mapping table, the function sets this parameter equal to the required buffer size.

bOrder

Specifies whether the returned mapping table should be sorted in ascending order by IP address. If this parameter is TRUE, the table is sorted.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

MIB_IPADDRTABLE

GetIpForwardTable

The **GetIpForwardTable** function retrieves the IP routing table.

```
DWORD GetIpForwardTable(
  PMIB_IPFORWARDTABLE pIpForwardTable,  // buffer for
                                        //routing table
  PULONG pdwSize,                       // size of buffer
  BOOL bOrder                           // sort the table?
);
```

Parameters

pIpForwardTable

Pointer to a buffer that, on successful return, contains the IP routing table as a **MIB_IPFORWARDTABLE** structure.

pdwSize

Specifies the size of the buffer pointed to by the *pIpForwardTable* parameter. If the buffer is not large enough to hold the returned routing table, the function sets this parameter equal to the required buffer size.

bOrder

Specifies whether the returned table should be sorted. If this parameter is TRUE, the table is sorted in order of:

1. Destination address
2. Protocol that generated the route
3. Multipath routing policy
4. Next-hop address

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

MIB_IPFORWARDTABLE

GetIpNetTable

The **GetIpNetTable** function retrieves the IP-to-physical address mapping table.

```
DWORD GetIpNetTable(
  PMIB_IPNETTABLE pIpNetTable    // buffer for mapping table
  PULONG pdwSize,                // size of buffer
  BOOL bOrder                    // sort by IP address
);
```

Parameters

pIpNetTable
Pointer to a buffer that, on successful return, contains the IP-to-physical address mapping table as a **MIB_IPNETTABLE** structure.

pdwSize
Specifies the size of the buffer pointed to by the *pIpNetTable* parameter. If the buffer is not large enough to hold the returned mapping table, the function sets this parameter equal to the required buffer size.

bOrder
Specifies whether the returned mapping table should be sorted in ascending order by IP address. If this parameter is TRUE, the table is sorted.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

MIB_IPNETTABLE

GetIpStatistics

The **GetIpStatistics** function retrieves the IP statistics for the current computer.

```
DWORD GetIpStatistics(
  PMIB_IPSTATS pStats    // pointer to IP stats
);
```

Parameters

pStats
Pointer to a **MIB_IPSTATS** structure that, on successful return, contains the IP statistics for the local computer.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetIcmpStatistics, **GetTcpStatistics**, **GetUdpStatistics**, **MIB_IPSTATS**

GetNetworkParams

The **GetNetworkParams** function retrieves network parameters for the local computer.

```
DWORD GetNetworkParams(
  PFIXED_INFO pFixedInfo,   // pointer to buffer to
                            // receive data
  PULONG pOutBufLen         // size of buffer
);
```

Parameters

pFixedInfo

Pointer to a **FIXED_INFO** structure that, on successful return, contains the network parameters for the local computer.

pOutBufLen

Pointer to a **ULONG** variable that specifies the size of the **FIXED_INFO** structure. If this size is insufficient to hold the information, **GetNetworkParams** fills in this variable with the required size, and returns an error code of ERROR_BUFFER_OVERFLOW.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is one of the following error codes.

Value	Meaning
ERROR_BUFFER_OVERFLOW	The buffer size indicated by the *pOutBufLen* parameter is too small to hold the adapter information. The *pOutBufLen* parameter points to the required size.
ERROR_INVALID_PARAMETER	The *pOutBufLen* parameter is NULL, or the calling process does not have read/write access to the memory pointed to by *pOutBufLen,* or the calling process does not have write access to the memory pointed to by the *pAdapterInfo* parameter.
ERROR_NO_DATA	No adapter information exists for the local computer.
ERROR_NOT_SUPPORTED	**GetNetworkParams** is not supported by the operating system running on the local computer.
Other	If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

FIXED_INFO

GetNumberOfInterfaces

The **GetNumberOfInterfaces** functions retrieves the number of interfaces on the local computer.

```
DWORD GetNumberOfInterfaces(
  PDWORD pdwNumIf    // pointer to number of interfaces
);
```

Parameters

pdwNumIf

Pointer to a **DWORD** variable that, on successful return, contains the number of interfaces on the local computer.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetIfEntry

GetPerAdapterInfo

The **GetPerAdapterInfo** function retrieves information about the adapter corresponding to the specified interface.

```
DWORD GetPerAdapterInfo(
  ULONG IfIndex,           // index of the interface
  PIP_PER_ADAPTER_INFO pPerAdapterInfo,
                           // buffer to  receive info
  PULONG pOutBufLen        // size of buffer to receive info
);
```

Parameters

IfIndex

Specifies the index of an interface. **GetPerAdapterInfo** will retrieve information for the adapter corresponding to this interface.

pPerAdapterInfo

Pointer to an **IP_PER_ADAPTER_INFO** structure that, on successful return, contains information about the adapter.

pOutBufLen

Pointer to a **ULONG** variable that specifies the size of the **IP_PER_ADAPTER_INFO** structure. If this size is insufficient to hold the information, **GetPerAdapterInfo** fills in this variable with the required size, and returns an error code of ERROR_BUFFER_OVERFLOW.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is one of the following error codes.

Value	Meaning
ERROR_BUFFER_OVERFLOW	The buffer size indicated by the *pOutBufLen* parameter is too small to hold the adapter information. The *pOutBufLen* parameter points to the required size.
ERROR_INVALID_PARAMETER	The *pOutBufLen* parameter is NULL, or the calling process does not have read/write access to the memory pointed to by *pOutBufLen,* or the calling process does not have write access to the memory pointed to by the *pAdapterInfo* parameter.
ERROR_NOT_SUPPORTED	**GetPerAdapterInfo** is not supported by the operating system running on the local computer.
Other	If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

IP_PER_ADAPTER_INFO

GetRTTAndHopCount

The **GetRTTAndHopCount** function determines the round-trip time (RTT) and hop count to the specified destination.

```
BOOL GetRTTAndHopCount(
  IPAddr DestIpAddress, // destination IP address
  PULONG HopCount,      // returned hop count
  ULONG MaxHops,        // limit on number of hops to search
  PULONG RTT            // round-trip time
);
```

Parameters

DestIpAddress
Specifies the IP address of the destination for which to determine the RTT and hop count.

HopCount
Pointer to a **ULONG** variable. On successful return, this variable contains the hop count to the destination specified by the *DestIpAddress* parameter.

MaxHops
Specifies the maximum number of hops to search for the destination. If the number of hops to the destination exceeds this number, the function terminates the search and returns FALSE.

RTT
Round-trip time in milliseconds to the destination specified by *DestIpAddress*.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call **GetLastError** to obtain the error code for the failure.

Remarks

For information about the **IPAddr** data type, see *Types*. To convert an IP address between dotted decimal notation and **IPAddr** format, use the **inet_addr** and **inet_ntoa** functions.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetBestInterface, **GetBestRoute**

GetTcpStatistics

The **GetTcpStatistics** functions retrieves the TCP statistics for the local computer.

```
DWORD GetTcpStatistics(
  PMIB_TCPSTATS pStats    // pointer to TCP stats
);
```

Parameters

pStats

Pointer to a **MIB_TCPSTATS** structure that, on successful return, contains the TCP statistics for the local computer.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetIcmpStatistics, **GetIpStatistics**, **GetUdpStatistics**, **MIB_TCPSTATS**

GetTcpTable

The **GetTcpTable** function retrieves the TCP connection table.

```
DWORD GetTcpTable(
  PMIB_TCPTABLE pTcpTable,    // buffer for the
                              // connection table
  PDWORD pdwSize,             // size of the buffer
  BOOL bOrder                 // sort the table?
);
```

Parameters

pTcpTable

Pointer to a buffer that, on successful return, contains the TCP connection table as a **MIB_TCPTABLE** structure.

pdwSize

Specifies the size of the buffer pointed to by the *pTcpTable* parameter. If the buffer is not large enough to hold the returned connection table, the function sets this parameter equal to the required buffer size.

bOrder

Specifies whether the connection table should be sorted. If this parameter is TRUE, the table is sorted in order of:

1. Local IP address
2. Local port
3. Remote IP address
4. Remote port

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

MIB_TCPTABLE

GetUdpStatistics

The **GetUdpStatistics** function retrieves the User Datagram Protocol (UDP) statistics for the local computer.

```
DWORD GetUdpStatistics(
  PMIB_UDPSTATS pStats    // pointer to UDP stats
);
```

Parameters

pStats

Pointer to a **MIB_UDPSTATS** structure that, on successful return, contains the UDP statistics for the local computer.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

GetIcmpStatistics, **GetIpStatistics**, **GetTcpStatistics**, **MIB_UDPSTATS**

GetUdpTable

The **GetUdpTable** function retrieves the User Datagram Protocol (UDP) listener table.

```
DWORD GetUdpTable(
  PMIB_UDPTABLE pUdpTable,  // buffer for the listener table
  PDWORD pdwSize,           // size of buffer
  BOOL bOrder               // sort the table?
);
```

Parameters

pUdpTable

Pointer to a buffer that, on successful return, contains the UDP listener table as a **MIB_UDPTABLE** structure.

pdwSize

Specifies the size of the buffer pointed to by the *pUdpTable* parameter. If the buffer is not large enough to hold the returned listener table, the function sets this parameter equal to the required buffer size.

bOrder

Specifies whether the returned table should be sorted. If this parameter is TRUE, the table is sorted in order of:

1. IP address
2. Port

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

MIB_UDPTABLE

GetUniDirectionalAdapterInfo

The **GetUniDirectionalAdapterInfo** function retrieves information about the unidirectional adapters installed on the local computer. A unidirectional adapter is an adapter that can receive datagrams, but not transmit them.

```
DWORD GetUniDirectionalAdapterInfo(
  OUT PIP_UNIDIRECTIONAL_ADAPTER_ADDRESS pIPIfInfo,
  OUT PULONG dwOutBufLen
);
```

Parameters

pIPIfInfo

Pointer to an **IP_UNIDIRECTIONAL_ADAPTER_ADDRESS** structure that receives information about the unidirectional adapters installed on the local computer.

dwOutBufLen

Pointer to a **ULONG** variable that, on successful return, contains the size of the structure pointed to by the *pIPIfInfo* parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Unsupported.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

IP_UNIDIRECTIONAL_ADAPTER_ADDRESS

IpReleaseAddress

The **IpReleaseAddress** function releases an IP address previously obtained through Dynamic Host Configuration Protocol (DHCP).

```
DWORD IpReleaseAddress(
  PIP_ADAPTER_INDEX_MAP AdapterInfo
            // identifies the adapter
);
```

Parameters

AdapterInfo
Pointer to an **IP_ADAPTER_INDEX_MAP** structure that identifies the adapter associated with the IP address to release.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

IpRenewAddress

IpRenewAddress

The **IpRenewAddress** function renews a lease on an IP address previously obtained through Dynamic Host Configuration Protocol (DHCP).

```
DWORD IpRenewAddress(
  PIP_ADAPTER_INDEX_MAP AdapterInfo
          // identifies the adapter
);
```

Parameters

AdapterInfo
Pointer to an **IP_ADAPTER_INDEX_MAP** structure that identifies the adapter associated with the IP address to renew.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

IpReleaseAddress

NotifyAddrChange

The **NotifyAddrChange** function causes a notification to be sent to the caller whenever a change occurs in the table that maps IP addresses to interfaces.

```
DWORD NotifyAddrChange(
  OUT PHANDLE Handle,
  IN LPOVERLAPPED overlapped
);
```

Parameters

Handle
Pointer to a HANDLE variable that receives a handle to use in asynchronous notification.

overlapped

Pointer to an **OVERLAPPED** structure that will notify the caller of any changes in the table that maps IP addresses to interfaces.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Remarks

If the caller specifies NULL for the *Handle* and *overlapped* parameters, the call to **NotifyAddrChange** blocks until an IP address change occurs.

If the caller specifies a handle variable and an **OVERLAPPED** structure, the caller can use the returned handle with the **OVERLAPPED** structure to receive asynchronous notification of IP address changes. See ***GetQueuedCompletionStatus*** and the *I/O Completion Ports* overview for information about using the handle and **OVERLAPPED** structure to receive notifications.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

+ See Also

NotifyRouteChange, **OVERLAPPED**

NotifyRouteChange

The **NotifyRouteChange** function causes a notification to be sent to the caller whenever a change occurs in the IP routing table.

```
DWORD NotifyRouteChange(
  OUT PHANDLE Handle,
  IN LPOVERLAPPED overlapped
);
```

Parameters

Handle

Pointer to a HANDLE variable that receives a handle to use in asynchronous notification.

overlapped

Pointer to an **OVERLAPPED** structure that will notify the caller of any changes in the routing table.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Remarks

If the caller specifies NULL for the *Handle* and *overlapped* parameters, the call to **NotifyRouteChange** blocks until a routing table change occurs.

If the caller specifies a handle variable and an **OVERLAPPED** structure, the caller can use the returned handle with the **OVERLAPPED** structure to receive asynchronous notification of routing table changes. See ***GetQueuedCompletionStatus*** and the *I/O Completion Ports* overview for information about using the handle and **OVERLAPPED** structure to receive notifications.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

NotifyAddrChange, **OVERLAPPED**

SendARP

The **SendARP** function sends an ARP request to obtain the physical address that corresponds to the specified destination IP address.

```
DWORD SendARP(
  IPAddr DestIP,      // destination IP address
  IPAddr SrcIP,       // IP address of sender
  PULONG pMacAddr,    // returned physical address
  PULONG PhyAddrLen   // length of returned physical addr.
);
```

Parameters

DestIP

Specifies the destination IP address. The ARP request attempts to obtain the physical address that corresponds to this IP address.

SrcIP

Specifies the IP address of the sender. This parameter is optional. The caller may specify zero for the parameter.

pMacAddr

Pointer to a **ULONG** variable. On successful return, this variable contains the physical address that corresponds to the IP address specified by the *DestIP* parameter.

PhyAddrLen

Pointer to a **ULONG** variable. On successful return, this variable contains the length of the physical address pointed to by the *pMacAddr* parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Remarks

For information about the **IPAddr** data type, see *Win32 Simple Data Types*. To convert an IP address between dotted decimal notation and **IPAddr** format, use the **inet_addr** and **inet_ntoa** functions.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

+ See Also

CreateIpNetEntry, **DeleteIpNetEntry**, **FlushIpNetTable**, **SetIpNetEntry**

SetIfEntry

Use the **SetIfEntry** function to set the administrative status of an interface.

```
DWORD SetIfEntry(
  PMIB_IFROW pIfRow    // specifies interface and status
);
```

Parameters

pIfRow

Pointer to a **MIB_IFROW** structure. The **dwIndex** member of this structure should specify the interface on which to set administrative status. The **dwAdminStatus** member specifies the new administrative status. The **dwAdminStatus** member can be one of the following values.

Value	Meaning
MIB_IF_ADMIN_STATUS_UP	The interface is administratively enabled.
MIB_IF_ADMIN_STATUS_DOWN	The interface is administratively disabled.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

MIB_IFROW

SetIpForwardEntry

The **SetIpForwardEntry** function modifies an existing route in the local computer's IP routing table.

```
DWORD SetIpForwardEntry(
  PMIB_IPFORWARDROW pRoute   // pointer to route information
);
```

Parameters

pRoute

Pointer to a **MIB_IPFORWARDROW** structure that specifies the new information for the existing route. The caller must specify PROTO_IP_NETMGMT for the **dwForwardProto** member of this structure. The caller must also specify values for the **dwForwardIfIndex**, **dwForwardDest**, **dwForwardMask**, **dwForwardNextHop**, and **dwForwardPolicy** members of the structure.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes.

Value	Meaning
ERROR_INVALID_PARAMETER	The *pRoute* parameter is NULL, or **SetIpFowardEntry** is unable to read from the memory pointed to by *pRoute*, or one of the members of the **MIB_IPFORWARDROW** structure is invalid.
ERROR_NOT_SUPPORTED	The IP transport is not configured on the local computer.
Other	Use **FormatMessage** to obtain the message string for the returned error.

Remarks

To create a new route in the IP routing table, use the **CreateIpForwardEntry** function.

The caller should not specify a routing protocol, such as PROTO_IP_OSPF, for the **dwForwardProto** member of the **MIB_IPFORWARDROW** structure. Routing protocol identifiers are used to identify route information received through the specified routing protocol only. For example, PROTO_IP_OSPF is used to identify route information received through the OSPF routing protocol only.

The **dwForwardPolicy** member of the **MIB_IPFORWARDROW** structure is currently unused. The caller should specify zero for this member.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

+ See Also

CreateIpForwardEntry, **DeleteIpForwardEntry**, **MIB_IPFORWARDROW**

SetIpNetEntry

The **SetIpNetEntry** function modifies an existing ARP entry in the ARP table on the local computer.

```
DWORD SetIpNetEntry(
  PMIB_IPNETROW pArpEntry    // pointer to new information
);
```

Parameters

pArpEntry

Pointer to a **MIB_IPNETROW** structure. The information in this structure identifies the entry to modify and specifies the new information for the entry. The caller must specify values for all members of this structure.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

CreateIpNetEntry, **DeleteIpNetEntry**, **MIB_IPNETROW**

SetIpStatistics

The **SetIpStatistics** function toggles IP forwarding on or off and sets the default Time-To-Live (TTL) value for the local computer.

```
DWORD SetIpStatistics(
  PMIB_IPSTATS pIpStats   // new forwarding and TTL settings
);
```

Parameters

pIpStats

Pointer to a **MIB_IPSTATS** structure. The caller should set the **dwForwarding** and **dwDefaultTTL** members of this structure to the new values. To keep one of the members at its current value, use MIB_USE_CURRENT_TTL or MIB_USE_CURRENT_FORWARDING.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Remarks

To set only the default TTL, the caller can also use the **SetIpTTL** function.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

MIB_IPSTATS, **SetIpTTL**

SetIpTTL

The **SetIpTTL** function sets the default Time-To-Live (TTL) value for the local computer.

```
DWORD SetIpTTL(
  UINT nTTL    // new default TTL
);
```

Parameters

nTTL
Specifies the new TTL value for the local computer.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Remarks

The default TTL can also be set using the **SetIpStatistics** function.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

MIB_IPSTATS, **SetIpStatistics**

SetTcpEntry

The **SetTcpEntry** function sets the state of a TCP connection.

```
DWORD SetTcpEntry(
  PMIB_TCPROW pTcpRow    // pointer to struct. with new
                         // state info
);
```

Parameters

pTcpRow

Pointer to a **MIB_TCPROW** structure. This structure contains information to identify the TCP connection to modify. It also specifies the new state for the TCP connection. The caller must specify values for all the members in this structure.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, use **FormatMessage** to obtain the message string for the returned error.

Remarks

Currently, the only state to which a TCP connection can be set is MIB_TCP_STATE_DELETE_TCB.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

MIB_TCPROW

IP Helper Structures

Use the following structures to retrieve and modify configuration settings for the TCP/IP transport suite on the local computer:

FIXED_INFO
IP_ADAPTER_INDEX_MAP
IP_ADAPTER_INFO
IP_INTERFACE_INFO
IP_PER_ADAPTER_INFO
IP_UNIDIRECTIONAL_ADAPTER_ADDRESS

FIXED_INFO

The **FIXED_INFO** structure contains information that is the same across all the interfaces in a computer.

```
typedef struct {
  char HostName[MAX_HOSTNAME_LEN + 4] ;
  char DomainName[MAX_DOMAIN_NAME_LEN + 4];
  PIP_ADDR_STRING CurrentDnsServer;
  IP_ADDR_STRING DnsServerList;
  UINT NodeType;
  char ScopeId[MAX_SCOPE_ID_LEN + 4];
  UINT EnableRouting;
  UINT EnableProxy;
  UINT EnableDns;
} FIXED_INFO, *PFIXED_INFO;
```

Members

HostName[MAX_HOSTNAME_LEN + 4]
Specifies the host name for the local computer.

DomainName[MAX_DOMAIN_NAME_LEN + 4]
Specifies the domain in which the local computer is registered.

CurrentDnsServer
Specifies the current DNS server.

DnsServerList
Specifies the set of DNS servers used by the local computer.

NodeType
Specifies whether the local computer uses dynamic host configuration protocol (DHCP).

ScopeId[MAX_SCOPE_ID_LEN + 4]
Specifies the DHCP scope name.

EnableRouting
Specifies whether routing is enabled on the local computer.

EnableProxy
Specifies whether the local computer is acting as an ARP proxy.

EnableDns
Specifies whether DNS is enabled on the local computer.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Iptypes.h.

See Also

GetNetworkParams

IP_ADAPTER_INDEX_MAP

The **IP_ADAPTER_INDEX_MAP** structure pairs an adapter name with the index of that adapter.

```
typedef struct _IP_ADAPTER_INDEX_MAP {
  ULONG Index;                      // adapter index
  WCHAR Name[MAX_ADAPTER_NAME];     // name of the adapter
} IP_ADAPTER_INDEX_MAP, * PIP_ADAPTER_INDEX_MAP;
```

Members

Index
Specifies the index of the adapter.

Name[MAX_ADAPTER_NAME]
Pointer to a Unicode string that contains the name of the adapter.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Ipexport.h.

See Also

IP_INTERFACE_INFO, **GetInterfaceInfo**

IP_ADAPTER_INFO

The **IP_ADAPTER_INFO** structure contains information about a particular network adapter on the local computer.

```
typedef struct _IP_ADAPTER_INFO {
  struct _IP_ADAPTER_INFO* Next;
  DWORD ComboIndex;
  char AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
  char Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
  UINT AddressLength;
  BYTE Address[MAX_ADAPTER_ADDRESS_LENGTH];
  DWORD Index;
  UINT Type;
  UINT DhcpEnabled;
  PIP_ADDR_STRING CurrentIpAddress;
  IP_ADDR_STRING IpAddressList;
  IP_ADDR_STRING GatewayList;
  IP_ADDR_STRING DhcpServer;
  BOOL HaveWins;
  IP_ADDR_STRING PrimaryWinsServer;
  IP_ADDR_STRING SecondaryWinsServer;
  time_t LeaseObtained;
  time_t LeaseExpires;
} IP_ADAPTER_INFO, *PIP_ADAPTER_INFO;
```

Members

Next
Pointer to the next adapter in the linked list of adapters.

ComboIndex
This member is unused.

AdapterName[MAX_ADAPTER_NAME_LENGTH + 4]
Specifies the name of the adapter.

Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4]
Specifies a description for the adapter.

AddressLength
Specifies the length of hardware address for the adapter.

Address[MAX_ADAPTER_ADDRESS_LENGTH]
Specifies the hardware address for the adapter.

Index
Specifies the adapter index.

Type
Specifies the adapter type.

DhcpEnabled
Specifies whether Dynamic Host Configuration Protocol (DHCP) is enabled for this adapter.

CurrentIpAddress
Specifies the current IP address for this adapter.

IpAddressList
Specifies the list of IP addresses associated with this adapter.

GatewayList
Specifies the IP address of the default gateway for this adapter.

DhcpServer
Specifies the IP address of the DHCP server for this adapter.

HaveWins
Specifies whether this adapter uses Windows Internet Name Service (WINS).

PrimaryWinsServer
Specifies the IP address of the primary WINS server.

SecondaryWinsServer
Specifies the IP address of the secondary WINS server.

LeaseObtained
Specifies the time when the current DHCP lease was obtained.

LeaseExpires
Specifies the time when the current DHCP lease will expire.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Iptypes.h.

See Also

GetAdaptersInfo

IP_INTERFACE_INFO

The **IP_INTERFACE_INFO** structure contains a list of the network interface adapters on the local system.

```
typedef struct _IP_INTERFACE_INFO {
  LONG NumAdapters;        // number of adapters in array
  IP_ADAPTER_INDEX_MAP Adapter[1];
                           // adapter indices and names
} IP_INTERFACE_INFO,*PIP_INTERFACE_INFO;
```

Members

NumAdapters
Specifies the number of adapters listed in the array pointed to by the **Adapter** member.

Adapter[1]
Specifies an array of **IP_ADAPTER_INDEX_MAP** structures. Each structure maps an adapter index to that adapter's name.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Ipexport.h.

See Also

IP_ADAPTER_INDEX_MAP, **GetInterfaceInfo**

IP_PER_ADAPTER_INFO

The **IP_PER_ADAPTER_INFO** function contains information specific to a particular adapter.

```
typedef struct _IP_PER_ADAPTER_INFO {
  UINT AutoconfigEnabled;
  UINT AutoconfigActive;
  PIP_ADDR_STRING CurrentDnsServer;
  IP_ADDR_STRING DnsServerList;
} IP_PER_ADAPTER_INFO, *PIP_PER_ADAPTER_INFO;
```

Members

AutoconfigEnabled
Specifies whether auto-configuration is enabled on this adapter.

AutoconfigActive
Specifies whether auto-configuration is active on this adapter.

CurrentDnsServer
Specifies the IP address of the current DNS server for this adapter.

DnsServerList
Specifies the list of possible DNS servers for this adapter.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Iptypes.h.

+ See Also

GetPerAdapterInfo

IP_UNIDIRECTIONAL_ADAPTER_ADDRESS

The **IP_UNIDIRECTIONAL_ADAPTER_ADDRESS** structure contains the number of unidirectional adapters on the local computer, and the IP addresses that are associated with those adapters.

```
typedef struct _IP_UNIDIRECTIONAL_ADAPTER_ADDRESS {
  ULONG NumAdapters;
  IPAddr Address[1];
} IP_UNIDIRECTIONAL_ADAPTER_ADDRESS, *PIP_UNIDIRECTIONAL_ADAPTER_ADDRESS;
```

Members

NumAdapters
Specifies the number of unidirectional adapters on the local computer.

Address[1]
Specifies an array of **IPAddr** values. These are the IP addresses of the unidirectional adapters on the local computer.

Remarks

For information about the **IPAddr** data type, see *Win32 Simple Data Types*. To convert an IP address between dotted decimal notation and **IPAddr** format, use the **inet_addr** and **inet_ntoa** functions.

! Requirements

Windows NT/2000: Unsupported.
Windows 95/98: Requires Windows 98.
Header: Declared in Ipexport.h.

+ See Also

GetUniDirectionalAdapterInfo

CHAPTER 14

Simple Network Management Protocol (SNMP)

The Simple Network Management Protocol (SNMP) is the Internet standard protocol for exchanging management information between management console applications such as HP Openview, Novell NMS, IBM NetView, or Sun Net Manager, and managed entities. The managed entities can include hosts, routers, bridges, and hubs.

New SNMP Programming Elements

The Microsoft® SNMP service for Windows® 2000 adds support for the following programming features and elements:

- New SNMP extension agent functions
- New SNMP utility functions

In addition, the SNMP service adds support for the following structure:

- **AsnCounter64**

The SNMP service also introduces the following changes:

- Modified SNMP variable types
- Modified SNMP PDU request types

New SNMP Extension Agent Functions

The following new SNMP extension agent functions are available to SNMP extension agents that run on Windows 2000.

- **SnmpExtensionClose**
- **SnmpExtensionMonitor**
- **SnmpExtensionQueryEx**

New SNMP Utility Functions

The following new SNMP utility functions simplify manipulation of octet strings and **AsnAny** structures, and provide functionality that is useful during the development of SNMP applications.

- **SnmpSvcGetUptime**
- **SnmpSvcSetLogLevel**
- **SnmpSvcSetLogType**
- **SnmpUtilAsnAnyCpy**
- **SnmpUtilAsnAnyFree**
- **SnmpUtilDbgPrint**
- **SnmpUtilIdsToA**
- **SnmpUtilOctetsCmp**
- **SnmpUtilOctetsCpy**
- **SnmpUtilOctetsFree**
- **SnmpUtilOctetsNCmp**
- **SnmpUtilOidToA**
- **SnmpUtilPrintOid**

Modified SNMP Variable Types

The definitions for some SNMP variable types have changed. The SNMP.H file maps old variable types to the corresponding new types.

You should use the new SNMP variable type when you develop manager applications that use the Microsoft SNMP Management API. The following table lists the old SNMP variable types with the corresponding new variable type.

Old Variable Type	New Variable Type
ASN_RFC1155_IPADDRESS	ASN_IPADDRESS
ASN_RFC1155_COUNTER	ASN_COUNTER32
ASN_RFC1155_GAUGE	ASN_GAUGE32
ASN_RFC1155_TIMETICKS	ASN_TIMETICKS
ASN_RFC1155_OPAQUE	ASN_OPAQUE
ASN_RFC1213_DISPSTRING	ASN_OCTETSTRING

Modified SNMP PDU Request Types

The definitions for some SNMP PDU types have changed. The SNMP.H file maps old PDU types to the corresponding new types.

You should use the new SNMP PDU type when you develop manager applications that use the Microsoft SNMP Management API. The following table lists the old SNMP PDU types with the corresponding new PDU type.

Old PDU Type	New PDU Type
ASN_RFC1157_GETREQUEST	SNMP_PDU_GET
ASN_RFC1157_GETNEXTREQUEST	SNMP_PDU_GETNEXT
ASN_RFC1157_GETRESPONSE	SNMP_PDU_RESPONSE
ASN_RFC1157_SETREQUEST	SNMP_PDU_SET
ASN_RFC1157_TRAP	SNMP_PDU_V1TRAP

About SNMP

SNMP uses a distributed architecture consisting of *managers* and *agents*. An *agent* is an SNMP application that responds to queries from SNMP manager applications. The SNMP agent is responsible for retrieving and updating local management information based on the requests of the SNMP manager. The agent also notifies registered managers when significant events or *traps* occur. A *manager* is an SNMP application that generates queries to SNMP agent applications and receives traps from SNMP agent applications.

On computers running Microsoft® Windows NT®/Windows® 2000 the SNMP agent is implemented by the SNMP service (SNMP.EXE). The SNMP manager is typically a third-party SNMP management console application. The management console application does not need to run on the same host as the SNMP agent. To use the information the Microsoft SNMP service provides, you need at least one SNMP management console application. The system includes libraries that support SNMP management console applications, but it does not include an SNMP management console application at this time.

How SNMP Works

The following steps outline how a third-party SNMP management console application returns information from the SNMP service:

1. The SNMP management console application formulates an SNMP *message* based on input from the user. The message includes a protocol data unit (PDU) and authentication information. The management console application can use the Microsoft SNMP Management API library (MGMTAPI.DLL) or the Microsoft WinSNMP API library (WSNMP32.DLL) to perform this step.
2. The SNMP management console application sends the SNMP message to the SNMP service, using the SNMP service libraries.
3. The SNMP service receives the request. It verifies the authentication information and the source IP address.
4. The SNMP service selects the appropriate extension agent and requests that the agent retrieve the requested information.
5. The SNMP service sends the response to the SNMP management console application.

The SNMP Management Information Base (MIB)

A Management Information Base (MIB) describes a set of managed objects. An SNMP management console application can manipulate the objects on a specific computer if the SNMP service has an extension agent DLL that supports the MIB.

Each managed object in a MIB has a unique identifier. The identifier includes the object's type (such as counter, string, gauge, or address), the object's access level (such as read or read/write), size restrictions, and range information.

The following table contains a partial list of the MIBs that ship with Windows NT/Windows 2000. They are installed with the SNMP service. For a complete listing of MIBs, refer to the *Windows NT/Windows 2000 Resource Kit*.

MIB	Description
DHCP.MIB	Microsoft-defined MIB that contains object types for monitoring the network traffic between remote hosts and DHCP servers
HOSTMIB.MIB	Contains object types for monitoring and managing host resources
LMMIB2.MIB	Covers workstation and server services
MIB_II.MIB	Contains the Management Information Base (MIB-II), which provides a simple, workable architecture and system for managing TCP/IP-based internets
WINS.MIB	Microsoft-defined MIB for the Windows Internet Name Service (WINS)

The extension agent DLLs for MIB-II, LAN Manager MIB-II, and the Host Resources MIB are installed with the SNMP service. The extension agent DLLs for the other MIBs are installed when their respective services are installed. At service startup time, the SNMP service loads all of the extension agent DLLs that are listed in the registry.

Users can add other extension agent DLLs that implement additional MIBs. To do this, they must add a registry entry for the new DLL under the SNMP service. They must also register the new MIB with the SNMP management console application. For more information, see the documentation included with your management console application.

MIB Name Tree

The name space for MIB object identifiers is hierarchical. It is structured so that each manageable object can be assigned a globally unique name.

Authority for parts of the name space is assigned to individual organizations. This allows organizations to assign names without consulting an Internet authority for each assignment. For example, the name space assigned to Microsoft is 1.3.6.1.4.1.311, which is defined in MSFT.MIB. Microsoft has the authority to assign names to objects anywhere below that name space.

The object identifier in the hierarchy is written as a sequence of subidentifiers beginning at the root and ending at the object. Subidentifiers are separated with a period.

Relevant RFCs

TCP/IP standards are defined in Requests for Comments (RFCs), which are published by the Internet Engineering Task Force (IETF). The following table lists the RFCs that are relevant to SNMP.

RFC number	Title
1155	"Structure and Identification of Management Information for TCP/IP-based Internets." It defines SMI.MIB.
1157	"A Simple Network Management Protocol (SNMP)." It defines SNMP itself.
1213	"Management Information Base for Network Management of TCP/IP-based internets: MIB-II." It defines MIB_II.MIB.
1901	"Introduction to Community-based SNMPv2"
1902	"Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1903	"Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1904	"Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1905	"Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1906	"Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1907	"Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1908	"Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework"
2089	"V2ToV1 Mapping SNMPv2 onto SNMPv1 within a bi-lingual SNMP agent"

System Files for SNMP

The following table describes the principal files that relate to the SNMP service.

Filename	Description
DHCPMIB.DLL	Extension agent DLL that implements the Microsoft-defined DHCP MIB. Installed only on DHCP servers, and not available on Windows NT 3.1.
EVNTAGNT.DLL	SNMP DLL that translates event logs into SNMP traps; also known as the SNMP event translator.
HOSTMIB.DLL	Extension agent DLL that implements the Host Resources MIB.
LMMIB2.DLL	Extension agent DLL that implements LAN Manager MIB-II.
MGMTAPI.DLL	Microsoft Windows 2000-based SNMP Management API library. This API allows SNMP manager applications to "listen" for SNMP manager requests, and send requests to and receive responses from SNMP agents.

(continued)

(continued)

Filename	Description
MIB.BIN	Compiled MIB information used by MGMTAPI.DLL.
SNMP.EXE	SNMP service. This is the master agent that receives SNMP requests and delivers them to the appropriate extension agent DLL.
SNMPAPI.DLL	SNMP utilities DLL used by SNMP extension agent DLLs and manager applications. This DLL contains a framework for developing extension agent DLLs.
SNMPSNAP.DLL	SNMP configuration application that is a Microsoft Management Console (MMC) snap-in component. The snap-in adds several pages to the SNMP Service Properties sheet. For more information, see the online help for the SNMP service.
SNMPTRAP.EXE	SNMP trap service. Receives SNMP traps and forwards them to SNMP manager applications.
WINSMIB.DLL	Extension agent DLL that implements the Microsoft-defined WINS MIB. Installed only on WINS servers, and not available on Windows NT 3.1.
WSNMP32.DLL	Microsoft Windows 2000-based WinSNMP API library. This API allows SNMP manager applications to "listen" for SNMP manager requests, and send requests to and receive responses from SNMP agents.

For additional information, see *The SNMP Management Information Base (MIB)*. (You can also refer to the *Windows NT/Windows 2000 Resource Kit*.)

SNMP Utilities

The following table lists the SNMP utilities that are available in the Microsoft Windows Resource Kit.

Filename	Description
EVNTCMD.EXE	A command-line application for configuring the SNMP event translator
EVNTWIN.EXE	An application that provides a user interface for configuring the SNMP event translator
MIBCC.EXE	The SNMP MIB Compiler
SNMPUTIL.EXE	A sample SNMP manager console application

Configuring the SNMP Service

On occasion, you may need to reconfigure SNMP. In these instances, you need to know community names in your network, the trap destination for each community, and IP addresses or computer names for SNMP management hosts before you use or reconfigure SNMP services. For more information, refer to the *Windows NT/Windows 2000 Resource Kit* and to the SNMP service online documentation.

The following topics contain information relevant to configuring the SNMP service:

- Community names
- Host names and IP addresses
- Configuring SNMP security
- Configuring SNMP agent information

Community Names

A *community name* identifies a collection of SNMP managers and agents. The use of a community name provides primitive security and context checking for both agents and managers that receive requests and initiate trap operations. An agent won't accept a request from a manager outside the community.

Host Names and IP Addresses

If the computer does not have access to a WINS server, the SNMP service uses the HOSTS file to resolve host names to IP addresses. The HOST file is merely a text file listing explicit host names and IP addresses. If you use host names, be sure to add all host name and IP address mappings of the participating systems.

Configuring SNMP Security

SNMP security allows you to specify the communities and hosts from which a computer accepts requests, as well as the type of operations to accept from the computers belonging to a community. The security also allows you to specify whether to send an authentication trap when an unauthorized community or host requests information.

Configuring SNMP Agent Information

SNMP agent information allows you to specify comments about the user and the physical location of the computer and to indicate the types of service to report. The types of service that can be reported are based on the computer's configuration.

SNMP Reference

This section provides the SNMP functions and structures. These elements support the development of SNMP agent applications and SNMP manager applications for Windows NT®/Windows® 2000.

SNMP Functions

The SNMP functions fall into the following three functional groupings:

- SNMP Extension Agent API functions
- SNMP Management API functions
- SNMP Utility API functions

SNMP Extension Agent API Functions

The SNMP Extension Agent API functions define the interface between the SNMP service and the third-party SNMP extension agent DLLs. Applications use these functions to resolve the variable bindings specified by incoming SNMP PDUs.

SnmpExtensionClose
SnmpExtensionInit
SnmpExtensionInitEx
SnmpExtensionMonitor
SnmpExtensionQuery
SnmpExtensionQueryEx
SnmpExtensionTrap

SNMP Management API Functions

The SNMP Management API functions define the interface between third-party SNMP manager applications and the management function dynamic-link library MGMTAPI.DLL. The DLL works in conjunction with the SNMP trap service (SNMPTRAP.EXE), and can interact with one or more third-party SNMP manager applications. Third-party manager applications can use the management functions to perform SNMP manager operations.

SnmpMgrClose
SnmpMgrGetTrap
SnmpMgrOidToStr
SnmpMgrOpen
SnmpMgrRequest
SnmpMgrStrToOid
SnmpMgrTrapListen

SNMP Utility API Functions

The SNMP Utility API functions simplify manipulation of SNMP data structures and provide functionality that is useful during the development of SNMP applications.

SnmpSvcGetUptime
SnmpSvcSetLogLevel
SnmpSvcSetLogType
SnmpUtilAsnAnyCpy
SnmpUtilAsnAnyFree
SnmpUtilDbgPrintSnmpUtilIdsToA
SnmpUtilMemAlloc
SnmpUtilMemFree
SnmpUtilMemReAlloc
SnmpUtilOctetsCmp
SnmpUtilOctetsCpy
SnmpUtilOctetsFree
SnmpUtilOctetsNCmp
SnmpUtilOidAppend
SnmpUtilOidCmp
SnmpUtilOidCpy
SnmpUtilOidFree
SnmpUtilOidNCmpSnmpUtilOidToA
SnmpUtilPrintAsnAnySnmpUtilPrintOid
SnmpUtilVarBindCpy
SnmpUtilVarBindListCpy
SnmpUtilVarBindFree
SnmpUtilVarBindListFree

SnmpExtensionClose

The Microsoft SNMP service calls the **SnmpExtensionClose** function to request that the SNMP extension agent deallocate resources and terminate operations. This function is an element of the SNMP Extension Agent API.

Note The SNMP service calls the **SnmpExtensionClose** function only if the extension agent is running on Windows 2000. For more information, see the following *Remarks* section.

```
VOID SnmpExtensionClose();
```

Parameters

This function has no parameters.

Return Values

None.

Remarks

An SNMP extension agent that runs on Windows NT 3.51 or 4.0 can also export the **SnmpExtensionClose** function. Because the SNMP service does not call this function under these conditions, the extension agent must call **SnmpExtensionClose** manually. It should do this when the SNMP service calls the extension agent's **DllMain** function with the value DLL_PROCESS_DETACH. The extension agent must clean up allocated resources and terminate services at this time.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

+ See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpExtensionInit**, **SnmpExtensionInitEx**

SnmpExtensionInit

The Microsoft SNMP service calls the **SnmpExtensionInit** function to initialize the SNMP extension agent DLL. This function is an element of the SNMP Extension Agent API.

```
BOOL SnmpExtensionInit(
  DWORD dwUptimeReference,            // see NOTE below
  HANDLE *phSubagentTrapEvent,        // trap event handle
  AsnObjectIdentifier *pFirstSupportedRegion  // first MIB
                                              // subtree
);
```

Parameters

dwUptimeReference

[in] Specifies a time-zero reference for the extension agent.

Note Extension agents should ignore this parameter. The SNMP extension agent DLL should use the **SnmpSvcGetUptime** function to retrieve the number of centiseconds the SNMP service has been running. For more information, see the following *Remarks* section.

---

phSubagentTrapEvent

[out] Pointer to an event handle the extension agent passes back to the SNMP service. This handle is used to notify the service that the extension agent has one or more traps to send. For additional information about allocating and deallocating the event handle, see the following *Remarks* section.

pFirstSupportedRegion

[out] Pointer to an **AsnObjectIdentifier** structure to receive the first MIB subtree that the extension agent supports. For additional information about allocating and deallocating resources for this structure, see the following Remarks section.

The extension agent can register additional MIB subtrees by implementing the **SnmpExtensionInitEx** entry point function.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

Remarks

Extension agents should ignore the *dwUptimeReference* parameter. Instead, they should call the **SnmpSvcGetUptime** function to retrieve the number of centiseconds that the Microsoft SNMP service has been running. Because the *dwUptimeReference* parameter stores the elapsed time as a **DWORD** value in milliseconds, the time can wrap to zero and reflect an inaccurate time interval.

The extension agent notifies the SNMP service that it needs to send one or more traps by setting the event handle passed back in the *phSubagentTrapEvent* parameter to the signaled state. After this event has been signaled, the SNMP service repeatedly calls the extension agent's **SnmpExtensionTrap** entry point until the function returns a value of FALSE. This indicates that the extension agent has no more traps to send. If the extension agent does not generate traps, the *phSubagentTrapEvent* parameter should return a value of NULL.

The SNMP extension agent must allocate and deallocate resources for the trap event handle. When the SNMP service calls the **SnmpExtensionInit** function, the extension agent must call the **CreateEvent** function to allocate the event handle. The extension agent passes the handle to the SNMP service in the *phSubagentTrapEvent* parameter. When the SNMP service calls the **SnmpExtensionClose** function, the extension agent must deallocate resources for the trap event handle.

The SNMP service makes a copy of the **AsnObjectIdentifier** structure the extension agent returns in the *pFirstSupportedRegion* parameter. The extension agent must allocate and deallocate the resources associated with the original structure. It can do this when the SNMP service calls the **SnmpExtensionClose** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **AsnObjectIdentifier**, **CreateEvent**, **SnmpExtensionTrap**, **SnmpSvcGetUptime**, **SnmpExtensionClose**, **SnmpExtensionMonitor**

SnmpExtensionInitEx

The Microsoft SNMP service calls the **SnmpExtensionInitEx** function to identify any additional management information base (MIB) subtrees the SNMP extension agent supports. This function is an element of the SNMP Extension Agent API.

```
BOOL SnmpExtensionInitEx(
  AsnObjectIdentifier *pNextSupportedRegion  // next MIB
                                             //subtree
);
```

Parameters

pNextSupportedRegion
: [out] Pointer to an **AsnObjectIdentifier** structure to receive the next MIB subtree that the extension agent supports.

Return Values

If the *pNextSupportedRegion* parameter has been initialized with an additional MIB subtree, the return value is TRUE.

If there are no more MIB subtrees to register, the return value is FALSE.

Remarks

The SNMP service repeatedly calls the **SnmpExtensionInitEx** function entry point so the extension agent can register support for additional MIB subtrees.

The SNMP service makes a copy of the **AsnObjectIdentifier** structure the extension agent returns in the *pNextSupportedRegion* parameter. The extension agent must allocate and deallocate the resources associated with the original structure. It can do this when the SNMP service calls the **SnmpExtensionClose** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpExtensionInit**, **SnmpExtensionClose**, **AsnObjectIdentifier**, **SnmpExtensionMonitor**

SnmpExtensionMonitor

The Microsoft SNMP service calls the **SnmpExtensionMonitor** function to provide the SNMP extension agent with a view to the service's internal counters and parameters. This function is an element of the SNMP Extension Agent API.

The **SnmpExtensionMonitor** function is optional. Extension agents should implement the function if they are interested in a view of the SNMP service's internal management objects, as defined in RFC 1213, "Management Information Base for Network Management of TCP/IP-based internets: MIB-II."

```
BOOL SnmpExtensionMonitor(
  LPVOID pAgentMgmtData
);
```

Parameters

pAgentMgmtData
[in] Pointer to an array of **AsnAny** objects (structures). The number of objects, and the type and description of each object, are in accordance with RFC 1213. For more information, see the following *Remarks* section.

Return Values

Unless an unexpected error occurs while the SNMP extension agent is processing the value of the *pAgentMgmtData* parameter, the extension agent should return TRUE. If the extension agent returns FALSE, the SNMP service does not load the extension agent, and the service stops directing SNMP requests to the extension agent.

Remarks

If the extension agent exports the **SnmpExtensionMonitor** function, the SNMP service calls the function during initialization of the extension agent, immediately after the service calls the **SnmpExtensionInit** and the **SnmpExtensionInitEx** functions.

The SNMP service dynamically updates the SNMP counters (for example, the snmpInPkts and the snmpOutNoSuchNames counters) in the array pointed to by the *pAgentMgmtData* parameter. In order to be able to read these values while the SNMP service is running, the extension agent must store the pointer to *pAgentMgmtData*.

Note that an SNMP extension agent should not update the memory pointed to by the *pAgentMgmtData* parameter. This is because the values of the SNMP service's internal counters would no longer be valid, and the behavior of the SNMP service could become unpredictable. As long as the extension agent does not alter it, the memory pointed to by *pAgentMgmtData* is valid while the SNMP service is running.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Snmp.h.

+ See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpExtensionInit**, **SnmpExtensionInitEx**, **SnmpExtensionClose**, **AsnAny**

SnmpExtensionQuery

The Microsoft SNMP service calls the **SnmpExtensionQuery** function to resolve SNMP requests that contain variables within one or more of the SNMP extension agent's registered MIB subtrees. This function is an element of the SNMP Extension Agent API.

Note The extension agent must export the **SnmpExtensionQuery** function if the extension agent runs on Windows NT 3.51 or 4.0. However, it is recommended that you use the **SnmpExtensionQueryEx** function, which supports SNMP version 2C (SNMPv2C) data types and multiphase SNMP SET operations.

```
BOOL SnmpExtensionQuery(
  BYTE bPduType,                    // SNMPv1 PDU request type
  SnmpVarBindList *pVarBindList,    // pointer to variable
                                    // bindings
  AsnInteger32 *pErrorStatus,       // pointer to SNMPv1 error
                                    // status
  AsnInteger32 *pErrorIndex         // pointer to the error index
);
```

Parameters

bPduType

[in] Specifies the SNMP version 1 (SNMPv1) PDU request type. This parameter can be one of the following values.

Value	Meaning
SNMP_PDU_GET	Retrieve the value or values of the specified variables.
SNMP_PDU_GETNEXT	Retrieve the value or values of the lexicographic successor of the specified variable.
SNMP_PDU_SET	Write a value within a specific variable.

Note that PDU request types have been renamed. For additional information, see *Modified SNMP PDU Request Types*.

pVarBindList

[in/out] Pointer to the variable bindings list.

pErrorStatus

[out] Pointer to a variable in which the error status result will be returned. This parameter can be one of the following values defined by SNMPv1.

Value	Meaning
SNMP_ERRORSTATUS_NOERROR	The agent reports that no errors occurred during transmission.
SNMP_ERRORSTATUS_TOOBIG	The agent could not place the results of the requested operation into a single SNMP message.
SNMP_ERRORSTATUS_NOSUCHNAME	The requested operation identified an unknown variable.
SNMP_ERRORSTATUS_BADVALUE	The requested operation tried to change a variable but it specified either a syntax or value error.

Value	Meaning
SNMP_ERRORSTATUS_READONLY	The requested operation tried to change a variable that was not allowed to change according to the community profile of the variable.
SNMP_ERRORSTATUS_GENERR	An error other than one of those listed here occurred during the requested operation.

pErrorIndex
[out] Pointer to a variable in which the error index result will be returned.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

Remarks

When the SNMP service receives an SNMP PDU request, it calls the **SnmpExtensionQuery** function to process the request. The extension agent must follow the rules in RFC 1157 to either resolve the variable bindings or generate an error.

If the extension agent cannot resolve the variable bindings on a **Get Next** request, it must change the **name** field of the **SnmpVarBind** structure to the value of the object identifier immediately following that of the currently supported MIB subtree view. For example, if the extension agent supports view ".1.3.6.1.4.1.77.1", a **Get Next** request on ".1.3.6.1.4.1.77.1.5.1" would result in a modified **name** field of ".1.3.6.1.4.1.77.2". This signals the SNMP service to continue the attempt to resolve the variable bindings with other extension agents.

It is important to note that the SNMP service and the extension agent may need to exchange dynamically allocated memory during a call to the **SnmpExtensionQuery** function. The service dynamically allocates the object identifier in each **SnmpVarBind** structure it passes to the extension agent. However, the extension agent must release this memory in order to replace the object identifier when it processes a **Get Next** request. The extension agent allocates dynamic memory for variable-length object types. The SNMP service releases this memory after the object is placed in the response PDU.

In order to avoid heap corruption and memory leaks, both the SNMP service and the extension agent must use memory allocation routines that resolve to the same heap. The extension agent must use the **SnmpUtilMemAlloc** function to allocate memory that it passes to the SNMP service. It must use the **SnmpUtilMemFree** function to release the memory the service passes back to the extension agent. These functions are located in the utility dynamic-link library SNMPAPI.DLL.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpVarBind**, **SnmpExtensionInit**, **SnmpUtilMemAlloc**, **SnmpUtilMemFree**

SnmpExtensionQueryEx

The Microsoft SNMP service calls the **SnmpExtensionQueryEx** function to process SNMP requests that specify variables in one or more MIB subtrees registered by SNMP extension agents. This function is an element of the SNMP Extension Agent API:

Note It is recommended that you use the **SnmpExtensionQueryEx** function, which supports SNMP version 2C (SNMPv2C) data types and multiphase SNMP SET operations. However, the extension agent must also export the **SnmpExtensionQuery** function if the extension agent runs on Windows NT 3.51 or 4.0. The SNMP service does not call the **SnmpExtensionQuery** function if the extension agent exports the **SnmpExtensionQueryEx** function.

```
BOOL SnmpExtensionQueryEx(
  DWORD dwRequestType,            // extension agent request
                                  // type
  DWORD dwTransactionId,          // identifier of the
                                  // incoming PDU
  SnmpVarBindList *pVarBindList,  // pointer to variable
                                  // binding list
  AsnOctetString *pContextInfo,   // pointer to context
                                  // information
  AsnInteger32 *pErrorStatus,     // pointer to SNMPv2 error
                                  // status
  AsnInteger32 *pErrorIndex       // pointer to the error
                                  // index
);
```

Parameters

dwRequestType

Specifies the type of operation that the SNMP service is requesting the extension agent to perform. This parameter can be one of the following values.

Value	Meaning
SNMP_EXTENSION_GET	Retrieve the value or values of the specified variables.
SNMP_EXTENSION_GET_NEXT	Retrieve the value or values of the lexicographic successor of the specified variables.
SNMP_EXTENSION_SET_TEST	Validate the values of the specified variables. This operation maximizes the probability of a successful write during the COMMIT request.
SNMP_EXTENSION_SET_COMMIT	Write the new values to the specified variables.
SNMP_EXTENSION_SET_UNDO	Reset the values of the specified variables to their values before the COMMIT request.
SNMP_EXTENSION_SET_CLEANUP	Release the resources allocated in previous requests and operations.

For additional information about the SET request types, that is, those that begin with SNMP_EXTENSION_SET_, see the following *Remarks* section.

dwTransactionId

Specifies a **DWORD** variable that is the unique identifier of the incoming SNMP request PDU. The extension agent can use this value to correlate multiple calls by the SNMP service that involve the same PDU.

pVarBindList

Pointer to the variable binding list containing the variables of interest.

pContextInfo

Pointer to an octet string that contains user-defined context information.

The extension agent can use this parameter to store context information used during multiphase SNMP SET operations. The extension agent must release resources associated with this parameter during the CLEANUP request. The SNMP service does not release any resources associated with this parameter. For additional information, see the following *Remarks* section.

pErrorStatus

Pointer to a variable to receive the error status result. This parameter can be one of the following values defined by SNMPv2C.

Error Code	Meaning
SNMP_ERRORSTATUS_ NOERROR	The agent reports that no errors occurred during transmission.
SNMP_ERRORSTATUS_ TOOBIG	The agent could not place the results of the requested SNMP operation into a single SNMP message.

(continued)

(continued)

Error Code	Meaning
SNMP_ERRORSTATUS_NOSUCHNAME	The requested SNMP operation identified an unknown variable.
SNMP_ERRORSTATUS_BADVALUE	The requested SNMP operation tried to change a variable but it specified either a syntax or value error.
SNMP_ERRORSTATUS_READONLY	The requested SNMP operation tried to change a variable that was not allowed to change, according to the community profile of the variable.
SNMP_ERRORSTATUS_GENERR	An error other than one of those listed here occurred during the requested SNMP operation.
SNMP_ERRORSTATUS_NOACCESS	The specified SNMP variable is not accessible.
SNMP_ERRORSTATUS_WRONGTYPE	The value specifies a type that is inconsistent with the type required for the variable.
SNMP_ERRORSTATUS_WRONGLENGTH	The value specifies a length that is inconsistent with the length required for the variable.
SNMP_ERRORSTATUS_WRONGENCODING	The value contains an Abstract Syntax Notation One (ASN.1) encoding that is inconsistent with the ASN.1 tag of the field.
SNMP_ERRORSTATUS_WRONGVALUE	The value cannot be assigned to the variable.
SNMP_ERRORSTATUS_NOCREATION	The variable does not exist, and the agent cannot create it.
SNMP_ERRORSTATUS_INCONSISTENTVALUE	The value is inconsistent with values of other managed objects.
SNMP_ERRORSTATUS_RESOURCEUNAVAILABLE	Assigning the value to the variable requires allocation of resources that are currently unavailable.
SNMP_ERRORSTATUS_COMMITFAILED	No validation errors occurred, but no variables were updated.
SNMP_ERRORSTATUS_UNDOFAILED	No validation errors occurred. Some variables were updated because it was not possible to undo their assignment.
SNMP_ERRORSTATUS_AUTHORIZATIONERROR	An authorization error occurred.
SNMP_ERRORSTATUS_NOTWRITABLE	The variable exists but the agent cannot modify it.
SNMP_ERRORSTATUS_INCONSISTENTNAME	The variable does not exist; the agent cannot create it because the named object instance is inconsistent with the values of other managed objects.

pErrorIndex
Pointer to a variable to receive the error index result.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

Remarks

The SNMP service calls the **SnmpExtensionQueryEx** function multiple times to process an incoming SNMP SET request. The service can call **SnmpExtensionQueryEx** during the TEST request phase, the COMMIT request phase, the UNDO request phase, and the CLEANUP request phase.

TEST request

The SNMP service processes an SNMP SET request type by first calling the **SnmpExtensionQueryEx** function with a *dwRequestType* of SNMP_EXTENSION_SET_TEST. The service calls each extension agent responsible for the variable bindings in the request. Each extension agent must validate the variables in the variable binding list. They can optionally store any context information required for the following requests in the variable pointed to by the *pContextInfo* parameter.

If the TEST request fails, the service initiates a CLEANUP request. The service calls each extension agent that previously returned TRUE to the TEST request again with the **SnmpExtensionQueryEx** function. The service calls each extension agent using the SNMP_EXTENSION_SET_CLEANUP *dwRequestType*.

COMMIT request

If all extension agents return TRUE to the TEST request, the SNMP service calls each extension agent with the **SnmpExtensionQueryEx** function, using the SNMP_EXTENSION_SET_COMMIT *dwRequestType*. The service returns to the extension agent context information that the extension agent passed to the service. This is the context information the extension agent passed in the *pContextInfo* parameter during the TEST request. The extension agent can use the context information to update the values of the specified variables in an instrumentation-specific manner.

If the extension agent supports rollback processing, it can update the context information in the *pContextInfo* parameter at this time. The SNMP service passes the information back to the extension agent during the UNDO request.

If all extension agents return TRUE to the COMMIT request, the service calls each extension agent with the **SnmpExtensionQueryEx** function, using the SNMP_EXTENSION_SET_CLEANUP *dwRequestType*.

If any extension agent fails the COMMIT request, the service also initiates a CLEANUP request. The service calls each extension agent that previously returned TRUE to the COMMIT request again with the **SnmpExtensionQueryEx** function. The service calls each extension agent using the SNMP_EXTENSION_SET_CLEANUP *dwRequestType*.

CLEANUP request

The service returns to the extension agent the context information passed in the *pContextInfo* parameter during the TEST or COMMIT request. The extension agent must release the resources associated with the parameter at this time.

UNDO request

If any extension agent returns FALSE to the COMMIT request, the SNMP service terminates the COMMIT request. The service calls each extension agent that returned TRUE to the COMMIT request with a *dwRequestType* of SNMP_EXTENSION_SET_UNDO. This signals the extension agents that the COMMIT request failed, and they must initiate rollback processing.

The extension agents must attempt to reset the values of the variables of interest, back to the values they were before the COMMIT request failed. To do this, the extension agents use the context information returned in the *pContextInfo* parameter during the COMMIT request.

If any extension agent returns FALSE to the UNDO request, the entire SET operation fails with the error code SNMP_ERRORSTATUS_UNDOFAILED. If all extension agents return TRUE to the UNDO request, the SNMP SET operation fails with the error code set by the extension agent that failed the COMMIT request.

After the UNDO request the service always calls each extension agent with the **SnmpExtensionQueryEx** function, using the SNMP_EXTENSION_SET_CLEANUP *dwRequestType*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpExtensionQuery**, **AsnOctetString**, **SnmpVarBindList**

SnmpExtensionTrap

The Microsoft SNMP service calls the **SnmpExtensionTrap** function to retrieve information the service needs to generate traps for the SNMP extension agent. The service calls this function only after the extension agent sets the trap event handle to the signaled state during a call to the **SnmpExtensionInit** function. The **SnmpExtensionTrap** function is an element of the SNMP Extension Agent API.

```
BOOL SnmpExtensionTrap(
  AsnObjectIdentifier *pEnterpriseOid,
        // generating enterprise
  AsnInteger32 *pGenericTrapId,
        // generating trap type
  AsnInteger32 *pSpecificTrapId,
        // enterprise-specific type
  AsnTimeticks *pTimeStamp,
        // time stamp
  SnmpVarBindList *pVarBindList
        // variable bindings
);
```

Parameters

pEnterpriseOid

[out] Pointer to an **AsnObjectIdentifier** structure to receive the object identifier of the enterprise that generated the trap. The SNMP service does not free the memory for this variable.

pGenericTrapId

[out] Pointer to a variable to receive an indication of the generic trap. This parameter can be one of the following values.

Value	Meaning
SNMP_GENERICTRAP_ COLDSTART	The agent is initializing protocol entities on the managed mode. It may alter objects in its view.
SNMP_GENERICTRAP_ WARMSTART	The agent is re-initializing itself but will not alter objects within its view.
SNMP_GENERICTRAP_ LINKDOWN	An attached interface has changed from the **up** state to the **down** state. The first variable identifies the interface.
SNMP_GENERICTRAP_ LINKUP	An attached interface has changed from the **down** state to the **up** state. The first variable identifies the interface.
SNMP_GENERICTRAP_ AUTHFAILURE	An SNMP entity has sent an SNMP message, but has falsely claimed to belong to a known community.
SNMP_GENERICTRAP_ EGPNEIGHLOSS	An EGP peer has changed to the **down** state. The first variable identifies the IP address of the EGP peer.
SNMP_GENERICTRAP_ ENTERSPECIFIC	Signals an extraordinary event that is identified in the *pSpecificTrapId* parameter.

pSpecificTrapId

[out] Pointer to a variable to receive an indication of the specific trap generated.

pTimeStamp

[out] Pointer to a variable to receive the time stamp. It is recommended that you initialize this parameter with the value returned by a call to the **SnmpSvcGetUptime** function.

pVarBindList
[out] Pointer to the variable bindings list. The extension agent must allocate the memory for this parameter. The SNMP service frees the memory with a call to the **SnmpUtilVarBindListFree** function.

Return Values

If the **SnmpExtensionTrap** function returns a trap, the return value is TRUE. The SNMP service repeatedly calls the function until it returns a value of FALSE. For additional information, see the following *Remarks* section.

Remarks

The SNMP service repeatedly calls the **SnmpExtensionTrap** function when the *phSubagentTrapEvent* event handle is set to the signaled state. This handle is passed back during the call to the **SnmpExtensionInit** entry point function. The **SnmpExtensionTrap** function must return TRUE to indicate that the parameters contain valid data for a single trap. The function must return FALSE to indicate that the parameters do not represent valid trap data, and to stop the service's repeated calls.

Note that after the SNMP service sends a trap, it frees the memory associated with the variable binding list.

It is important to note that earlier documentation stated that the extension agent should dynamically allocate memory for the enterprise object identifier because the SNMP service would attempt to release the memory after sending a trap. The service will not release the memory associated with the enterprise object identifier. It is recommended that you return a pointer to a static **AsnObjectIdentifier** structure instead.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

+ See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SetEvent**, **SnmpUtilMemAlloc**, **SnmpUtilVarBindListFree**, **SnmpVarBindList**, **SnmpSvcGetUptime**, **SnmpExtensionInit**

SnmpMgrClose

The **SnmpMgrClose** function closes communications sockets and data structures associated with the specified session. This function is an element of the SNMP Management API.

```
BOOL SnmpMgrClose(
  LPSNMP_MGR_SESSION session  // SNMP session pointer
);
```

Parameters

session

[in] Pointer to an internal structure that specifies the session to close.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. This function may return Windows Sockets error codes.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Mgmtapi.h.
Library: Use Mgmtapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpMgrOpen**, **SnmpMgrRequest**

SnmpMgrGetTrap

The **SnmpMgrGetTrap** function returns outstanding trap data that the caller has not received if trap reception is enabled. This function is an element of the SNMP Management API.

```
BOOL SnmpMgrGetTrap(
  AsnObjectIdentifier *enterprise,
        // generating enterprise
  AsnNetworkAddress *IPAddress,
        // generating IP address
  AsnInteger *genericTrap,
        // generic trap type
  AsnInteger *specificTrap,
        // enterprise-specific type
  AsnTimeticks *timeStamp,
        // time stamp
  SnmpVarBindList *variableBindings
        // variable bindings
);
```

Parameters

enterprise
[out] Pointer to an object identifier to receive the enterprise that generated the SNMP trap.

IPAddress
[out] Pointer to a variable to receive the IP address of the enterprise that generated the SNMP trap.

genericTrap
[out] Pointer to a variable to receive an indicator of the generic trap. This parameter can be one of the following values.

Value	Meaning
SNMP_GENERICTRAP_ COLDSTART	The agent is initializing protocol entities on the managed mode. It may alter objects in its view.
SNMP_GENERICTRAP_ WARMSTART	The agent is re-initializing itself but it will not alter objects in its view.
SNMP_GENERICTRAP_ LINKDOWN	An attached interface has changed from the **up** state to the **down** state. The first variable in the variable bindings list identifies the interface.
SNMP_GENERICTRAP_ LINKUP	An attached interface has changed from the **down** state to the **up** state. The first variable in the variable bindings list identifies the interface.
SNMP_GENERICTRAP_ AUTHFAILURE	An SNMP entity has sent an SNMP message, but it has falsely claimed to belong to a known community.
SNMP_GENERICTRAP_ EGPNEIGHLOSS	An EGP peer has changed to the **down** state. The first variable in the variable bindings list identifies the IP address of the EGP peer.
SNMP_GENERICTRAP_ ENTERSPECIFIC	An extraordinary event has occurred and it is identified in the *specificTrap* parameter with an enterprise-specific value.

specificTrap
[out] Pointer to a variable to receive an indication of the specific trap generated.

timeStamp
[out] Pointer to a variable to receive the time stamp.

variableBindings
[out] Pointer to an **SnmpVarBindList** structure to receive the variable bindings list.

Return Values

If the function returns a trap, the return value is nonzero.

You should call the **SnmpMgrGetTrap** function repeatedly until **GetLastError** returns zero. **GetLastError** may also return the following error codes.

Error Code	Meaning
SNMP_MGMTAPI_TRAP_ERRORS	Indicates errors were encountered; traps are not accessible.
SNMP_MGMTAPI_NOTRAPS	Indicates no traps are available.
SNMP_MEM_ALLOC_ERROR	Indicates a memory allocation error.

Remarks

The application must always call the **SnmpMgrTrapListen** function before calling the **SnmpMgrGetTrap** function. This is because the event handle pointed to by the *phTrapAvailable* parameter of the **SnmpMgrTrapListen** function enables the event-driven acquisition of SNMP traps. The SNMP Management API signals an application's event when the SNMP Trap Service delivers a trap.

The application can also poll the **SnmpMgrGetTrap** function for traps at regular intervals. In this case, the application should repeatedly call **SnmpMgrGetTrap** until the function returns zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Mgmtapi.h.
Library: Use Mgmtapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpMgrTrapListen**, **SnmpVarBindList**

SnmpMgrOidToStr

The **SnmpMgrOidToStr** function converts an internal object identifier structure to its string representation. This function is an element of the SNMP Management API.

```
BOOL SnmpMgrOidToStr(
  AsnObjectIdentifier *oid, // object identifier to convert
  LPSTR *string             // string object identifier representation
);
```

Parameters

oid
[in] Pointer to an object identifier variable to convert.

string
[out] Pointer to a null-terminated string to receive the converted value.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. This function may return Windows Sockets error codes.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Mgmtapi.h.
Library: Use Mgmtapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpMgrStrToOid**

SnmpMgrOpen

The **SnmpMgrOpen** function initializes communications sockets and data structures, allowing communications with the specified SNMP agent. This function is an element of the SNMP Management API.

```
LPSNMP_MGR_SESSION SnmpMgrOpen(
  LPSTR lpAgentAddress,
      // name and address of target SNMP agent
  LPSTR lpAgentCommunity,
      // community for target SNMP agent
  INT nTimeOut,
      // communication time-out in milliseconds
  INT nRetries
      // communication time-out or retry count
);
```

Parameters

lpAgentAddress
[in] Pointer to a null-terminated string specifying either a dotted-decimal IP address or a host name that can be resolved to an IP address, an IPX address (in 8.12 notation), or an ethernet address.

lpAgentCommunity
[in] Pointer to a null-terminated string specifying the SNMP community name used when communicating with the agent specified in the *lpAgentAddress* parameter.

nTimeOut
[in] Specifies the communications time-out in milliseconds.

nRetries

[in] Specifies the communications retry count. The time-out specified in the *nTimeOut* parameter is doubled each time a retry attempt is transmitted.

Return Values

If the function succeeds, the return value is a pointer to an **LPSNMP_MGR_SESSION** structure. This structure is used internally and the programmer should not alter it.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**. **GetLastError** may return the SNMP_MEM_ALLOC_ERROR error code, which indicates a memory allocation error.

This function may also return Windows Sockets error codes.

The name and address of the SNMP target, or the string pointed to by the *lpAgentAddress* parameter, should conform to one of the following forms.

Name/Address	Form (example)
IP Address	157.57.8.160
IP Hostname	merlin.microsoft.com
Ethernet Address	00aa00bbccdd
IPX Address	00006112.00aa00bbccdd

Remarks

Names can be provided for agents only if TCP/IP is loaded and the names are TCP/IP host names. NetBIOS names cannot be supplied for IPX hosts.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Mgmtapi.h.
Library: Use Mgmtapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpMgrClose**, **SnmpMgrRequest**

SnmpMgrRequest

The **SnmpMgrRequest** function requests the specified operation be performed with the specified agent. This function is an element of the SNMP Management API.

```
SNMPAPI SnmpMgrRequest(
  LPSNMP_MGR_SESSION session,        // SNMP session pointer
  BYTE requestType,                  // Get, GetNext, or Set
  SnmpVarBindList *variableBindings, // variable bindings
  AsnInteger *errorStatus,           // SNMPv1 error status
  AsnInteger *errorIndex             // error index
);
```

Parameters

session

[in] Pointer to an internal structure that specifies the session that will perform the request.

requestType

[in] Specifies the SNMP request type. This parameter can be one of the following values defined by SNMPv1:

Value	Meaning
SNMP_PDU_GET	Retrieve the value or values of the specified variables.
SNMP_PDU_GETNEXT	Retrieve the value or values of the lexicographic successor of the specified variable.
SNMP_PDU_SET	Write a value within a specific variable.

Note that PDU request types have been renamed. For additional information, see *Modified SNMP PDU Request Types*.

variableBindings

[in/out] Pointer to the variable bindings list.

errorStatus

[out] Pointer to a variable in which the error status result will be returned. This parameter can be one of the following values defined by SNMPv1:

Value	Meaning
SNMP_ERRORSTATUS_NOERROR	The agent reports that no errors occurred during transmission.
SNMP_ERRORSTATUS_TOOBIG	The agent could not place the results of the requested operation into a single SNMP message.
SNMP_ERRORSTATUS_NOSUCHNAME	The requested operation identified an unknown variable.
SNMP_ERRORSTATUS_BADVALUE	The requested operation tried to change a variable but it specified either a syntax or value error.
SNMP_ERRORSTATUS_READONLY	The requested operation tried to change a variable that was not allowed to change according to the community profile of the variable.
SNMP_ERRORSTATUS_GENERR	An error other than one of those listed here occurred during the requested operation.

errorIndex
[out] Pointer to a variable in which the error index result will be returned.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**. **GetLastError** may return one of the following error codes.

Error Code	Meaning
SNMP_MGMTAPI_TIMEOUT	The request timed-out.
SNMP_MGMTAPI_SELECT_FDERRORS	Unexpected error file descriptors indicated by the Windows Sockets **select** function.

Remarks

Retries and time-outs are supplied to the **SnmpMgrOpen** function. Each variable in the variable bindings list must be initialized to type ASN_NULL for **Get** and **Get Next** requests.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Mgmtapi.h.
Library: Use Mgmtapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, SnmpMgrOpen, SnmpMgrClose

SnmpMgrStrToOid

The **SnmpMgrStrToOid** function converts the string format of an object identifier to its internal object identifier structure. This function is an element of the SNMP Management API.

```
BOOL SnmpMgrStrToOid(
  LPSTR string,              // string to convert
  AsnObjectIdentifier *oid   // object identifier
                             // representation
);
```

Parameters

string
[in] Pointer to a null-terminated string to convert.

oid
[out] Pointer to an object identifier variable to receive the converted value.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. This function does not return Windows Sockets error codes.

Remarks

If an application passes a valid object identifier to **SnmpMgrStrToOid**, yet is unable to obtain the requested variable, then the syntax of the system group and object identifier is incorrect. This occurs because **SnmpMgrStrToOid** assumes that the object identifier is under the Internet MIB of the management subtree.

You must always precede the object identifier with a period (.) to obtain the correct system group (for example, ".1.3.6.1.2.1.1"). If an application passes the variable "1.3.6.1.2.1.1", **SnmpMgrStrToOid** cannot interpret the object identifier correctly.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Mgmtapi.h.
Library: Use Mgmtapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpMgrOidToStr**

SnmpMgrTrapListen

The **SnmpMgrTrapListen** function registers the ability of an SNMP manager application to receive SNMP traps from the SNMP Trap Service. This function is an element of the SNMP Management API.

```
BOOL SnmpMgrTrapListen(
  HANDLE *phTrapAvailable  // event handle indicating a trap
                           //  is available
);
```

Parameters

phTrapAvailable

[out] Pointer to an event handle to receive an indication that there are traps available, and that the application should call the **SnmpMgrGetTrap** function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. **GetLastError** may return any of the following error codes.

Error Code	Description
SNMP_MEM_ALLOC_ERROR	Indicates a memory allocation error.
SNMP_MGMTAPI_TRAP_DUPINIT	Indicates that this function has already been called.
SNMP_MGMTAPI_AGAIN	Indicates an error occurred; the application can attempt to call the function again.

This function may return other system errors as well.

Remarks

It is important to note that the **SnmpMgrTrapListen** function succeeds on Windows NT® 4.0 and Windows® 2000 only if the SNMP trap service has been started.

The application must always call the **SnmpMgrTrapListen** function before calling the **SnmpMgrGetTrap** function. This is because the event handle pointed to by the *phTrapAvailable* parameter enables the event-driven acquisition of SNMP traps. The SNMP Management API signals an application's event when the SNMP Trap Service delivers a trap.

The application can also poll the **SnmpMgrGetTrap** function for traps at regular intervals. In this case, the application should repeatedly call **SnmpMgrGetTrap** until the function returns zero.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Mgmtapi.h.
Library: Use Mgmtapi.lib.

+ See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpMgrGetTrap**

SnmpSvcGetUptime

The **SnmpSvcGetUptime** function retrieves the number of centiseconds that the SNMP service has been running. This function is an element of the SNMP Utility API.

```
DWORD SnmpSvcGetUptime();
```

Parameters

This function has no parameters.

Return Values

The function returns a **DWORD** value that is the number of centiseconds the SNMP service has been running.

Remarks

An extension agent should call the **SnmpSvcGetUptime** function only if the extension agent DLL is loaded within the address space of the SNMP service.

The SNMP extension agent DLL is encouraged to use the **SnmpSvcGetUptime** function to retrieve the number of centiseconds that the SNMP service has been running. Extension agents should use **SnmpSvcGetUptime** rather than calculate the uptime using the *dwUptimeReference* parameter. The service passes this parameter to the extension agent as the result of a call to the **SnmpExtensionInit** function. Because the *dwUptimeReference* parameter stores the elapsed time as a **DWORD** value in milliseconds, the time can wrap to zero and reflect an inaccurate time interval.

An extension agent that sends traps must initialize the *timeStamp* parameter to the **SnmpExtensionTrap** function with the value returned by a call to the **SnmpSvcGetUptime** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpExtensionInit**, **SnmpExtensionTrap**

SnmpSvcSetLogLevel

The **SnmpSvcSetLogLevel** function adjusts the level of detail of the debug output from the SNMP service and from SNMP extension agents using the **SnmpUtilDbgPrint** function. This function is an element of the SNMP Utility API.

```
VOID SnmpSvcSetLogLevel(
  INT nLogLevel  // level of severity of the event
);
```

Parameters

nLogLevel

Specifies a signed integer variable that indicates the level of detail of the debug output from the **SnmpUtilDbgPrint** function. This parameter can be one of the following values.

Value	Meaning
SNMP_LOG_SILENT	Disable all debugging output.
SNMP_LOG_FATAL	Display fatal errors only.
SNMP_LOG_ERROR	Display recoverable errors.
SNMP_LOG_WARNING	Display warnings and recoverable errors.
SNMP_LOG_TRACE	Display trace information.
SNMP_LOG_VERBOSE	Display verbose trace information.

Return Values

None.

Remarks

Extension agents are encouraged to use the **SnmpSvcSetLogType** and **SnmpSvcSetLogLevel** functions during development to adjust the output of debugging information. Extension agents can integrate the information with the debug output from the SNMP service.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilDbgPrint**, **SnmpSvcSetLogType**

SnmpSvcSetLogType

The **SnmpSvcSetLogType** function adjusts the destination for the debug output from the SNMP service and from SNMP extension agents using the **SnmpUtilDbgPrint** function. This function is an element of the SNMP Utility API.

```
VOID SnmpSvcSetLogType(
  INT nLogType  // destination for debug output
);
```

Parameters

nLogType

Specifies a signed integer variable that represents the destination for the debug output from the **SnmpUtilDbgPrint** function. This parameter can be one of the following values.

Value	Meaning
SNMP_OUTPUT_TO_CONSOLE	The destination for the debug output is a console window.
SNMP_OUTPUT_TO_LOGFILE	The destination for the debug output is the SNMPDBG.LOG file in the SYSTEM32 directory.
SNMP_OUTPUT_TO_DEBUGGER	The destination for the debug output is a debugger utility.

Return Values

None.

Remarks

Extension agents are encouraged to use the **SnmpSvcSetLogType** and **SnmpSvcSetLogLevel** functions during development to adjust the output of debugging information. Extension agents can integrate the information with the debug output from the SNMP service.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilDbgPrint**, **SnmpSvcSetLogLevel**

SnmpUtilAsnAnyCpy

The **SnmpUtilAsnAnyCpy** function copies the variable pointed to by the *pAnySrc* parameter to the *pAnyDst* parameter. The function allocates any necessary memory for the destination's copy. The **SnmpUtilAsnAnyCpy** function is an element of the SNMP Utility API.

```
SNMPAPI SnmpUtilAsnAnyCpy(
  AsnAny *pAnyDst,  // destination structure
  AsnAny *pAnySrc   // source structure
);
```

Parameters

pAnyDst
Pointer to an **AsnAny** structure to receive the copy.

pAnySrc
Pointer to an **AsnAny** structure to copy.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

Call the **SnmpUtilAsnAnyFree** function to free the memory that the **SnmpUtilAsnAnyCpy** function allocates for the destination structure.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

+ See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **AsnAny**, **SnmpUtilAsnAnyFree**

SnmpUtilAsnAnyFree

The **SnmpUtilAsnAnyFree** function frees the memory allocated for the specified **AsnAny** structure. This function is an element of the SNMP Utility API.

```
VOID SnmpUtilAsnAnyFree(
  AsnAny *pAny  // pointer to structure to free
);
```

Parameters

pAny
Pointer to an **AsnAny** structure whose memory should be freed.

Return Values

None.

Remarks

Call the **SnmpUtilAsnAnyFree** function to free the memory that the **SnmpUtilAsnAnyCpy** function allocates.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **AsnAny**, **SnmpUtilAsnAnyCpy**

SnmpUtilDbgPrint

The **SnmpUtilDbgPrint** function enables debugging output from the SNMP service. This function is an element of the SNMP Utility API:

```
VOID SnmpUtilDbgPrint(
  INT nLogLevel,   // level of severity of event
  LPSTR szFormat   // pointer to a format string
);
```

Parameters

nLogLevel
Specifies a signed integer variable that indicates the level of detail of the log event. This parameter can be one of the following values shown on the next page.

Value	Meaning
SNMP_LOG_SILENT	Disable all debugging output.
SNMP_LOG_FATAL	Display fatal errors only.
SNMP_LOG_ERROR	Display recoverable errors.
SNMP_LOG_WARNING	Display warnings and recoverable errors.
SNMP_LOG_TRACE	Display trace information.
SNMP_LOG_VERBOSE	Display verbose trace information.

szFormat
Pointer to a null-terminated format string that is similar to the standard C library function **printf** style.

Return Values

None.

Remarks

Extension agents are encouraged to use this function during development to enable debug output from the SNMP service.

Use the **SnmpSvcSetLogLevel** function to set the level of detail of the debug output from the SNMP service or from an extension agent's call to the **SnmpUtilDbgPrint** function. Call the **SnmpSvcSetLogType** function to specify the destination for the debug output.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpSvcSetLogType**, **SnmpSvcSetLogLevel**

SnmpUtilIdsToA

The **SnmpUtilIdsToA** function converts an object identifer (OID) to a null-terminated string. This function is an element of the SNMP Utility API.

```
LPSTR SnmpUtilIdsToA(
  UINT *Ids,      // object identifier to convert
  UINT IdLength  // number of elements
);
```

Parameters

Ids

[in] Pointer to an array of unsigned integers. The array contains the sequence of numbers that the OID contains. The *IdLength* parameter specifies the array's length.

For more information, see the following Return Values and Remarks sections.

IdLength

[in] Specifies the number of elements in the array pointed to by the *Ids* parameter.

Return Values

The function returns a null-terminated string that contains the string representation of the array of numbers pointed to by the *Ids* parameter. The string contains a sequence of numbers separated by periods ("."); for example, 1.3.6.1.4.1.311.

If the *Ids* parameter is null, or if the *IdLength* parameter specifies zero, the function returns the string "<null oid>".

The maximum length of the returned string is 256 characters. If the string's length exceeds 256 characters, the string is truncated and terminated with a sequence of three periods ("...").

Remarks

The **SnmpUtilIdsToA** function can assist with the debugging of SNMP applications.

Note that the following memory restrictions apply when you call **SnmpUtilIdsToA**:

- The *Ids* parameter must point to a valid memory block of at least *IdLength* integers, or the function call results in an access violation exception.
- The string returned by **SnmpUtilIdsToA** resides in memory that the SNMP Utility API allocates. The application should not make any assumptions about the memory allocation. The data is guaranteed to be valid until you call **SnmpUtilIdsToA** again, so before calling the function again you should copy the data to another location.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilOidToA**

SnmpUtilMemAlloc

The **SnmpUtilMemAlloc** function allocates dynamic memory from the process heap. This function is an element of the SNMP Utility API.

```
LPVOID SnmpUtilMemAlloc(
  UINT nBytes  // bytes to allocate for object
);
```

Parameters

nBytes

[in] Specifies the number of bytes to allocate for the memory object.

Return Values

If the function succeeds, the return value is a pointer to the newly allocated memory object.

If the function fails, the return value is NULL.

Remarks

Use the **SnmpUtilMemFree** function to release memory that the **SnmpUtilMemAlloc** function allocates.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilMemFree**, **SnmpUtilMemReAlloc**

SnmpUtilMemFree

The **SnmpUtilMemFree** function frees the specified memory object. This function is an element of the SNMP Utility API.

```
VOID SnmpUtilMemFree(
  LPVOID pMem  // pointer to memory object to release
);
```

Parameters

pMem
[in/out] Pointer to the memory object to release.

Return Values

None.

Remarks

Call the **SnmpUtilMemAlloc** function to allocate the memory for the object.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilMemAlloc**, **SnmpUtilMemReAlloc**

SnmpUtilMemReAlloc

The **SnmpUtilMemReAlloc** function changes the size of the specified memory object. This function is an element of the SNMP Utility API.

```
LPVOID SnmpUtilMemReAlloc(
  LPVOID pMem,  // pointer to memory object
  UINT nBytes   // bytes to allocate
);
```

Parameters

pMem
[in] Pointer to the memory object to resize.

nBytes
[in] Specifies the number of bytes to allocate for the new memory object.

Return Values

If the function succeeds, the return value is a pointer to the newly allocated memory object.

If the function fails, the return value is NULL.

Remarks

Call the **SnmpUtilMemFree** function to release memory that the **SnmpUtilMemReAlloc** function allocates.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilMemAlloc**, **SnmpUtilMemFree**

SnmpUtilOctetsCmp

The **SnmpUtilOctetsCmp** function compares two octet strings. This function is an element of the SNMP Utility API.

```
SNMPAPI SnmpUtilOctetsCmp(
  AsnOctetString *pOctets1,  // first octet string
  AsnOctetString *pOctets2   // second octet string
);
```

Parameters

pOctets1
  Pointer to an **AsnOctetString** structure to compare.

pOctets2
  Pointer to a second **AsnOctetString** structure to compare.

Return Values

The function returns a value greater than zero if *pOctets1* is greater than *pOctets2*, zero if *pOctets1* equals *pOctets2*, and less than zero if *pOctets1* is less than *pOctets2.*

Remarks

The **SnmpUtilOctetsCmp** function calls the **SnmpUtilOctetsNCmp** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **AsnOctetString**, **SnmpUtilOctetsNCmp**

SnmpUtilOctetsCpy

The **SnmpUtilOctetsCpy** function copies the variable pointed to by the *pOctetsSrc* parameter to the variable pointed to by the *pOctetsDst* parameter. The function allocates any necessary memory for the destination's copy. The **SnmpUtilOctetsCpy** function is an element of the SNMP Utility API.

```
SNMPAPI SnmpUtilOctetsCpy(
  AsnOctetString *pOctetsDst,  // destination octet string
  AsnOctetString *pOctetsSrc   // source octet string
);
```

Parameters

pOctetsDst
Pointer to an **AsnOctetString** structure to receive the copy.

pOctetsSrc
Pointer to an **AsnOctetString** structure to copy.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

Call the **SnmpUtilOctetsFree** function to free the memory that the **SnmpUtilOctetsCpy** function allocates for the destination structure.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **AsnOctetString**, **SnmpUtilOctetsFree**

SnmpUtilOctetsFree

The **SnmpUtilOctetsFree** function frees the memory allocated for the specified octet string. This function is an element of the SNMP Utility API.

```
VOID SnmpUtilOctetsFree(
  AsnOctetString *pOctets  // octet string to free
);
```

Parameters

pOctets
Pointer to an **AsnOctetString** structure whose memory should be freed.

Return Values

None.

Remarks

Call the **SnmpUtilOctetsFree** function to free the memory that the **SnmpUtilOctetsCpy** function allocates.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **AsnOctetString**, **SnmpUtilOctetsCpy**

SnmpUtilOctetsNCmp

The **SnmpUtilOctetsNCmp** function compares two octet strings. The function compares the subidentifiers in the strings until it reaches the number of subidentifiers specified by the *nChars* parameter. **SnmpUtilOctetsNCmp** is an element of the SNMP Utility API.

```
SNMPAPI SnmpUtilOctetsNCmp(
  AsnOctetString *pOctets1,  // first octet string
  AsnOctetString *pOctets2,  // second octet string
  UINT nChars                // maximum length to compare
);
```

Parameters

pOctets1
Pointer to an **AsnOctetString** structure to compare.

pOctets2
Pointer to a second **AsnOctetString** structure to compare.

nChars
Specifies the number of subidentifiers to compare.

Return Values

The function returns a value greater than zero if *pOctets1* is greater than *pOctets2*, zero if *pOctets1* equals *pOctets2*, and less than zero if *pOctets1* is less than *pOctets2.*

Remarks

The **SnmpUtilOctetsCmp** function calls the **SnmpUtilOctetsNCmp** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **AsnOctetString**, **SnmpUtilOctetsCmp**

SnmpUtilOidAppend

The **SnmpUtilOidAppend** function appends the source object identifier to the destination object identifier. This function is an element of the SNMP Utility API.

```
SNMPAPI SnmpUtilOidAppend(
  AsnObjectIdentifier *pOidDst, // destination object
                                // identifier
  AsnObjectIdentifier *pOidSrc  // source object identifier
);
```

Parameters

pOidDst

[in/out] Pointer to an **AsnObjectIdentifier** structure to receive the source structure.

pOidSrc

[in] Pointer to an **AsnObjectIdentifier** structure to append.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. This function does not generate Windows Sockets errors. The application should call the **GetLastError** function. **GetLastError** may return the following error codes.

Error Code	Description
SNMP_BERAPI_OVERFLOW	Indicates an overflow condition
SNMP_MEM_ALLOC_ERROR	Indicates a memory allocation error

Remarks

The **SnmpUtilOidAppend** function calls the **SnmpUtilMemReAlloc** function. The **SnmpUtilMemReAlloc** function expands the buffer for the destination object identifier.

Call the **SnmpUtilOidFree** function to free memory that the **SnmpUtilOidAppend** function allocates for the destination.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

+ See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilMemReAlloc**, **SnmpUtilOidFree**

SnmpUtilOidCmp

The **SnmpUtilOidCmp** function compares two object identifiers. This function is an element of the SNMP Utility API.

```
SNMPAPI SnmpUtilOidCmp(
  AsnObjectIdentifier *pOid1,  // first object identifier
  AsnObjectIdentifier *pOid2   // second object identifier
);
```

Parameters

pOid1
  [in] Pointer to an **AsnObjectIdentifier** structure to compare.

pOid2
  [in] Pointer to a second **AsnObjectIdentifier** structure to compare.

Return Values

The function returns a value greater than zero if *pOid1* is greater than *pOid2*, zero if *pOid1* equals *pOid2*, and less than zero if *pOid1* is less than *pOid2*.

Remarks

The **SnmpUtilOidCmp** function calls the **SnmpUtilOidNCmp** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilOidNCmp**

SnmpUtilOidCpy

The **SnmpUtilOidCpy** function copies the variable pointed to by the *pOidSrc* parameter to the *pOidDst* parameter, allocating any necessary memory for the destination's copy. This function is an element of the SNMP Utility API.

```
SNMPAPI SnmpUtilOidCpy(
  AsnObjectIdentifier *pOidDst,  // destination object
                                 // identifier
  AsnObjectIdentifier *pOidSrc   // source object identifier
);
```

Parameters

pOidDst
: [out] Pointer to an **AsnObjectIdentifier** structure to receive the copy.

pOidSrc
: [in] Pointer to an **AsnObjectIdentifier** structure to copy.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

Call the **SnmpUtilOidFree** function to free memory that the **SnmpUtilOidCpy** function allocates for the destination structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilOidFree**

SnmpUtilOidFree

The **SnmpUtilOidFree** function frees the memory allocated for the specified object identifier. This function is an element of the SNMP Utility API.

```
VOID SnmpUtilOidFree(
  AsnObjectIdentifier *pOid  // object identifier to free
);
```

Parameters

pOid
: [in/out] Pointer to an **AsnObjectIdentifier** structure whose memory should be freed.

Return Values

None.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilOidAppend**

SnmpUtilOidNCmp

The **SnmpUtilOidNCmp** function compares two object identifiers. The function compares the subidentifiers in the variables until it reaches the number of subidentifiers specified by the *nSubIds* parameter. **SnmpUtilOidNCmp** is an element of the SNMP Utility API.

```
SNMPAPI SnmpUtilOidNCmp(
  AsnObjectIdentifier *pOid1,  // first object identifier
  AsnObjectIdentifier *pOid2,  // second object identifier
  UINT nSubIds                 // maximum length to compare
);
```

Parameters

pOid1
: [in] Pointer to an **AsnObjectIdentifier** structure to compare.

pOid2
: [in] Pointer to a second **AsnObjectIdentifier** structure to compare.

nSubIds
: [in] Specifies the number of subidentifiers to compare.

Return Values

The function returns a value greater than zero if *pOid1* is greater than *pOid2*, zero if *pOid1* equals *pOid2*, and less than zero if *pOid1* is less than *pOid2*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilOidCmp**

SnmpUtilOidToA

The **SnmpUtilOidToA** function converts an object identifier (OID) to a null-terminated string. This function is an element of the SNMP Utility API.

```
LPSTR SnmpUtilOidToA(
  AsnObjectIdentifier *Oid  // object identifier to convert
);
```

Parameters

Oid

[in] Pointer to an **AsnObjectIdentifier** structure to convert.

Return Values

The function returns a null-terminated string of characters that contains the string representation of the object identifier pointed to by the *Oid* parameter.

Remarks

The **SnmpUtilOidToA** function can assist with the debugging of SNMP applications.

For more information, see the **SnmpUtilIdsToA** function. **SnmpUtilOidToA** calls **SnmpUtilIdsToA** internally to format the string.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilIdsToA**, **AsnObjectIdentifier**

SnmpUtilPrintAsnAny

The **SnmpUtilPrintAsnAny** function prints the value of the *Any* parameter to the standard output. This function is an element of the SNMP Utility API.

```
VOID SnmpUtilPrintAsnAny(
  AsnAny *pAny  // pointer to value to print
);
```

Parameters

pAny
[in] Pointer to an **AsnAny** structure for a value to print.

Return Values

None.

Remarks

Use the **SnmpUtilPrintAsnAny** function for debugging and development purposes. This function does not generally print the data in a form that a manager application would typically need.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **AsnAny**

SnmpUtilPrintOid

The **SnmpUtilPrintOid** function formats the specified object identifier (OID) and prints the result to the standard output device. This function is an element of the SNMP Utility API.

```
VOID SnmpUtilPrintOid(
  AsnObjectIdentifier *Oid  // object identifier to print
);
```

Parameters

Oid
[in] Pointer to an **AsnObjectIdentifier** structure to print.

Return Values

None.

Remarks

The **SnmpUtilPrintOid** function can assist with the debugging of command-line SNMP applications. The function prints the object identifier as a sequence of numbers separated by periods ("."); for example, 1.3.6.1.4.1.311.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpUtilDbgPrint**, **AsnObjectIdentifier**

SnmpUtilVarBindCpy

The **SnmpUtilVarBindCpy** function copies the specified **SnmpVarBind** structure, and allocates any memory necessary for the destination structure. The **SnmpUtilVarBindCpy** function is an element of the SNMP Utility API.

```
SNMPAPI SnmpUtilVarBindCpy(
  SnmpVarBind *pVbDst,  // destination variable bindings
  SnmpVarBind *pVbSrc   // source variable bindings
);
```

Parameters

pVbDst
[out] Pointer to an **SnmpVarBind** structure to receive the copy.

pVbSrc
[in] Pointer to an **SnmpVarBind** structure to copy.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

Call the **SnmpUtilVarBindFree** function to free memory that the **SnmpUtilVarBindCpy** function allocates for the destination structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpVarBind**, **SnmpUtilVarBindFree**

SnmpUtilVarBindListCpy

The **SnmpUtilVarBindListCpy** function copies the specified **SnmpVarBindList** structure, and allocates any necessary memory for the destination's copy. This function is an element of the SNMP Utility API.

```
SNMPAPI SnmpUtilVarBindListCpy(
  SnmpVarBindList *pVblDst, // destination variable
                            // bindings list
  SnmpVarBindList *pVblSrc  // source variable bindings list
);
```

Parameters

pVblDst
[out] Pointer to an **SnmpVarBindList** structure to receive the copy.

pVblSrc
[in] Pointer to an **SnmpVarBindList** structure to copy.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

Call the **SnmpUtilVarBindListFree** function to free memory that the **SnmpUtilVarBindListCpy** function allocates for the destination structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpVarBindList**, **SnmpUtilVarBindListFree**, **SnmpUtilOidCpy**

SnmpUtilVarBindFree

The **SnmpUtilVarBindFree** function frees the memory allocated for an **SnmpVarBind** structure. This function is an element of the SNMP Utility API.

```
VOID SnmpUtilVarBindFree(
  SnmpVarBind *pVb  // variable binding to free
);
```

Parameters

pVb

[in/out] Pointer to an **SnmpVarBind** structure whose memory should be freed.

Return Values

None.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpVarBind**, **SnmpUtilVarBindListFree**

SnmpUtilVarBindListFree

The **SnmpUtilVarBindListFree** function frees the memory allocated for an **SnmpVarBindList** structure. This function is an element of the SNMP Utility API.

```
VOID SnmpUtilVarBindListFree(
  SnmpVarBindList *pVbl  // variable bindings list to free
);
```

Parameters

pVbl

[in/out] Pointer to an **SnmpVarBindList** structure whose allocated memory should be freed.

Return Values

No return value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.
Library: Use Snmpapi.lib.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Functions, **SnmpVarBindList**, **SnmpUtilVarBindFree**

SNMP Structures

The following structures are used with SNMP.

AsnAny
AsnCounter64
AsnObjectIdentifier
AsnOctetString
SnmpVarBind
SnmpVarBindList

AsnAny

The **AsnAny** structure contains an SNMP variable type and value. This structure is a member of the **SnmpVarBind** structure that is used as a parameter in many of the SNMP functions. This structure is not used by the WinSNMP API functions.

```
typedef struct {
    BYTE asnType;
     union {
       AsnInteger32         number;       // ASN_INTEGER
                                          // ASN_INTEGER32
       AsnUnsigned32        unsigned32;   // ASN_UNSIGNED32
       AsnCounter64         counter64;    // ASN_COUNTER64
       AsnOctetString       string;       // ASN_OCTETSTRING
       AsnBits              bits;         // ASN_BITS
       AsnObjectIdentifier  object;   // ASN_OBJECTIDENTIFIER
```

```
        AsnSequence         sequence;     // ASN_SEQUENCE
        AsnIPAddress        address;      // ASN_IPADDRESS
        AsnCounter32        counter;      // ASN_COUNTER32
        AsnGauge32          gauge;        // ASN_GAUGE32
        AsnTimeticks        ticks;        // ASN_TIMETICKS
        AsnOpaque           arbitrary;    // ASN_OPAQUE
} asnValue;
} AsnAny;
```

Members

asnType

Indicates the variable's type. This member must be only one of the following values.

Value	Meaning
ASN_INTEGER	Indicates a 32-bit signed integer variable.
ASN_INTEGER32	Indicates a 32-bit signed integer variable.
ASN_UNSIGNED32	Indicates a 32-bit unsigned integer variable.
ASN_COUNTER64	Indicates a counter variable that increases until it reaches a maximum value of (2^64)–1.
ASN_OCTETSTRING	Indicates an octet string variable.
ASN_BITS	Indicates a variable that is an enumeration of named bits.
ASN_OBJECTIDENTIFIER	Indicates an object identifier variable.
ASN_SEQUENCE	Indicates an ASN sequence variable.
ASN_IPADDRESS	Indicates an IP address variable.
ASN_COUNTER32	Indicates a counter variable.
ASN_GAUGE32	Indicates a gauge variable.
ASN_TIMETICKS	Indicates a timeticks variable.
ASN_OPAQUE	Indicates an opaque variable.

asnValue

Contains the variable's value. This member can be only one of the following values.

Value	Meaning
number	Accesses a 32-bit signed integer variable.
unsigned32	Accesses a 32-bit unsigned integer variable.
counter64	Accesses a counter variable that increases until it reaches a maximum value of (2^64)–1.
String	Accesses an octet string variable.

(continued)

(continued)

Value	Meaning
bits	Accesses a variable that is an enumeration of named bits with non-negative, contiguous values, starting at zero.
Object	Accesses an object identifier variable.
sequence	Accesses an ASN sequence variable.
address	Accesses an IP address variable.
counter	Accesses a counter variable that increases until it reaches a maximum value of (2^32)–1.
Gauge	Accesses a gauge variable.
ticks	Accesses a timeticks counter variable that is relative to a specific timer event.
Arbitrary	Accesses an opaque variable.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Structures, **SnmpVarBind**, **SnmpExtensionMonitor**

AsnCounter64

The **AsnCounter64** structure contains a 64-bit unsigned integer value and represents a 64-bit counter. This structure is used by multiple SNMP functions. This structure is not used by the WinSNMP API functions.

```
typedef struct {
    ULONG   LowPart;    // low-order 32 bits
    ULONG   HighPart;   // high-order 32 bits
} AsnCounter64;
```

Members

LowPart
Specifies the low-order 32 bits of the counter.

HighPart
Specifies the high-order 32 bits of the counter.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Structures, **AsnAny**

AsnObjectIdentifier

The **AsnObjectIdentifier** structure represents object identifiers. This structure is used by multiple SNMP functions. This structure is not used by the WinSNMP API functions.

```
typedef struct {
    UINT idLength;  // length of oid
    UINT * ids ;    // pointer to oid array
} AsnObjectIdentifier;
```

Members

idLength
Specifies the number of integers in the object identifier.

ids
Pointer to an array of integers that represents the object identifier.

Remarks

None.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Structures

AsnOctetString

The **AsnOctetString** structure contains octet quantities, usually bytes. This structure is used by multiple SNMP functions. This structure is not used by the WinSNMP API functions.

```
typedef struct {
    BYTE  * stream;  // pointer to octet stream
    UINT  length;    // number of octets in stream
    BOOL  dynamic;   // dynamic allocation flag
} AsnOctetString;
```

Members

stream

Pointer to the octet stream.

length

The number of octets in the data stream.

dynamic

Flag that specifies whether the data stream has been dynamically allocated with the **SnmpUtilMemAlloc** function.

Remarks

Use the **AsnOctetString** structure to transfer string values. For example, use it to transfer the string that represents a computer name as a MIB object value.

You must check the flag specified in the **dynamic** member before you release the data stream of an octet string. Call the **SnmpUtilMemFree** function only if the **dynamic** member is set to TRUE.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Structures, **SnmpUtilMemFree**, **SnmpUtilMemAlloc**

SnmpVarBind

The **SnmpVarBind** structure represents an SNMP variable binding. This structure is used by multiple SNMP functions. This structure is not used by the WinSNMP API functions.

```
typedef struct {
    AsnObjectName   name;   // variable name
    AsnObjectSyntax value;  // variable value
} SnmpVarBind;
```

Members

name

Indicates the variable's name, as an object identifier.

value

Contains the variable's value.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Structures

SnmpVarBindList

The **SnmpVarBindList** structure represents an SNMP variable bindings list. This structure is used by multiple SNMP functions. This structure is not used by the WinSNMP API functions.

```
typedef struct {
    SnmpVarBind *list;   // pointer to variable bindings
    UINT         len;    // number of variable binding entries
} SnmpVarBindList;
```

Members

list

A pointer that references an array to access individual variable bindings.

len

Contains the number of variable bindings in the list.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Snmp.h.

See Also

Simple Network Management Protocol (SNMP) Overview, SNMP Structures, **SnmpVarBind**

CHAPTER 15

The WinSNMP API

The Microsoft® Windows® SNMP Application Programming Interface (the WinSNMP API) versions 1.1a and 2.0 allow you to develop SNMP-based network management applications that execute in the Windows® 2000 operating environment. The Simple Network Management Protocol (SNMP) is a request-response protocol that transfers management information between protocol entities.

WinSNMP has been jointly developed with the cooperation, input, and support from several companies, associations and individuals.

The first part of this overview provides information about the WinSNMP 2.0 Addendum, SNMP versions, and a list of the relevant Requests for Comments (RFCs). It also describes the WinSNMP model, and the components and features of the Microsoft WinSNMP implementation. It also provides information about data management and communications concepts you need to program in the WinSNMP environment.

The second section discusses the WinSNMP functions and the following related WinSNMP programming tasks:

- Opening and closing a WinSNMP application
- Opening and closing a WinSNMP session
- Managing traps and notifications
- Working with variable binding lists
- Working with protocol data units
- Sending SNMP messages
- Receiving SNMP messages
- Managing object identifiers
- Freeing WinSNMP descriptors
- Setting the entity and context translation mode
- Managing the retransmission policy
- Writing WinSNMP applications with multiple threads
- Registering an SNMP agent application

You should be familiar with the basic concepts of SNMP and Windows programming before reading this overview. For more information about SNMP, see *Simple Network Management Protocol* and the relevant Requests for Comments (RFCs) which are published by the Internet Engineering Task Force (IETF).

New WinSNMP Programming Elements

The Microsoft® implementation of the WinSNMP API for Windows® 2000 adds support for the following functions. These additions are documented in the WinSNMP version 2.0 Addendum, dated 12/05/97. For more information, see *About the WinSNMP 2.0 Addendum*.

New WinSNMP Functions

The following new WinSNMP 2.0 functions are available to SNMP applications that are compliant with WinSNMP. These functions enable an application to cancel retransmission attempts and time-out notifications for an SNMP message. You can also register and unregister applications as SNMP agents, and modify the port assigned to a destination entity.

When developing new WinSNMP applications, it is recommended that you call the **SnmpCreateSession** function to open a WinSNMP session instead of calling the **SnmpOpen** function.

- **SnmpCancelMsg**
- **SnmpCreateSession**
- **SnmpGetVendorInfo**
- **SnmpListen**
- **SnmpSetPort**

For information about WinSNMP 2.0 features that the Microsoft WinSNMP implementation supports, see the following topics:

- Levels of SNMP Support
- Support for IPX Address Strings in WinSNMP
- Registering an SNMP Agent Application

About the WinSNMP API

The WinSNMP API is an interface for the development of SNMP-enabled network management applications.

The WinSNMP API provides the following features:

- SNMP-enabling technology for network management applications
- SNMP version 1 (SNMPv1) and SNMP version 2C (SNMPv2C) support

In addition to SNMP manager operations, WinSNMP API version 2.0 also supports SNMP agent operations.

The WinSNMP API supports 32-bit applications, and it executes in single- and multi-threaded environments. Support for the WinSNMP API is available for applications that execute in the Windows 2000 operating environment.

About the WinSNMP 2.0 Addendum

The WinSNMP version 2.0 Addendum is an update to the WinSNMP Manager API specification, version 1.1a. (At this time no specification defines version 2.0 of the WinSNMP API.) The Addendum was jointly developed with the cooperation, input, and support from several companies, associations, and individuals.

The WinSNMP 1.1a specification defines the WinSNMP Manager application programming interface for developing SNMP-based network management applications. In the WinSNMP version 2.0 Addendum, the interface is renamed as the WinSNMP API. This is because WinSNMP 2.0 supports both agent and manager SNMP operations. You should be familiar with both documents if you are programming in the WinSNMP environment.

The Microsoft WinSNMP implementation is compliant with WinSNMP 2.0. It supports all the functions and operations defined in both the WinSNMP 1.1a specification and the WinSNMP 2.0 Addendum.

For information about new functionality the addendum provides, see *New WinSNMP Programming Elements*.

About SNMP Versions

The original Internet standard Network Management Framework, described in RFCs 1155, 1157, and 1213, is called the SNMP version 1 (SNMPv1) framework. Relevant portions of the proposed framework for version 2C of the Simple Network Management Protocol (SNMPv2C) are described in RFCs 1901 through 1908.

The WinSNMP API supports the SNMP protocol functionality described in the relevant Internet RFCs. WinSNMP places no constraints on the use of SNMPv1 or SNMPv2C by WinSNMP applications.

A management entity can support a different version of SNMP than the one the WinSNMP application supports. The Microsoft WinSNMP implementation performs the appropriate translations from SNMPv1 to SNMPv2C in accordance with the relevant RFC.

RFCs Relevant to WinSNMP

TCP/IP standards are defined in Requests for Comments (RFCs), which are published by the Internet Engineering Task Force (IETF). The RFCs that are relevant to WinSNMP features are listed in the table on the following page.

RFC number	Title
1155	"Structure and Identification of Management Information for TCP/IP-based Internets"
1157	"A Simple Network Management Protocol (SNMP)"
1213	"Management Information Base for Network Management of TCP/IP-based internets: MIB-II"
1901	"Introduction to Community-based SNMPv2"
1902	"Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1903	"Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1904	"Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1905	"Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1906	"Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1907	"Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)"
1908	"Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework"
2089	"V2ToV1 Mapping SNMPv2 onto SNMPv1 within a bi-lingual SNMP agent"

Software Requirements for the WinSNMP API

A WinSNMP application must access the WinSNMP API through the dynamic-link library WSNMP32.DLL.

The following files are required to support the functionality of the WinSNMP API.

Filename	Description
WSNMP32.LIB	WinSNMP Library
WSNMP32.DLL	Provides WinSNMP interface
WINSNMP.H	WinSNMP header file
SNMPTRAP.EXE	Receives SNMP traps and forwards them to MGMTAPI.DLL, the Microsoft SNMP Manager API library
SNMPAPI.DLL	Provides SNMP utilities

The WinSNMP Model

The WinSNMP-compliant model includes the following basic components:

- A WinSNMP-enabled SNMP network management application, that is, a WinSNMP-compliant *application*
- The WinSNMP function layer
- A WinSNMP-enabled SNMP service provider, that is, a WinSNMP-compliant *implementation*

Network management applications that must convey SNMP messages operate efficiently in an event-driven environment. This is because SNMP is a datagram-based or connectionless protocol between remote entities that do not establish virtual circuits.

Since Microsoft Windows applications are also event-driven, WinSNMP uses a programming model in which the receipt and processing of asynchronous *message-events* drive management applications. An asynchronous message-event can be an SNMP operation request by a manager application, or the response to a request by an agent application.

SNMP sends requests and responses as SNMP messages. An SNMP message is an SNMP Protocol Data Unit (PDU) plus additional message header elements defined by the relevant RFC. For more information, see *About SNMP Messages*, *Working with Variable Binding Lists* and *Working with Protocol Data Units*.

About the Microsoft WinSNMP Implementation

The Microsoft WinSNMP implementation complies with WinSNMP version 2.0. It supports all the functions and operations defined in both the WinSNMP version 1.1a specification and the WinSNMP version 2.0 Addendum. The implementation provides the following services for WinSNMP applications:

- Manages communications to and from management entities. The entity can reside on the local computer or be connected through a local or wide-area network, or the Internet. For more information, see *Levels of SNMP Support*.
- Hides the details of SNMP protocol, Abstract Syntax Notation One (ASN.1), and the Basic Encoding Rules (BER) for describing transfer syntax.
- Verifies the correctness of PDUs and rejects invalid PDUs. For more information, see *Working with Protocol Data Units*.
- Transforms SNMPv2C PDU types when necessary in accordance with the relevant RFCs.
- Converts SNMPv1 traps from an SNMPv1 agent to SNMPv2C traps in accordance with RFC 1908, "Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework." For more information, see *Translating Traps from SNMPv1 to SNMPv2C*.

- Supports the application's retransmission policy and provides retransmission execution support. For more information, see *The WinSNMP Database* and *About Retransmission*.
- Provides entity and context translation support for the application. For more information, see *The WinSNMP Database* and *Setting the Entity and Context Translation Mode*.

For additioinal information about the relationship between a WinSNMP application and the implementation, see *WinSNMP Data Management Concepts* and *WinSNMP Sessions*.

The WinSNMP Database

The Microsoft WinSNMP implementation maintains a store of administrative information in a database. The database includes the following information:

- **Network and version properties**. The implementation uses these properties to determine which network transport protocol and SNMP version framework to use to complete transmission requests. For more information, see *About SNMP Versions*.
- **Entity and context translation mode**. The implementation uses this mode to interpret user-friendly names for SNMP entities and contexts. For more information, see *Setting the Entity and Context Translation Mode*.
- **Retransmission policy setting**. The implementation uses this setting to determine whether or not it should execute the application's retransmission policy. The implementation stores a time-out value and a retry count for each destination entity. For more information, see *About Retransmission* and *Managing the Retransmission Policy*.

Levels of SNMP Support

The Microsoft WinSNMP implementation provides level 2 SNMP communications support. Level 2 supports the Internet Engineering Task Force (IETF) standard SNMPv2 protocol as defined in RFCs 1902-1908. It also supports the community-based message wrapper as defined in RFC 1901 (SNMPv2C).

Level 2 communications support includes message encoding and decoding services, previously called Level 0 communications support in WinSNMP version 1.1a. Level 2 also supports all SNMPv1 protocol operations, previously called Level 1 communications support in WinSNMP version 1.1a.

The implementation returns the maximum level of SNMP communications it supports in response to a call by the WinSNMP application to the **SnmpStartup** function.

If the WinSNMP application uses the implementation for SNMP message encoding and decoding only, the application must perform required transformations that the implementation would have performed. This includes translating SNMPv1 traps returned by a call to the **SnmpRecvMsg** function, to SNMPv2C traps. It also includes translating PDU types defined by SNMPv1 to the relevant PDU type defined by SNMPv2C, in accordance with RFC 1908.

WinSNMP Sessions

A *session* is the basic unit of resource and communications management between a calling WinSNMP application and the Microsoft WinSNMP implementation. It is recommended that an application use sessions to organize its operations and reduce the demand on the implementation's resources.

For more information, see *Opening and Closing a WinSNMP Session*.

WinSNMP Data Management Concepts

The topics in this section cover the major concepts of data management that apply to programming with the WinSNMP API.

- Object identifiers
- WinSNMP Descriptors
- Resource handle objects
- C-style strings
- Allocating WinSNMP memory objects

Object Identifiers

An SNMP *object identifier* uniquely names an object and identifies its location within a Management Information Base (MIB) tree structure. Object identifiers are application-independent Abstract Syntax Notation One (ASN.1) data types that consist of a sequence of non-negative integers, or subidentifiers. Object identifiers must have a minimum of two subidentifiers and they must not exceed 128 subidentifiers.

The WinSNMP programming environment uses the **smiOID** structure to manage object identifiers. The format of the object identifier array in an **smiOID** structure is one subidentifier per array element.

The dotted numeric string representation of an object identifier separates its subidentifiers with periods; for example, “1.2.3.4.5.6”.

For more information, see *The SNMP Management Information Base (MIB)* and the relevant RFCs.

WinSNMP Descriptors

In the WinSNMP programming environment a *descriptor* is one of the following two structures:

- An **smiOCTETS** structure which describes an octet string variable
- An **smiOID** structure which describes an SNMP object identifier variable

A WinSNMP descriptor is a structure that has two members: a length member, **len**, and a pointer member, **ptr**. The **ptr** member points to the octet string or object identifier of interest. The **ptr** member can be either the **smiLPBYTE** or **smiLPUINT32** data type.

An **smiOCTETS** descriptor or an **smiOID** descriptor can be the **value** member of an **smiVALUE** structure. The **smiVALUE** structure describes the value associated with a variable name in a variable binding entry.

The Microsoft WinSNMP implementation allocates and deallocates memory for all output **smiOCTETS** and **smiOID** structures. Therefore, the application must call the **SnmpFreeDescriptor** function to free the memory for the **ptr** member of these structures.

String members in descriptors do not require a NULL terminating byte. For additional information about managing the memory allocated for descriptors, see *Allocating WinSNMP Memory Objects*.

Resource Handle Objects

The structure of a resource object is restricted to the Microsoft WinSNMP implementation. A WinSNMP application can access a resource object with a handle.

The implementation can allocate one of the following types of resource handles for a WinSNMP application.

Handle type	Description
HSNMP_SESSION	Handle to a WinSNMP session
HSNMP_ENTITY	Handle to an SNMP entity
HSNMP_CONTEXT	Handle to a WinSNMP context
HSNMP_PDU	Handle to a protocol data unit
HSNMP_VBL	Handle to a variable binding list

A WinSNMP application can request that the implementation create or delete resource handles, but the implementation performs the operation. For additional information about freeing individual resources, see the ***SnmpFreeDescriptor***, ***SnmpFreeVbl***, ***SnmpFreePdu***, ***SnmpFreeEntity***, and ***SnmpFreeContext*** functions.

C-Style Strings

A WinSNMP application can use NULL-terminated C-style strings to convert entity and object identifier (OID) objects to and from their string representations.

The WinSNMP functions that manipulate C-style strings include **SnmpStrToEntity**, **SnmpEntityToStr**, **SnmpStrToOid**, and **SnmpOidToStr**. Because **SnmpEntityToStr** and **SnmpOidToStr** return a pointer to a C-style string variable, the WinSNMP application must pass an appropriate value in the *size* parameter when it makes calls to these functions.

Note The *context* parameter of the **SnmpStrToContext** and **SnmpContextToStr** functions must be an octet string structure, that is, an **smiOCTETS** structure. The *context* parameter cannot be a C-style string. The string contained in an **smiOCTETS** structure does not require a NULL-terminating byte.

Allocating WinSNMP Memory Objects

Descriptors, resource handles and C-style strings are the three types of memory objects in the WinSNMP programming environment.

The type of object determines whether the Microsoft WinSNMP implementation or the WinSNMP application allocates and deallocates the memory for the object. This reduces unnecessary allocation of temporary buffer space and unnecessary copying of buffers.

The following table summarizes the allocation and deallocation of resources for WinSNMP memory objects.

Object type	Description
smiOID or **smiOCTETS** descriptor	If the WinSNMP application allocates the memory, it must deallocate the memory with a call to an appropriate function. If the implementation allocates the memory, the application must call the **SnmpFreeDescriptor** function to deallocate the memory.
smiVALUE structure	If the **value** member is an **smiOID** or an **smiOCTETS** descriptor, the application must proceed as indicated above for descriptors.
Resource handle	The implementation allocates, manages, and frees the memory.
C-style string	The WinSNMP application must manage and free the memory it allocates.

For more information, see *Freeing WinSNMP Descriptors*.

WinSNMP Communications Management Concepts

The WinSNMP API provides network transport independence for SNMP-based network management applications that execute in the Microsoft Windows® 2000 operating environment. The topics in this section cover concepts of communications management that apply to programming with WinSNMP.

About SNMP Messages

The Simple Network Management Protocol uses *messages* to communicate and exchange information between remote SNMP entities. SNMP messages contain Protocol Data Units (PDUs) plus additional message header elements defined by the relevant RFC. A PDU is a data packet that contains SNMP data components (or fields).

The format of an SNMP message is the same for both SNMPv1 and SNMPv2C. However, SNMPv2C supports additional PDU types. For example, it supports the **SNMP_PDU_GETBULK** request type, which enables an application to efficiently retrieve large blocks of data from target entities.

Translating Message Requests

If a WinSNMP application requests functionality that is available under the SNMP version 2C framework (SNMPv2C), but the target entity supports only the SNMP version 1 framework (SNMPv1), the Microsoft WinSNMP implementation attempts to translate the request to the SNMPv1 format. To do this, the implementation uses the procedures defined in RFC 1908, "Coexistence Between Version 1 and Version 2 of the Internet-Standard Network Management Framework." If translation is not possible, the **SnmpSendMsg** function fails with the extended error code SNMPAPI_OPERATION_INVALID.

About Traps and Notifications

One type of message an SNMP agent application can send to a WinSNMP application is an asynchronous message that informs the application of a significant event. An example of a significant event is when a network link shuts down or when an authentication failure occurs.

These types of messages are called *traps* under SNMPv1 and *notifications* under SNMPv2C. The Microsoft WinSNMP implementation always translates SNMPv1 traps to the SNMPv2C format, as defined by RFC 1908.

When you call the **SnmpCreatePdu** function to create a trap PDU, you can create only an SNMPv2C trap PDU. The only type of trap PDU you can update with a call to the **SnmpSetPduData** function is an SNMPv2C trap PDU.

For more information, see the following topics:

- Translating Traps from SNMPv1 to SNMPv2C
- Trap Formats

Translating Traps from SNMPv1 to SNMPv2C

When the Microsoft WinSNMP implementation receives traps from an entity operating under the SNMPv1 framework, it translates the traps to the SNMPv2C format. Therefore, when **SnmpRecvMsg** delivers a trap it is always in the SNMPv2C format. RFC 1908, "Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework," specifies the rules for translating from the SNMPv1 to the SNMPv2C trap format.

A WinSNMP application can check the last variable binding entry in a variable binding list to determine if the entry is a trap translated from the SNMPv1 to the SNMPv2C format. If so, the last variable binding will always be equal to the value "snmpTrapEnterpriseOID.0".

For more information, see *Managing Traps and Notifications*.

Trap Formats

The format of trap PDUs is different than that of other PDUs. The format of SNMPv1 traps and SNMPv2C traps is also different.

Under the SNMPv2C framework, the PDU trap format is a variable binding list of *n* variable binding entries organized in the following manner:

- The first variable binding entry contains a time-stamp.
- The second variable binding entry is an object identifier that identifies the trap.
- The third through *n* variable binding entries, if present, contain additional information associated with the trap.

Under the SNMPv1 framework, the PDU trap format is as follows.

Field	Description
enterprise	Identifies the type of device that generated the trap
agent-addr	Identifies the IP address of the device that generated the trap
generic-trap/specific-trap	Identifies a predefined trap type
time-stamp	Identifies when the trap was generated
variable-bindings	Contains additional information associated with the trap

The **SnmpRecvMsg** function always returns a message in the SNMPv2C format. If the message contains the operation type **SNMP_PDU_TRAP**, the application can read the variable binding entries of the message and retrieve the information associated with the trap.

About Retransmission

A WinSNMP application can make SNMP operation requests in various ways. The application can issue several requests to an SNMP agent, without waiting for a response, or it can issue a single request and wait for the response. Since SNMP can be implemented on multiple transport protocols, delivery mechanisms and reliability characteristics can vary.

When you code the WinSNMP application you must determine the level of reliability you need for communications operations, based on the way the application issues operation requests. Then you must select a retransmission strategy and implement a *retransmission policy.*

A retransmission policy includes a *time-out period* and a *retry count.* A time-out period is the elapsed time, in hundredths of a second, between an application's issuance of an **SnmpSendMsg** request and its receipt of the corresponding message. The application receives the message as a result of a call to the **SnmpRecvMsg** function. The time-out value is the period of time the Microsoft WinSNMP implementation waits for an entity to respond to a communication request. If there is no response within the time-out period, the implementation either retransmits the request if the retry count value specifies retransmission attempts, or it fails the call to **SnmpSendMsg**. A retry count is the maximum number of retransmission attempts the implementation makes if an SNMP transmission request fails.

The implementation stores time-out values and retry counts in its database for the application. The implementation stores individual values for each destination entity.

Applications must establish their own polling frequencies and they must manage timer variables. For more information, see *Managing the Retransmission Policy*.

WinSNMP Programming Tasks

The following table summarizes the basic programming procedures that you must perform to code a WinSNMP application, and the topics that provide information about these tasks.

Programming task	Task-related function and topics
Open the WinSNMP application.	Use **SnmpStartup**. See *Opening and Closing a WinSNMP Application*.
Open one or more WinSNMP sessions.	Use **SnmpCreateSession**. See *Opening and Closing a WinSNMP Session*.
Register to receive traps or notifications.	Use **SnmpRegister**. See *Managing Traps and Notifications*.
Create one or more variable binding lists for incorporation in a PDU.	Use **SnmpCreateVbl**, **SnmpDuplicateVbl**, **SnmpSetVb**. See Working with Variable Binding Lists. **Note** The application may need to call other variable binding functions to create the variable binding list.
Create one or more PDUs for transmission and processing.	Use **SnmpCreatePDU**, **SnmpSetPduData**, **SnmpDuplicatePDU**. See Working with Protocol Data Units. **Note** The application may need to call other PDU functions and WinSNMP utility functions to create the PDU.
Submit one or more SNMP operation requests.	Use **SnmpSendMsg**. See *Sending SNMP Messages*.

Programming task	Task-related function and topics
Retrieve the response to the SNMP operation request.	Use **SnmpRecvMsg**. See *Receiving SNMP Messages*.
Process the request response.	Use application-specific logic.
Close each WinSNMP session.	Use **SnmpClose**. See *Opening and Closing a WinSNMP Session*.
Close the WinSNMP application.	Use **SnmpCleanup**. See *Opening and Closing a WinSNMP Application*.

The following topics contain additional information about other general programming concepts specific to the WinSNMP environment.

General programming tasks	Managing Object Identifiers Freeing WinSNMP Descriptors Setting the Entity and Context Translation Mode Managing the Retransmission Policy Writing WinSNMP Applications with Multiple Threads Registering an SNMP Agent Application

In addition, the WinSNMP application may need to incorporate calls to the following WinSNMP functions: **SnmpFreeVbl**, **SnmpFreeEntity**, **SnmpFreeDescriptor**, **SnmpFreeContext**, and **SnmpFreePdu**. This enables the Microsoft WinSNMP implementation to free WinSNMP memory objects. As a general rule, the WinSNMP application should free all resources allocated as the result of a call to a WinSNMP function. For additional information about deallocating resources, see *Allocating WinSNMP Memory Objects*.

Opening and Closing a WinSNMP Application

The WinSNMP application must call the **SnmpStartup** function successfully before it calls any other WinSNMP function. The **SnmpStartup** function enables the Microsoft WinSNMP implementation to perform initialization procedures. The function also returns to the application the version of the WinSNMP API that the implementation supports, the level of SNMP communications it supports, and the implementation's default translation and retransmission modes.

The WinSNMP application must call the **SnmpCleanup** function when the application no longer requires the implementation's services. Even though **SnmpCleanup** enables the implementation to deallocate all resources allocated to the application, it is recommended that the application first call the **SnmpClose** function once for each open WinSNMP session, to maximize the implementation's performance. The WinSNMP application must call **SnmpCleanup** as the last WinSNMP function before it terminates.

Opening and Closing a WinSNMP Session

To open a session, an application calls the **SnmpCreateSession** function. If the function completes successfully, the Microsoft WinSNMP implementation opens a session, and the function returns a session identifier in the form of an **HSNMP_SESSION** handle. All WinSNMP functions that return handle variables include the session identifier as an input parameter. This enables the implementation to use the handle to efficiently manage resources at the session level.

An application can have multiple sessions open at one time. Multiple sessions within an application can share handle variables.

If the implementation cannot open a session because of resource limitations, it returns SNMPAPI_FAILURE when the application calls **SnmpCreateSession**. If the application then calls the **SnmpGetLastError** function, it returns SNMPAPI_ALLOC_ERROR.

A call to the **SnmpClose** function enables the implementation to free any remaining resources and to close the session.

For more information, see *Resource Handle Objects* and *WinSNMP Sessions*.

Managing Traps and Notifications

The WinSNMP application must *register* to receive traps and notifications by calling the **SnmpRegister** function with SNMPAPI_ON. The application can *unregister* and disable traps and notifications by calling the function with SNMPAPI_OFF.

Several options are available when the application calls **SnmpRegister**. The application can register or unregister for the following traps and notifications:

- One type of trap or notification
- All traps and notifications
- All sources of trap and notification requests
- Traps and notifications from all management entities
- Traps and notifications for every context

To register and receive a predefined trap or notification type, the application must define an object identifier (an **smiOID** structure) for each predefined type. The structure must contain a pattern-matching sequence for the type of trap or notification. RFC 1907, "Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)," defines trap and notification object identifiers.

To retrieve outstanding trap data and notifications for a WinSNMP session, a WinSNMP application must call the **SnmpRecvMsg** function with the session handle returned by the **SnmpCreateSession** function.

For more information, see *Sending SNMP Messages* and *Receiving SNMP Messages*. For additional information about allocation and deallocation of resources for traps and notifications, see *Allocating WinSNMP Memory Objects*.

Multiple Trap Registrations

Several options are available when a WinSNMP application registers a WinSNMP session for traps or notifications. Because of this, an application can call the **SnmpRegister** function multiple times, in effect defining a custom filter for the reception of traps and notifications. For example, you can register for one type of trap or notification, or for all traps and notifications, depending on the value of the *notification* parameter. Additionally, the application can specify values in other parameters to the **SnmpRegister** function to further define the traps and notifications that should reach an application. For more information, see *Managing Traps and Notifications*.

Following are instances in which multiple calls to **SnmpRegister** are redundant. In these instances **SnmpRegister** returns SNMPAPI_SUCCESS if the function completes successfully, but the redundant registration is ineffective.

1. A call to the **SnmpRegister** function requesting filtered delivery of traps and notifications to the session, after a previous call to **SnmpRegister** requesting delivery of all traps and notifications (unfiltered delivery). This call is redundant because the session is already receiving all traps and notifications, including the single type defined by the filter.
2. A duplicate call to **SnmpRegister**, or one in which the parameter list is identical to the parameter list in a previous call to **SnmpRegister** for the session.
3. A call to the **SnmpRegister** function requesting filtered delivery of traps and notifications based on an object identifier (OID) whose prefix is an OID specified in a previous call to **SnmpRegister**. For example, you can specify "1.3.6.1.4.1.311" in the *notification* parameter to receive notifications originating from any Microsoft SNMP agent entity. If you call **SnmpRegister** again and specify "1.3.6.1.4.1.311.1", the second call is redundant because the session is already receiving all traps and notifications that contain the OID prefix "1.3.6.1.4.1.311".

To unregister the session, you must match each unique registration call to the **SnmpRegister** function. Call **SnmpRegister** to unregister the session, and ensure that the first five parameters to **SnmpRegister** are identical to those in the initial registration call. The only difference between the initial call and the unregistering call is that when registering you must specify the value SNMPAPI_ON in the *status* parameter, and when you call the function to unregister, you must specify SNMPAPI_OFF. You do not need to match redundant registration calls to the **SnmpRegister** function. You need only match the first unique registration call.

To change filtering criteria, it may be necessary for an application to first unregister and disable delivery of certain traps or notifications. Then the application can create a new filter by calling **SnmpRegister**, passing appropriate values.

Working with Variable Binding Lists

A *variable binding* is the pairing of an SNMP object instance name with an associated value. A *variable binding list* is a series of variable binding entries. In the WinSNMP programming environment, a Protocol Data Unit (PDU) includes a variable binding list.

The details of the variable binding list structure are restricted to the Microsoft WinSNMP implementation. A WinSNMP application can access a variable binding list with a handle of the type **HSNMP_VBL**. For more information, see *Resource Handle Objects*.

The WinSNMP application can construct and manipulate variable binding lists, and include them in PDUs. To perform these operations, the application uses the WinSNMP variable binding functions. For additional information about working with WinSNMP and variable binding lists, see the following topics:

- Creating a Variable Binding List
- Managing a Variable Binding List

For additional information about variable bindings and variable binding lists, see *RFC1905, "Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2),"* and the WinSNMP *Variable Binding Functions*.

Creating a Variable Binding List

The **SnmpCreateVbl** function creates a new variable binding list. If the WinSNMP application specifies a variable name and a value, the function creates the list and adds the name and value as the first entry in the list. If the application specifies NULL for the variable name, the function creates an empty list.

To copy a variable binding list, call the **SnmpDuplicateVbl** function. The function creates a new variable binding list and initializes the new list with a copy of the data in the source variable binding list.

The **SnmpCreateVbl** function and the **SnmpDuplicateVbl** function allocate any necessary memory for the new variable binding list. The WinSNMP application must release the resources associated with these lists. It is recommended that the application do this by matching each call to **SnmpCreateVbl** and **SnmpDuplicateVbl** with a corresponding call to the **SnmpFreeVbl** function when it is appropriate to free the allocated memory.

Managing a Variable Binding List

The **SnmpGetVb** function retrieves variable binding information from a variable binding list. The function retrieves the variable name and the variable's associated value from the variable binding entry specified by the WinSNMP application.

To update variable binding entries in a variable binding list, call the **SnmpSetVb** function. The **SnmpSetVb** function also appends new variable binding entries to an existing variable binding list.

The WinSNMP application must call the **SnmpDeleteVb** function to remove entries from a variable binding list.

To retrieve, modify or delete a variable binding entry, the WinSNMP application must specify the position of the entry in the variable binding list.

A call to the **SnmpSetPduData** function associates a variable binding list with a PDU. A call to the **SnmpGetPduData** function retrieves a variable binding list from a PDU. An individual variable binding is not directly associated with a PDU, but it is indirectly associated through its inclusion in a variable binding list.

Working with Protocol Data Units

The Simple Network Management Protocol sends operation requests and responses as SNMP messages. An SNMP message is an SNMP protocol data unit (PDU) plus additional message header elements defined by the relevant RFC. A PDU includes a variable binding list.

The structure of a PDU is restricted to the Microsoft WinSNMP implementation. A WinSNMP application can access a PDU with a handle of the type **HSNMP_PDU**. The WinSNMP application must create a PDU before it calls the **SnmpSendMsg** function or the **SnmpEncodeMsg** function.

The application can extract and update the data elements of a PDU, and release resources allocated for PDUs. To perform these operations, the application uses the WinSNMP PDU functions. For additional information about working with PDUs in the WinSNMP programming environment, see the following topics:

- Creating a PDU
- Updating a PDU
- Duplicating a PDU
- Validating a PDU
- Matching Response and Request PDUs

For more information, see *Working with Variable Binding Lists* and *Resource Handle Objects*.

Creating a PDU

To create and initialize a PDU a WinSNMP application calls the **SnmpCreatePdu** function. The application must call **SnmpCreatePdu** before it calls the **SnmpSendMsg** function to request that the Microsoft WinSNMP implementation transmit a PDU. The application must also call **SnmpCreatePdu** before it calls the **SnmpEncodeMsg** function to request encoding of an SNMP message.

The application must call the **SnmpFreePdu** function to release the resources that the **SnmpCreatePdu** function allocates for new PDUs.

Updating a PDU

A WinSNMP application can retrieve and update selected PDU fields by using the **SnmpGetPduData** and the **SnmpSetPduData** functions.

The application can retrieve the PDU type, request identifier, error status, and error index fields from a specific PDU. It can also retrieve the handle to the variable binding list. The application can update the **PDU_type** and **request_id** fields.

If the PDU type is equal to **SNMP_PDU_GETBULK**, the application can also update the **non_repeaters** and the **max_repetitions** fields of the PDU.

Duplicating a PDU

The **SnmpDuplicatePdu** function duplicates a PDU, allocating any necessary memory. To release resources allocated by **SnmpDuplicatePdu** for a new PDU, a WinSNMP application must call the **SnmpFreePdu** function.

Validating a PDU

When the WinSNMP application calls the **SnmpSendMsg** function or the **SnmpEncodeMsg** function, the Microsoft WinSNMP implementation verifies the validity of the PDU and the other function parameters.

The value of one PDU data component (or field) can be valid individually, but it may be invalid in combination with values for other fields. For example, unless the **PDU_type** field of the PDU is **SNMP_PDU_GETBULK** or **SNMP_PDU_RESPONSE**, both the **error_status** and **error_index** fields must be equal to zero. Any other value combination constitutes an invalid PDU.

The implementation rejects invalid PDUs and returns the error status SNMPAPI_FAILURE. It sets an extended error code equal to SNMPAPI_PDU_INVALID.

Matching Response and Request PDUs

The order in which the WinSNMP application receives SNMP responses may not match the order in which the application submits SNMP operation requests. To match the response with the request, the application must use the request identifier field (the **request_id**) of the response.

The **request_id** field is a unique numeric value that identifies the PDU. Applications can assign request identifiers by calling the **SnmpCreatePdu** function or the **SnmpSetPduData** function. Call the **SnmpGetPduData** function to retrieve a PDU's **request_id**.

Assigning Request Identifiers

A WinSNMP application can call the **SnmpCreatePdu** function or the **SnmpSetPduData** function to assign an application-generated request identifier to a PDU. The application must pass the value in the *request_id* parameter.

An application can request that the Microsoft WinSNMP implementation generate and assign a request identifier to a PDU by passing zero in the *request_id* parameter of the **SnmpCreatePdu** function. The application can retrieve the assigned request identifier with a call to the **SnmpGetPduData** function.

To assign a request identifier equal to a specific value, including zero, the application must pass that value in the *request_id* parameter of the **SnmpSetPduData** function.

When the implementation executes the application's retransmission policy, it includes the **request_id** field of the original PDU in each retransmitted SNMP message. For more information, see *About Retransmission* and *Managing the Retransmission Policy*.

Note When the implementation receives traps from an SNMPv1 entity, it assigns a non-zero value to the **request_id** field of the PDU. This value may duplicate a request identifier used by the application in a request PDU. Applications must check for duplication.

Sending SNMP Messages

A WinSNMP application initiates a transmission request by sending an SNMP message. SNMP messages include an SNMP protocol data unit. For more information, see *Working with Protocol Data Units*.

A WinSNMP application must call the **SnmpSendMsg** function to request that the Microsoft WinSNMP implementation transmit the PDU, with the other required message elements defined by the relevant RFC. In addition to the destination entity, the application must specify the source entity and a context for the request. The **SnmpSendMsg** function executes asynchronously.

The WinSNMP application must call the **SnmpRecvMsg** function to retrieve the response to an **SnmpSendMsg** request.

The implementation verifies the validity of the PDU and the other message elements when an application calls **SnmpSendMsg**.

Receiving SNMP Messages

The WinSNMP application must call the **SnmpRecvMsg** function to retrieve the response to an **SnmpSendMsg** request.

The **SnmpCreateSession** function passes an application window handle and a notification message identifier to the Microsoft WinSNMP implementation. When the application window receives this message, it signals the application to call the **SnmpRecvMsg** function using the session handle returned by **SnmpCreateSession**.

The **SnmpRecvMsg** function returns two entity handles, a context handle, and the handle to a PDU. It is recommended that the WinSNMP application free these resources using the **SnmpFreeEntity**, **SnmpFreeContext**, and **SnmpFreePdu** functions.

For additional information about managing the time between a call to the **SnmpSendMsg** function and the receipt of the corresponding response, see *About Retransmission*. For additional information about using the **request_id** PDU field to match a response PDU with its request PDU, see *Matching Response and Request PDUs*.

General WinSNMP Programming Tasks

The following topics contain information about general programming concepts specific to the WinSNMP environment.

Managing Object Identifiers

The WinSNMP API provides several WinSNMP utility functions that simplify the manipulation of object identifiers for WinSNMP applications.

The **SnmpOidToStr** function converts the internal binary representation of an object identifier to its dotted numeric string format. When you call **SnmpOidToStr**, specify a string buffer of MAXOBJIDSTRSIZE length (1408 bytes) to ensure that the output buffer is large enough to hold the converted string. To convert an object identifier from the dotted numeric string format to its internal binary representation, call the **SnmpStrToOid** function.

To copy an SNMP object identifier call the **SnmpOidCopy** function. This function allocates any necessary memory for the new object identifier.

A WinSNMP application must call the **SnmpFreeDescriptor** function to free resources allocated for the **ptr** member of the **smiOID** structure specified by both the **SnmpStrToOid** and the **SnmpOidCopy** functions.

The **SnmpOidCompare** function compares two SNMP object identifiers. The WinSNMP application can specify the number of subidentifiers to compare. Call **SnmpOidCompare** to determine whether two object identifiers have common prefixes.

For additional information about managing the memory allocated for object identifiers, see *Allocating WinSNMP Memory Objects*.

Freeing WinSNMP Descriptors

The WinSNMP programming environment assigns the deallocation of descriptor resources to the WinSNMP implementation or the WinSNMP application, depending on which component allocates the memory for the descriptor.

To free the resources for an **smiOID** or an **smiOCTETS** descriptor, the rules on the following page apply.

- For input parameters

  Because the WinSNMP application allocates the memory for input objects with variable lengths, the application must free that memory using an appropriate function. For example, if the application allocated the resources with a call to the **GlobalAlloc** function, it should use the **GlobalFree** function to deallocate the resources. If the application allocated the resources with a call to the **HeapAlloc** function, it should call the **HeapFree** function.
- For output parameters

  A call to any of the following functions results in the implementation allocating memory for an **smiOID** or an **smiOCTETS** descriptor: **SnmpGetVb**, **SnmpEncodeMsg**, **SnmpOidCopy**, **SnmpContextToStr**, and **SnmpStrToOid**.

  Because the implementation allocates the memory for these output objects, the application must call the **SnmpFreeDescriptor** function to deallocate the resources. This function enables the implementation to free the memory allocated for the **ptr** member of these structures.

To free the resources for an **smiVALUE** structure, the following rules apply:

> A WinSNMP application must check the **syntax** member of an **smiVALUE** structure to correctly evaluate the **value** member of the structure. If the **syntax** member indicates that the **value** member is an **smiOCTETS** or an **smiOID** descriptor, and the implementation allocated the resources for the descriptor, the application must call **SnmpFreeDescriptor**. This enables the implementation to free the memory. If the application allocated the resources, it must free the memory using an appropriate function, as indicated earlier.

For more information, see *Allocating WinSNMP Memory Objects*.

Setting the Entity and Context Translation Mode

The WinSNMP application can specify the interpretation and translation of entity and context parameters by setting the entity and context translation mode. The Microsoft WinSNMP implementation stores the mode in a database.

The setting of the entity and context translation mode determines the manner in which the **SnmpStrToEntity** function and the **SnmpStrToContext** function interpret input strings. The setting also determines the type of output string that the **SnmpEntityToStr** and the **SnmpContextToStr** functions return. For more information, see *Support for IPX Address Strings in WinSNMP*.

The implementation returns the current default entity and context translation mode in the *nTranslateMode* parameter of the **SnmpStartup** function. To retrieve the current entity and context translation mode in effect for the implementation, an application can call the **SnmpGetTranslateMode** function at any time.

The valid entity and context translation modes follow.

Mode	Meaning
SNMPAPI_TRANSLATED	The implementation uses its database to translate user-friendly names for SNMP entities and managed objects. The implementation translates them into their SNMPv1 or SNMPv2C components.
SNMPAPI_UNTRANSLATED_V1	The implementation interprets SNMP entity parameters as literal SNMP transport addresses, and context parameters as literal SNMP community strings. For SNMPv2 destination entities, the implementation creates outgoing SNMP messages that contain a value of zero in the version field.
SNMPAPI_UNTRANSLATED_V2	The implementation interprets SNMP entity parameters as SNMP transport addresses, and context parameters as literal SNMP community strings. For SNMPv2 destination entities, the implementation creates outgoing SNMP messages that contain a value of 1 in the version field.

The implementation tries to associate resources in its database with the literal transport address of the management entity.

To change the entity and context translation mode setting a WinSNMP application must call the **SnmpSetTranslateMode** function. If the requested translation mode is invalid, the function fails, and **SnmpGetLastError** returns the error code SNMPAPI_MODE_INVALID.

Support for IPX Address Strings in WinSNMP

WinSNMP 2.0 formalizes the use of IPX address strings. If you specify an IPX address string as an input parameter to the **SnmpStrToEntity** function, you must format the string using the following standards:

- A network number that consists of eight hexadecimal digits (zero-filled if necessary)
- A separator (either ":", "." or "–")
- A node number that consists of 12 hexadecimal digits (zero-filled if necessary)

For example, 00000001:00081A0D01C2 or 00000001.00081a0d01c2. Hexadecimal digits can be uppercase or lowercase.

This is the format the **SnmpEntityToStr** function uses to return an IPX address string. The string is returned when an application that is operating in one of the SNMPAPI_UNTRANSLATED modes calls the **SnmpEntityToStr** function. The string can also be returned when the application is operating in SNMPAPI_TRANSLATED mode and no user-friendly name is available for the entity.

Managing the Retransmission Policy

The WinSNMP application can request that the Microsoft WinSNMP implementation execute the application's retransmission policy. When the implementation manages retransmission, it uses the time-out period and the retry count values in its database.

The implementation identifies the default retransmission mode in a return value from the **SnmpStartup** function during initialization. The mode can be one of the following values.

Value	Meaning
SNMPAPI_ON	The implementation is executing the application's retransmission policy.
SNMPAPI_OFF	The implementation is not executing the application's retransmission policy.

A WinSNMP application can retrieve at any time the current retransmission mode in effect for the implementation by calling the **SnmpGetRetransmitMode** function. The WinSNMP API provides other database functions that simplify management of the retransmission policy.

At any time during program execution, the WinSNMP application can adjust execution of the policy by performing one of the following steps:

- Request that the implementation start or stop executing the retransmission policy by calling the **SnmpSetRetransmitMode** function. For more information, see *Turning Retransmission On and Off.*
- Modify the time-out period and retry count values in the implementation's database. For more information, see *Changing the Retransmission Policy.*
- Call the the **SnmpCancelMsg** function to cancel the retransmission cycle and release internal data structures associated with a single SNMP message request. For more information, see *Canceling Retransmission.*

The application can execute its own retransmission policy. In this case, execution may or may not be based on the values in the database.

Turning Retransmission On and Off

An application can set the retransmission mode with a call to the **SnmpSetRetransmitMode** function. The new retransmission mode applies to subsequent calls to the **SnmpSendMsg** function.

When the application calls **SnmpSetRetransmitMode** with the retransmission mode value SNMPAPI_ON, the Microsoft WinSNMP implementation begins execution of the application's retransmission policy. The new retransmission mode does not affect outstanding SNMP messages. An outstanding message is one that has no response at the time the application changes the retransmission mode.

When the WinSNMP application calls the **SnmpSetRetransmitMode** function with the retransmission mode value SNMPAPI_OFF, the implementation stops executing the retransmission policy. The implementation cancels retransmission attempts for outstanding SNMP messages. This is because it may not be possible for the implementation to handle all outstanding SNMP requests and operations plus new requests, before an application time-out or retry counter signals an event.

Canceling Retransmission

If there is no response to a communication request within the time-out period specified for a destination entity, and if retransmissions are specified for the entity, the Microsoft WinSNMP implementation retransmits the request. A call to the **SnmpCancelMsg** function can cancel this retransmission cycle and release internal data structures associated with the message request.

Note that it is possible for a destination entity to receive an SNMP message that has been cancelled by a call to the **SnmpCancelMsg** function. It is also possible that a destination entity can respond to a cancelled SNMP message. This is because transaction queuing occurs at multiple levels. However, once a message has been cancelled by a call to **SnmpCancelMsg**, the Microsoft WinSNMP implementation will not retransmit the message, submit a response PDU, or send a time-out notification to the application for that message.

Changing the Retransmission Policy

The Microsoft WinSNMP implementation provides retransmission policy support by storing a time-out value and a retry count for the WinSNMP application in a database. The implementation stores values for individual destination entities. The implementation initially supplies default values for these elements, but an application can add or modify values for individual entities.

A call to the **SnmpGetTimeout** and **SnmpGetRetry** functions returns the time-out and retry count values for a specific destination entity. To change the time-out value, a WinSNMP application must call the **SnmpSetTimeout** function. To change the retry count value the application must call the **SnmpSetRetry** function. The updated settings affect new SNMP message requests to the destination entity.

Writing WinSNMP Applications with Multiple Threads

The Microsoft WinSNMP implementation ensures that the WinSNMP operations of one process do not modify the WinSNMP settings of another process.

A WinSNMP application with multiple threads must ensure that WinSNMP operations that set application-level parameters are thread-safe. The functions that set application-level parameters are **SnmpSetTranslateMode** and **SnmpSetRetransmitMode**. These functions modify settings for the entity and context translation mode and the retransmission mode.

For more information, see *Multiple Threads*.

Registering an SNMP Agent Application

In addition to SNMP manager operations, the WinSNMP API version 2.0 also supports SNMP agent operations.

To register a WinSNMP application as an SNMP agent, the application can call the **SnmpListen** function. This function informs the Microsoft WinSNMP implementation that an SNMP entity will be acting in the role of an SNMP agent. The application can also call **SnmpListen** to inform the implementation when it will no longer be acting as an agent.

If an application calls the **SnmpListen** function and passes the value SNMPAPI_ON in the *lStatus* parameter, the following actions occur:

1. The entity that will be functioning in an SNMP agent role binds to its assigned port, and "listens" for incoming SNMP message requests.
2. The agent uses application-specific logic to process each SNMP request.
3. The agent forms appropriate responses to each request.
4. The agent transmits the response to the requesting entity by calling the **SnmpSendMsg** function. When the agent calls **SnmpSendMsg**, it specifies the address of the agent in the *srcEntity* parameter, and the address of the remote manager entity in the *dstEntity* parameter. (These values are the reverse of the values the agent entity received in these parameters when it called the **SnmpRecvMsg** function to retrieve an SNMP request.)

For more information about SNMP management applications and agent applications, see *About SNMP*.

WinSNMP API Reference

The following sections describe in detail the functions, structures, data types, function return values, and common error codes of the WinSNMP API.

- WinSNMP Data Types
- WinSNMP Common Error Codes
- WinSNMP Function Return Values
- WinSNMP Functions
- WinSNMP Structures

WinSNMP Data Types

The standard WinSNMP data types are defined in the WINSNMP.H file.

Note that WinSNMP specifies some parameters with the signed long integer type, **smiINT**. This enables WinSNMP to comply with the data elements, especially the PDU components, defined in the relevant RFCs.

WinSNMP Error Codes

Note The errors described in this topic are distinct from the SNMP error codes defined by the relevant RFCs.

All WinSNMP functions return the value SNMPAPI_FAILURE if the function fails. The WinSNMP application must immediately call the **SnmpGetLastError** function to retrieve extended error information when a WinSNMP function fails. For additional information about the extended error codes WinSNMP functions return, see the following topics:

- *WinSNMP Common Error Codes*
- *Network Transport Errors*

The WinSNMP errors that convey context-specific information are noted in each function's reference page.

WinSNMP Common Error Codes

The **SnmpGetLastError** function can return a general error code after a WinSNMP function fails. The following table lists the WinSNMP common error codes.

Error code	Meaning	Recommended action
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully either since program execution began, or since a call to the **SnmpCleanup** function completed successfully.	The application should call **SnmpGetLastError** before it calls any other WinSNMP API function when **SnmpStartup** fails. The **SnmpGetLastError** function returns extended error information about the failure of **SnmpStartup**.
SNMPAPI_ALLOC_ERROR	The application specified an invalid pointer, or an error occurred during memory allocation. The Microsoft WinSNMP implementation could not obtain sufficient resources to execute the request.	The application should provide valid memory pointers for all output parameters. It should free resources, reduce resource requirements, or facilitate a graceful shutdown. A graceful shutdown includes multiple calls to the **SnmpClose** function, one for each open WinSNMP session. It also includes a call to the **SnmpCleanup** function.

Error code	Meaning	Recommended action
SNMPAPI_NOOP	The function did not complete successfully because all output parameters are NULL.	The application must specify at least one output parameter that is not NULL when calling a function that returns information to the application.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.	The application should usually respond with a graceful shutdown. A graceful shutdown includes multiple calls to the **SnmpClose** function, one for each open WinSNMP session. It also includes a call to the **SnmpCleanup** function.

The WinSNMP errors that convey context-specific information are noted in each function's reference page.

Network Transport Errors

The Microsoft WinSNMP implementation can detect a network transport error after it transmits an SNMP message. When this occurs, the implementation sends a data receipt notification to the WinSNMP session that initiated the communication request. The implementation also returns SNMPAPI_FAILURE on the next call to the **SnmpRecvMsg** function for the session. The **SnmpRecvMsg** function fails with an extended error code that indicates a network transport layer error.

For a list of specific transport layer errors, see the reference pages for the ***SnmpRegister***, ***SnmpSendMsg***, and ***SnmpRecvMsg*** functions.

WinSNMP Function Return Values

The return value from a WinSNMP function call can be a handle to a resource that the Microsoft WinSNMP implementation allocates for the WinSNMP application.

The following table lists the five types of resource handles that the implementation allocates.

Handle type	Description
HSNMP_SESSION	Handle to a WinSNMP session
HSNMP_ENTITY	Handle to an SNMP entity
HSNMP_CONTEXT	Handle to an SNMP context
HSNMP_PDU	Handle to a protocol data unit
HSNMP_VBL	Handle to a variable binding list

For more information, see *Resource Handle Objects*.

The return value can also be an unsigned long integer value of the **smiUINT32** type that represents an SNMPAPI_STATUS value.

The following table lists the WinSNMP status values and their meaning.

Status	Description
SNMPAPI_FAILURE	Indicates an error. Equates to 0 or NULL.
SNMPAPI_SUCCESS	Indicates the function completed successfully. Equates to 1 or a positive count.

WinSNMP Functions

The functions used with WinSNMP fall into the following functional groupings. An alphabetic list follows.

- Communications Functions
- Entity and Context Functions
- Database Functions
- PDU Functions
- Utility Functions
- Variable Binding Functions
- WinSNMP Functions—Alphabetic List

WinSNMP Communications Functions

The WinSNMP communications functions provide an interface between the calling WinSNMP application and the Microsoft WinSNMP implementation. The implementation handles communication between the application and other management entities.

Function	Description
SnmpCancelMsg	Requests that the Microsoft WinSNMP implementation cancel retransmission attempts and time-out notifications for an SNMP request message.
SnmpCleanup	Informs the implementation that an application is disconnecting and no longer requires allocated resources. An application must call the **SnmpCleanup** function as the last WinSNMP function before it terminates.
SnmpClose	Enables the implementation to deallocate resources associated with a session, and to close communications mechanisms.

Function	Description
SnmpCreateSession	Requests the implementation to open a WinSNMP session and allocate resources and communications mechanisms. When developing new WinSNMP applications, it is recommended that you call the **SnmpCreateSession** function to open a WinSNMP session instead of calling the **SnmpOpen** function.
SnmpListen	Registers or unregisters a WinSNMP application as an SNMP agent.
SnmpOpen	Requests the implementation to open a WinSNMP session and allocate resources and communications mechanisms. When developing new WinSNMP applications, tt is recommended that you call the **SnmpCreateSession** function to open a WinSNMP session instead of calling the **SnmpOpen** function.
SnmpRecvMsg	Returns SNMP messages and outstanding trap data and notifications.
SnmpRegister	Informs the implementation that the application needs to register or unregister for traps and notifications.
SnmpSendMsg	Requests that the implementation transmit a protocol data unit.
SnmpStartup	Notifies the implementation to perform initialization procedures for the application. An application must call the **SnmpStartup** function successfully before it calls any other WinSNMP function.
SNMPAPI_CALLBACK	Notifies a WinSNMP session that an SNMP message or asynchronous event is available. **Note**: This callback function applies only to sessions opened as a result of a call to the **SnmpCreateSession** function.

WinSNMP Entity and Context Functions

The WinSNMP entity and context functions enable a WinSNMP application to specify user-friendly names for SNMP entities and contexts. The Microsoft WinSNMP implementation translates the name to its SNMPv1 or SNMPv2C components using the implementation's database.

Function	Description
SnmpContextToStr	Returns a string that identifies an SNMP context (a set of managed object resources).
SnmpEntityToStr	Returns a string that identifies an SNMP management entity.

(continued)

(continued)

Function	Description
SnmpFreeContext	Releases resources allocated by the **SnmpStrToContext** function for an SNMP context.
SnmpFreeEntity	Releases resources allocated by the **SnmpStrToEntity** function for an SNMP management entity.
SnmpSetPort	Changes the port assigned to an SNMP destination entity.
SnmpStrToContext	Returns a handle to SNMP context information that is specific to the implementation.
SnmpStrToEntity	Returns a handle to SNMP management entity information that is specific to the implementation.

WinSNMP Database Functions

The WinSNMP database functions provide a WinSNMP application with access to the current settings in the Microsoft WinSNMP implementation's store of administrative information. These functions permit changing the retransmission mode and the entity and context translation mode. The database functions also provide the application with the ability to manipulate the time-out and retry count values.

Function	Description
SnmpGetRetransmitMode	Returns the current setting of the retransmission mode.
SnmpGetRetry	Returns the retry count value, in units, for the retransmission of SNMP message requests.
SnmpGetTimeout	Returns the time-out value, in hundredths of a second, for the transmission of SNMP message requests.
SnmpGetTranslateMode	Returns the current setting of the entity and context translation mode.
SnmpGetVendorInfo	Retrieves information that identifies the WinSNMP vendor.
SnmpSetRetransmitMode	Changes the retransmission mode.
SnmpSetRetry	Changes the retry count value for the retransmission of SNMP message requests.
SnmpSetTimeout	Changes the time-out value for the transmission of SNMP message requests.
SnmpSetTranslateMode	Changes the entity and context translation mode.

WinSNMP PDU Functions

The WinSNMP PDU functions enable WinSNMP applications to extract and update the data elements (or fields) of a PDU. This includes PDUs returned by a call to the **SnmpRecvMsg** function or the **SnmpDecodeMsg** function. The PDU functions also construct PDUs for use in the **SnmpSendMsg** and **SnmpEncodeMsg** functions.

Function	Description
SnmpCreatePdu	Creates and initializes an SNMP protocol data unit.
SnmpDuplicatePdu	Duplicates an SNMP protocol data unit.
SnmpFreePdu	Releases resources associated with an SNMP protocol data unit created by the **SnmpCreatePdu** or the **SnmpDuplicatePdu** function.
SnmpGetPduData	Returns selected data elements from a specified SNMP protocol data unit.
SnmpSetPduData	Updates selected data elements in a specified SNMP protocol data unit.

WinSNMP Utility Functions

The WinSNMP utility functions enable a WinSNMP application to manage objects and SNMP messages across the WinSNMP interface.

Function	Description
SnmpDecodeMsg	Decodes an encoded or serialized SNMP message into its constituent components.
SnmpEncodeMsg	Creates an encoded SNMP message.
SnmpFreeDescriptor	Signals the Microsoft WinSNMP implementation that it should free the memory it allocated for a specific descriptor.
SnmpGetLastError	Returns the last-error code value for the last SNMP operation.
SnmpOidCompare	Compares two SNMP object identifiers.
SnmpOidCopy	Copies an SNMP object identifier.
SnmpOidToStr	Converts the internal binary representation of an SNMP object identifier to its dotted numeric string format.
SnmpStrToOid	Converts the dotted numeric string format of an SNMP object identifier to its internal binary representation.

WinSNMP Variable Binding Functions

The WinSNMP variable binding functions enable WinSNMP applications to construct and manipulate variable binding lists, and include them in PDUs.

Function	Description
SnmpCountVbl	Enumerates the variable binding entries in a specified variable binding list.
SnmpCreateVbl	Creates a new variable binding list.

(continued)

(continued)

Function	Description
SnmpDeleteVb	Removes a variable binding entry from a variable binding list.
SnmpDuplicateVbl	Copies a variable binding list.
SnmpFreeVbl	Releases resources for a variable binding list allocated previously by the **SnmpCreateVbl** or the **SnmpDuplicateVbl** function.
SnmpGetVb	Retrieves information from a specified variable binding entry.
SnmpSetVb	Changes variable binding entries in a variable binding list; appends new variable binding entries to an existing variable binding list.

WinSNMP Functions—Alphabetic List

SNMPAPI_CALLBACK
SnmpCancelMsg
SnmpCleanup
SnmpClose
SnmpContextToStr
SnmpCountVbl
SnmpCreatePdu
SnmpCreateSession
SnmpCreateVbl
SnmpDecodeMsg
SnmpDeleteVb
SnmpDuplicatePdu
SnmpDuplicateVbl
SnmpEncodeMsg
SnmpEntityToStr
SnmpFreeContext
SnmpFreeDescriptor
SnmpFreeEntity
SnmpFreePdu
SnmpFreeVbl
SnmpGetLastError
SnmpGetPduData
SnmpGetRetransmitMode
SnmpGetRetry
SnmpGetTimeout
SnmpGetTranslateMode
SnmpGetVb
SnmpGetVendorInfo
SnmpListen
SnmpOidCompare
SnmpOidCopy
SnmpOidToStr
SnmpOpen
SnmpRecvMsg
SnmpRegister
SnmpSendMsg
SnmpSetPduData
SnmpSetPort
SnmpSetRetransmitMode
SnmpSetRetry
SnmpSetTimeout
SnmpSetTranslateMode
SnmpSetVb
SnmpStartup
SnmpStrToContext
SnmpStrToEntity
SnmpStrToOid

SNMPAPI_CALLBACK

The Microsoft WinSNMP implementation calls the **SNMPAPI_CALLBACK** function to notify a WinSNMP session that an SNMP message or asynchronous event is available.

SNMPAPI_CALLBACK is a placeholder for an application- or library-defined callback function name.

```
SNMPAPI_STATUS CALLBACK SNMPAPI_CALLBACK(
  HSNMP_SESSION hSession,  // handle to the WinSNMP session
  HWND hWnd,               // handle to the notification
                           // window
  UINT wMsg,               // window notification message
                           // number
  WPARAM wParam,           // type of notification
  LPARAM lParam,           // request identifier of PDU
  LPVOID lpClientData      // optional application-defined
                           // data
);
```

Parameters

hSession
[in] Handle to the WinSNMP session.

hWnd
[in] Handle to a window of the WinSNMP application to notify when an asynchronous request completes, or when trap notification occurs. This parameter does not have significance for the WinSNMP session, but the implementation always passes the value to the callback function.

wMsg
[in] Specifies an unsigned integer that identifies the notification message to send to the WinSNMP application window. This parameter does not have significance for the WinSNMP session, but the implementation always passes the value to the callback function.

wParam
[in] Specifies an application-defined 32-bit value that identifies the type of notification. If this parameter is equal to zero, an SNMP message is available for the session. The application should call the **SnmpRecvMsg** function to retrieve the message. If this parameter is not equal to zero, it indicates an asynchronous event notification for the session. For additional information, see the following Remarks section.

lParam
[in] Specifies an application-defined 32-bit value that specifies the request identifier of the PDU being processed.

lpClientData

[in] If the *lpClientData* parameter was not NULL on the call to the **SnmpCreateSession** function for this session, this parameter is a pointer to application-defined data.

Return Values

The function must return SNMPAPI_SUCCESS to continue execution of the application. If the function returns any other value, the implementation responds as if the application called the **SnmpClose** function for the indicated session.

Remarks

When the implementation is executing the retransmission policy for the WinSNMP application and a transmission time-out occurs, the implementation informs the session of the error. In this situation the value of the *wParam* parameter would be SNMPAPI_TL_TIMEOUT. For a list of other transport layer errors, see the reference pages for the **SnmpRegister**, **SnmpSendMsg**, and **SnmpRecvMsg** functions.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpCreateSession**, **SnmpClose**

SnmpCancelMsg

A WinSNMP application calls the **SnmpCancelMsg** function to request that the Microsoft WinSNMP implementation cancel retransmission attempts and time-out notifications for an SNMP request message. The **SnmpCancelMsg** function is an element of the WinSNMP API, version 2.0.

```
SNMPAPI_STATUS SnmpCancelMsg(
  HSNMP_SESSION session, // handle to the WinSNMP session
  smiINT32 reqId         // request_id field of the PDU
);
```

Parameters

session

[in] Handle to the WinSNMP session that submitted the SNMP request message (PDU) to be canceled.

reqId

[in] Specifies a unique numeric value that identifies the PDU of interest. This parameter must match the request identifier (**request_id**) of the *PDU* parameter specified in the application's initial call to the **SnmpSendMsg** function.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error code	Description
SNMPAPI_SESSION_INVALID	The *session* parameter is invalid.
SNMPAPI_PDU_INVALID	The *reqId* parameter does not identify an outstanding message for the specified session.
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

Calling the **SnmpCancelMsg** function is equivalent to calling the **SnmpSetRetransmitMode** function, for a specific SNMP message, with the retransmission mode equal to SNMPAPI_OFF.

For more information, see *Canceling Retransmission* and *Managing the Retransmission Policy*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpSendMsg**, **SnmpSetRetransmitMode**

SnmpCleanup

The **SnmpCleanup** function informs the Microsoft WinSNMP implementation that the calling WinSNMP application no longer requires the implementation's services.

Note A WinSNMP application must call the **SnmpCleanup** function as the last WinSNMP function before it terminates.

```
SNMPAPI_STATUS SnmpCleanup(VOID);
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS. Until the WinSNMP application successfully recalls the **SnmpStartup** function, any other call to a WinSNMP function returns SNMPAPI_FAILURE, with an extended error code of SNMPAPI_NOT_INITIALIZED.

If the function fails, the return value is SNMPAPI_FAILURE, but the WinSNMP application does not need to retry the call to **SnmpCleanup**. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

Before the WinSNMP application calls **SnmpCleanup**, it should call the **SnmpClose** function once for each session the implementation opens as a result of a call to the **SnmpCreateSession** function.

When a WinSNMP application calls the **SnmpCleanup** function, the implementation deallocates all resources allocated to the application. However, it is recommended that a WinSNMP application free the specific resources that the implementation allocates for it with the WinSNMP function that corresponds to the resource. For additional information about freeing individual resources, see ***SnmpFreeEntity***, ***SnmpFreeVbl***, ***SnmpFreeDescriptor***, ***SnmpFreeContext***, and ***SnmpFreePdu***.

If a WinSNMP application must perform an emergency exit, and it calls **SnmpCleanup** without freeing individual resources and without calling **SnmpClose** for every open session, the implementation deallocates all resources allocated to the WinSNMP application. However, to enable this functionality in the implementation, the application must still call **SnmpCleanup**.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpCreateSession**, **SnmpStartup**, **SnmpClose**, **SnmpFreeEntity**, **SnmpFreeVbl**, **SnmpFreeDescriptor**, **SnmpFreeContext**, **SnmpFreePdu**

SnmpClose

The **SnmpClose** function enables the Microsoft WinSNMP implementation to deallocate memory, resources, and data structures associated with a WinSNMP session. The WinSNMP **SnmpClose** function also closes communications mechanisms the implementation opened as a result of a call to the **SnmpCreateSession** function.

```
SNMPAPI_STATUS SnmpClose(
  HSNMP_SESSION session    // handle to the session to close
);
```

Parameters

session
[in] Handle to the WinSNMP session to close.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the errors on the next page.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_SESSION_INVALID	The *session* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

It is recommended that a WinSNMP application call the **SnmpClose** function once for each session that the application opened using the **SnmpCreateSession** function. If a WinSNMP application terminates unexpectedly, it must call **SnmpCleanup** before it terminates to enable the implementation to deallocate all resources. The implementation processes one **SnmpCleanup** call as if it were a series of **SnmpClose** calls, one call for each session opened as a result of a call to **SnmpCreateSession**.

When the implementation closes a session it discards all outstanding incoming and outgoing asynchronous requests and replies for the session. For additional information, see *WinSNMP Sessions*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpCleanup**, **SnmpCreateSession**

SnmpContextToStr

The WinSNMP **SnmpContextToStr** function returns a string that identifies an SNMP context, which is a set of managed object resources. The function returns the string in an **smiOCTETS** structure.

```
SNMPAPI_STATUS SnmpContextToStr(
  HSNMP_CONTEXT context,  // handle to the context
  smiLPOCTETS string      // pointer to a structure to
                          // receive the context string
);
```

Parameters

context

[in] Handle to the SNMP context of interest.

string

[out] Pointer to an **smiOCTETS** structure to receive the string that identifies the context of interest. The string can have a null-terminating byte.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_CONTEXT_INVALID	The *context* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The current setting of the entity and context translation mode determines the type of output string **SnmpContextToStr** returns. For additional information, see *Setting the Entity and Context Translation Mode*.

The WinSNMP application must provide the address of a valid **smiOCTETS** structure for the *string* parameter. If the **SnmpContextToStr** function completes successfully, the Microsoft WinSNMP implementation initializes the **len** and **ptr** members of the structure. The WinSNMP application must call the **SnmpFreeDescriptor** function to enable the implementation to free the resources for these members.

When the entity and context translation mode is SNMPAPI_TRANSLATED, and the entry exists in the implementation's database, the implementation returns the associated user-friendly name of the context. If an entry does not exist for the context name, **SnmpContextToStr** returns the SNMP community string.

When the entity and context translation mode is SNMPAPI_UNTRANSLATED_V1 or SNMPAPI_UNTRANSLATED_V2, the implementation also returns the SNMP community string.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpFreeDescriptor**, **smiOCTETS**

SnmpCountVbl

A WinSNMP application calls the WinSNMP **SnmpCountVbl** function to enumerate the variable binding entries in the specified variable bindings list.

```
SNMPAPI_STATUS SnmpCountVbl(
  HSNMP_VBL vbl   // handle to the variable bindings list
);
```

Parameters

vbl

[in] Handle to the variable bindings list to enumerate.

Return Values

If the function succeeds, the return value is the count of variable binding entries in the variable bindings list.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_NOOP	The variable bindings list does not contain any variable binding entries at this time.
SNMPAPI_VBL_INVALID	The *vbl* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The **SnmpCountVbl** function returns an unsigned integer value that is the maximum value the WinSNMP application can specify for the *index* parameter in the **SnmpGetVb**, **SnmpSetVb**, and **SnmpDeleteVb** functions.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpGetVb**, **SnmpSetVb**, **SnmpDeleteVb**

SnmpCreatePdu

The WinSNMP **SnmpCreatePdu** function creates and initializes an SNMP Protocol Data Unit (PDU).

```
HSNMP_PDU SnmpCreatePdu(
  HSNMP_SESSION session, // handle to the WinSNMP session
  smiINT PDU_type,       // PDU type
  smiINT32 request_id,   // PDU request identifier
  smiINT error_status,   // PDU error status, unless type
                         // is SNMP_PDU_GETBULK
  smiINT error_index,    // PDU error index, unless type
                         // is SNMP_PDU_GETBULK
  HSNMP_VBL varbindlist  // handle to the variable bindings
                         // list
);
```

Parameters

session
[in] Handle to the WinSNMP session.

PDU_type
[in] Specifies a PDU type that identifies the SNMP operation. This parameter can be NULL, or it can be one of the following values. If this parameter is NULL, the Microsoft WinSNMP implementation supplies the default PDU type **SNMP_PDU_GETNEXT**. The only type of trap PDU you can create with a call to the **SnmpCreatePdu** function is an SNMPv2C trap PDU.

Value	Meaning
SNMP_PDU_GET	Search and retrieve a value from a specified SNMP variable.
SNMP_PDU_GETNEXT	Search and retrieve the value of an SNMP variable without knowing the exact name of the variable.
SNMP_PDU_RESPONSE	Reply to an **SNMP_PDU_GET** or an **SNMP_PDU_GETNEXT** request.
SNMP_PDU_SET	Store a value in a specified SNMP variable.
SNMP_PDU_GETBULK	Search and retrieve multiple values with a single request.
SNMP_PDU_TRAP	Alerts the management system to an event under SNMPv2C.

request_id
[in] Specifies a unique numeric value that the WinSNMP application supplies to identify the PDU. If this parameter is NULL, the implementation assigns a value.

error_status
[in] If the *PDU_type* parameter is equal to **SNMP_PDU_GETBULK**, this parameter specifies a value for the **non_repeaters** field of the PDU. For other PDU types, this parameter specifies a value for the **error_status** field of the PDU. This parameter can be NULL.

error_index
[in] If the *PDU_type* parameter is equal to **SNMP_PDU_GETBULK**, this parameter specifies a value for the **max_repetitions** field of the PDU. For other PDU types, this parameter specifies a value for the **error_index** field of the PDU. This parameter can be NULL.

varbindlist
[in] Handle to a structure that represents an SNMP variable bindings list. This parameter can be NULL.

Return Values

If the function succeeds, the return value is the handle to a new SNMP PDU.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.

Error code	Description
SNMPAPI_SESSION_INVALID	The session handle is invalid.
SNMPAPI_PDU_INVALID	The PDU type is invalid.
SNMPAPI_VBL_INVALID	The variable bindings list is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

A WinSNMP application must create a PDU before it calls the **SnmpSendMsg** or the **SnmpEncodeMsg** functions.

All of the parameters of the **SnmpCreatePdu** function are required. However, all parameters, except the *session* parameter, can be NULL. In this instance, the new PDU has the following default values.

Field	Contents
PDU_type	**SNMP_PDU_GETNEXT**
request_id	The implementation generates a numeric value.
error_status	SNMP_ERROR_NOERROR
error_index	0
varbindlist	NULL

The application must call the **SnmpFreePdu** function to release the resources that the **SnmpCreatePdu** function allocates for the new PDU.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpFreePdu**, **SnmpSendMsg**, **SnmpEncodeMsg**

SnmpCreateSession

The **SnmpCreateSession** function requests the Microsoft WinSNMP implementation to open a session for the WinSNMP application. The application can specify how the implementation should inform the WinSNMP session of available SNMP messages and asynchronous events. The application can specify a window notification message or an **SNMPAPI_CALLBACK** function to notify the session.

The **SnmpCreateSession** function is an element of the WinSNMP API, version 2.0. When developing new WinSNMP applications, it is recommended that you call **SnmpCreateSession** to open a WinSNMP session instead of calling the **SnmpOpen** function.

```
HSNMP_SESSION SnmpCreateSession(
  HWND hWnd,              // handle to the notification window
  UINT wMsg,              // window notification message number
  SNMPAPI_CALLBACK fCallback,   // notification callback
                                // function
  LPVOID lpClientData   // pointer to callback function data
);
```

Parameters

hWnd

[in] Handle to a window of the WinSNMP application to notify when an asynchronous request completes, or when trap notification occurs. This parameter is required for window notification messages for the session.

wMsg

[in] Specifies an unsigned integer that identifies the notification message to send to the WinSNMP application window. This parameter is required for window notification messages for the session.

fCallback

[in] Specifies the address of an application-defined, session-specific **SNMPAPI_CALLBACK** function. The implementation will call this function to inform the WinSNMP session when notifications are available.

This parameter is required to specify callback notifications for the session. This parameter must be NULL to specify window notification messages for the session. For additional information, see the following Remarks section.

lpClientData

[in] Pointer to application-defined data to pass to the callback function specified by the *fCallback* parameter. This parameter is optional and can be NULL. If the *fCallback* parameter is NULL, the implementation ignores this parameter.

Return Values

If the function succeeds, the return value is a handle that identifies the WinSNMP session that the implementation opens for the calling application.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the errors on the following page.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_HWND_INVALID	The *fCallback* parameter is NULL, but the *hWnd* parameter is not a valid window handle.
SNMPAPI_MSG_INVALID	The *fCallback* parameter is NULL, but the *wMsg* parameter is not a valid message value.
SNMPAPI_MODE_INVALID	The *fCallback* parameter is NULL and the *hWnd* and *wMsg* parameters are valid individually. However, the values are mutually incompatible on the Windows platform.
SNMPAPI_OPERATION_INVALID	The combination of parameter values does not specify callback notifications or window notification messages.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The **SnmpCreateSession** function returns a unique handle to each open WinSNMP session within the calling WinSNMP application. The application must use the session handle that **SnmpCreateSession** returns in other WinSNMP function calls to facilitate allocation and deallocation of resources by the implementation.

It is recommended that a WinSNMP application call the **SnmpClose** function once for each session that the implementation opens as a result of a call to the **SnmpCreateSession** function. If a WinSNMP application terminates unexpectedly, it must call **SnmpCleanup** before it terminates to enable the implementation to deallocate all resources.

Callback Notifications

If the *fCallback* parameter is not NULL on a successful call to the **SnmpCreateSession** function, the implementation alerts the session to notifications using the callback function specified by the *fCallback* parameter.

Following is an example of a call to the **SnmpCreateSession** function, requesting that the implementation signal the session about messages and events using callback notifications.

```
hSession = SnmpCreateSession (0, 0, myFunc, <NULL|myData>);
```

Window Notification Messages

The **SnmpCreateSession** function passes to the implementation the handle to an application window and a notification message identifier. When the application window receives the notification message specified by the *wMsg* parameter, the WinSNMP application must retrieve the incoming protocol data unit (PDU). The application does this by calling the **SnmpRecvMsg** function with the session handle returned by **SnmpCreateSession**.

One WinSNMP application can open multiple WinSNMP sessions. If an application opens multiple sessions using the same window handle, it is recommended that the WinSNMP application specify a unique *wMsg* parameter for each session.

Following is an example of a call to the **SnmpCreateSession** function, requesting that the implementation signal the session about messages and events using window notification messages.

```
hSession = SnmpCreateSession (myWnd, myMsg, NULL, NULL);
```

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SNMPAPI_CALLBACK**, **SnmpClose**, **SnmpCleanup**, **SnmpRecvMsg**

SnmpCreateVbl

The WinSNMP **SnmpCreateVbl** function creates a new variable bindings list for the calling WinSNMP application. If the *name* and *value* parameters are not NULL, **SnmpCreateVbl** uses their values to create the first variable binding entry for the new variable bindings list. The **SnmpCreateVbl** function returns a handle to the new variable bindings list and allocates any necessary memory for it.

```
HSNMP_VBL SnmpCreateVbl(
  HSNMP_SESSION session,  // handle to the WinSNMP session
  smiLPCOID name,         // pointer to the variable name
  smiLPCVALUE value       // pointer to the value to
                          //  associate with the variable
);
```

Parameters

session

[in] Handle to the WinSNMP session.

name

[in] Pointer to an **smiOID** structure that contains the variable name for the first variable binding entry. This parameter can be NULL. For additional information, see the following Remarks section.

value

[in] Pointer to an **smiVALUE** structure that contains a value to associate with the variable in the first variable binding entry. This parameter can be NULL. For additional information, see the following Remarks section.

Return Values

If the function succeeds, the return value is a handle to a new variable bindings list.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_SESSION_INVALID	The session handle is invalid.
SNMPAPI_OID_INVALID	The *name* parameter references an invalid **smiOID** structure.
SNMPAPI_SYNTAX_INVALID	The **syntax** member of the structure pointed to by the *value* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The **SnmpCreateVbl** function uses the values of the *name* and *value* parameters to create and initialize the first variable binding entry of a new variable bindings list. If the *name* parameter is NULL, the Microsoft WinSNMP implementation ignores the *value* parameter and creates an empty variable bindings list.

If the *name* parameter is not NULL, but the *value* parameter is NULL, the implementation creates and initializes the first variable binding entry in the variable bindings list. It initializes the **syntax** member of the structure pointed to by the *value* parameter with the value **SNMP_SYNTAX_NULL**.

The WinSNMP application must release the resources associated with each variable bindings list. It should do this by matching each call to the **SnmpCreateVbl** and **SnmpDuplicateVbl** functions with a corresponding call to the **SnmpFreeVbl** function.

To avoid memory leaks, a WinSNMP application must call **SnmpFreeVbl** before it reuses the handle to a variable bindings list in a subsequent call to **SnmpCreateVbl** or **SnmpDuplicateVbl**. For additional information, see *WinSNMP Data Management Concepts*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpDuplicateVbl**, **SnmpFreeVbl**, **smiOID**, **smiVALUE**

SnmpDecodeMsg

The WinSNMP **SnmpDecodeMsg** function decodes an encoded SNMP message into its components. This function performs the opposite action of the WinSNMP **SnmpEncodeMsg** function.

```
SNMPAPI_STATUS SnmpDecodeMsg(
  HSNMP_SESSION session,     // handle to the WinSNMP session
  LPHSNMP_ENTITY srcEntity,  // handle to the source entity
  LPHSNMP_ENTITY dstEntity,  // handle to the target entity
  LPHSNMP_CONTEXT context,   // handle to the context
  LPHSNMP_PDU pdu,           // handle to the PDU
  smiLPCOCTETS msgBufDesc    // pointer to the message buffer
);
```

Parameters

session
[in] Handle to the WinSNMP session. This parameter is required. For additional information, see the following Remarks section.

srcEntity
[out] Pointer to a variable that receives a handle to the source management entity. For more information, see the following Remarks section.

dstEntity
[out] Pointer to a variable that receives a handle to the target management entity. For more information, see the following Remarks section.

context
[out] Pointer to a variable that receives a handle to the context (a set of managed object resources) that the target management entity controls.

pdu
[out] Pointer to a variable that receives a handle to the SNMP protocol data unit (PDU).

msgBufDesc
[in] Pointer to an **smiOCTETS** structure that contains the SNMP message to decode into its components. The **len** member of the structure specifies the maximum number of bytes to process; the **ptr** member points to the encoded SNMP message.

Return Values

If the function succeeds, the return value is the number of decoded bytes. This value can be equal to, or less than, the **len** member of the **smiOCTETS** structure pointed to by the *msgBufDesc* parameter.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_SESSION_INVALID	The *session* parameter is invalid.
SNMPAPI_ENTITY_INVALID	One or both of the entity parameters is invalid.
SNMPAPI_CONTEXT_INVALID	The *context* parameter is invalid.
SNMPAPI_PDU_INVALID	The *pdu* parameter is invalid.
SNMPAPI_OUTPUT_TRUNCATED	The output buffer length is insufficient. No output parameters were created.
SNMPAPI_MESSAGE_INVALID	The SNMP message format in the buffer indicated by the *msgBufDesc* parameter is invalid. No output parameters were created.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The Microsoft WinSNMP implementation returns a value of zero in the *srcEntity* and the *dstEntity* parameters when an application submits an SNMPv1 or an SNMPv2C message to the **SnmpDecodeMsg** function. This is because the message format does not include the address information necessary to create WinSNMP entity resources.

The Microsoft WinSNMP implementation allocates resources to the WinSNMP application as a result of a successful call to the **SnmpDecodeMsg** function. It is recommended that the WinSNMP application free individual resources with the WinSNMP function that corresponds to the resource. For additional information, see *Freeing WinSNMP Descriptors* and *WinSNMP Data Management Concepts*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpEncodeMsg**, **SnmpFreeEntity**, **SnmpFreeContext**, **SnmpFreePdu**, **SnmpSendMsg**, **smiOCTETS**

SnmpDeleteVb

The WinSNMP **SnmpDeleteVb** function removes a variable binding entry from a variable bindings list.

```
SNMPAPI_STATUS SnmpDeleteVb(
  HSNMP_VBL vbl,    // handle to the variable bindings list
  smiUINT32 index   // position of the variable binding entry
                    //   in the list
);
```

Parameters

vbl

[in] Handle to the variable bindings list to update.

index

[in] Specifies an unsigned long integer variable that identifies the variable binding entry to remove. This variable contains the position of the variable binding entry, within the variable bindings list.

Valid values for this parameter are in the range from 1 to *n*, where 1 indicates the first variable binding entry in the variable bindings list, and *n* is the total number of entries in the variable bindings list. For additional information, see the following Remarks section.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_INDEX_INVALID	The *index* parameter is invalid.
SNMPAPI_VBL_INVALID	The *vbl* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

A WinSNMP application can use the **SnmpDeleteVb** function to delete invalid variable binding entries. When an **SNMP_PDU_RESPONSE** protocol data unit (PDU) includes an error that indicates an invalid variable binding entry, the application can call **SnmpDeleteVb** to delete the entry. Then the application can resubmit the request PDU with a call to the **SnmpSendMsg** function, without the invalid variable binding entry in the variable bindings list. Request PDUs include the **SNMP_PDU_GET**, **SNMP_PDU_GETNEXT**, and **SNMP_PDU_GETBULK** PDU data types.

After the **SnmpDeleteVb** function deletes a variable binding entry, the index value of all entries after the deleted entry will decrement by one. A call to the **SnmpCountVbl** function returns the new total number of entries in the variable bindings list. The new total is one less than the count returned by a call to **SnmpCountVbl** before the current call to **SnmpDeleteVb**.

If a WinSNMP application calls the **SnmpDeleteVb** function and deletes the last variable binding entry in a variable bindings list, the result is an empty variable bindings list. The variable bindings list still has a valid handle and the WinSNMP application must release the handle with a call to the **SnmpFreeVbl** function.

The following are valid values to use for the *index* parameter:

- The return value from a call to the **SnmpCountVbl** function
- The error index field of an **SNMP_PDU_RESPONSE** PDU returned by a call to the **SnmpRecvMsg** function

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpCountVbl**, **SnmpRecvMsg**, **SnmpFreeVbl**

SnmpDuplicatePdu

The WinSNMP **SnmpDuplicatePdu** function duplicates the SNMP protocol data unit (PDU) that the *PDU* parameter identifies, allocating any necessary memory for the duplicate PDU.

```
HSNMP_PDU SnmpDuplicatePdu(
  HSNMP_SESSION session,    // handle to the WinSNMP session
  HSNMP_PDU PDU             // handle to the PDU to duplicate
);
```

Parameters

session

[in] Handle to the WinSNMP session.

PDU

[in] Handle to the PDU to duplicate. The **SnmpDuplicatePdu** function provides a unique handle to each PDU within the calling application.

Return Values

If the function succeeds, the return value is a handle that identifies the new duplicate PDU.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_SESSION_INVALID	The session handle is invalid.
SNMPAPI_PDU_INVALID	The PDU handle is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

To release resources allocated by the **SnmpDuplicatePdu** function for a new PDU, a WinSNMP application must call the **SnmpFreePdu** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpFreePdu**, **SnmpGetPduData**

SnmpDuplicateVbl

The WinSNMP **SnmpDuplicateVbl** function copies a variable bindings list for the specified WinSNMP session. This function returns a handle to the copied variable bindings list and allocates any necessary memory for it.

```
HSNMP_VBL SnmpDuplicateVbl(
  HSNMP_SESSION session,  // handle to the WinSNMP session
  HSNMP_VBL vbl           // handle to the variable bindings
                          //   list to duplicate
);
```

Parameters

session

[in] Handle to the WinSNMP session.

vbl

[in] Handle to the variable bindings list to copy. The source variable bindings list can be empty.

Return Values

If the function succeeds, the return value is a handle to a new variable bindings list.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_SESSION_INVALID	The session handle is invalid.
SNMPAPI_VBL_INVALID	The *vbl* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The **SnmpDuplicateVbl** function creates a new variable bindings list for the specified WinSNMP session. This function initializes the new list with a copy of the data in the source variable bindings list.

The handle the **SnmpDuplicateVbl** function returns is unique among the variable bindings list handles that are active within the WinSNMP application.

The WinSNMP application must release the resources associated with each variable bindings list. It should do this by matching each call to the **SnmpCreateVbl** and **SnmpDuplicateVbl** functions with a corresponding call to the **SnmpFreeVbl** function. To avoid memory leaks, a WinSNMP application must call **SnmpFreeVbl** before it reuses the handle to a variable bindings list in a subsequent call to **SnmpCreateVbl** or **SnmpDuplicateVbl**. For additional information, see *WinSNMP Data Management Concepts*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpFreeVbl**, **SnmpCreateVbl**

SnmpEncodeMsg

The Microsoft WinSNMP implementation uses the parameters passed in the WinSNMP **SnmpEncodeMsg** function to encode an SNMP message. The implementation returns the encoded SNMP message to the WinSNMP application in the buffer specified by the *msgBufDesc* parameter.

```
SNMPAPI_STATUS SnmpEncodeMsg(
  HSNMP_SESSION session,    // handle to the WinSNMP session
  HSNMP_ENTITY srcEntity,   // handle to the source entity
  HSNMP_ENTITY dstEntity,   // handle to the target entity
  HSNMP_CONTEXT context,    // handle to the context
  HSNMP_PDU pdu,            // handle to the PDU
  smiLPOCTETS msgBufDesc    // pointer to the message buffer
);
```

Parameters

session
[in] Handle to the WinSNMP session.

srcEntity
[in] Handle to the management entity that initiates the request to encode the SNMP message.

dstEntity
[in] Handle to the target management entity.

context
[in] Handle to the context (a set of managed object resources) that the target management entity controls.

pdu
[in] Handle to the PDU that contains the SNMP operation request.

msgBufDesc
[out] Pointer to an **smiOCTETS** structure that receives the encoded SNMP message.

Return Values

If the function succeeds, the return value is the length, in bytes, of the encoded SNMP message. This number is also the value of the **len** member of the **smiOCTETS** structure pointed to by the *msgBufDesc* parameter.

If the function fails, the return value is SNMPAPI_FAILURE. For additional information, see the following Remarks section. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_SESSION_INVALID	The *session* parameter is invalid.
SNMPAPI_ENTITY_INVALID	One or both of the entity parameters is invalid.
SNMPAPI_CONTEXT_INVALID	The *context* parameter is invalid.
SNMPAPI_PDU_INVALID	The *pdu* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The first five parameters passed to the **SnmpEncodeMsg** function are the same parameters that are passed to the **SnmpSendMsg** function.

The WinSNMP application must call the **SnmpFreeDescriptor** function to free resources allocated for the **ptr** member of the **smiOCTETS** structure. This is the structure pointed to by the *msgBufDesc* parameter. For additional information, see WinSNMP Data Management Concepts.

On input, the **SnmpEncodeMsg** function ignores the members of the structure pointed to by the *msgBufDesc* parameter. The implementation overwrites the members of the structure if the function completes successfully.

The implementation verifies the format of the first five input parameters. If one of the parameters is invalid, **SnmpEncodeMsg** returns SNMPAPI_FAILURE, and **SnmpGetLastError** returns an extended error code.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpFreeDescriptor**, **SnmpDecodeMsg**, **SnmpSendMsg**, **smiOCTETS**

SnmpEntityToStr

The WinSNMP **SnmpEntityToStr** function returns a string that identifies an SNMP management entity.

```
SNMPAPI_STATUS SnmpEntityToStr(
  HSNMP_ENTITY entity, // handle to the entity
  smiUINT32 size,      // buffer size, in bytes, for
                       // output string
  LPSTR string         // pointer to the buffer to receive
                       // the output string
);
```

Parameters

entity
[in] Handle to the SNMP management entity of interest.

size
[in] Specifies the size, in bytes, of the buffer pointed to by the *string* parameter. The WinSNMP application must allocate a buffer that is large enough to contain the output string.

string
[out] Pointer to a buffer to receive the null-terminated string that identifies the SNMP management entity of interest.

Return Values

If the function succeeds, the return value is the number of bytes, including a terminating null byte, that **SnmpEntityToStr** returns in the *string* buffer. This value can be less than or equal to the value of the *size* parameter, but it cannot be greater.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_ENTITY_INVALID	The *entity* parameter is invalid.
SNMPAPI_OUTPUT_TRUNCATED	The output buffer length is insufficient.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The current setting of the entity and context translation mode determines the type of output string **SnmpEntityToStr** returns. For additional information, see *Support for IPX Address Strings in WinSNMP* and *Setting the Entity and Context Translation Mode*.

When the entity and context translation mode is SNMPAPI_TRANSLATED, and an entry exists in the implementation's database, the implementation returns the associated user-friendly name of the management entity. If an entry does not exist for the management entity, **SnmpEntityToStr** returns the literal SNMP transport address of the management entity.

When the entity and context translation mode is SNMPAPI_UNTRANSLATED_V1 or SNMPAPI_UNTRANSLATED_V2, the Microsoft WinSNMP implementation also returns the literal SNMP transport address of the management entity.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpStrToEntity**

SnmpFreeContext

The WinSNMP **SnmpFreeContext** function releases resources associated with an SNMP context, which is a set of managed object resources.

```
SNMPAPI_STATUS SnmpFreeContext(
  HSNMP_CONTEXT context  // handle to the context to release
);
```

Parameters

context
[in] Handle to the SNMP context that will have its resources released.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_CONTEXT_INVALID	The *context* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

When the WinSNMP application calls the **SnmpClose** function or the **SnmpCleanup** function, the Microsoft WinSNMP implementation frees all resources it allocated for the WinSNMP session. However, it is recommended that the WinSNMP application free individual resources with the WinSNMP function that corresponds to the resource. For example, applications should call the **SnmpFreeContext** function to release resources allocated by a call to the **SnmpStrToContext** function. This reduces the implementation's work load, and should enhance the service of the implementation to all applications.

For additional information, see *WinSNMP Data Management Concepts*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpClose**, **SnmpCleanup**, **SnmpStrToContext**

SnmpFreeDescriptor

A WinSNMP application uses the **SnmpFreeDescriptor** function to inform the Microsoft WinSNMP implementation that it no longer requires access to a descriptor object. This WinSNMP function signals the implementation to free the memory it allocated for the descriptor object.

```
SNMPAPI_STATUS SnmpFreeDescriptor(
  smiUINT32 syntax,        // data type of target descriptor
                           // object
  smiLPOPAQUE descriptor   // pointer to the target
                           // descriptor object
);
```

Parameters

syntax
[in] Specifies the syntax data type of the target descriptor object.

descriptor
[in] Pointer to an **smiOPAQUE** structure that contains the target descriptor object to release.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_SYNTAX_INVALID	The *syntax* parameter is invalid.
SNMPAPI_OPERATION_INVALID	The *descriptor* parameter is invalid. For additional information, see the following Remarks section.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The implementation allocates and deallocates memory for output descriptor objects with variable lengths. This memory allocation and deallocation are restricted to the implementation, except for the interface that the **SnmpFreeDescriptor** function provides. For additional information, see *Freeing WinSNMP Descriptors*.

The implementation returns the SNMPAPI_OPERATION_INVALID error code if the *descriptor* parameter specifies a memory allocation that the implementation released in a prior call to **SnmpFreeDescriptor**. The function returns the same error code if the *descriptor* parameter specifies a memory allocation that the implementation did not make for the calling WinSNMP application.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpStrToOid**, **SnmpOidCopy**, **SnmpEncodeMsg**

SnmpFreeEntity

The WinSNMP **SnmpFreeEntity** function releases resources associated with an SNMP management entity.

```
SNMPAPI_STATUS SnmpFreeEntity(
  HSNMP_ENTITY entity   // handle to the entity to release
);
```

Parameters

entity
[in] Handle to the SNMP management entity that will have its resources released.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_ENTITY_INVALID	The *entity* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

When the WinSNMP application calls the **SnmpClose** function or the **SnmpCleanup** function, the Microsoft WinSNMP implementation frees all resources it allocated for the WinSNMP session. However, it is recommended that the WinSNMP application free individual resources by using the WinSNMP function that corresponds to the resource. For example, applications should call the **SnmpFreeEntity** function to release resources allocated by a call to the **SnmpStrToEntity** function. This reduces the implementation's work load, and should enhance the implementation's service to all applications.

For additional information, see *WinSNMP Data Management Concepts*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpClose**, **SnmpCleanup**, **SnmpStrToEntity**

SnmpFreePdu

The WinSNMP **SnmpFreePdu** function releases resources associated with an SNMP protocol data unit (PDU) created by the **SnmpCreatePdu** or the **SnmpDuplicatePdu** function.

```
SNMPAPI_STATUS SnmpFreePdu(
  HSNMP_PDU PDU     // handle to the PDU to free
);
```

Parameters

PDU
[in] Handle to the SNMP PDU to free.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the errors on the next page.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_PDU_INVALID	The PDU handle is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

If the application calls the **SnmpClose** or the **SnmpCleanup** function, the Microsoft WinSNMP implementation frees all resources it allocates for the WinSNMP session. However, it is recommended that the application free individual resources with the WinSNMP function that corresponds to the resource. This reduces the implementation's work load, and should enhance the implementation's service to all applications. The application should use the **SnmpFreeVbl** function to deallocate variable bindings list resources. For additional information, see *WinSNMP Data Management Concepts*.

Under WinSNMP, a variable binding entry exists only within a variable bindings list, even if the variable bindings list contains just one entry.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpFreeVbl**, **SnmpClose**, **SnmpCleanup**

SnmpFreeVbl

The WinSNMP **SnmpFreeVbl** function releases resources associated with a variable bindings list. These are resources allocated previously by a call to the **SnmpCreateVbl** function or the **SnmpDuplicateVbl** function in a WinSNMP application.

```
SNMPAPI_STATUS SnmpFreeVbl(
  HSNMP_VBL vbl      // handle to the variable bindings list
);
```

Parameters

vbl
[in] Handle to the variable bindings list to release.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_VBL_INVALID	The *vbl* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The WinSNMP application must release the resources associated with each variable bindings list. It should do this by matching each call to the **SnmpCreateVbl** and **SnmpDuplicateVbl** functions with a corresponding call to the **SnmpFreeVbl** function. To avoid memory leaks, a WinSNMP application must call **SnmpFreeVbl** before it reuses the handle to a variable bindings list in a subsequent call to **SnmpCreateVbl** or **SnmpDuplicateVbl**.

If the application calls the **SnmpClose** or the **SnmpCleanup** function, the Microsoft WinSNMP implementation frees all resources it allocates for the WinSNMP session. However, even if the application does not reuse a variable bindings list handle, it is recommended that the application free individual variable bindings resources with the **SnmpFreeVbl** function. This reduces the implementation's work load, and should enhance its service to all applications. For additional information, see *WinSNMP Data Management Concepts*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpDuplicateVbl**, **SnmpCreateVbl**, **SnmpClose**, **SnmpCleanup**

SnmpGetLastError

The WinSNMP **SnmpGetLastError** function returns the calling application's last-error code value. The value indicates the reason why the last function call executed by the WinSNMP application failed.

```
SNMPAPI_STATUS SnmpGetLastError(
  HSNMP_SESSION session    // handle to the WinSNMP session
);
```

Parameters

session

[in] Handle to the WinSNMP session. This parameter can also be NULL.

In certain cases, when a function call fails you can pass a NULL *session* value to the **SnmpGetLastError** function to retrieve the last-error code value. This is true for function calls that do not involve a *session* parameter, and cases in which the *session* parameter value is invalid. These cases are noted in the Return Values section on the function's reference page.

A single-thread application can pass a NULL *session* value to **SnmpGetLastError** to retrieve last-error information for the entire application.

For more information, see the following *Remarks* and *Return Values* sections.

Return Values

If the *session* parameter is a valid WinSNMP session handle, the **SnmpGetLastError** function returns the last WinSNMP error that occurred for the indicated session.

If the *session* parameter is NULL—for example, if the **SnmpStartup** function fails, **SnmpGetLastError** returns the last WinSNMP error that occurred.

Remarks

A WinSNMP application must call **SnmpGetLastError** immediately after a function fails, to retrieve the last-error code. If another function fails, it overwrites the last-error code set by the most recently failed function. For more information, see *WinSNMP Error Codes.*

Although the *session* parameter accommodates both multithread and single-thread Windows operating environments, the potential still exists for the last-error code from one thread to overwrite the last-error code from another thread.

Note that **SnmpGetLastError** must be able to return the last-error code to a WinSNMP application under the following conditions:

- After the **SnmpStartup** function fails
- Before the **SnmpCreateSession** function creates any WinSNMP sessions for the instance of the application

- After the **SnmpClose** function closes all WinSNMP sessions for the instance of the application
- After the **SnmpCleanup** function disconnects the WinSNMP application from the Microsoft WinSNMP implementation

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpStartup**, **SnmpCreateSession**, **SnmpClose**, **SnmpCleanup**

SnmpGetPduData

The WinSNMP **SnmpGetPduData** function returns selected data fields from a specified SNMP protocol data unit (PDU).

```
SNMPAPI_STATUS SnmpGetPduData(
  HSNMP_PDU PDU,            // handle to the PDU
  smiLPINT PDU_type,        // PDU_type field of the PDU
  smiLPINT32 request_id,    // request_id field of the PDU
  smiLPINT error_status,    // error_status field of the PDU
  smiLPINT error_index,     // error_index field of the PDU
  LPHSNMP_VBL varbindlist   // handle to the variable
                            // bindings list
);
```

Parameters

PDU

[in] Handle to the SNMP PDU.

PDU_type

[out] Pointer to a variable that receives the **PDU_type** field of the specified PDU. This parameter can be NULL, or one of the following values.

Value	Meaning
SNMP_PDU_GET	Search and retrieve a value from a specified SNMP variable.
SNMP_PDU_GETNEXT	Search and retrieve the value of an SNMP variable without knowing the exact name of the variable.
SNMP_PDU_RESPONSE	Reply to an **SNMP_PDU_GET** or an **SNMP_PDU_GETNEXT** request.
SNMP_PDU_SET	Store a value in a specified SNMP variable.
SNMP_PDU_GETBULK	Search and retrieve multiple values with a single request.
SNMP_PDU_TRAP	Alerts the management system to an extraordinary event under SNMPv2C.

request_id

[out] Pointer to a variable that receives the **request_id** field of the specified PDU. This parameter can be NULL.

error_status

[out] Pointer to a variable that receives the **error_status** field of the specified PDU. If the *PDU_type* parameter is equal to **SNMP_PDU_GETBULK**, this parameter receives the value of the **non_repeaters** field of the PDU.

This parameter can be NULL, or one of the following values. The first six errors are common to the SNMP version 1 (SNMPv1) and SNMP version 2C frameworks (SNMPv2C). The remaining errors are available under SNMPv2C only.

Error Code	Meaning
SNMP_ERROR_NOERROR	The agent reports that no errors occurred during transmission.
SNMP_ERROR_TOOBIG	The agent could not place the results of the requested SNMP operation into a single SNMP message.
SNMP_ERROR_NOSUCHNAME	The requested SNMP operation identified an unknown variable.
SNMP_ERROR_BADVALUE	The requested SNMP operation tried to change a variable but it specified either a syntax or value error.
SNMP_ERROR_READONLY	The requested SNMP operation tried to change a variable that was not allowed to change, according to the community profile of the variable.

Error Code	Meaning
SNMP_ERROR_GENERR	An error other than one of those listed here occurred during the requested SNMP operation.
SNMP_ERROR_NOACCESS	The specified SNMP variable is not accessible.
SNMP_ERROR_WRONGTYPE	The value specifies a type that is inconsistent with the type required for the variable.
SNMP_ERROR_WRONGLENGTH	The value specifies a length that is inconsistent with the length required for the variable.
SNMP_ERROR_WRONGENCODING	The value contains an Abstract Syntax Notation One (ASN.1) encoding that is inconsistent with the ASN.1 tag of the field.
SNMP_ERROR_WRONGVALUE	The value cannot be assigned to the variable.
SNMP_ERROR_NOCREATION	The variable does not exist, and the agent cannot create it.
SNMP_ERROR_INCONSISTENTVALUE	The value is inconsistent with values of other managed objects.
SNMP_ERROR_ RESOURCEUNAVAILABLE	Assigning the value to the variable requires allocation of resources that are currently unavailable.
SNMP_ERROR_COMMITFAILED	No validation errors occurred, but no variables were updated.
SNMP_ERROR_UNDOFAILED	No validation errors occurred. Some variables were updated because it was not possible to undo their assignment.
SNMP_ERROR_ AUTHORIZATIONERROR	An authorization error occurred.
SNMP_ERROR_NOTWRITABLE	The variable exists but the agent cannot modify it.
SNMP_ERROR_INCONSISTENTNAME	The variable does not exist; the agent cannot create it because the named object instance is inconsistent with the values of other managed objects.

error_index

[out] Pointer to a variable that receives the **error_index** field of the specified PDU.

If the *PDU_type* parameter is equal to **SNMP_PDU_GETBULK**, this parameter receives the value of the **max_repetitions** field of the specified PDU. This parameter can be NULL.

varbindlist

[out] Pointer to a variable that receives a handle to the variable bindings list field of the specified PDU. This parameter can be NULL. For additional information, see the following *Remarks* section.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_NOOP	All output parameters are NULL. The SNMP operation was not performed.
SNMPAPI_PDU_INVALID	The PDU type is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

All parameters of the **SnmpGetPduData** function are required. However, all parameters, except the *PDU* parameter, can be NULL. In parameters the application passes as NULL, the **SnmpGetPduData** function does not return a value.

The **SnmpGetPduData** function always returns a handle to a new variable bindings list object if the *varbindlist* parameter is not NULL. Additionally, if the *PDU* parameter specifies a new PDU, the function also attaches a handle to the new PDU.

When an application calls **SnmpGetPduData** with a *varbindlist* parameter that is not NULL, but the *PDU* parameter specifies an existing PDU, the function returns a handle to a new duplicate variable bindings list. The function call does not disturb the handle attached to the existing PDU. An existing PDU is one that an application creates with a call to the **SnmpCreatePdu** function, or one that the application receives and then reads using a call to **SnmpGetPduData**.

When an application creates a PDU with **SnmpCreatePdu**, or after the application reads a PDU using **SnmpGetPduData**, the Microsoft WinSNMP implementation expects that the application "knows" the values of the PDU fields. If an application reads a PDU a second time with **SnmpGetPduData**, the call results in a copy of the variable bindings list of the specified PDU. This type of call to **SnmpGetPduData** also duplicates the handle to the PDU.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpDuplicateVbl**, **SnmpCreatePdu**

SnmpGetRetransmitMode

The WinSNMP **SnmpGetRetransmitMode** function returns the current setting of the retransmission mode to a WinSNMP application. The Microsoft WinSNMP implementation uses the retransmission mode to govern transmission time-outs and retransmission attempts on calls to the **SnmpSendMsg** function.

```
SNMPAPI_STATUS SnmpGetRetransmitMode(
  smiLPUINT32 nRetransmitMode  //current retransmission mode
);
```

Parameters

nRetransmitMode

[out] Pointer to an unsigned long integer variable to receive the current retransmission mode in effect for the implementation. This parameter can be one of the following values.

Value	Meaning
SNMPAPI_ON	The implementation is executing the WinSNMP application's retransmission policy.
SNMPAPI_OFF	The implementation is not executing the WinSNMP application's retransmission policy.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. If **SnmpGetRetransmitMode** fails, the value of the *nRetransmitMode* parameter has no meaning for the application. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

Typically a WinSNMP application, rather than an agent application, calls the **SnmpGetRetransmitMode** function. For additional information, see *About Retransmission* and *Managing the Retransmission Policy*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpStartup**, **SnmpSetRetransmitMode**

SnmpGetRetry

The WinSNMP **SnmpGetRetry** function returns the retry count value, in units, for the retransmission of SNMP message requests. The retry count applies to calls that a WinSNMP application makes to the **SnmpSendMsg** function for a specified management entity.

```
SNMPAPI_STATUS SnmpGetRetry(
  HSNMP_ENTITY hEntity,      // destination management entity
  smiLPUINT32 nPolicyRetry,  // retry count value from
                             // the database
  smiLPUINT32 nActualRetry   // last actual or estimated
                             // response retry count
);
```

Parameters

hEntity

[in] Handle to the destination management entity of interest.

nPolicyRetry

[out] Pointer to an unsigned long integer variable to receive the retry count value for the specified management entity. This is a value that the Microsoft WinSNMP implementation stores in a database. If you do not need the information returned in this parameter, *nPolicyRetry* must be a NULL pointer.

nActualRetry

[out] Pointer to an unsigned long integer variable to receive the last actual or estimated response retry count for the destination entity, as reported by the implementation. If you do not need the information returned in this parameter, *nActualRetry* must be a NULL pointer.

Note This feature has not yet been implemented. If this parameter is a valid pointer, the function returns 0. For additional information, see the following Remarks section.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_ENTITY_INVALID	The *hEntity* parameter is invalid.
SNMPAPI_NOOP	The *nPolicyRetry* and *nActualRetry* parameters are both NULL. The operation was not performed.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

Typically a WinSNMP application, rather than an agent application, calls the **SnmpGetRetry** function.

A WinSNMP application can modify the retry count value with a call to the **SnmpSetRetry** function.

The WinSNMP application can monitor the value of the *nActualRetry* parameter and compare it to the value of the *nPolicyRetry* parameter to optimize transmission performance. For additional information, see *About Retransmission* and *Managing the Retransmission Policy*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpSetRetry**, **SnmpSetRetransmitMode**, **SnmpGetRetransmitMode**

SnmpGetTimeout

The WinSNMP **SnmpGetTimeout** function returns the time-out value, in hundredths of a second, for the transmission of SNMP message requests. The time-out value applies to calls that a WinSNMP application makes to the **SnmpSendMsg** function for a specified management entity.

```
SNMPAPI_STATUS SnmpGetTimeout(
  HSNMP_ENTITY hEntity,          // destination management
                                 // entity
  smiLPTIMETICKS nPolicyTimeout, // time-out value from the
                                 // database
  smiLPTIMETICKS nActualTimeout  // last actual or estimated
                                 // response time
);
```

Parameters

hEntity

[in] Handle to the destination management entity of interest.

nPolicyTimeout

[out] Pointer to an integer variable to receive the time-out value, in hundredths of a second, for the specified management entity. This is a value that the Microsoft WinSNMP implementation stores in a database. If you do not need the information returned in this parameter, *nPolicyTimeout* must be a NULL pointer.

nActualTimeout

[out] Pointer to an integer variable to receive the last actual or estimated response interval for the destination entity, as reported by the implementation. If you do not need the information returned in this parameter, *nActualTimeout* must be a NULL pointer.

Note This feature has not yet been implemented. If this parameter is a valid pointer, the function returns 0. For additional information, see the following *Remarks* section.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_ENTITY_INVALID	The *hEntity* parameter is invalid.
SNMPAPI_NOOP	The *nPolicyTimeout* and *nActualTimeout* parameters are both NULL. The operation was not performed.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

Typically a WinSNMP application, rather than an agent application, calls the **SnmpGetTimeout** function.

The time-out period is the interval between the application's call to the **SnmpSendMsg** function and its call to the **SnmpRecvMsg** function.

A WinSNMP application can modify the time-out value with a call to the **SnmpSetTimeout** function.

The WinSNMP application can monitor the value of the *nActualTimeout* parameter and compare it to the value of the *nPolicyTimeout* parameter to optimize transmission performance. For additional information, see *About Retransmission* and *Managing the Retransmission Policy*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpSetTimeout**, **SnmpSetRetransmitMode**, **SnmpGetRetransmitMode**

SnmpGetTranslateMode

The WinSNMP **SnmpGetTranslateMode** function returns the current setting of the entity and context translation mode to a WinSNMP application. The entity and context translation mode affects the interpretation and return of WinSNMP input and output string parameters.

```
SNMPAPI_STATUS SnmpGetTranslateMode(
  smiLPUINT32 nTranslateMode  // current entity/context
                              // translation mode
);
```

Parameters

nTranslateMode

[out] Pointer to an unsigned long integer variable to receive the entity and context translation mode in effect for the Microsoft WinSNMP implementation. This parameter can be one of the following values.

Value	Meaning
SNMPAPI_TRANSLATED	The implementation uses its database to translate user-friendly names for SNMP entities and managed objects. The implementation translates them into their SNMPv1 or SNMPv2C components.
SNMPAPI_UNTRANSLATED_V1	The implementation interprets SNMP entity parameters as SNMP transport addresses, and context parameters as SNMP community strings. For SNMPv2 destination entities, the implementation creates outgoing SNMP messages that contain a value of zero in the version field.
SNMPAPI_UNTRANSLATED_V2	The implementation interprets SNMP entity parameters as SNMP transport addresses, and context parameters as SNMP community strings. For SNMPv2 destination entities, the implementation creates outgoing SNMP messages that contain a value of 1 in the version field.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. If **SnmpGetTranslateMode** fails, the value of the *nTranslateMode* parameter has no meaning for the application. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The entity and context translation mode affects calls to the **SnmpStrToEntity**, **SnmpStrToContext**, **SnmpContextToStr** and **SnmpEntityToStr** functions. For additional information, see *Setting the Entity and Context Translation Mode.*

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpStrToContext**, **SnmpStrToEntity**, **SnmpContextToStr**, **SnmpEntityToStr**, **SnmpStartup**, **SnmpSetTranslateMode**

SnmpGetVb

A WinSNMP application calls the **SnmpGetVb** function to retrieve information from a variable bindings list. This WinSNMP function retrieves a variable name and its associated value from the variable binding entry specified by the *index* parameter.

```
SNMPAPI_STATUS SnmpGetVb(
  HSNMP_VBL vbl,     // handle to the variable bindings list
  smiUINT32 index,   // position of the variable binding
                     // entry in the list
  smiLPOID name,     // pointer to the structure to receive
                     // the variable name
  smiLPVALUE value   // pointer to the structure to receive
                     // the associated value
);
```

Parameters

vbl

[in] Handle to the variable bindings list to retrieve.

index

[in] Specifies an unsigned long integer variable that identifies the variable binding entry to retrieve. This variable contains the position of the variable binding entry, within the variable bindings list.

Valid values for this parameter are in the range from 1 to *n*, where 1 indicates the first variable binding entry in the variable bindings list, and *n* is the total number of entries in the list. For additional information, see the following Remarks section.

name

[out] Pointer to an **smiOID** structure to receive the variable name of the variable binding entry.

value

[out] Pointer to an **smiVALUE** structure to receive the value associated with the variable identified by the *name* parameter.

If the function succeeds, the **syntax** member of the structure pointed to by the *value* parameter can be one of the following syntax data types. For additional information, see RFC 1902, "Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)."

Syntax data type	Meaning
SNMP_SYNTAX_INT	Indicates a 32-bit signed integer variable.
SNMP_SYNTAX_OCTETS	Indicates an octet string variable that is binary or textual data.
SNMP_SYNTAX_NULL	Indicates a NULL value.
SNMP_SYNTAX_OID	Indicates an object identifier variable that is an assigned name with a maximum of 128 subidentifiers.
SNMP_SYNTAX_INT32	Indicates a 32-bit signed integer variable.
SNMP_SYNTAX_IPADDR	Indicates a 32-bit Internet address variable.
SNMP_SYNTAX_CNTR32	Indicates a counter variable that increases until it reaches a maximum value of (2^32) –1.
SNMP_SYNTAX_GAUGE32	Indicates a gauge variable that is a non-negative integer that can increase or decrease, but never exceed a maximum value.
SNMP_SYNTAX_TIMETICKS	Indicates a counter variable that measures the time in hundredths of a second, until it reaches a maximum value of (2^32) –1. It is a non-negative integer that is relative to a specific timer event.

Syntax data type	Meaning
SNMP_SYNTAX_OPAQUE	This type provides backward compatibility, and should not be used for new object types. It supports the capability to pass arbitrary Abstract Syntax Notation One (ASN.1) syntax.
SNMP_SYNTAX_CNTR64	Indicates a counter variable that increases until it reaches a maximum value of (2^64) –1.
SNMP_SYNTAX_UINT32	Indicates a 32-bit unsigned integer variable.
SNMP_SYNTAX_ NOSUCHOBJECT	Indicates that the agent does not support the object type that corresponds to the variable.
SNMP_SYNTAX_ NOSUCHINSTANCE	Indicates that the object instance does not exist for the operation.
SNMP_SYNTAX_ ENDOFMIBVIEW	Indicates the WinSNMP application is attempting to reference an object identifier that is beyond the end of the MIB tree that the agent supports.

The last three syntax types describe exception conditions under the SNMP version 2C(SNMPv2C) framework.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_INDEX_INVALID	The *index* parameter is invalid.
SNMPAPI_VBL_INVALID	The *vbl* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The **SnmpGetVb** function returns the variable name of the variable binding entry in the structure pointed to by the *name* parameter. It returns the variable's associated value in the structure pointed to by the *value* parameter.

On input, the **SnmpGetVb** function ignores the members of the **smiOID** and **smiVALUE** structures pointed to by the *name* and *value* parameters respectively. The Microsoft WinSNMP implementation overwrites the members if the function completes successfully.

Valid values for a WinSNMP application to use for the *index* parameter are as follows:

- The return value from a call to the **SnmpCountVbl** function
- The error index field of an **SNMP_PDU_RESPONSE** protocol data unit (PDU) returned by a call to the **SnmpRecvMsg** function

The WinSNMP application must call the **SnmpFreeDescriptor** function to free resources allocated for the **ptr** member of the **smiOID** structure pointed to by the *name* parameter. The application must also call the **SnmpFreeDescriptor** function to release resources allocated for the **smiVALUE** structure pointed to by the *value* parameter under the conditions following. If the **value** member is an **smiOCTETS** or an **smiOID** structure, the application must call **SnmpFreeDescriptor** to free the resources allocated for these structures. For additional information, see *WinSNMP Data Management Concepts*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpCountVbl**, **SnmpRecvMsg**, **SnmpFreeDescriptor**, **smiOID**, **smiVALUE**, **smiOCTETS**

SnmpGetVendorInfo

A WinSNMP application calls the **SnmpGetVendorInfo** function to retrieve information about the Microsoft WinSNMP implementation. The function returns the information in an **smiVENDORINFO** structure. The **SnmpGetVendorInfo** function is an element of the WinSNMP API, version 2.0.

```
SNMPAPI_STATUS SnmpGetVendorInfo(
  smiLPVENDORINFO vendorInfo  // pointer to structure to
                              // receive information
);
```

Parameters

vendorInfo

[out] Pointer to an **smiVENDORINFO** structure to receive information. The information includes a way to contact Microsoft and the enterprise number assigned to Microsoft by the Internet Assigned Numbers Authority (IANA).

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_NOOP	The *vendorInfo* parameter is NULL.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **smiVENDORINFO**

SnmpListen

The WinSNMP **SnmpListen** function registers a WinSNMP application as an SNMP agent. An agent application calls this function to inform the Microsoft WinSNMP implementation that an entity will be acting in the role of an SNMP agent. An application also calls this function to inform the implementation when an entity will no longer be acting in this role. The **SnmpListen** function is an element of the WinSNMP API, version 2.0.

```
SNMPAPI_STATUS SnmpListen(
  HSNMP_ENTITY hEntity,    // handle to the entity
                           // that will receive notifications
  SNMPAPI_STATUS lStatus   // flag to indicate agent role
);
```

Parameters

hEntity

[in] Handle to the WinSNMP entity to notify when the Microsoft WinSNMP implementation receives an incoming SNMP request message (PDU) This parameter identifies the agent application. For more information, see the following Remarks and Return Values sections.

When you call the **SnmpCreateSession** function, you can specify whether the implementation should use a window notification message or an **SNMPAPI_CALLBACK** function to notify the application when an SNMP message or asynchronous event is available.

lStatus

[in] Specifies an unsigned long integer variable that indicates whether the WinSNMP entity identified by the *hEntity* parameter is acting in an SNMP agent role, or if it is no longer acting in this role. This parameter can be one of the following values.

Value	Meaning
SNMPAPI_ON	The specified WinSNMP entity is functioning in an agent role.
SNMPAPI_OFF	The specified WinSNMP entity is not functioning in an agent role.

Passing a value of SNMPAPI_OFF releases both the resources allocated to the entity and the port assigned it. For more information, see the following Remarks section.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_ENTITY_INVALID	The *hEntity* parameter is invalid. This parameter must be a handle returned by a previous call to the **SnmpStrToEntity** function.
SNMPAPI_MODE_INVALID	The *lStatus* parameter is invalid.
SNMPAPI_NOOP	The entity specified by the *hEntity* parameter is already functioning in the role of an SNMP agent.

Error Code	Description
SNMPAPI_TL_RESOURCE_ERROR	There is a network transport layer error. A socket could not be created for the entity specified by the *hEntity* parameter.
SNMPAPI_TL_OTHER	An error occurred in the network transport layer while trying to bind a socket for the entity specified by the *hEntity* parameter.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

When you specify an entity, you explicitly specify the address family, interface address, and port for the entity. This is because WinSNMP assigns these attributes to each WinSNMP entity as a result of a call to the **SnmpStrToEntity** function. The implementation uses the address and port settings currently assigned to the entity specified by the *hEntity* parameter when it sends notifications to the agent application. For more information, see **SnmpSetPort**.

When you call the **SnmpClose** function for a WinSNMP session and the **SnmpCleanup** function for a WinSNMP application, you must release all ports associated with WinSNMP agent applications.

For more information about SNMP management applications and agent applications, see *Registering an SNMP Agent Application* and *About SNMP*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpRecvMsg**, **SnmpSendMsg**, **SnmpSetPort**, **SnmpStrToEntity**, **SnmpClose**, **SnmpCleanup**

SnmpOidCompare

The WinSNMP **SnmpOidCompare** function lexicographically compares two SNMP object identifiers, up to the length specified by the *maxlen* parameter.

```
SNMPAPI_STATUS SnmpOidCompare(
  smiLPCOID xOID,     // first object identifier to compare
  smiLPCOID yOID,     // second object identifier to compare
```

(continued)

(continued)

```
    smiUINT32 maxlen,    // maximum length to compare
    smiLPINT result      // result of comparison
);
```

Parameters

xOID
[in] Pointer to the first **smiOID** object identifier to compare. The length of the object identifier can be zero.

yOID
[in] Pointer to the second **smiOID** object identifier to compare. The length of the object identifier can be zero.

maxlen
[in] If not equal to zero, specifies the number of subidentifiers to compare. This parameter must be less than MAXOBJIDSIZE: 128 subidentifiers, the maximum number of components in an object identifier. For additional information, see the following *Remarks* section.

result
[out] Pointer to an integer variable to receive the result of the comparison. The variable can receive one of the following results.

Result	Meaning
Greater than 0	*xOID* is greater than *yOID*
Equal to 0	*xOID* equals *yOID*
Less than 0	*xOID* is less than *yOID*

For additional comparison conditions, see the following Remarks section.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_OID_INVALID	One or both of the *xOID* and *yOID* parameters is invalid.
SNMPAPI_SIZE_INVALID	The *maxlen* parameter is invalid. The parameter size is greater than MAXOBJIDSIZE.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

A WinSNMP application can call the **SnmpOidCompare** function to determine whether two object identifiers have common prefixes.

If the *maxlen* parameter is not equal to zero, and not greater than MAXOBJIDSIZE, the value of *maxlen* sets the upper limit for the number of subidentifiers to compare. The maximum number of subidentifiers that the **SnmpOidCompare** function compares defaults to whichever is the smallest number—the *maxlen* parameter, or the **len** member of one of the **smiOID** structures pointed to by the *xOID* and *yOID* parameters.

If the *maxlen* parameter is equal to zero, the maximum number of subidentifiers that the **SnmpOidCompare** function compares defaults to the number that is the smaller of the **len** members of the two **smiOID** structures.

The value of the *result* parameter will indicate that *xOID* equals *yOID* if the two **smiOID** structures are lexicographically equal and one of the following occurs:

- **SnmpOidCompare** compares a *maxlen* number of subidentifiers.
- **SnmpOidCompare** compares the maximum number of subidentifiers, and the **len** members of both **smiOID** structures are equal, but less than the *maxlen* parameter.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **smiOID**

SnmpOidCopy

The WinSNMP **SnmpOidCopy** function copies an SNMP object identifier, allocating any necessary memory for the copy.

```
SNMPAPI_STATUS SnmpOidCopy(
  smiLPCOID srcOID,      // source object identifier
  smiLPOID dstOID        // destination object identifier
);
```

Parameters

srcOID
[in] Pointer to an **smiOID** structure to copy.

dstOID

[out] Pointer to an **smiOID** structure to receive a copy of the object identifier specified by the *srcOID* parameter.

Return Values

If the function succeeds, the return value is the number of subidentifiers in the copied object identifier. This number is also the value of the **len** member of the **smiOID** structure pointed to by the *dstOID* parameter.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_OID_INVALID	The *srcOID* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

On input, the **SnmpOidCopy** function ignores the members of the **smiOID** structure pointed to by the *dstOID* parameter. The Microsoft WinSNMP implementation overwrites the **smiOID** members if the function completes successfully.

The WinSNMP application must call the **SnmpFreeDescriptor** function to enable the implementation to free resources allocated for the **ptr** member of the **smiOID** structure pointed to by the *dstOID* parameter. For additional information, see *WinSNMP Data Management Concepts* and *Freeing WinSNMP Descriptors*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpFreeDescriptor**, **smiOID**

SnmpOidToStr

The WinSNMP **SnmpOidToStr** function converts the internal binary representation of an SNMP object identifier to its dotted numeric string format, for example, to "1.2.3.4.5.6".

```
SNMPAPI_STATUS SnmpOidToStr(
  smiLPCOID srcOID,  // object identifier to convert
  smiUINT32 size,    // buffer size for string
  LPSTR string       // pointer to buffer for converted
                     // string object identifier
);
```

Parameters

srcOID
[in] Pointer to an **smiOID** structure with an object identifier to convert.

size
[in] Specifies the size, in bytes, of the buffer indicated by the *string* parameter.

string
[out] Pointer to a buffer to receive the converted string object identifier that specifies the SNMP management entity.

Return Values

If the function succeeds, the return value is the length, in bytes, of the string that the WinSNMP application writes to the *string* parameter. The return value includes a null-terminating byte. This value may be less than or equal to the value of the *size* parameter, but it may not be greater.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_SIZE_INVALID	The *size* parameter is invalid. This parameter cannot be equal to zero; it must indicate the size of the buffer pointed to by the *string* parameter.
SNMPAPI_OID_INVALID	The *srcOID* parameter is invalid. For additional information, see the following Remarks section.
SNMPAPI_OUTPUT_TRUNCATED	The output buffer length is insufficient.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

It is recommended that a WinSNMP application specify, with the *size* parameter, a string buffer of MAXOBJIDSTRSIZE length (1408 bytes). This ensures that the output buffer is large enough to hold the converted string. Because the converted string is usually less than MAXOBJIDSTRSIZE, the WinSNMP application can copy the converted string to a smaller buffer. The application can then reuse or free the memory that it allocated for the initial buffer. For additional information, see *WinSNMP Data Management Concepts*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpStrToOid**, **smiOID**

SnmpOpen

The **SnmpOpen** function requests the Microsoft WinSNMP implementation to open a session for the WinSNMP application. This WinSNMP function enables the implementation to allocate and initialize memory, resources, data structures, and communications mechanisms. The **SnmpOpen** function returns a handle to the new WinSNMP session.

Note When developing new WinSNMP applications, it is recommended that you call the **SnmpCreateSession** function to open a WinSNMP session instead of calling the **SnmpOpen** function.

```
HSNMP_SESSION SnmpOpen(
  HWND hWnd,      // handle to the notification window
  UINT wMsg       // window notification message number
);
```

Parameters

hWnd
: [in] Handle to a window of the WinSNMP application to notify when an asynchronous request completes, or when trap notification occurs.

wMsg
: [in] Specifies an unsigned integer that identifies the notification message to send to the WinSNMP application window.

Return Values

If the function succeeds, the return value is a handle that identifies the WinSNMP session that the implementation opens for the calling application.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_HWND_INVALID	The *hWnd* parameter is not a valid window handle.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The **SnmpOpen** function returns a unique handle to each open WinSNMP session within the calling WinSNMP application. The application must use the session handle that **SnmpOpen** returns in other WinSNMP function calls to facilitate allocation and deallocation of resources by the implementation. When the implementation allocates resources to an individual session, it performs an orderly release of resources in response to a call to **SnmpClose** for the session.

The **SnmpOpen** function passes to the implementation the handle to an application window and a notification message identifier. If the *wParam* component of the notification message specified by the *wMsg* parameter is equal to zero, the WinSNMP application must retrieve the incoming protocol data unit (PDU). The application does this by calling the **SnmpRecvMsg** function with the session handle returned by **SnmpOpen**. If the *wParam* parameter of the notification message is not equal to zero, it represents a WinSNMP error code. The error code applies to the PDU identified by the request identifier in the *lParam* parameter of the notification message.

One WinSNMP application can open multiple WinSNMP sessions. If an application opens multiple sessions using the same window handle, it is recommended that the WinSNMP application specify a unique *wMsg* parameter for each session.

It is recommended that a WinSNMP application call the **SnmpClose** function once for each session that the implementation opens as a result of a call to the **SnmpOpen** function. If a WinSNMP application terminates unexpectedly, it must call **SnmpCleanup** before it terminates to enable the implementation to deallocate all resources. The implementation treats one **SnmpCleanup** call as if it were a series of **SnmpClose** calls, one call for each session the implementation opens as a result of a call to **SnmpOpen**.

For information about opening a WinSNMP session and specifying the method used to inform the session of available SNMP messages and asynchronous events, see **SnmpCreateSession**. When you call **SnmpCreateSession** you can specify a window notification message or an **SNMPAPI_CALLBACK** function to notify the session.

For more information, see WinSNMP Sessions.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpClose, SnmpCleanup**, **SnmpRecvMsg**, **SNMPAPI_CALLBACK**, **SnmpCreateSession**

SnmpRecvMsg

The WinSNMP **SnmpRecvMsg** function retrieves the results of a completed asynchronous request submitted by a call to the **SnmpSendMsg** function, in the form of an SNMP message. The **SnmpRecvMsg** function also returns outstanding trap data and notifications registered for a WinSNMP session.

```
SNMPAPI_STATUS SnmpRecvMsg(
  HSNMP_SESSION session,     // handle to the WinSNMP session
  LPHSNMP_ENTITY srcEntity,  // handle to the source entity
  LPHSNMP_ENTITY dstEntity,  // handle to the target entity
  LPHSNMP_CONTEXT context,   // handle to the context
  LPHSNMP_PDU PDU            // handle to the PDU
);
```

Parameters

session
[in] Handle to the WinSNMP session.

srcEntity
[out] Pointer to a variable that receives a handle to the entity that sends the message. Note that the *srcEntity* parameter to the **SnmpRegister** function specifies a handle to the management entity that registers for trap notification.

dstEntity
[out] Pointer to a variable that receives a handle to the entity that receives the message. Note that the *dstEntity* parameter to the **SnmpRegister** function specifies a handle to the management entity that sends traps.

context

[out] Pointer to a variable that receives a handle to the context, which is a set of managed object resources. The entity specified by the *srcEntity* parameter issues the message from this context.

PDU

[out] Pointer to a variable that receives a handle to the Protocol Data Unit (PDU) component of the message.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS, and the output parameters contain the values indicated in the preceding parameter descriptions.

If the function fails, the return value is SNMPAPI_FAILURE. If the function fails with an extended error code that indicates a network transport layer error, that is, one that begins with SNMPAPI_TL_, the output parameters also contain the values indicated preceding to enable the WinSNMP application to recover gracefully.

To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function may return one of the following WinSNMP or network transport layer errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_SESSION_INVALID	The *session* parameter is invalid.
SNMPAPI_NOOP	The specified session has no messages in its queue at this time.
SNMPAPI_TL_NOT_INITIALIZED	The network transport layer was not initialized.
SNMPAPI_TL_NOT_SUPPORTED	The network transport layer does not support the SNMP protocol.
SNMPAPI_TL_NOT_AVAILABLE	The network subsystem failed.
SNMPAPI_TL_RESOURCE_ERROR	A resource error occurred in the network transport layer.
SNMPAPI_TL_UNDELIVERABLE	The entity specified by the *dstEntity* parameter is unavailable.
SNMPAPI_TL_SRC_INVALID	The entity specified by the *srcEntity* parameter was not initialized.
SNMPAPI_TL_INVALID_PARAM	A network transport layer function call received an invalid input parameter.
SNMPAPI_TL_PDU_TOO_BIG	The PDU is too large for the network transport layer to send or receive.

(continued)

(continued)

Error Code	Description
SNMPAPI_TL_OTHER	An undefined network transport layer error occurred.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

For additional information, see *Network Transport Errors*.

Remarks

The **SnmpCreateSession** function passes an application window handle and notification message identifier to the Microsoft WinSNMP implementation. When the application window receives the notification message specified by the *wMsg* parameter, the WinSNMP application must call the **SnmpRecvMsg** function with the session handle returned by **SnmpCreateSession** to retrieve an incoming protocol data unit (PDU). For additional information, see About SNMP Messages.

The **SnmpRecvMsg** function instantiates four objects and allocates their resources: two entity handles, a context handle, and a PDU handle. The handle to the variable bindings list component of the returned PDU is not instantiated until the WinSNMP application calls the **SnmpGetPduData** function. When it no longer needs the resources **SnmpRecvMsg** returns, the WinSNMP application must free the individual resources using the WinSNMP function that corresponds to the resource. For additional information, see ***SnmpFreePdu***, ***SnmpFreeEntity***, and ***SnmpFreeContext***.

When the implementation receives traps from an entity operating under the SNMP version 1 framework (SNMPv1), it translates the traps to the SNMP version 2C (SNMPv2C) format. Therefore, when **SnmpRecvMsg** delivers a trap it is always in the SNMPv2C format. For additional information, see *Translating Traps from SNMPv1 to SNMPv2C* and *WinSNMP Programming Tasks*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpFreePdu**, **SnmpFreeEntity**, **SnmpFreeContext**, **SnmpSendMsg**, **SnmpRegister**, **SnmpGetPduData**

SnmpRegister

The WinSNMP **SnmpRegister** function registers or unregisters a WinSNMP application for trap and notification reception. The application can register and receive traps and notifications, or unregister and disable traps and notifications.

A WinSNMP application can register or unregister for one type of trap or notification, or for all traps and notifications, depending on the value of the *notification* parameter.

```
SNMPAPI_STATUS SnmpRegister(
  HSNMP_SESSION session,    // handle to the WinSNMP session
  HSNMP_ENTITY srcEntity,   // handle to the entity that is
                            //   the source of the request
  HSNMP_ENTITY dstEntity,   // handle to the entity that
                            //   receives the request
  HSNMP_CONTEXT context,    // handle to the context
  smiLPCOID notification,   // trap-matching sequence
  smiUINT32 state           // flag for trap reception
);
```

Parameters

session

[in] Handle to the WinSNMP session that is registering or unregistering for traps and notifications.

srcEntity

[in] Handle to the management entity that is the source of the registration request. This entity, acting in an SNMP manager role, will receive the trap or notification.

If this parameter is NULL, the Microsoft WinSNMP implementation registers or unregisters all sources of trap and notification requests.

Note that the *srcEntity* parameter to the **SnmpRecvMsg** function has a different role. In that function, *srcEntity* receives a handle to the entity that sent the trap.

dstEntity

[in] Handle to the management entity that is the recipient of the registration request. This entity, acting in an SNMP agent role, will send the trap or notification.

If this parameter is NULL, the implementation registers or unregisters the WinSNMP application for traps and notifications from all management entities.

Note that the *dstEntity* parameter to the **SnmpRecvMsg** function receives a handle to the management entity that registers for trap notification.

context

[in] Handle to the context, which is a set of managed object resources.

If this parameter is NULL, the implementation registers or unregisters the WinSNMP application for traps and notifications for every context.

notification

[in] Pointer to an **smiOID** structure that contains the pattern-matching sequence for one type of trap or notification. The implementation uses this sequence to identify the type of trap or notification for which the WinSNMP application is registering or unregistering. For additional information, see the following *Remarks* section.

If this parameter is NULL, the implementation registers or unregisters the WinSNMP application for all traps and notifications from the management entity or entities specified by the *dstEntity* parameter.

state

[in] Specifies an unsigned long integer variable that indicates whether the WinSNMP application is registering to receive traps and notifications, or if it is unregistering. This parameter should be equal to one of the following values, but if it contains a different value, the implementation registers the application.

Value	Meaning
SNMPAPI_OFF	Disable traps and notifications.
SNMPAPI_ON	Register to receive traps and notifications.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function may return one of the following WinSNMP or network transport layer errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_SESSION_INVALID	The *session* parameter is invalid.
SNMPAPI_ENTITY_INVALID	One or both of the entity parameters is invalid.
SNMPAPI_CONTEXT_INVALID	The *context* parameter is invalid.
SNMPAPI_OID_INVALID	The *notification* parameter is invalid.
SNMPAPI_TL_NOT_INITIALIZED	The network transport layer was not initialized.
SNMPAPI_TL_IN_USE	The trap port is not available.
SNMPAPI_TL_NOT_AVAILABLE	The network subsystem failed.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

For additional information, see *Network Transport Errors*.

Remarks

Typically a WinSNMP manager application, rather than an agent application, calls the **SnmpRegister** function.

If a WinSNMP application passes NULL in a call to the **SnmpRegister** function in the *srcEntity*, *dstEntity*, *context*, or *notification* parameters, the implementation does not use that parameter to filter traps and notifications from reaching the WinSNMP application. If an application passes NULL in all of the parameters mentioned previously, the implementation delivers all received notifications to the session.

If a WinSNMP application registers to receive a specific type of trap or notification, it must define an object identifier, that is, an **smiOID** structure, that corresponds to that type of trap. The *notification* parameter must point to this structure. RFC 1907, "Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)," defines trap and notification object identifiers. For additional information, see *Managing Traps and Notifications* and *Translating Traps from SNMPv1 to SNMPv2C*.

The implementation uses the value of the *notification* parameter as a pattern to match against received traps and notifications. For example, if the WinSNMP application passes *n* number of subidentifiers in the *notification* parameter, and the first *n* subidentifiers in a received trap match all the passed subidentifiers, then the trap object identifier is a match. If a received trap has fewer subidentifiers than *n*, the object identifier does not match. If there is a match, the implementation sends the trap or notification to the WinSNMP application.

If any or all of the *dstEntity, srcEntity,* or *context* parameters are NULL, the implementation may need to allocate resources on a subsequent call to the **SnmpRecvMsg** function, for that function's corresponding parameters. When the WinSNMP application no longer needs the resources **SnmpRecvMsg** returns, the application must free the individual resources with the function that corresponds to the resource. For additional information, see ***SnmpFreeEntity*** and ***SnmpFreeContext***.

For more information, see *Multiple Trap Registrations*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, *WinSNMP Functions*, **SnmpCreateSession**, **SnmpRecvMsg**, **SnmpFreeEntity**, **SnmpFreeContext**

SnmpSendMsg

A WinSNMP application calls the **SnmpSendMsg** function to request that the Microsoft WinSNMP implementation transmit an SNMP Protocol Data Unit (PDU), in the form of an SNMP message. The WinSNMP application specifies a source entity, a destination entity, and a context for the request.

If a WinSNMP application expects a PDU in response to a **SnmpSendMsg** request, it must retrieve the PDU. To do this, the application must call the **SnmpRecvMsg** function using the session handle returned by **SnmpCreateSession**.

```
SNMPAPI_STATUS SnmpSendMsg(
  HSNMP_SESSION session,     // handle to the WinSNMP session
  HSNMP_ENTITY srcEntity,    // handle to the source entity
  HSNMP_ENTITY dstEntity,    // handle to the target entity
  HSNMP_CONTEXT context,     // handle to the context
  HSNMP_PDU PDU              // handle to the PDU
);
```

Parameters

session
[in] Handle to the WinSNMP session.

srcEntity
[in] Handle to the management entity that initiates the request to send the SNMP message.

dstEntity
[in] Handle to the target entity that will respond to the SNMP request.

context
[in] Handle to the context, (a set of managed object resources), that the target management entity controls.

PDU
[in] Handle to the protocol data unit that contains the SNMP operation request.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function may return one of the following WinSNMP or network transport layer errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.

Error Code	Description
SNMPAPI_SESSION_INVALID	The *session* parameter is invalid.
SNMPAPI_ENTITY_INVALID	One or both of the entity parameters is invalid.
SNMPAPI_CONTEXT_INVALID	The *context* parameter is invalid.
SNMPAPI_PDU_INVALID	The PDU parameter is invalid.
SNMPAPI_OPERATION_INVALID	The operation specified in the **PDU_type** field of the PDU is inappropriate for the destination entity. For more information, see the following *Remarks* section.
SNMPAPI_TL_NOT_INITIALIZED	The network transport layer was not initialized.
SNMPAPI_TL_NOT_SUPPORTED	The network transport layer does not support the SNMP protocol.
SNMPAPI_TL_NOT_AVAILABLE	The network subsystem failed.
SNMPAPI_TL_RESOURCE_ERROR	A resource error occurred in the network transport layer.
SNMPAPI_TL_SRC_INVALID	The entity specified by the *srcEntity* parameter was not initialized.
SNMPAPI_TL_INVALID_PARAM	A network transport layer function call received an invalid input parameter.
SNMPAPI_TL_PDU_TOO_BIG	The PDU is too large for the network transport layer to send or receive.
SNMPAPI_TL_OTHER	An undefined network transport layer error occurred.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

For additional information, see *Network Transport Errors*.

Remarks

The **SnmpSendMsg** function executes asynchronously and therefore returns immediately.

The implementation notifies the WinSNMP application when the asynchronous request is completed. The implementation does this by sending a notification message to the window specified by the *wMsg* and *hWnd* parameters, respectively, in the initial call to **SnmpCreateSession** for the session. When the application window receives the notification message, the WinSNMP application must retrieve the incoming PDU. The application does this by calling the **SnmpRecvMsg** function with the session handle returned by **SnmpCreateSession**.

When a WinSNMP application calls the **SnmpSendMsg** function, the implementation determines which network transport protocol and SNMP version framework to use to complete the transmission request. The implementation determines this by matching its capabilities with properties associated with the requesting session and with the target management entity. This information is available from values in the implementation's database.

If a WinSNMP application requests functionality that is available under the SNMP version 2C framework (SNMPv2C), but the target entity uses the SNMP version 1 framework (SNMPv1), the implementation attempts to translate the request to SNMPv1. To do this, the implementation uses the procedures defined in RFC1908, "Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework." If translation is not possible, **SnmpSendMsg** fails with the extended error code SNMPAPI_OPERATION_INVALID. This situation occurs, for example, when an application attempts to send a PDU with the **SNMP_PDU_InformRequest** data type to an SNMPv1 destination entity.

For additional information, see *WinSNMP Programming Tasks* and *About SNMP Messages*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpCreateSession**, **SnmpRecvMsg**

SnmpSetPduData

The WinSNMP **SnmpSetPduData** function updates selected data fields in the specified SNMP Protocol Data Unit (PDU).

```
SNMPAPI_STATUS SnmpSetPduData(
  HSNMP_PDU PDU,                    // handle to the PDU
  const smiINT *PDU_type,           // pointer to the PDU type
  const smiINT32 *request_id,       // pointer to the PDU
                                    // request identifier
  const smiINT *non_repeaters,      // valid only for an
                                    // SNMP_PDU_GETBULK request
  const smiINT *max_repetitions,    // valid only for an
                                    // SNMP_PDU_GETBULK request
  const HSNMP_VBL *varbindlist      // handle to variable
                                    // bindings list
);
```

Parameters

PDU

[in] Handle to an SNMP PDU.

PDU_type

[in] Pointer to a variable with a value to update the **PDU_type** field of the specified PDU. This parameter can also be NULL.

request_id

[in] Pointer to a variable with a value to update the **request_id** field of the specified PDU. This parameter can also be NULL.

non_repeaters

[in] If the *PDU_type* parameter is equal to **SNMP_PDU_GETBULK**, this parameter points to a variable with a value to update the **non_repeaters** field of the specified PDU. The Microsoft WinSNMP implementation ignores this parameter for other PDU types. This parameter can also be NULL.

max_repetitions

[in] If the *PDU_type* parameter is equal to **SNMP_PDU_GETBULK**, this parameter points to a variable with a value to update the **max_repetitions** field of the specified PDU. The implementation ignores this parameter for other PDU types. This parameter can also be NULL.

varbindlist

[in] Pointer to a variable with a value that updates the handle to the variable bindings list field of the specified PDU. This parameter can also be NULL.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_PDU_INVALID	The PDU type is invalid.
SNMPAPI_VBL_INVALID	The variable bindings list is invalid.
SNMPAPI_NOOP	All input parameters are NULL. The SNMP operation was not performed.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

All parameters of the **SnmpSetPduData** function are required. However, all parameters, except the *PDU* parameter, can be NULL. If the WinSNMP application passes NULL in a parameter, **SnmpSetPduData** does not update the corresponding field in the PDU. Because **SnmpSetPduData** passes parameters as pointers to values, an application can still update a PDU field with NULL.

The value of one PDU field can be valid alone, but may be invalidated in combination with values for other fields. The implementation validates the PDU and the other message elements when the application calls the **SnmpSendMsg** or the **SnmpEncodeMsg** functions. The implementation rejects invalid PDUs.

The only type of trap PDU you can update with a call to the **SnmpSetPduData** function is an SNMPv2C trap PDU.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpSendMsg**, **SnmpEncodeMsg**

SnmpSetPort

A WinSNMP application calls the **SnmpSetPort** function to change the port assigned to a destination entity. The **SnmpSetPort** function is an element of the WinSNMP API, version 2.0.

```
SNMPAPI_STATUS SnmpSetPort(
  HSNMP_ENTITY hEntity,  // handle to the destination entity
  UINT nPort             // new port assignment
);
```

Parameters

hEntity

[in] Handle to a WinSNMP destination entity. This parameter can specify the handle to an entity acting in the role of an SNMP agent application as a result of a call to the **SnmpListen** function. For more information, see the following Remarks section.

nPort

[in] Specifies an unsigned integer that identifies the new port assignment for the destination entity. If you specify a local address that is busy, or if you specify a remote address that is unavailable, a call to the **SnmpSetPort** function fails.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_OPERATION_INVALID	The entity specified by the *hEntity* parameter is already functioning in an agent role as the result of a call to the **SnmpListen** function. For more information, see the following *Remarks* section.
SNMPAPI_ENTITY_INVALID	The *hEntity* parameter is invalid. This parameter must be a handle returned by a previous call to the **SnmpStrToEntity** function.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The Microsoft WinSNMP implementation assigns a port to each management entity as a result of a WinSNMP application's call to the **SnmpStrToEntity** function. If the SNMPAPI_UNTRANSLATED mode is in effect when the implementation creates an entity, the implementation typically assigns the standard SNMP request port for the respective protocol family to the entity; for example, UDP 161 or IPX 36879. If the SNMPAPI_TRANSLATED mode is in effect, the implementation assigns the port specified for the entity in the WinSNMP database. To retrieve the current entity and context translation mode in effect for the implementation, an application can call the **SnmpGetTranslateMode** function. For more information, see *Setting the Entity and Context Translation Mode* and *The WinSNMP Database*.

A call to the **SnmpSetPort** function fails if the entity specified by the *hEntity* parameter is currently functioning in an agent role. This is because the entity has already been assigned to a port other than the one specified by the *nPort* parameter. To ensure assignment of an agent application to a specific port, a WinSNMP application can perform the steps outlined in the following code sample.

```
hAgent = SnmpStrToEntity (hSession, <addrString>);
lStatus = SnmpSetPort (hAgent, <nPort>);
lStatus = SnmpListen (hAgent, SNMPAPI_ON);
```

where `<addrString>` contains the string representation of an IP address or an IPX address, and `<nPort>` contains the new port assignment for the agent application.

Note that an IPX address contains a network number that consists of eight hexadecimal digits (zero-filled if necessary); a separator (either ":", "." or "–"); and a node number that consists of 12 hexadecimal digits (zero-filled if necessary). For example, 00000001:00081A0D01C2. For more information, see *Support for IPX Address Strings in WinSNMP*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpListen**, **SnmpStrToEntity**, **SnmpSetPort**, **SnmpGetTranslateMode**

SnmpSetRetransmitMode

The WinSNMP **SnmpSetRetransmitMode** function enables a WinSNMP application to set the retransmission mode. The Microsoft WinSNMP implementation uses the new retransmission mode to govern transmission time-outs and retransmission attempts on subsequent calls to the **SnmpSendMsg** function.

```
SNMPAPI_STATUS SnmpSetRetransmitMode(
  smiUINT32 nRetransmitMode  // new retransmission mode
);
```

Parameters

nRetransmitMode

[in] Specifies a value for the new retransmission mode. This parameter must be one of the following values.

Value	Meaning
SNMPAPI_ON	The implementation executes the WinSNMP application's retransmission policy.
SNMPAPI_OFF	The implementation does not execute the WinSNMP application's retransmission policy.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_MODE_INVALID	The implementation does not support the requested retransmission mode.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

Typically a WinSNMP manager application, rather than an agent application, calls the **SnmpSetRetransmitMode** function.

If a WinSNMP application sets the retransmission mode to SNMPAPI_OFF, the implementation does not initiate retransmission attempts for new SNMP communications operations. The new setting affects all subsequent calls to the **SnmpSendMsg** function, until the WinSNMP application sets the retransmission mode back to SNMPAPI_ON.

Calling the **SnmpCancelMsg** function is equivalent to calling the **SnmpSetRetransmitMode** function, for a specific SNMP message, with the retransmission mode equal to SNMPAPI_OFF.

Note If the implementation returns the error SNMPAPI_MODE_INVALID to a call to **SnmpSetRetransmitMode**, the WinSNMP application must execute the retransmission policy.

For additional information, see *About Retransmission* and *Managing the Retransmission Policy*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpSendMsg**, **SnmpRegister**, **SnmpGetRetransmitMode**, **SnmpGetTimeout**, **SnmpGetRetry**, **SnmpCancelMsg**

SnmpSetRetry

The WinSNMP **SnmpSetRetry** function enables a WinSNMP application to change the retry count value for the retransmission of SNMP message requests. The retry count applies to calls that a WinSNMP application makes to the **SnmpSendMsg** function for a specified management entity. The Microsoft WinSNMP implementation stores the value in a database.

```
SNMPAPI_STATUS SnmpSetRetry(
  HSNMP_ENTITY hEntity,     // destination management entity
  smiUINT32 nPolicyRetry    // new retry count value for
                            // database
);
```

Parameters

hEntity

[in] Handle to the destination management entity of interest.

nPolicyRetry

[in] Specifies a new value for the retry count for the management entity. This value replaces the value currently stored in the implementation's database.

If this parameter is equal to zero, and the current retransmission mode is equal to SNMPAPI_ON, the implementation selects a value for the retry count. The implementation uses this value when it executes the WinSNMP application's retransmission policy.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_ENTITY_INVALID	The *hEntity* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

Typically a WinSNMP manager application, rather than an agent application, calls the **SnmpSetRetry** function.

For additional information, see *About Retransmission* and *Managing the Retransmission Policy*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpSetRetransmitMode**, **SnmpGetRetry**, **SnmpGetRetransmitMode**

SnmpSetTimeout

The WinSNMP **SnmpSetTimeout** function enables a WinSNMP application to change the time-out value for the transmission of SNMP message requests. The time-out value applies to calls that a WinSNMP application makes to the **SnmpSendMsg** function for a specified management entity. The Microsoft WinSNMP implementation stores the value in a database.

```
SNMPAPI_STATUS SnmpSetTimeout(
  HSNMP_ENTITY hEntity,         // destination management
                                // entity
  smiTIMETICKS nPolicyTimeout   // new time-out value for
                                // database
);
```

Parameters

hEntity
[in] Handle to the destination management entity of interest.

nPolicyTimeout
[in] Specifies a new time-out value, in hundredths of a second, for the management entity. This value replaces the value currently stored in the implementation's database.

If this parameter is equal to zero, and the current retransmission mode is equal to SNMPAPI_ON, the implementation selects a time-out value. The implementation uses this time-out value when it executes the WinSNMP application's retransmission policy.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the errors on the next page.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_ENTITY_INVALID	The *hEntity* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

Typically a WinSNMP manager application, rather than an agent application, calls the **SnmpSetTimeout** function.

For additional information, see *About Retransmission* and *Managing the Retransmission Policy*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpGetTimeout**, **SnmpSetRetransmitMode**, **SnmpGetRetransmitMode**

SnmpSetTranslateMode

The WinSNMP **SnmpSetTranslateMode** function enables a WinSNMP application to change the entity and context translation mode. The entity and context translation mode affects the interpretation and return of WinSNMP input and output string parameters.

```
SNMPAPI_STATUS SnmpSetTranslateMode(
  smiUINT32 nTranslateMode    // new entity/context
                              // translation mode
);
```

Parameters

nTranslateMode

[in] Specifies a value for the new entity and context translation mode. This parameter must be one of the following values.

Value	Meaning
SNMPAPI_TRANSLATED	The Microsoft WinSNMP implementation uses its database to translate user-friendly names for SNMP entities and managed objects. The implementation translates them into their SNMPv1 or SNMPv2C components.
SNMPAPI_UNTRANSLATED_V1	The implementation interprets SNMP entity parameters as SNMP transport addresses, and context parameters as SNMP community strings. For SNMPv2 destination entities, the implementation creates outgoing SNMP messages that contain a value of zero in the version field.
SNMPAPI_UNTRANSLATED_V2	The implementation interprets SNMP entity parameters as SNMP transport addresses, and context parameters as SNMP community strings. For SNMPv2 destination entities, the implementation creates outgoing SNMP messages that contain a value of 1 in the version field.

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_MODE_INVALID	The implementation does not support the requested translation mode.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The new entity and context translation mode affects subsequent calls to the **SnmpStrToEntity**, **SnmpStrToContext**, **SnmpContextToStr**, and **SnmpEntityToStr** functions. The WinSNMP application can change the entity and context translation mode again by making another call to **SnmpSetTranslateMode** with a different *nTranslateMode* value.

For additional information, see *Setting the Entity and Context Translation Mode*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpStrToContext**, **SnmpStrToEntity**, **SnmpContextToStr**, **SnmpEntityToStr**, **SnmpGetTranslateMode**

SnmpSetVb

The WinSNMP **SnmpSetVb** function changes variable binding entries in a variable bindings list. This function also appends new variable binding entries to an existing variable bindings list.

```
SNMPAPI_STATUS SnmpSetVb(
  HSNMP_VBL vbl,      // handle to the variable bindings list
  smiUINT32 index,    // position of the variable binding
                      // entry in the list
  smiLPCOID name,     // pointer to the variable name portion
                      // of the entry
  smiLPCVALUE value   // pointer to the variable value
                      // portion of the entry
);
```

Parameters

vbl
[in] Handle to the variable bindings list to update.

index
[in] Specifies an unsigned long integer variable that contains the position of the variable binding entry, within the variable bindings list, if this is an update operation. If this is an append operation, this parameter must be equal to zero. For more information, see the following *Remarks* section.

name
[in] Pointer to an **smiOID** structure that represents the name of the variable to append or change. For more information, see the following *Remarks* section.

value
[in] Pointer to an **smiVALUE** structure. The structure contains the value associated with the variable specified by the *name* parameter.

Return Values

If the function succeeds, the return value is the position of the updated or appended variable binding entry in the variable bindings list. For additional information, see the following Remarks section.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_VBL_INVALID	The *vbl* parameter is invalid.
SNMPAPI_INDEX_INVALID	The *index* parameter is invalid.
SNMPAPI_OID_INVALID	The *name* parameter is invalid.
SNMPAPI_SYNTAX_INVALID	The **syntax** member of the structure pointed to by the *value* parameter is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

Valid values for the *index* parameter range from zero to *n*. The value zero indicates an append operation. The value *n* is the total number of variable binding entries in the variable bindings list. A WinSNMP application should call the **SnmpCountVbl** function before it calls **SnmpSetVb** to obtain the total number of variable binding entries.

If the function successfully performs an update operation, the return value equals the value of the *index* parameter. If the function appends a variable binding entry, the return value is $n + 1$.

If the *name* parameter is not NULL, but the *value* parameter is NULL, the Microsoft WinSNMP implementation initializes the new variable binding entry with the **value** member set to NULL and with the **syntax** member set to **SNMP_SYNTAX_NULL**.

If the *index* parameter is not equal to zero, and the *name* parameter is NULL, the Microsoft WinSNMP implementation updates only the value of the variable pointed to by the *index* parameter.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpCountVbl**, **smiOID**, **smiVALUE**

SnmpStartup

The **SnmpStartup** function notifies the Microsoft WinSNMP implementation that the WinSNMP application requires the implementation's services. The WinSNMP **SnmpStartup** function enables the implementation to initialize and to return to the application the version of the Windows SNMP Application Programming Interface (WinSNMP API), the level of SNMP communications that the implementation supports, and the implementation's default translation and retransmission modes.

Note A WinSNMP application must call the **SnmpStartup** function successfully before it calls any other WinSNMP function.

```
SNMPAPI_STATUS SnmpStartup(
  smiLPUINT32 nMajorVersion,     // major version number of
                                 // the WinSNMP API
  smiLPUINT32 nMinorVersion,     // minor version number of
                                 // the WinSNMP API
  smiLPUINT32 nLevel,            // level of SNMP the
                                 // implementation supports
  smiLPUINT32 nTranslateMode,    // default entity/context
                                 // translation mode
  smiLPUINT32 nRetransmitMode    // default retransmission
                                 // mode
);
```

Parameters

nMajorVersion

[out] Pointer to an unsigned long integer variable to receive the major version number of the WinSNMP API that the implementation supports. For example, to indicate that the implementation supports WinSNMP version 2.0, the function returns a value of 2.

nMinorVersion

[out] Pointer to an unsigned long integer variable to receive the minor version number of the WinSNMP API that the implementation supports. For example, to indicate that the implementation supports WinSNMP version 2.0, the function returns a value of 0.

nLevel

[out] Pointer to an unsigned long integer variable to receive the highest level of SNMP communications the implementation supports. Upon successful return, this parameter contains a value of 2. For a description of level 2 support, see *Levels of SNMP Support*.

nTranslateMode

[out] Pointer to an unsigned long integer variable to receive the default translation mode in effect for the implementation. The translation mode applies to the implementation's interpretation of the *entity* parameter that the WinSNMP application passes to the **SnmpStrToEntity** function. The translation mode also applies to the *string* parameter that the WinSNMP application passes to the **SnmpStrToContext** function. This parameter can be one of the following values.

Value	Meaning
SNMPAPI_TRANSLATED	The implementation uses its database to translate user-friendly names for SNMP entities and managed objects. The implementation translates them into their SNMPv1 or SNMPv2C components.
SNMPAPI_UNTRANSLATED_V1	The implementation interprets SNMP entity parameters as SNMP transport addresses, and context parameters as SNMP community strings. For SNMPv2 destination entities, the implementation creates outgoing SNMP messages that contain a value of zero in the version field.
SNMPAPI_UNTRANSLATED_V2	The implementation interprets SNMP entity parameters as SNMP transport addresses, and context parameters as SNMP community strings. For SNMPv2 destination entities, the implementation creates outgoing SNMP messages that contain a value of 1 in the version field.

For additional information, see *Setting the Entity and Context Translation Mode.*

nRetransmitMode

[out] Pointer to an unsigned long integer variable to receive the default retransmission mode in effect for the implementation. This parameter can be one of the following values.

Value	Meaning
SNMPAPI_OFF	The implementation is not executing the retransmission policy of the WinSNMP application.
SNMPAPI_ON	The implementation is executing the retransmission policy of the WinSNMP application.

For additional information, see *About Retransmission.*

Return Values

If the function succeeds, the return value is SNMPAPI_SUCCESS, and the parameters contain appropriate values, as indicated in the preceding parameter descriptions.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors. For additional information, see the Remarks section that follows.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

A WinSNMP application must call the **SnmpStartup** function successfully at least once, before it calls any other WinSNMP function. If a WinSNMP application does call another WinSNMP function, before it successfully calls **SnmpStartup**, the implementation returns the error SNMPAPI_NOT_INITIALIZED.

The WinSNMP application can call **SnmpGetLastError** for error information, or retry **SnmpStartup** if a call to the **SnmpStartup** function fails. When **SnmpStartup** returns SNMPAPI_FAILURE, and a subsequent call to **SnmpGetLastError** returns SNMP_ALLOC_ERROR, the WinSNMP application can elect to wait. It can retry the call to **SnmpStartup** when the implementation has adequate free resources.

A WinSNMP application can call **SnmpStartup** multiple times. For example, it may need to retry the function call for the reasons discussed preceding. A WinSNMP application must also call **SnmpCleanup** at least once, as the last WinSNMP function call before it terminates. Multiple **SnmpStartup** calls do not require multiple **SnmpCleanup** calls.

For additional information, see *Levels of SNMP Support* and *About SNMP Versions*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpStrToEntity**, **SnmpStrToContext**, **SnmpCleanup**

SnmpStrToContext

The WinSNMP **SnmpStrToContext** function returns a handle to SNMP context information that is specific to the Microsoft WinSNMP implementation. The handle is a valid value that a WinSNMP application can use as the *context* parameter in a call to the **SnmpSendMsg** and **SnmpRegister** functions.

```
HSNMP_CONTEXT SnmpStrToContext(
  HSNMP_SESSION session,   // handle to the WinSNMP session
  smiLPCOCTETS string      // pointer to a string structure
);
```

Parameters

session

[in] Handle to the WinSNMP session.

string

[in] Pointer to an **smiOCTETS** structure that contains a string to interpret. The string can identify a collection of managed objects, or it can be a community string.

The current setting of the entity and context translation mode determines the way **SnmpStrToContext** interprets the input string structure as shown in the following table.

Entity/Context Translation Mode	Meaning
SNMPAPI_TRANSLATED	The implementation interprets the *string* parameter as a user-friendly name for a collection of managed objects. The implementation translates the name into its SNMPv1 or SNMPv2C components using the implementation's database.
SNMPAPI_UNTRANSLATED_V1	The implementation interprets the *string* parameter as a literal SNMP community string.
SNMPAPI_UNTRANSLATED_V2	The implementation interprets the *string* parameter as a literal SNMP community string.

Return Values

If the function succeeds, the return value is a handle to the context of interest.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the errors on the following page.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_SESSION_INVALID	The *session* parameter is invalid.
SNMPAPI_CONTEXT_INVALID	The *string* parameter format is invalid. For example, the **len** member or the **ptr** member of the **smiOCTETS** structure pointed to by the *string* parameter is NULL.
SNMPAPI_CONTEXT_UNKNOWN	The value referenced in the *string* parameter does not exist.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The current setting of the entity and context translation mode determines the manner in which **SnmpStrToContext** interprets the input string structure. For additional information, see *Setting the Entity and Context Translation Mode*.

The WinSNMP application must call the **SnmpFreeContext** function to release the context handle allocated by the **SnmpStrToContext** function. For additional information about releasing resources, see *WinSNMP Data Management Concepts*.

The WinSNMP application should free the memory associated with the **ptr** member of the **smiOCTETS** structure pointed to by the *string* parameter. This is because the application defines and allocates the resources. For example, if the application allocated resources with a call to the **GlobalAlloc** function, it should use the **GlobalFree** function to deallocate the resources. For additional information, see *Freeing WinSNMP Descriptors*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpSendMsg**, **SnmpRegister**, **SnmpFreeDescriptor**, **SnmpFreeContext**, **smiOCTETS**

SnmpStrToEntity

The WinSNMP **SnmpStrToEntity** function returns a handle to information about an SNMP management entity that is specific to the Microsoft WinSNMP implementation.

```
HSNMP_ENTITY SnmpStrToEntity(
  HSNMP_SESSION session,  // handle to the WinSNMP session
  LPCSTR string           // pointer to a string that
                          // identifies the entity
);
```

Parameters

session
[in] Handle to the WinSNMP session.

string
[in] Pointer to a null-terminated string that identifies the SNMP management entity of interest. The current setting of the entity and context translation mode determines the manner in which **SnmpStrToEntity** interprets the input string as follows.

Entity/Context Translation Mode	Meaning
SNMPAPI_TRANSLATED	The implementation interprets the *string* parameter as a user-friendly name. The implementation translates the name into its SNMPv1 or SNMPv2C components using the implementation's database.
SNMPAPI_UNTRANSLATED_V1	The implementation interprets the *string* parameter as a literal SNMP transport address.
SNMPAPI_UNTRANSLATED_V2	The implementation interprets the *string* parameter as a literal SNMP transport address.

Return Values

If the function succeeds, the return value is a handle to the SNMP management entity of interest.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError**. The **SnmpGetLastError** function can return one of the following errors.

Error Code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.

(continued)

(continued)

Error Code	Description
SNMPAPI_SESSION_INVALID	The *session* parameter is invalid.
SNMPAPI_ENTITY_UNKNOWN	The entity string is invalid.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The current setting of the entity and context translation mode determines the manner in which **SnmpStrToEntity** interprets the input string that identifies the management entity of interest. For additional information, see *Support for IPX Address Strings in WinSNMP* and *Setting the Entity and Context Translation Mode*.

The WinSNMP application should call the **SnmpFreeEntity** function to release the entity handle allocated by the **SnmpStrToEntity** function. For additional information, see *WinSNMP Data Management Concepts*.

The **SnmpStrToEntity** function returns a valid entity handle that a WinSNMP application can use as the *srcEntity* or the *dstEntity* parameter in multiple WinSNMP functions. These functions include the **SnmpSendMsg**, **SnmpRecvMsg**, **SnmpRegister**, **SnmpEncodeMsg**, and **SnmpDecodeMsg** functions.

The implementation returns the current entity and context translation mode in the *nTranslateMode* parameter of the **SnmpStartup** function. A WinSNMP application can change the setting of the entity and context translation mode with a call to the **SnmpSetTranslateMode** function.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

+ See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpFreeEntity**, **SnmpSetTranslateMode**, **SnmpStartup**, **SnmpSendMsg**, **SnmpRecvMsg**, **SnmpRegister**, **SnmpEncodeMsg**, **SnmpDecodeMsg**

SnmpStrToOid

The WinSNMP **SnmpStrToOid** function converts the dotted numeric string format of an SNMP object identifier, for example, “1.2.3.4.5.6”, to its internal binary representation.

```
SNMPAPI_STATUS SnmpStrToOid(
  LPCSTR string,        // string object identifier to convert
  smiLPOID dstOID       // object identifier internal
                        // representation
);
```

Parameters

string

[in] Pointer to a null-terminated object identifier string to convert.

dstOID

[out] Pointer to an **smiOID** structure that receives the converted value.

Return Values

If the function succeeds, the return value is the number of subidentifiers in the converted object identifier. This number is also the value of the **len** member of the **smiOID** structure pointed to by the *dstOID* parameter.

If the function fails, the return value is SNMPAPI_FAILURE. To get extended error information, call **SnmpGetLastError** specifying a NULL value in its *session* parameter. The **SnmpGetLastError** function can return one of the following errors.

Error code	Description
SNMPAPI_NOT_INITIALIZED	The **SnmpStartup** function did not complete successfully.
SNMPAPI_ALLOC_ERROR	An error occurred during memory allocation.
SNMPAPI_OID_INVALID	The *string* parameter is invalid. For additional information, see the following Remarks section.
SNMPAPI_OTHER_ERROR	An unknown or undefined error occurred.

Remarks

The WinSNMP application must call the **SnmpFreeDescriptor** function to free resources allocated for the **ptr** member of the **smiOID** structure pointed to by the *dstOID* parameter. On input, **SnmpFreeDescriptor** ignores the members of this **smiOID** structure. The Microsoft WinSNMP implementation overwrites the **smiOID** members if the function completes successfully.

The **SnmpStrToOid** function fails and returns the SNMPAPI_OID_INVALID error code if the *string* parameter meets one of the following conditions:

- Is not null-terminated.
- Is not the textual form of a valid object identifier.
- Is insufficient in length; all object identifiers must have two subidentifiers.
- Exceeds the MAXOBJIDSTRSIZE of 1408 bytes.

For additional information, see *WinSNMP Data Management Concepts* and *Freeing WinSNMP Descriptors.*

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.
Library: Use Wsnmp32.lib.

See Also

WinSNMP API Overview, WinSNMP Functions, **SnmpFreeDescriptor**, **smiOID**

WinSNMP Structures

The WinSNMP API functions use the following structures:

smiCNTR64
smiOCTETS
smiOID
smiVALUE
smiVENDORINFO

smiCNTR64

The WinSNMP **smiCNTR64** structure contains a 64-bit unsigned integer value. The structure represents a 64-bit counter.

```
typedef struct {
    smiUINT32   hipart;    // high-order 32 bits
    smiUINT32   lopart;    // low-order 32 bits
} smiCNTR64, *smiLPCNTR64;
```

Members

hipart
Specifies the high-order 32 bits of the counter.

lopart
Specifies the low-order 32 bits of the counter.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.

See Also

WinSNMP API Overview, WinSNMP Structures, **SnmpGetVb**, **smiVALUE**

smiOCTETS

The WinSNMP **smiOCTETS** structure passes context strings to multiple WinSNMP functions. The structure also describes and receives encoded SNMP messages.

The **smiOCTETS** structure contains a pointer to an SNMP octet string of variable length. The structure can be a member of the **smiVALUE** structure.

```
typedef struct {
    smiUINT32   len;  // number of bytes in the octet string
    smiLPBYTE   ptr;  // pointer to an octet string
} smiOCTETS, *smiLPOCTETS;
```

Members

len

Specifies an unsigned long integer value that indicates the number of bytes in the octet string array pointed to by the **ptr** member.

ptr

Pointer to a byte array that contains the octet string of interest. A NULL-terminating byte is not required.

Remarks

The Microsoft WinSNMP implementation allocates and deallocates memory for all output **smiOCTETS** structures. The WinSNMP application should not free memory that the implementation allocates for the **ptr** member of an **smiOCTETS** structure. Instead, the application must call the **SnmpFreeDescriptor** function to free the memory.

Because the WinSNMP application allocates memory for input descriptor objects with variable lengths, it must free that memory. For more information, see *WinSNMP Data Management Concepts.*

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.

See Also

WinSNMP API Overview, WinSNMP Structures, **SnmpStrToContext**, **SnmpContextToStr**, **SnmpEncodeMsg**, **SnmpDecodeMsg**, **SnmpFreeDescriptor**, **smiVALUE**

smiOID

The WinSNMP **smiOID** structure passes object identifiers to multiple WinSNMP functions. The structure also receives the variable name of a variable binding entry in a call to the **SnmpGetVb** function.

The **smiOID** structure contains a pointer to a variable length array of a named object's subidentifiers. The structure can be a member of the **smiVALUE** structure.

```
typedef struct {
    smiUINT32     len;    // number of array elements
    smiLPUINT32   ptr;    // pointer to an array of
                          // subidentifiers
} smiOID, *smiLPOID;
```

Members

len

Specifies an unsigned long integer value that indicates the number of elements in the array pointed to by the **ptr** member.

ptr

Pointer to an array of unsigned long integers that represent the object identifier's subidentifiers.

Remarks

In an **smiOID** structure, the format of the array pointed to by the **ptr** member is one subidentifier per array element. For example, the string "1.3.6.1" would be an array of four elements {1,3,6,1}.

The Microsoft WinSNMP implementation allocates and deallocates memory for all output **smiOID** structures. The WinSNMP application should not free memory that the implementation allocates for the **ptr** member of an **smiOID** structure. Instead, the application must call the **SnmpFreeDescriptor** function to free the memory.

Because the WinSNMP application allocates memory for input descriptor objects with variable lengths, it must free that memory. For more information, see *WinSNMP Data Management Concepts*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.

See Also

WinSNMP API Overview, WinSNMP Structures, **SnmpGetVb**, **SnmpStrToOid**, **SnmpOidToStr**, **SnmpOidCopy**, **SnmpOidCompare**, **SnmpFreeDescriptor**, **smiVALUE**

smiVALUE

The WinSNMP **smiVALUE** structure describes the value associated with a variable name in a variable binding entry.

The **syntax** member of the **smiVALUE** structure contains a WinSNMP data type that indicates the type of data in the **value** member. The **value** member of the structure is the union of all possible WinSNMP data types.

```
typedef struct {                    // smiVALUE portion of VarBind
   smiUINT32    syntax;             // Insert SNMP_SYNTAX_<type>
     union {
       smiINT     sNumber;          // SNMP_SYNTAX_INT
                                    // SNMP_SYNTAX_INT32
       smiUINT32 uNumber;           // SNMP_SYNTAX_UINT32
                                    // SNMP_SYNTAX_CNTR32
                                    // SNMP_SYNTAX_GAUGE32
                                    // SNMP_SYNTAX_TIMETICKS
       smiCNTR64 hNumber;           // SNMP_SYNTAX_CNTR64
       smiOCTETS string;            // SNMP_SYNTAX_OCTETS
                                    // SNMP_SYNTAX_OPAQUE
                                    // SNMP_SYNTAX_IPADDR
                                    // SNMP_SYNTAX_NSAPADDR
       smiOID     oid;              // SNMP_SYNTAX_OID
       smiBYTE    empty;            // SNMP_SYNTAX_NULL
                                    // SNMP_SYNTAX_NOSUCHOBJECT
                                    // SNMP_SYNTAX_NOSUCHINSTANCE
                                    // SNMP_SYNTAX_ENDOFMIBVIEW
     } value;    // union
} smiVALUE, *smiLPVALUE;
```

Members

syntax

Specifies an unsigned long integer that indicates the syntax data type of the **value** member. This member can be only one of the types listed in the following table. For more information, see *WinSNMP Data Types* and *RFC 1902, "Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)."*

Syntax data type	Meaning
SNMP_SYNTAX_INT	Indicates a 32-bit signed integer variable.
SNMP_SYNTAX_OCTETS	Indicates an octet string variable that is binary or textual data.
SNMP_SYNTAX_NULL	Indicates a NULL value.
SNMP_SYNTAX_OID	Indicates an object identifier variable that is an assigned name with a maximum of 128 subidentifiers.
SNMP_SYNTAX_INT32	Indicates a 32-bit signed integer variable.
SNMP_SYNTAX_IPADDR	Indicates a 32-bit Internet address variable.
SNMP_SYNTAX_CNTR32	Indicates a counter variable that increases until it reaches a maximum value of (2^32) –1.
SNMP_SYNTAX_GAUGE32	Indicates a gauge variable that is a non-negative integer that can increase or decrease, but never exceed a maximum value.
SNMP_SYNTAX_TIMETICKS	Indicates a counter variable that measures the time in hundredths of a second, until it reaches a maximum value of (2^32) –1. It is a non-negative integer that is relative to a specific timer event.
SNMP_SYNTAX_OPAQUE	This type provides backward compatibility, and should not be used for new object types. It supports the capability to pass arbitrary Abstract Syntax Notation One (ASN.1) syntax.
SNMP_SYNTAX_CNTR64	Indicates a counter variable that increases until it reaches a maximum value of (2^64) –1.
SNMP_SYNTAX_UINT32	Indicates a 32-bit unsigned integer variable.
SNMP_SYNTAX_NOSUCHOBJECT	Indicates that the agent does not support the object type that corresponds to the variable.

Syntax data type	Meaning
SNMP_SYNTAX_NOSUCHINSTANCE	Indicates that the object instance does not exist for the operation.
SNMP_SYNTAX_ENDOFMIBVIEW	Indicates the WinSNMP application is attempting to reference an object identifier that is beyond the end of the MIB tree that the agent supports.

The last three syntax types describe exception conditions under the SNMP version 2C (SNMPv2C) framework.

value

Specifies the union of all possible WinSNMP syntax data types, including the **smiOID** or **smiOCTETS** descriptor types.

Remarks

A WinSNMP application must check the **syntax** member of an **smiVALUE** structure to correctly dereference the **value** member. The **value** member can contain a simple scalar value or a non-scalar value like an **smiOCTETS** or an **smiOID** descriptor structure.

If the **syntax** member indicates that the **value** member is an **smiOCTETS** or an **smiOID** descriptor structure, the WinSNMP application must determine whether to free the resources allocated for the structure. The Microsoft WinSNMP implementation allocates and deallocates memory for all output **smiOCTETS** and **smiOID** structures. The application must call the **SnmpFreeDescriptor** function to free the memory for the **ptr** member of these structures.

Because the WinSNMP application allocates memory for input descriptors with variable lengths, it must free that memory. For more information, see *WinSNMP Data Management Concepts*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.

+ See Also

WinSNMP API Overview, WinSNMP Structures, **smiOCTETS**, **smiOID**, **SnmpGetVb**, **SnmpCreateVbl**, **SnmpFreeDescriptor**

smiVENDORINFO

The **smiVENDORINFO** structure contains information about the Microsoft WinSNMP implementation. A WinSNMP application can call the **SnmpGetVendorInfo** function to retrieve this structure. The **smiVENDORINFO** structure is an element of the WinSNMP API, version 2.0.

```
typedef struct {
    CHAR  vendorName[MAXVENDORINFO*2];
    CHAR  vendorContact[MAXVENDORINFO*2];
    CHAR  vendorVersionId[MAXVENDORINFO];
    CHAR  vendorVersionDate[MAXVENDORINFO];
    smiUINT32  vendorEnterprise;
} smiVENDORINFO, FAR *smiLPVENDORINFO;
```

Members

vendorName

Contains the null-terminated string "Microsoft Corporation". The string is suitable for display to end users.

vendorContact

Specifies a null-terminated character string that indicates how Microsoft can be contacted for WinSNMP-related information. For example, this member can contain a postal address, a telephone number or a fax number, a URL, or an e-mail address such as "snmpinfo@microsoft.com". The string is suitable for display.

vendorVersionId

Specifies a null-terminated character string that identifies the version number of the WinSNMP API the Microsoft WinSNMP implementation is currently supporting. The string is suitable for display.

vendorVersionDate

Specifies a null-terminated character string that indicates the release date of the version of the WinSNMP API the Microsoft WinSNMP implementation is currently supporting. The string is suitable for display.

vendorEnterprise

Contains the value 311, Microsoft's enterprise number (permanent address) assigned by the Internet Assigned Numbers Authority (IANA).

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Winsnmp.h.

See Also

SnmpGetVendorInfo

CHAPTER 16

Network Management

Microsoft® Windows NT®, Windows® 2000, Windows® 95, and Windows® 98 support a variety of networking APIs. The network management functions provide the ability to manage user accounts and network resources. Many of the capabilities provided by the network management functions are not provided by other networking functions. However, if the capabilities are provided by another set of functions, the documentation for the network management functions will refer you to other functions you can use for the same task.

About Network Management

The network management functions allow you to manage network shares as Windows Explorer and the Net command do. You can also manage user accounts as User Manager does.

Network Management Function Groups

The network management functions can be divided into the following groups:

- Access functions (Windows 95 and Windows 98 only)
- Alert functions
- ApiBuffer functions
- Directory Service functions
- Distributed File System (Dfs) functions
- Get functions
- Group functions
- Local group functions
- Message functions
- NetFile functions
- Remote Utility functions
- Replicator functions
- Schedule functions
- Server functions
- Server and workstation transport functions
- Session functions
- Share functions
- Statistics functions
- Use functions
- User functions
- User modal functions
- Workstation and workstation user functions

If you are programming for Active Directory, you may be able to call certain ADSI interface methods to achieve the same functionality you can achieve by calling certain network management functions. For more information, see *Mapping ADSI Interfaces to the Network Management Functions.*

The system also provides a network-independent set of network functions (WNet functions) that allow network functions to work across different network vendors' products. If your application could be converted to use a WNet function, you should perform the conversion. There are at least two reasons to make the change:

1. The WNet functions are network independent, while the network management functions work only on Microsoft networks.
2. Some of the Win32 functions documented in this SDK may not be supported in future releases of Microsoft operating systems if they have been superseded. Microsoft does not plan to remove specific functions unless equivalent or better functionality is available.

Windows NT/2000: The following groups of network management functions are obsolete on this platform:

- Access functions
- Audit functions
- Configuration functions
- Error Logging functions
- NetService functions

Access Functions

Windows NT/2000: The access functions are obsolete on Windows NT/Windows 2000. These functions work only when the function is accessed with a LAN Manager 2.*x* system. The Win32 API provides a full set of access control functions. Use these in place of the network management access functions.

Windows 95/98: The access functions examine or modify user or group access permissions for particular network resources. An Access-Control List (ACL) contains the name of a resource, an audit attribute field, and a list of access-control entries. An Access-Control Entry (ACE) is a user name or a group name, and the corresponding access permissions.

The access functions that are supported on Windows 95 and Windows 98 are listed following.

Function	Description
NetAccessAdd	Creates a new ACL for a resource and sets the user or group access permissions.
NetAccessCheck	Verifies whether a user has permission to perform a specified operation on a particular resource.
NetAccessDel	Deletes the ACL for a resource.
NetAccessEnum	Retrieves information about all ACLs.
NetAccessGetInfo	Returns the ACL for a particular resource.

Function	Description
NetAccessGetUserPerms	Returns a user's or group's access permissions for a particular resource.
NetAccessSetInfo	Changes the ACL for a resource and grants access permissions.
NetSecurityGetInfo	Returns access control information in a **security_info_1** structure.

Only users or applications with admin group membership or special permission for the resource can define or examine access permissions. Users have special permissions for a resource when they are granted ACCESS_PERM permission for that resource; this is also known as P permission.

Access permission information is available at the following levels:

access_info_0
access_info_1
access_info_2
access_info_12

Access list information is available at the following level:

access_list

Alert Functions

The network management alert functions notify network service programs and applications of network events. An *event* is a particular instance of a process, occurrence, or state of hardware as defined by an application. The alert functions allow applications to indicate when predefined events occur.

The alert functions are listed following.

Function	Description
NetAlertRaise	Notifies all registered clients that a particular event has occurred.
NetAlertRaiseEx	Simplifies notifying registered clients that a particular event has occurred, because, unlike **NetAlertRaise**, **NetAlertRaiseEx** does not require a **STD_ALERT** structure.

The alerter service must be running on the client computer when you call the **NetAlertRaise** function or the **NetAlertRaiseEx** function. If the service is not running, the functions fail with ERROR_FILE_NOT_FOUND. The alerter service on the client calls the **NetMessageBufferSend** function to send information to recipients.

Applications, network services, and internal network components use the network management alert functions to raise an alert, notifying various applications or users when a particular type of event occurs. The alert category functions, data types, structures, and constants are defined in the LMCONS.H, LMERR.H, and LMALERT.H header files. To access these definitions, define the constants INCL_NETERRORS and INCL_NETALERT, and include the header file LM.H.

The LMALERT.H file predefines the following alert classes (types of network events) for sending alerts:

- Network events requiring administrative assistance
- Addition of an entry to an error log file
- Reception of a broadcast message by a user or an application
- Completion of a print job
- Use of certain applications or resources by users

You can define other classes of alerts for network applications as needed. For example, if an application on a server routinely writes large amounts of data to a disk drive, the application runs the risk of filling the disk. In this case, you might want to add the event "no free disk space" to trigger an alert that notifies the application to pause or to terminate the process that is filling the disk. The event name for an alert can be any text string.

When you raise an alert with a call to the **NetAlertRaise** function, the message data should consist of one **STD_ALERT** header structure, followed by additional fixed-length data that is alert-specific in one **ADMIN_OTHER_INFO**, **ERRLOG_OTHER_INFO**, **PRINT_OTHER_INFO**, or **USER_OTHER_INFO** structure. Additional variable-length data can follow the alert-specific structure. (Calls to the **NetAlertRaiseEx** function do not require a **STD_ALERT** structure.) The calling application must allocate the memory for all structures and variable-length data, and free the memory after the call returns.

The following macros are available for use with alert data buffers.

Macro	Description
ALERT_OTHER_INFO	Returns a pointer to the fixed-length data that follows the **STD_ALERT** structure in an alert message.
ALERT_VAR_DATA	Returns a pointer to the variable-length data that follows the alert-specific data in an alert message.

Instead of using the network management alert functions, you may be able to use the Windows Management Instrumentation (WMI) SDK for event notification. For more information about the platforms that support the WMI event model, see *Event Notification* in the WMI documentation.

ApiBuffer Functions

The network management ApiBuffer functions are used to manage memory allocation. However, in general, you should use the memory management functions provided by the Win32 API.

The ApiBuffer functions are listed following.

Function	Description
NetApiBufferAllocate	Allocates memory from the heap. Call this function when you require compatibility with the **NetApiBufferFree** function.
NetApiBufferFree	Frees memory allocated by the **NetApiBufferAllocate** function and other network management functions.
NetApiBufferReallocate	Changes the size of a buffer allocated by a call to the **NetApiBufferAllocate** function.
NetApiBufferSize	Returns the size, in bytes, of a buffer allocated by a call to the **NetApiBufferAllocate** function.

Windows NT/2000: For remotable functions that return information to the caller, the RPC run-time library allocates the buffer containing the return information. When the caller has finished processing the information, it must call the **NetApiBufferFree** function to free the allocated buffer.

Audit Functions

Note The network management auditing functions are obsolete on Windows NT/Windows 2000 because the system uses an integrated event logging mechanism for reporting both audits and errors.

The network management auditing functions and error logging functions are provided to access LAN Manager 2.*x* logs. They will report ERROR_NOT_SUPPORTED if called on a Windows NT/Windows 2000 system.

Configuration Functions

Note The network management configuration functions are obsolete on Windows NT/Windows 2000. They are only for LAN Manager 2.*x* support. Use the registry functions to retrieve configuration information for Windows NT/Windows 2000 computers.

Directory Service Functions

The network management directory service functions allow developers to work with the domain controller and domain membership in the directory service.

The network management directory service functions are listed following.

Function	Description
NetGetJoinableOUs	Retrieves a list of organizational units (OUs) in which a computer account can be created.
NetGetJoinInformation	Retrieves join status information for the specified computer.
NetJoinDomain	Joins a computer to a workgroup or domain.
NetRenameMachineInDomain	Changes the name of a computer in a domain.
NetUnjoinDomain	Unjoins a computer from a workgroup or a domain.
NetValidateName	Verifies the validity of a computer name, workgroup name, or domain name.

For more information, see the *Active Directory Reference*.

Distributed File System (Dfs) Functions

The Distributed File System (Dfs) functions provide the ability to logically group shares on multiple servers and to transparently link shares into a single hierarchical name space. Dfs organizes shared resources on a network in a treelike structure.

Dfs supports *stand-alone* implementations of Dfs, those with one host server, and *domain-based* implementations that have multiple host servers and high availability. The Dfs *topology data* for domain-based implementations is stored in Active Directory. The data includes the Dfs root, Dfs links and a replica set.

Each Dfs tree structure has one or more *root shares,* which are stored on a physical server running the Dfs process. A root share can contain one or more *Dfs links*. Each Dfs link points to one or more shared folders on the network. You can add, modify and delete Dfs links from a Dfs root share. When you remove the last share associated with a Dfs link, Dfs deletes the Dfs link in the Dfs root share. (In earlier documentation, Dfs links were called junction points.)

When a Dfs link points to more than one shared folder, the folders are called *replicas.* When users access a Dfs link, the Dfs server selects one of the replicas based on site information, if it is available, and connects the user to the replica. This helps to distribute client requests across the replicas and can improve performance.

An application can use the Dfs functions to:

- Add a Dfs link to a Dfs root.
- Create or remove stand-alone and domain-based Dfs roots.
- Add shares to an existing Dfs link.
- Remove a Dfs link from a Dfs root.
- Remove a Dfs path from a Dfs link.
- View and configure information about the Dfs links in a named Dfs root.

The Dfs functions are listed following.

Function	Description
NetDfsAdd	Creates a new Dfs link or adds a share to an existing link.
NetDfsAddFtRoot	Creates a new domain-based Dfs root, or adds a new server and share to an existing domain-based Dfs implementation.
NetDfsAddStdRoot	Creates the root for a new stand-alone Dfs implementation.
NetDfsAddStdRootForced	Creates the root for a new stand-alone Dfs implementation in a cluster technology environment, allowing an offline share to host the Dfs root.
NetDfsEnum	Enumerates all Dfs links in a named Dfs root.
NetDfsGetClientInfo	Returns the client's cached information about a specific Dfs link.
NetDfsGetInfo	Returns information about a specific Dfs link.
NetDfsManagerInitialize	Reinitializes the Dfs service on a specified server.
NetDfsRemove	Removes a share from a Dfs link; removes the Dfs link if the share is the last associated with the specified link.
NetDfsRemoveFtRoot	Removes a server and share from a domain-based Dfs implementation; deletes the Dfs root if there are no more associated shares.
NetDfsRemoveFtRootForced	Removes the specified server from a domain-based Dfs implementation, even if the server is offline.
NetDfsRemoveStdRoot	Deletes the root of a stand-alone Dfs implementation.
NetDfsSetClientInfo	Modifies cached information about a Dfs link on a client computer.
NetDfsSetInfo	Associates information with a Dfs link.

Dfs functions are available at the following information levels:

DFS_INFO_1
DFS_INFO_2
DFS_INFO_3
DFS_INFO_4
DFS_INFO_100
DFS_INFO_101
DFS_INFO_102
DFS_INFO_200
DFS_STORAGE_INFO

Shares on computers that are running Windows NT Workstation, Windows 2000 Professional, Windows 95, Windows 98, or Windows for Workgroups can be published in a Dfs name space. You can also publish any non-Microsoft shares for which client redirectors are available in a Dfs name space. However, unlike a share that is published on a server that is running Windows NT Server 4.0 or Windows 2000 Server, they cannot host a Dfs share or point to other Dfs shares.

Dfs uses the Windows 2000 file replication service to copy changes between replicated shares. Users can modify files stored on one replica, and the file replication service propagates the changes to the other designated replicas. The service preserves the most recent change to a document or files.

Error Logging Functions

Note The network management error logging functions are obsolete on Windows NT/Windows 2000 because the system uses an integrated event logging mechanism for reporting both audits and errors.

The network management auditing functions and error logging functions are provided to access LAN Manager 2.*x* logs. They will report ERROR_NOT_SUPPORTED if called on a Windows NT/Windows 2000 system.

Get Functions

The network management get functions retrieve information about a domain. You can also call these functions to retrieve information about local, global, workstation, and server user accounts.

The network management get functions are listed following.

Function	Description
NetGetAnyDCName	Returns the name of any domain controller for a domain that is directly trusted by a specified server.
NetGetDCName	Returns the name of the Primary Domain Controller (PDC) for the specified domain.
NetGetDisplayInformationIndex	Returns the index of the first display information entry whose name begins with a specified string or alphabetically follows the string.
NetQueryDisplayInformation	Returns user, computer, or global group account information.

The information returned by the **NetQueryDisplayInformation** function is available at the following levels:

NET_DISPLAY_GROUP
NET_DISPLAY_MACHINE
NET_DISPLAY_USER

Group Functions

A *global group* contains user accounts from one domain that are grouped together under one group account name. A global group can contain only members (users) from the domain where the global group is created; it cannot contain local groups or other global groups. A global group is available within its own domain and within any trusting domain.

The network management group functions control global groups. The group functions are listed following.

Function	Description
NetGroupAdd	Creates a global group.
NetGroupAddUser	Adds one user to an existing global group.
NetGroupDel	Removes a global group whether or not the group has any members.
NetGroupDelUser	Removes one user name from a global group.
NetGroupEnum	Lists all global groups on a server.
NetGroupGetInfo	Returns information about a particular global group.
NetGroupGetUsers	Lists all members of a particular global group.
NetGroupSetInfo	Sets general information about a global group.
NetGroupSetUsers	Assigns members to a new global group; replaces the members of an existing group.

When you call the **NetGroupAdd** function to create a global group, you must supply a group name. Initially, a new group has no members.

User accounts automatically belong to one of the special global groups **Domain**, **Users**, or **None**, according to the user's security requirements. Membership in these groups is indirectly controlled by the **NetUserAdd**, **NetUserDel**, and **NetUserSetInfo** functions.

Global group account information is available at the following levels:

GROUP_INFO_0
GROUP_INFO_1
GROUP_INFO_2
GROUP_INFO_1002
GROUP_INFO_1005

The 1002 and 1005 levels are valid only for the **NetGroupSetInfo** function.

Global group member information is available at the following information level:

GROUP_USERS_INFO_0

For more information, see the network management Local Group Functions.

If you are programming for Active Directory™, you may be able to call certain Active Directory Service Interface (ADSI) methods to achieve the same functionality you can achieve by calling the network management group functions. For more information, see **IADsGroup**.

Local Group Functions

A *local group* can contain user accounts or global group accounts from one or more domains. (Global groups can contain users from only one domain.) A local group shares common privileges and rights only within its own domain.

The network management local group functions control members of local groups in a way that the functions can only be called locally on the system on which the local group is defined. On a Windows NT/Windows 2000 workstation, or on a server that is not a domain controller, you can use only a local group defined on that system. A local group defined on the primary domain controller is replicated to all other domain controllers in the domain. Therefore, a local group is available on all domain controllers within the domain in which it was created.

The local group functions create or delete local groups, and review or adjust the memberships of local groups. These functions are listed following.

Function	Description
NetLocalGroupAdd	Creates a local group.
NetLocalGroupAddMembers	Adds one or more users or global groups to an existing local group.
NetLocalGroupDel	Deletes a local group, removing all existing members from the group.
NetLocalGroupDelMembers	Removes one or more members from an existing local group.
NetLocalGroupEnum	Returns information about each local group account on a server.
NetLocalGroupGetInfo	Returns information about a particular local group account on a server.
NetLocalGroupGetMembers	Lists all members of a specified local group.
NetLocalGroupSetInfo	Sets general information about a local group.
NetLocalGroupSetMembers	Assigns members to a local group.

You can add a member to a local group by specifying the Security Identifier (SID) of the member. To translate a member account name to a SID, call the **LookupAccountName** function.

When you create a local group by calling the **NetLocalGroupAdd** function, you must supply a local group name. Initially, the local group has no members.

Local group account information is available at the following levels:

LOCALGROUP_INFO_0
LOCALGROUP_INFO_1
LOCALGROUP_INFO_1002

Local group membership information is available at the following information levels:

LOCALGROUP_MEMBERS_INFO_0
LOCALGROUP_MEMBERS_INFO_1
LOCALGROUP_MEMBERS_INFO_2
LOCALGROUP_MEMBERS_INFO_3

You can retrieve the names of the local groups to which a user belongs by calling the **NetUserGetLocalGroups** function, specifying the following information level:

LOCALGROUP_USERS_INFO_0

For more information, see the network management Group Functions.

If you are programming for Active Directory™, you may be able to call certain Active Directory Service Interface (ADSI) methods to achieve the same functionality you can achieve by calling the network management local group functions. For more information, see ***IADsGroup***.

Message Functions

The network management message functions send messages and maintain message aliases. The message functions are listed following.

Function	Description
NetMessageBufferSend	Sends a message to a registered message alias.
NetMessageNameAdd	Registers a message alias in the message name table.
NetMessageNameDel	Deletes a message alias from the message name table.
NetMessageNameEnum	Lists all the message aliases stored in the message name table.
NetMessageNameGetInfo	Returns information about a particular message alias in the message name table.

A *message* is a buffer of text data sent to a user or application on the network. To receive a message, a user or application must register a message alias in a computer's table of message names. This can be done by calling the **NetMessageNameAdd** function. A *message name table* contains a list of registered message aliases (users and applications) permitted to receive messages. The aliases registered in the message name table are case insensitive.

The messenger service must be running on the receiving computer to display a pop-up message when the message is received. In addition, the Workstation service must be running on the local computer. **Netbios** is the transport mechanism used between the sender and receiver.

Message functions are available at two information levels:

MSG_INFO_0
MSG_INFO_1

MSG_INFO_1 exists only for compatibility. The messenger service does not forward names or allow names to be forwarded to it.

NetFile Functions

The network management file functions provide a way to monitor and close the file, device, and pipe resources open on a server. The file functions are listed following.

Function	Description
NetFileClose	Forces a resource to close.
NetFileEnum	Returns information about open files on a server.
NetFileGetInfo	Returns information about a particular opening of a server resource.

Call the **NetFileClose** function when the file cannot be closed by any other means. This function should be used with caution because **NetFileClose** does not write data cached on the client system to the file before closing the file.

The **NetFileEnum** function returns information about resources open on a server. A file can be opened one or more times by one or more applications. Each file opening is uniquely identified. The **NetFileEnum** function returns an entry for each file opening. The **NetFileGetInfo** function returns information about one opening of a resource.

File information is available at the following levels:

FILE_INFO_2
FILE_INFO_3

Levels 0 and 1 are not supported. Level 2 returns only the identification number assigned to the resource when it was opened. Level 3 returns the identification number, permissions, file locks, and the name of the user who opened the resource.

If you are programming for Active Directory™, you may be able to call certain Active Directory Service Interface (ADSI) methods to achieve the same functionality you can achieve by calling the **NetFileEnum** and **NetFileGetInfo** functions. For more information, see ***IADsResource*** and ***IADsFileServiceOperations***.

Windows 95/98: The **NetFileClose2** function forces a resource to close. This function can be used when an error prevents closure by any other means. The **file_info_50** structure is supported on Windows 95 and Windows 98.

NetService Functions

Note The network management NetService functions are obsolete on Windows NT/Windows 2000 because the system provides a complete set of service functions. These should be used in place of the NetService functions, unless you need to control services on a LAN Manager 2.*x* server.

A service can be started using the service functions. At startup time, the service defines whether it can be stopped, paused, and continued. Windows NT/Windows 2000 networking provides several standard services, such as the workstation, server, and messenger services.

Remote Utility Functions

The network management remote utility functions are listed following.

Function	Description
NetRemoteComputerSupports	Queries the redirector to retrieve the optional features that a remote system supports.
NetRemoteTOD	Enables applications to access the time-of-day information on a remote server.

The remote time-of-day information is available at one information level:

TIME_OF_DAY_INFO Replicator Functions

The Windows NT replicator service maintains identical sets of files and directories on different servers and workstations. When you update files on one server, the file replicator service replaces the corresponding files on other servers and workstations with the updated files. The replication process simplifies the task of updating and coordinating files, and maintains the integrity of the replicated data. For more information about replication, see the following topics:

- About export and import servers
- About the file replicator service

Replication is controlled by options you set in the LANMAN.INI file using the following categories of network management replicator functions.

- Replicator configuration functions
- Replicator export directory functions
- Replicator import directory functions

About Export and Import Servers

To replicate a set of files and directories on several computers, you must create a master set of files and directories on a server that you designate as an *export server*. Export servers maintain the master files in an export directory tree with a maximum of 32 levels. Each directory can contain as many as 1000 files.

Import servers are servers and workstations that receive replicated files. Import servers and workstations have import directories that correspond to the export directories. When a file changes in the export directory, the file replicator service copies the changes to the corresponding import directories on all designated import servers and workstations. The service replicates all directory and file additions and deletions to the import servers.

A network can have any number of export and import servers. A server can be designated as both an export and an import server. You can configure workstations only as import servers.

About the File Replicator Service

Before using the replicator functions, check the Control Panel Services application to make sure the file replicator service is configured to log on using a specific user account. The user account must be a member of the Replicator local group, the account must not be disabled, and the account must have permission to access the import and export trees on the respective servers.

Changing from file system replication control in LAN Manager 2.*x* to replicator function control in Windows NT has the following implications:

- To terminate replication from a client's master, applications can no longer delete a directory in the client's import path.
- To control the method of replication, applications can no longer use the REPL.INI file located in each replicated directory on the master.
- To control replication of a master directory, applications can no longer create or delete USERLOCK.* files.
- To prevent replication on a client from a master directory, applications can no longer create or delete USERLOCK.* files.
- You will need to modify applications that depend on the LAN Manager 2.*x* behavior of ignoring locks for file integrity trees. (The Windows NT/Windows 2000 policy differs from LAN Manager 2.*x* policy; under Windows NT/Windows 2000 the locks are always respected.)

Each of the options listed preceding can be specified to the file replicator service by calling the appropriate network management replicator function.

Any user or application logged on as a member of either the administration group or the server operator group can modify the parameters that control the file replicator service. (This applies to both local and remote export servers.)

You can use the Windows NT file replicator service to copy relatively small directory trees. If you are attempting to replicate multimegabyte directory trees or trees containing thousands of directories and files, you should consider some other means of doing so.

Replicator Configuration Functions

You can use the replicator configuration functions to examine and modify the configuration parameters for the file replicator service. The replicator configuration functions are listed following.

Function	Description
NetReplGetInfo	Returns configuration information for the file replicator service.
NetReplSetInfo	Modifies configuration information for the file replicator service.

Configuration information for the file replicator service is available at the following levels:

REPL_INFO_0
REPL_INFO_1000
REPL_INFO_1001
REPL_INFO_1002
REPL_INFO_1003

Replicator Export Directory Functions

The replicator export directory functions control top-level directories under the export path on the master directory.

The replicator export directory functions are listed following.

Function	Description
NetReplExportDirAdd	Registers an existing directory in the export path for replication.
NetReplExportDirDel	Removes the registration of a replicated directory.
NetReplExportDirEnum	Lists the replicated directories in the export path.
NetReplExportDirGetInfo	Returns the control information for a replicated directory.
NetReplExportDirLock	Locks a directory so that replication of it can be suspended.
NetReplExportDirSetInfo	Modifies the control information for a replicated directory.
NetReplExportDirUnlock	Unlocks a directory so that replication of it can resume.

A master directory can be registered for replication in a variety of ways.

- The file replicator service automatically registers a new directory for replication when a user creates it under the master directory's export path. In this case, the file replicator service gives the directory the REPL_INTEGRITY_FILE and REPL_EXTENT_TREE settings for the **integrity** and **extent** replication controls.
- An application can call the **NetReplExportDirAdd** function to register a pre-existing directory in the export path for replication. When adding a directory in this manner, you must specify the settings for the **integrity** and **extent** replication controls using the **rpedX_integrity** and **rpedX_extent** members of the appropriate **REPL_EDIR_INFO** structure.

The **integrity** control determines when a master updates a client. The control can be one of the following values.

Value	Meaning
REPL_INTEGRITY_FILE	The client receives a replica of a file within the directory when the directory isn't being modified or replicated.
REPL_INTEGRITY_TREE	Before the file replicator service updates the client, each file and directory within the replicated directory must be stable for a specific period of time. This time is specified by the **rp0_guardtime** member of the **REPL_INFO_0** structure. (Call the **NetReplGetInfo** function to retrieve this type of configuration information for the file replicator service.)

The **extent** control specifies the selection of files for replication within the main export directory. The control can be one of the following values.

Value	Meaning
REPL_EXTENT_TREE	The file replicator service replicates the entire tree within the directory.
REPL_EXTENT_FILE	The file replicator service replicates only the files in the first-level directory.

For additional information about these replication controls, see **REPL_EDIR_INFO_1**.

On systems running LAN Manager 2.*x*, the replication controls used to be specified in the REPL.INI file in each replicated directory. The controls could not be dynamically set. To examine the replication controls of a directory on Windows NT/Windows 2000, call the **NetReplExportDirGetInfo** function; to modify them, call **NetReplExportDirSetInfo**. Windows NT/Windows 2000 ignores the REPL.INI file.

You can call the replicator export directory functions whether or not the file replicator service is running. If the service is running on a master, modifications to the directory controls take effect immediately, and the changes persist after the file replicator service stops. If the service has not started, the changes to the directory controls are stored as persistent information and take effect when the file replicator service starts.

The replicator export directory functions are available at the following information levels:

REPL_EDIR_INFO_0
REPL_EDIR_INFO_1
REPL_EDIR_INFO_2
REPL_EDIR_INFO_1000
REPL_EDIR_INFO_1001

The **rped2_lockcount** and **rped2_locktime** members of the **REPL_EDIR_INFO_2** structure contain lock status information.

Replicator Import Directory Functions

The replicator import directory functions designate the top-level directories in the client's import path that should receive updates from the master. The functions also return status information about a replicated directory on the client. (On LAN Manager 2.*x*, after a user creates a directory under the import path, the file replicator service automatically replicates to it.)

The replicator import directory functions are listed following.

Function	Description
NetReplImportDirAdd	Registers an existing directory in the import path to receive replication from a master.
NetReplImportDirDel	Removes the registration of a directory in the import path so that it no longer receives updates from the master; the function does not delete the directory from the file system.
NetReplImportDirEnum	Lists the client directories that are registered for replication.
NetReplImportDirGetInfo	Returns status information for a replicated directory on an import server.
NetReplImportDirLock	Locks a directory so that replication to it can be suspended.
NetReplImportDirUnlock	Unlocks a directory so that replication to it can resume.

You can register a client directory for replication in one of the following ways:

- The file replicator service automatically adds a directory to the client's import path if the directory is exported by a master from which the import server is already importing.
- An application can call the **NetReplImportDirAdd** function to register a preexisting directory in the import path for replication. This can be useful if you want to modify the import directory's properties prior to importing the directory; for example, you may want to lock the directory and suspend replication. (The function does not create the directory.)

You can call the replicator import directory functions whether or not the file replicator service is running. If the service is running on a client, directory additions and deletions take effect immediately, and the changes persist after the file replicator service stops. If the service has not started, and if there is a master that exports the directory, directory additions receive updates when the file replicator service starts.

The replicator import directory functions are available at the following information levels:

REPL_IDIR_INFO_0
REPL_IDIR_INFO_1

Schedule Functions

The network management schedule service functions submit and manage jobs that execute on a specified computer at a particular time (or times) in the future. Jobs can include commands and programs. The functions manage jobs at remote and local computers, provided the schedule service is running on the computer.

The schedule service functions are listed following.

Function	Description
NetScheduleJobAdd	Submits a job to run at a specified future date and time.
NetScheduleJobDel	Cancels a range of jobs queued to run on a computer.
NetScheduleJobEnum	Lists the jobs queued on a specified computer.
NetScheduleJobGetInfo	Returns information about a particular job queued on a computer.

For the network management schedule functions to succeed, a caller must have administrator's privilege at a computer where the schedule service is running. The schedule service functions are also known as "Job" and "AT command" functions.

The **AT_INFO** structure is used by the **NetScheduleJobAdd** function to specify information when submitting a job, and by the **NetScheduleJobGetInfo** function to retrieve information about a job that has been submitted. The **AT_ENUM** structure is used by **NetScheduleJobEnum** to enumerate and return information about an entire queue of submitted jobs.

Server Functions

The network management server functions perform administrative tasks on a local or remote server. The server functions are listed following.

Function	Description
NetServerDiskEnum	Returns a list of local disk drives on a server.
NetServerEnum	Lists all visible servers of a particular type (or types) in the specified domain.

Function	Description
NetServerGetInfo	Returns configuration information about a specified server.
NetServerSetInfo	Sets the operating parameters for a server.

Any user or application with admin group membership on a local or remote server can perform administrative tasks on that server to control the server's operation, user access, and resource sharing. The low-level parameters that affect a server's operation can be examined and modified by calling the **NetServerGetInfo** and **NetServerSetInfo** functions. These parameters are defined in the server's LANMAN.INI file.

Most network management server functions execute only on a remote server. The **NetServerEnum** function executes on either a local workstation or a remote server. If you attempt to execute other server functions on a local workstation, the functions return the error NERR_RemoteOnly.

Server-specific information is available at the following levels, starting at level 100:

SERVER_INFO_100
SERVER_INFO_101
SERVER_INFO_102
SERVER_INFO_402
SERVER_INFO_403
SERVER_INFO_1501
SERVER_INFO_1502
SERVER_INFO_1503
SERVER_INFO_1506
SERVER_INFO_1509
SERVER_INFO_1510
SERVER_INFO_1511
SERVER_INFO_1512
SERVER_INFO_1513
SERVER_INFO_1515
SERVER_INFO_1516
SERVER_INFO_1518
SERVER_INFO_1523
SERVER_INFO_1528
SERVER_INFO_1529
SERVER_INFO_1530
SERVER_INFO_1533
SERVER_INFO_1536
SERVER_INFO_1538
SERVER_INFO_1539
SERVER_INFO_1540
SERVER_INFO_1541
SERVER_INFO_1542
SERVER_INFO_1544
SERVER_INFO_1550
SERVER_INFO_1552

The server information levels that were available in LAN Manager 2.*x* are no longer available. However, the following structures are supported on LAN Manager 2.*x* systems:

SERVER_INFO_1005
SERVER_INFO_1010
SERVER_INFO_1016
SERVER_INFO_1017
SERVER_INFO_1018
SERVER_INFO_1107

If you are programming for Active Directory™, you may be able to call certain Active Directory Service Interface (ADSI) methods to achieve the same functionality you can achieve by calling the network management server functions. For more information, see ***IADsComputer***.

Windows 95/98: The following information levels are supported on Windows 95 and Windows 98:

server_info_1
server_info_50

For more information, see the *Server and Workstation Transport Functions*.

Server and Workstation Transport Functions

The network management server and workstation transport functions handle binding and unbinding of transport protocols to and from the server and redirector. The server transport functions deal with transport protocols managed by the server; the workstation transport functions deal with transport protocols managed by the redirector.

File sharing between a transport device and a server has two components:

- The server computer where the files reside
- A Server Message Block (SMB) client that accesses the files

The client computer communicates with the server computer over a local area network using a transport protocol; for example, TCP, NetBEUI, or XNS. The client sends requests to the server to retrieve data. The software on the client computer that generates the file requests is called the *redirector* because it redirects local file requests to the server computer. The software on the computer that receives and acts on the file requests is called the *server* because it serves the clients. The format specific to these requests is called the SMB protocol.

The server transport functions are listed following.

Function	Description
NetServerComputerNameAdd	Binds an emulated server name to each of the transport protocols on which a server is active. (Combines the functionality of the **NetServerTransportEnum** function and the **NetServerTransportAddEx** function.)
NetServerComputerNameDel	Disconnects each network transport protocol from an emulated server name set by a previous call to the **NetServerComputerNameAdd** function.
NetServerTransportAdd	Binds the specified server to the transport protocol. (This function supports only the **SERVER_TRANSPORT_INFO_0** information level.)

Function	Description
NetServerTransportAddEx	Binds the specified server to the transport protocol. (This extended function supports the **SERVER_TRANSPORT_INFO_1**, **SERVER_TRANSPORT_INFO_2**, and **SERVER_TRANSPORT_INFO_3** information levels.)
NetServerTransportDel	Disconnects the transport protocol from the server.
NetServerTransportEnum	Enumerates the transport protocols managed by the server.

Server transport functions are available at the following information levels:

SERVER_TRANSPORT_INFO_0
SERVER_TRANSPORT_INFO_1
SERVER_TRANSPORT_INFO_2
SERVER_TRANSPORT_INFO_3

The workstation transport functions perform equivalent operations for the workstation. The workstation transport functions are listed following.

Function	Description
NetWkstaTransportAdd	Connects the redirector to the transport protocol.
NetWkstaTransportDel	Disconnects the transport protocol from the redirector.
NetWkstaTransportEnum	Lists the transport protocols that are managed by the redirector.

Workstation transport functions are available at one information level:

WKSTA_TRANSPORT_INFO_0

Session Functions

The network management session functions control network sessions established between workstations and servers. The functions require that the server service be started on the server.

The session functions are listed following.

Function	Description
NetSessionDel	Deletes the current connections between a workstation and server; terminates the network session.
NetSessionEnum	Returns information about all current sessions established for a server.
NetSessionGetInfo	Returns information about a particular session.

A *session* is a link between a workstation and a server. A session is established the first time a workstation makes a connection to a shared resource on the server. Until the session ends, all further connections between the workstation and the server are part of the same session. To end a session, an application on the server end of a connection calls the **NetSessionDel** function.

The network management session functions manage information on a per-user basis with the *username* parameter. Because there can be multiple users per session, this parameter is necessary to access the user-specific information for the session.

Session functions are available at five information levels:

SESSION_INFO_0
SESSION_INFO_1
SESSION_INFO_2
SESSION_INFO_10
SESSION_INFO_502

If you are programming for Active Directory™, you may be able to call certain Active Directory Service Interface (ADSI) methods to achieve the same functionality you can achieve by calling the network management session functions. For more information, see ***IADsSession*** and ***IADsFileServiceOperations***.

Windows 95/98: The following information levels are supported on Windows 95 and Windows 98:

session_info_0
session_info_1
session_info_2
session_info_10
session_info_50

Share Functions

The network management share functions control shared resources. A shared resource is a local resource on a server (for example, a disk directory, print device, or named pipe) that can be accessed by users and applications on the network.

The share functions are listed following.

Function	Description
NetShareAdd	Shares a resource on a server.
NetShareCheck	Queries whether a server is sharing a device.
NetShareDel	Deletes a share name from a server's list of shared resources.
NetShareEnum	Retrieves share information about each shared resource on a server.

Function	Description
NetShareGetInfo	Retrieves information about a specified shared resource on a server.
NetShareSetInfo	Sets a shared resource's parameters.

The **NetShareAdd** function allows a user or application to share a resource of a specific type using the specified share name. The **NetShareAdd** function requires the share name and local device name to share the resource. A user or application must have an account on the server to access the resource.

You can also specify a security descriptor to be associated with a share. Security descriptors specify which users are allowed to access files through the share, and with what type of access.

The network management functions use the following special share names for interprocess communication (IPC) and remote administration of the server.

- IPC$, reserved for interprocess communication
- ADMIN$, reserved for remote administration
- A$, B$, C$ (and other local disk names followed by a dollar sign), assigned to local disk devices

To list all connections made to a shared resource on a server, or to list all connections established from a particular computer, call the **NetConnectionEnum** function. You can call **NetConnectionEnum** at the **CONNECTION_INFO_0** and **CONNECTION_INFO_1** information levels.

Share functions are available at the following information levels:

SHARE_INFO_0
SHARE_INFO_1
SHARE_INFO_2
SHARE_INFO_501
SHARE_INFO_502
SHARE_INFO_1005

The following information levels are valid only for **NetShareSetInfo**:

SHARE_INFO_1004
SHARE_INFO_1006

If you are programming for Active Directory™, you may be able to call certain Active Directory Service Interface (ADSI) methods to achieve the same functionality you can achieve by calling the network management share functions. For more information, see ***IADsFileShare***.

Windows 2000: The **SHARE_INFO_1501** information level is supported only on Windows 2000.

Windows 95/98: The following information levels are supported on Windows 95 and Windows 98:

connection_info_0
connection_info_1
connection_info_50
share_info_0
share_info_1
share_info_2
share_info_50

Statistics Functions

Windows NT/Windows 2000 accumulates a set of operating statistics for workstations and servers from the time that the workstation or server service is started. To retrieve these statistics, you can call the following network management statistics function.

Function	Description
NetStatisticsGet	Retrieves operating statistics for a service; supports the workstation and server services.

Because Windows NT/Windows 2000 and LAN Manager 2.*x* workstations collect a different set of statistics, the caller must know whether the server is running Windows NT/Windows 2000 or LAN Manager 2.*x*. You can call the **NetServerGetInfo** function to determine the type of server and interpret the returned buffer accordingly.

The **NetStatisticsGet** function returns a **STAT_WORKSTATION_0** structure when workstation statistics are requested; the function returns a **STAT_SERVER_0** structure when server statistics are requested.

Use Functions

The network management use functions examine and manage connections (uses) between workstations and servers. The use functions are listed following.

Function	Description
NetUseAdd	Creates a connection between a local computer and a server.
NetUseDel	Ends a connection to a shared resource.
NetUseEnum	Lists all current connections between the local computer and resources on remote servers.
NetUseGetInfo	Returns information about a connection to a shared resource.

Connections are distinguished from sessions: a *session* is established the first time a workstation makes a connection to a shared resource on the server. All additional connections between the workstation and the server are part of this same session until the session ends. Two types of connections can be made: device-name connections (which can only be explicit) and Universal-Naming Convention (UNC) connections (which can be explicit or implicit).

Connections are made on a per-user basis. A connection made by a user is deleted when that user logs off. For this reason the network management use functions are local only, because a connection set up by a remote user would not be accessible to any other users, even the user that was interactively logged on to that computer.

The **NetUseAdd** function establishes an explicit connection between the local computer and a resource shared on a server by redirecting a local device name to the share name of a remote server resource (*servername**sharename*). Once a device-name connection is made, users or applications can use the remote resource by specifying the local device name.

Implicit UNC connections are made by the function responsible for the connection. To establish an implicit UNC connection, an application passes the share name of a resource to any function that accepts UNC paths. The function accepts the UNC name and makes a connection to the specified share name. All further requests on this connection require the full share name.

The use functions are available at the following information levels:

USE_INFO_0
USE_INFO_1
USE_INFO_2

Information level 2 is not available if the function is accessed with a LAN Manager 2.*x* system. In that case, the function returns ERROR_NOT_SUPPORTED.

User Functions

The network management user functions control a user's account in the security database. The user functions are listed following.

Function	Description
NetUserAdd	Adds a user account and assigns a password and privilege level.
NetUserChangePassword	Changes a user's password for a specified network server or domain.
NetUserDel	Deletes a user account from the server.
NetUserEnum	Lists all user accounts on a server.
NetUserGetGroups	Returns a list of global group names to which a user belongs.
NetUserGetInfo	Returns information about a particular user account on a server.

(continued)

(continued)

Function	Description
NetUserGetLocalGroups	Returns a list of local group names to which a user belongs.
NetUserSetGroups	Sets global group memberships for a specified user account.
NetUserSetInfo	Sets the password and other elements of a user account.

Each user or application that accesses network resources must have an account in the security database. The Windows NT/Windows 2000 Server directory services use this account to verify that the user or application has permission to connect to a resource. When a user or an application requests access to a resource, the security system checks for an appropriate user account or group account to permit the access.

Once you remove a user account by calling the **NetUserDel** function, the user can no longer access the server except by using the guest account.

Because a user's password is confidential, it is not returned by the **NetUserEnum** function or the **NetUserGetInfo** function. The password is initially assigned when you call **NetUserAdd**.

User account information is available at the following levels:

USER_INFO_0
USER_INFO_1
USER_INFO_2
USER_INFO_3
USER_INFO_10
USER_INFO_11
USER_INFO_20
USER_INFO_21
USER_INFO_22

In addition, the following information levels are valid when you call the **NetUserSetInfo** function:

USER_INFO_1003
USER_INFO_1005
USER_INFO_1006
USER_INFO_1007
USER_INFO_1008
USER_INFO_1009
USER_INFO_1010
USER_INFO_1011
USER_INFO_1012
USER_INFO_1013
USER_INFO_1014
USER_INFO_1017
USER_INFO_1018
USER_INFO_1020
USER_INFO_1023
USER_INFO_1024
USER_INFO_1025
USER_INFO_1051
USER_INFO_1052
USER_INFO_1053

If you are programming for Active Directory™, you may be able to call certain Active Directory Service Interface (ADSI) methods to achieve the same functionality you can achieve by calling the network management user functions. For more information, see ***IADsUser*** and ***IADsComputer***.

User Modal Functions

The network management user modal functions control the system-wide parameters that affect the Windows NT/Windows 2000 security system behavior.

The user modal functions are listed following.

Function	Description
NetUserModalsGet	Returns global information for all users and global groups in the security database.
NetUserModalsSet	Sets global information for all users and global groups in the security database.

The **NetUserModalsGet** and **NetUserModalsSet** functions examine and modify the modal settings, which are global parameters that affect every account in the security database (for example, the minimum allowable password length). All modal settings can be altered by calling **NetUserModalsSet**. Most of the modals can also be altered by using the **net accounts** command. The network management user modal functions do not require the server to have user-level security.

User modal information is available at the following levels:

USER_MODALS_INFO_0
USER_MODALS_INFO_1
USER_MODALS_INFO_2
USER_MODALS_INFO_3

The following information levels are valid only for **NetUserModalsSet** and replace the older way of passing in a *Parmnum* to set a specific field:

USER_MODALS_INFO_1001
USER_MODALS_INFO_1002
USER_MODALS_INFO_1003
USER_MODALS_INFO_1004
USER_MODALS_INFO_1005
USER_MODALS_INFO_1006
USER_MODALS_INFO_1007

If you are programming for Active Directory™, you may be able to call certain Active Directory Service Interface (ADSI) methods to achieve the same functionality you can achieve by calling the network management user modal functions. For more information, see ***IADsDomain***.

Workstation and Workstation User Functions

The network management workstation functions perform administrative tasks on a local or remote workstation. Any user or application with admin group membership, on a local or remote server, can perform administrative tasks on a workstation to control its operation, user access, and resource sharing.

The workstation functions are listed following.

Function	Description
NetWkstaGetInfo	Returns information about the configuration elements for a workstation.
NetWkstaSetInfo	Configures a workstation.

The workstation functions allow access to two discrete types of workstation information:

- System information
- Platform-specific information (Windows NT, OS/2, MS-DOS, and so on)

Within each type the data is categorized by security access. Data that is guest-accessible is a subset of the data that is user-accessible, which is in turn a subset of the admin-accessible data.

The workstation information structures have been restructured from those of LAN Manager 2.*x* to allow the information to be grouped by type and security accesses. The LAN Manager 2.*x* workstation information format has been discontinued due to the following:

- The base levels (0 and 1) were not grouped by accessibility. A non-superset level (level 10) was required to allow guest access to the information.
- Platform-specific implementation information was included in the base levels such that every platform had to return all information including a default for non-relevant fields. This increased the size of the information structures unnecessarily, making the functions cumbersome to use.

Workstation information is available at the following levels:

WKSTA_INFO_100
WKSTA_INFO_101
WKSTA_INFO_102

The network management workstation user functions allow access to user-specific information. The user-specific information is separated from the workstation information because there can be more than one user on a workstation.

The workstation user functions are listed following.

Function	Description
NetWkstaUserEnum	Lists information about all users currently logged on to the workstation.
NetWkstaUserGetInfo	Returns information about one currently logged-on user.
NetWkstaUserSetInfo	Sets the user-specific information for the configuration elements of a workstation.

Workstation user information is available at the following levels:

WKSTA_USER_INFO_0
WKSTA_USER_INFO_1
WKSTA_USER_INFO_1101

Network Management Data

The following topics discuss the data buffers, alignment, structures, and handles used by the network management functions.

- Network Management Function Buffers
- Network Management Function Buffer Lengths
- API Data Alignment
- Embedded Strings
- Enumeration Resume Handles
- Function Status
- NLS Support
- Parameter Error Reporting
- RPC Buffer Allocation Errors
- Obsolete Information Fields

Network Management Function Buffers

Windows NT/2000: The RPC run-time library handles the buffers required by the 32-bit data retrieval network management functions as follows:

- **Sending data to the server** (data specified by [in] parameters).

  The caller must allocate the buffer for the relevant information structure (or structures) and pass a pointer variable to the function. The caller does not need to specify the buffer length.

  Example: **NetGroupAdd**
- **Retrieving data from the server** (data specified by [out] parameters).

The system allocates the buffer for the returned information. The caller must pass a pointer variable to the function on input. On successful return, the pointer receives the address of the system-allocated buffer that contains the returned information. This simplifies the calling code, because the caller does not need to estimate the size of the buffer, or resize the buffer and reissue the function.

When the caller has finished processing the returned information, it must free the system-allocated memory by calling the **NetApiBufferFree** function. For more information about specifying buffer sizes, see *Network Management Function Buffer Lengths*.

Example: **NetGroupEnum**

Windows 95/98: The caller must provide and free all buffers required by the network management functions.

Network Management Function Buffer Lengths

Applications that specify buffer sizes when calling network management enumeration functions (and various data retrieval functions) must specify buffers large enough to hold the returned information structure (or structures) plus the strings to which their members point. If you do not specify a large enough buffer to receive all the available entries, the function returns ERROR_MORE_DATA. Enumeration calls do not return partial entries.

Windows NT/2000: The network management functions take an advisory maximum data-length parameter, *prefmaxlen*. This parameter allows an application to suggest the number of bytes the server should return from a function call.

If you specify the value MAX_PREFERRED_LENGTH in the *prefmaxlen* parameter, the network management functions allocate the amount of memory required for the data.

Windows 95/98: The caller must provide and free all buffers required by the network management functions.

For more information, see *Network Management Function Buffers*.

API Data Alignment

All structures specified for the network management functions must be 32-bit word aligned. The base size for a structure element is a **DWORD**.

Embedded Strings

Information structures will not contain embedded strings. This improves the alignment of the information structures and allows for OEM flexibility in the core functions.

Any information field that is returned in an enumeration call that can be subsequently used as a key for a GetInfo call is guaranteed to be present in the enumeration buffer. If the variable-length information string that would specify the key field value will not fit, then the entire fixed-length structure for the entry is not returned. Other variable-length fields will be returned as a NULL pointer for the case in which the string does not fit.

Enumeration Resume Handles

Enumeration resume handles are identifiers for the actual resume key contained in the instance data for the function. This is required for security, interoperability, and to simplify the caller code for the function.

If a NULL is passed for the pointer to the resume handle, no handle is stored and the enumeration search cannot be continued. This is useful in cases where the application does not want to enumerate all the items.

If an error is returned from an enumeration call, the resume handle must be treated as invalid and not used for any subsequent enumeration calls. When this occurs you must restart the enumeration from the beginning.

Function Status

The network management functions return zero on success; a nonzero return code indicates an error. Because the network management functions use RPC, the error definitions include RPC error codes. For more information, see *Net Error Codes.*

NLS Support

Windows NT/2000: The network management functions take Unicode strings as input and provide Unicode strings on output. If your application generally works with ANSI strings, care must be taken to convert to and from Unicode where appropriate.

Windows 95/98: Because the system does not support Unicode, the network management functions require ANSI strings.

Parameter Error Reporting

The Add and SetInfo functions return an index for a parameter in error. The caller may pass a NULL pointer for the *parm_err* parameter indicating that the field should not be set by the function. For functions that are accessed through LAN Manager 2.*x* servers, this field is returned as PARM_ERROR_UNKNOWN.

RPC Buffer Allocation Errors

Because the RPC run-time library allocates memory for send and receive buffers, the function should expect RPC allocation errors. In the event of an RPC allocation error, a resumable handle is invalidated. This is a requirement because resumable functions are not rewindable.

Obsolete Information Fields

Many of the information fields in the core information structures will be obsolete. These fields will remain in the information structure for compatibility with 16-bit versions of Windows and will return an intelligent default on 32-bit Windows systems.

Platform Support

The network management functions are implemented on all Win32 platforms. However, there are implementation differences between the platforms. The following sections contain platform-specific information.

- Windows 95/98 support
- Functions that only have support for remoting to LAN Manager 2.*x*
- Requests from 16-bit LAN Manager clients
- Calling 16-bit LAN Manager servers

Windows 95/98 Support

The following 32-bit network management functions are supported by Windows 95 and Windows 98:

- **NetAccessAdd**
- **NetAccessCheck**
- **NetAccessDel**
- **NetAccessEnum**
- **NetAccessGetInfo**
- **NetAccessGetUserPerms**
- **NetAccessSetInfo**
- **NetConnectionEnum**
- **NetFileClose2**
- **NetFileEnum**
- **NetSecurityGetInfo**
- **NetServerGetInfo**
- **NetSessionDel**
- **NetSessionEnum**
- **NetSessionGetInfo**
- **NetShareAdd**
- **NetShareDel**
- **NetShareEnum**
- **NetShareGetInfo**
- **NetShareSetInfo**

You can also thunk to the 16-bit LAN Manager functions on Windows 95/98. For information on these functions, please see the *16-bit LanMan Programmer's Toolkit.* This toolkit is available through the Microsoft Developer Network (MSDN). For information on thunking, see *Thunk Compiler.*

In addition, the following network management structures are supported by Windows 95 and Windows 98.

connection_info_0
connection_info_1
connection_info_50
file_info_50
security_info_1
server_info_1
server_info_50
session_info_0
session_info_1
session_info_2
session_info_10
session_info_50
share_info_0
share_info_1
share_info_2
share_info_50

For more information, see *Windows 95/98 network management code samples.*

Functions That Only Have Support for Remoting to LAN Manager 2.*x*

On Windows NT/Windows 2000, the following functions are supported only for remoting to a LAN Manager 2.*x* computer.

- **NetAccessAdd**
- **NetAccessCheck**
- **NetAccessDel**
- **NetAccessEnum**
- **NetAccessGetUserPerms**
- **NetAccessSetInfo**
- **NetAuditClear**
- **NetAuditRead**
- **NetAuditWrite**
- **NetConfigGet**
- **NetConfigGetAll**
- **NetConfigSet**
- **NetErrorLogClear**
- **NetErrorLogRead**
- **NetErrorLogWrite**

For information on these functions, please see the *16-bit LanMan Programmer's Toolkit*. This toolkit is available through the Microsoft Developer Network (MSDN).

Requests from 16-bit LAN Manager Clients

Windows NT/Windows 2000 provides support for most remote functions called from LAN Manager 2.*x* clients. However, the following function calls are *not* supported when they are accessed with a LAN Manager 2.*x* client to a Windows NT/Windows 2000 server:

- **DosPrintDriverEnum**
- **DosPrintQProcessorEnum**
- **DosPrintPortEnum**
- **DosPrintDest**
- **NetAccessCheck**
- **NetAlertRaise**
- **NetAlertStart**
- **NetAlertStop**
- **NetAuditClear**
- **NetAuditOpen**
- **NetAuditRead**
- **NetAuditWrite**
- **NetConfigGet2**
- **NetConfigGetAll2**
- **NetConfigSet**
- **NetErrorLogOpen**
- **NetErrorLogClear**
- **NetFileClose**
- **NetFileEnum**
- **NetFileGetInfo**
- **NetHandleGetInfo**
- **NetHandleSetInfo**
- **NetMessageFileSend**
- **NetMessageLogFileSet**
- **NetMessageLogFileGet**
- **NetMessageNameFwd**
- **NetMessageNameUnFwd**
- **NetNetBiosEnum**
- **NetNetBiosGetInfo**
- **NetProfileSave**
- **NetProfileLoad**
- **NetServerAdminCommand**
- **NetServerEnum**
- **NetServiceStatus**
- **NetStatisticsGet**
- **NetStatisticsClear**

- **NetUseAdd**
- **NetUseDel**
- **NetUseEnum**
- **NetUseGetInfo**
- **NetUserAdd**
- **NetUserSetInfo**
- **NetUserValidate2**
- **NetWkstaSetUID**

For information on these functions, please see the *16-bit LanMan Programmer's Toolkit.* This toolkit is available through the Microsoft Developer Network (MSDN).

Calling 16-bit LAN Manager Servers

When an RPC-based function fails to connect to the appropriate interface, the client-side stub may attempt to initiate a function request to activate the selected server. For most of the Win32 networking functions specified in this document, and any API where the functionality and data formats are changed only for 32-bit usage, the conversion is straightforward. For components that offer new functionality the caller of the function should generally be aware of the destination type. When the new function offers a superset of the functionality of the LAN Manager 2.*x* station the same function is used for both destinations, but the new function members must have either a reserved value of an associated field to inform the conversion layer the field may be ignored if going LAN Manager 2.*x* systems. This is required so that a function caller is not misled as to the action performed when the function was called.

Security Requirements for the Network Management Functions

Calling some of the network management functions does not require special group membership. Other functions require that users have a specific privilege level to execute successfully. When applicable, the Security Requirements section on a function's reference page indicates the privilege level a user must have to execute the particular function.

The security requirements that apply when you make calls to certain network management functions on Windows 2000 are different than the requirements that apply when you call the functions on Windows NT. The functions include, among others, all those that begin with **NetGroup**, **NetLocalGroup**, and **NetUser**. For a complete list of affected functions, see the end of this topic. For requirements that apply to an individual network management function, please see the function's reference page.

Windows 2000 Active Directory domain controllers: If you call one of the affected functions on a Windows 2000 domain controller running Active Directory, access to a securable object is allowed or denied based on the access-control list (ACL) for the object. (ACLs are specified in the directory.)

For *queries*, the default ACL permits all authenticated users and members of the "Pre-Windows 2000 compatible access" group to view information. For *updates*, the default ACL permits only Administrators and account operators to write information.

Note By default, the "Pre-Windows 2000 compatible access" group includes Everyone as a member. This enables anonymous access (Anonymous Logon) to information if the system allows anonymous access. Administrators can remove Everyone from the "Pre-Windows 2000 Compatible Access" group when installing a domain controller. Removing Everyone from the group restricts information access to authenticated users only.

Anonymous access to securable objects can also be restricted by setting the following key in the registry to the value 1. (This is also referred to as the RestrictAnonymous policy setting.)

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa

Windows 2000 servers and workstations: If you call one of the affected functions on a Windows 2000 member server or workstation to perform a *query*, all authenticated users can view the information. Anonymous access is also possible if the RestrictAnonymous policy setting allows anonymous access. For *updates*, only Administrators and account operators can write information.

The preceding security requirements apply to the following network management *query* functions when you call them on Windows 2000:

NetGroupEnum
NetGroupGetInfo
NetGroupGetUsers
NetLocalGroupEnum
NetLocalGroupGetInfo
NetLocalGroupGetMembers
NetQueryDisplayInformation
NetSessionGetInfo (levels 1 and 2 only)
NetShareEnum (level 2 only)
NetUserEnum
NetUserGetGroups
NetUserGetInfo
NetUserGetLocalGroups
NetUserModalsGet
NetWkstaGetInfo
NetWkstaUserEnum

The security requirements also apply to the following network management *update* functions on Windows 2000:

NetGroupAdd
NetGroupAddUser
NetGroupDel
NetGroupDelUser
NetGroupSetInfo
NetGroupSetUsers
NetLocalGroupAdd
NetLocalGroupAddMembers
NetLocalGroupDel
NetLocalGroupDelMembers
NetLocalGroupSetInfo
NetLocalGroupSetMembers
NetMessageBufferSend
NetUserAdd
NetUserChangePassword
NetUserDel
NetUserModalsSet
NetUserSetGroups
NetUserSetInfo

For more information about the Windows NT/Windows 2000 security model for controlling access to securable objects, see *Access Control*.

Using Network Management

This section discusses how to use the network management functions in your application.

- Looking up a user's full name
- Forcing a user to change the logon password
- Changing elements of user information
- Creating a new computer account
- Creating a local group and adding a user
- Determining the validating server on Windows 95/98
- Looking up text for error code numbers

This section also provides code samples demonstrating use of the 32-bit network management functions that are supported by Windows 95 and Windows 98. (You can find code samples demonstrating the network management functions supported by Windows NT/Windows 2000 on the appropriate reference page.)

- Windows 95/98 network management code samples

Looking Up a User's Full Name

Computers running Windows NT/Windows 2000 can be organized into a domain, which is a collection of computers on a Windows NT Server/Windows 2000 Server network. The domain administrator maintains centralized user and group account information.

To find the full name of a user, given the user name and domain name:

- Convert the user name and domain name to Unicode, if they are not already Unicode strings.
- Look up the computer name of the domain controller (DC) by calling **NetGetDCName**.
- Look up the user name on the DC computer by calling **NetUserGetInfo**.
- Convert the full user name to ANSI, unless the program is expecting to work with Unicode strings.

The following sample code is a function (GetFullName) that takes a user name and a domain name in the first two arguments and returns the user's full name in the third argument.

```
#include <windows.h>
#include <lm.h>
#include <stdio.h>

BOOL GetFullName( char *UserName, char *Domain, char *dest )
{
```

```
    WCHAR wszUserName[256];          // Unicode user name
    WCHAR wszDomain[256];
    LPBYTE ComputerName;

    struct _SERVER_INFO_100 *si100;  // Server structure
    struct _USER_INFO_2 *ui;         // User structure

// Convert ANSI user name and domain to Unicode

    MultiByteToWideChar( CP_ACP, 0, UserName,
        strlen(UserName)+1, wszUserName,
     sizeof(wszUserName)/sizeof(wszUserName[0]) );
    MultiByteTOWideChar( CP_ACP, 0, Domain,
        strlen(Domain)+1, wszDomain, sizeof(wszDomain)/sizeof(wszDomain[0]) );

// Get the computer name of a DC for the domain.

    NetGetDCName( NULL, wszDomain, &ComputerName );

// Look up the user on the DC.

    if( NetUserGetInfo( (LPWSTR) ComputerName,
        (LPWSTR) &wszUserName, 2, (LPBYTE *) &ui ) )
    {
        printf( "Error getting user information.\n" );
        return( FALSE );
    }

// Convert the Unicode full name to ANSI.

    WideCharToMultiByte( CP_ACP, 0, ui->usri2_full_name, -1,
        dest, 256, NULL, NULL );

    return (TRUE);
}
```

Forcing a User to Change the Logon Password

This code sample demonstrates how to force a user to change the logon password on the next logon using the **NetUserGetInfo** and **NetUserSetInfo** functions and the **USER_INFO_3** structure.

Set the **usri3_password_expired** member of the **USER_INFO_3** structure to a nonzero value using the following code fragment.

```
#define UNICODE
#include <windows.h>
#define INCL_NET
#include <lm.h>

#define USERNAME TEXT("your_user_name")
#define SERVER TEXT(\\\\server)


void main( void )
{
        PUSER_INFO_3 pUsr;
        DWORD netRet = 0, dwParmError = 0;
//
// First, retrieve the user information at level 3. This is
//  necessary to prevent resetting other user information when
//  the NetUserSetInfo call is made.
//
NetRet = NetUserGetInfo( SERVER, USERNAME, 3, &pUsr);
 if( netRet == NERR_Success )
 {
//
// The function was successful;
//  set the usri3_password_expired value to a nonzero value.
//  Call the NetUserSetInfo function.
//
pUsr->usri3_password_expired = TRUE;
netRet = NetUserSetInfo( PDC, USERNAME, 3, pUsr, &dwParmError);
//
// A zero return indicates success.
// If the return value is ERROR_INVALID_PARAMETER,
//  the dwParmError parameter will contain a value indicating the
//  invalid parameter within the user_info_3 structure. These values
//  are defined in the lmaccess.h file.
//
if( netRet == NERR_Success )
     printf("User %S will need to change password at next logon", USERNAME);
else printf("Error %d occurred.  Parm Error %d returned.\n", netRet,
dwParmError);
//
// Must free the buffer returned by NetUserGetInfo.
//
NetApiBufferFree( pUsr);
}
else printf("NetUserGetInfo failed: %d\n",netRet);
```

Changing Elements of User Information

The network management functions provide a variety of information levels to assist in changing user information. Some levels require administrative privileges to execute successfully.

The sample code in this topic demonstrates how to change several elements of user information by calling the **NetUserSetInfo** function. The code uses various network management information structures.

When changing user information, it is best to use the specific level for that piece of information. This prevents the accidental resetting of unrelated information when using the lower level values.

Setting some of the more commonly used levels is illustrated in the following code samples:

- Setting the User Password, Level 1003
- Setting the User Privilege, Level 1005
- Setting the User Home Directory, Level 1006
- Setting the User Comment Field, Level 1007
- Setting the User Flags, Level 1008
- Setting the User Script Path, Level 1009
- Setting The User Authority Flags, Level 1010
- Setting The User Full Name, Level 1011

All code fragments assume that the user has defined the UNICODE compile directive and included the appropriate SDK header files, as follows:

```
#define UNICODE
#include <windows.h>
#define INCL_NET
#include <lm.h>
```

Setting the User Password, Level 1003

The following code fragment illustrates how to set a user's password to a known value with a call to the **NetUserSetInfo** function. The **USER_INFO_1003** topic contains additional information.

```
#define PASSWORD TEXT("new_password")
.
.
.
USER_INFO_1003 usriSetPassword;
DWORD netRet = 0;
//
// Set the usri1003_password member to point to a Unicode string.
```

(continued)

(continued)

```
//
// SERVER and USERNAME can be hard-coded TEXT("") strings
//  or pointers to Unicode strings.
//
usriSetPassword.usri1003_password  = PASSWORD;
netRet = NetUserSetInfo( SERVER, USERNAME,  1003, (LPBYTE)&usriSetPassword);
if( netRet == NERR_Success ) printf("Success!\n");
else printf( "ERROR: %d Returned from NetUserSetInfo\n", netRet);
```

Setting the User Privilege, Level 1005

The following code fragment illustrates how to specify the level of privilege assigned to a user with a call to the **NetUserSetInfo** function. The **USER_INFO_1005** topic contains additional information.

```
USER_INFO_1005 usriPriv;
DWORD netRet = 0;
//
// Set the usri1005_priv member to the appropriate value;
//  then call NetUserSetInfo.
//
// SERVER and USERNAME can be hard-coded TEXT("") strings
//  or pointers to Unicode strings.
//
usriPriv.usri1005_priv  = USER_PRIV_USER;
netRet = NetUserSetInfo( SERVER, USERNAME,  1005, (LPBYTE)&usriPriv);
if( netRet == NERR_Success ) printf("Success!\n");
else printf( "ERROR: %d Returned from NetUserSetInfo\n", netRet);
```

Setting the User Home Directory, Level 1006

The following code fragment illustrates how to specify the path of a user's home directory with a call to the **NetUserSetInfo** function. The directory can be a hard-coded path or a valid Unicode path. The **USER_INFO_1006** topic contains additional information.

```
#define HOMEDIR TEXT("C:\\USER\USER_PATH")
USER_INFO_1006 usriHomeDir;
//
// Set the usri1006_home_dir member to point to a valid Unicode string
//  for the new home directory.
//
// SERVER and USERNAME can be hard-coded TEXT("") strings
//  or pointers to Unicode strings.
//
usriHomeDir.usri1006_home_dir  = HOMEDIR;
netRet = NetUserSetInfo( SERVER, USERNAME,  1006, (LPBYTE)&usriHomeDir);
if( netRet == NERR_Success ) printf("Success!\n");
else printf( "ERROR: %d Returned from NetUserSetInfo\n", netRet);
```

Setting the User Comment Field, Level 1007

The following code fragment illustrates how to associate a comment with a user by calling the **NetUserSetInfo** function. The **USER_INFO_1007** topic contains additional information.

```
#define COMMENT TEXT("This is my Comment Text for the user")
USER_INFO_1007 usriComment;
//
// Set the usri1007_comment member to point to
//  a valid Unicode comment string.
//
// SERVER and USERNAME can be hard-coded TEXT("") strings
//  or pointers to Unicode strings.
//
usriComment.usri1007_comment  = COMMENT;
netRet = NetUserSetInfo( SERVER, USERNAME,  1006, (LPBYTE)&usriComment);
if( netRet == NERR_Success ) printf("Success!\n");
else printf( "ERROR: %d Returned from NetUserSetInfo\n", netRet);
```

Setting the User Flags, Level 1008

The following code fragment illustrates how to set user flags with a call to the **NetUserSetInfo** function. The **USER_INFO_1008** topic contains a list of valid values for the flags and a description of each flag.

Note that the UF_SCRIPT flag must be set for Windows NT/Windows 2000 and LAN Manager networks. Trying to set other flags without setting UF_SCRIPT on a Windows NT/Windows 2000 or LAN Manager network will cause the **NetUserSetInfo** function to fail.

```
#define USR_FLAGS UF_SCRIPT | UF_NORMAL_ACCOUNT
USER_INFO_1008 usriFlags;
//
// Set the usri1008_flags member to the appropriate constant.
//
// SERVER and USERNAME can be hard-coded TEXT("") strings
//  or pointers to Unicode strings.
//
usriFlags.usri1008_flags  = USR_FLAGS;
netRet = NetUserSetInfo( SERVER, USERNAME,  1006, (LPBYTE)&usriFlags);
if( netRet == NERR_Success ) printf("Success!\n");
else printf( "ERROR: %d Returned from NetUserSetInfo\n", netRet);
```

Setting the User Script Path, Level 1009

The following code fragment illustrates how to set the path for the logon script file of a particular user with a call to the **NetUserSetInfo** function. The script file can be a .CMD file, an .EXE file, or a .BAT file. The string can also be null. The **USER_INFO_1009** topic contains additional information.

```
#define SCRIPT_PATH "C:\\BIN\\MYSCRIPT.BAT"
USER_INFO_1009 usriScrPath;
//
// Set the usri1009_script_path member to a Unicode
//  string constant or a pointer to a Unicode string.
//
// SERVER and USERNAME can be hard-coded TEXT("") strings
//  or pointers to Unicode strings.
//
usriScrPath.usri1009_script_path  = SCRIPT_PATH;
netRet = NetUserSetInfo( SERVER, USERNAME,  1006, (LPBYTE)&usriScrPath);
if( netRet == NERR_Success ) printf("Success!\n");
else printf( "ERROR: %d Returned from NetUserSetInfo\n", netRet);
```

Setting The User Authority Flags, Level 1010

The following code fragment illustrates how to set the operator privilege flags for a user with a call to the **NetUserSetInfo** function. The **USER_INFO_1010** topic contains a list of valid values for the flags and a description of each flag.

```
#define AUTHORITY_FLAGS AF_OP_ACCOUNTS
USER_INFO_1010 usriAuthFlags;
//
// Set the usri1010_auth_flags member to a constant
//  containing the appropriate flag values.
//
// SERVER and USERNAME can be hard-coded TEXT("") strings
//  or pointers to Unicode strings.
//
usriAuthFlags.usri1010_auth_flags  = SCRIPT_PATH;
netRet = NetUserSetInfo( SERVER, USERNAME,  1006, (LPBYTE)&usriAuthFlags);
if( netRet == NERR_Success ) printf("Success!\n");
else printf( "ERROR: %d Returned from NetUserSetInfo\n", netRet);
```

Setting The User Full Name, Level 1011

The following code fragment illustrates how to set a user's full name with a call to the **NetUserSetInfo** function. The **USER_INFO_1011** topic contains additional information.

```
#define USER_FULL_NAME TEXT("Joe B. User")
USER_INFO_1011 usriFullName;
//
// Set the usri1011_full_name member to a Unicode
//  string constant or a pointer to a Unicode string.
//
// SERVER and USERNAME can be hard-coded TEXT("") strings
//  or pointers to Unicode strings.
//
usriFullName.usri1011_full_name  = USER_FULL_NAME;
```

```
netRet = NetUserSetInfo( SERVER, USERNAME,  1006, (LPBYTE)&usriFullName);
if( netRet == NERR_Success ) printf("Success!\n");
else printf( "ERROR: %d Returned from NetUserSetInfo\n", netRet);
```

Creating a New Computer Account

The following code sample demonstrates how to create a new computer account using the **NetUserAdd** function.

The following are considerations for managing computer accounts:

- The computer account name should be all uppercase for consistency with Windows NT/Windows 2000 account management utilities.
- A computer account name always has a trailing dollar sign ($). Any functions used to manage computer accounts must build the computer name such that the last character of the computer account name is a dollar sign ($). For interdomain trust, the account name is TrustingDomainName$.
- The maximum computer name length is MAX_COMPUTERNAME_LENGTH (15). This length does not include the trailing dollar sign ($).
- The password for a new computer account should be the lowercase representation of the computer account name, without the trailing dollar sign ($). For interdomain trust, the password can be an arbitrary value that matches the value specified on the trust side of the relationship.
- The maximum password length is LM20_PWLEN (14). The password should be truncated to this length if the computer account name exceeds this length.
- The password provided at computer-account-creation time is valid only until the computer account becomes active on the domain. A new password is established during trust relationship activation.

```
#include <windows.h>
#include <lm.h>

BOOL AddMachineAccount(
    LPWSTR wTargetComputer,
    LPWSTR MachineAccount,
    DWORD AccountType
    )
{
    LPWSTR wAccount;
    LPWSTR wPassword;
    USER_INFO_1 ui;
    DWORD cbAccount;
    DWORD cbLength;
    DWORD dwError;
```

(continued)

(continued)

```
//
// Ensure a valid computer account type was passed.
//
if (AccountType != UF_WORKSTATION_TRUST_ACCOUNT &&
    AccountType != UF_SERVER_TRUST_ACCOUNT &&
    AccountType != UF_INTERDOMAIN_TRUST_ACCOUNT
    )
{
    SetLastError(ERROR_INVALID_PARAMETER);
    return FALSE;
}

//
// Obtain number of chars in computer account name.
//
cbLength = cbAccount = lstrlenW(MachineAccount);

//
// Ensure computer name doesn't exceed maximum length.
//
if(cbLength > MAX_COMPUTERNAME_LENGTH) {
    SetLastError(ERROR_INVALID_ACCOUNT_NAME);
    return FALSE;
}

//
// Allocate storage to contain Unicode representation of
// computer account name + trailing $ + NULL.
//
wAccount=(LPWSTR)HeapAlloc(GetProcessHeap(), 0,
    (cbAccount + 1 + 1) * sizeof(WCHAR)  // Account + '$' + NULL
    );

if(wAccount == NULL) return FALSE;

//
// Password is the computer account name converted to lowercase;
//  you will convert the passed MachineAccount in place.
//
wPassword = MachineAccount;

//
// Copy MachineAccount to the wAccount buffer allocated while
//  converting computer account name to uppercase.
//  Convert password (in place) to lowercase.
```

```
//
while(cbAccount--) {
    wAccount[cbAccount] = towupper( MachineAccount[cbAccount] );
    wPassword[cbAccount] = towlower( wPassword[cbAccount] );
}

//
// Computer account names have a trailing Unicode '$'.
//
wAccount[cbLength] = L'$';
wAccount[cbLength + 1] = L'\0'; // terminate the string

//
// If the password is greater than the max allowed, truncate.
//
if(cbLength > LM20_PWLEN) wPassword[LM20_PWLEN] = L'\0';

//
// Initialize the USER_INFO_1 structure.
//
ZeroMemory(&ui, sizeof(ui));

ui.usri1_name = wAccount;
ui.usri1_password = wPassword;

ui.usri1_flags = AccountType | UF_SCRIPT;
ui.usri1_priv = USER_PRIV_USER;

dwError=NetUserAdd(
            wTargetComputer,    // target computer name
            1,                  // info level
            (LPBYTE) &ui,       // buffer
            NULL
            );

//
// Free allocated memory.
//
if(wAccount) HeapFree(GetProcessHeap(), 0, wAccount);

//
// Indicate whether the function was successful.
//
if(dwError == NO_ERROR)
    return TRUE;
```

(continued)

(continued)

```
    else {
        SetLastError(dwError);
        return FALSE;
    }
}
```

The user that calls the account management functions must have Administrator privilege on the target computer. In the case of existing computer accounts, the creator of the account can manage the account, regardless of administrative membership.

The SeMachineAccountPrivilege can be granted on the target computer to give specified users the ability to create computer accounts. This gives non-administrators the ability to create computer accounts. The caller needs to enable this privilege prior to adding the computer account.

Creating a Local Group and Adding a User

Windows NT/Windows 2000 and Windows NT Server/Windows 2000 Server use the same functions that Microsoft LAN Manager uses to create and maintain user and local group-account information. A member of a Users group can create, maintain, and delete accounts in local groups. For example, to create a new local group, call the **NetLocalGroupAdd** function. To add a user to that group, call the **NetLocalGroupAddMembers** function.

The following program allows you to create a user and a local group and add the user to the local group.

```
#define UNICODE 1
#include <windows.h>
#include <lmcons.h>
#include <lmaccess.h>
#include <lmerr.h>
#include <lmapibuf.h>
#include <stdio.h>
#include <stdlib.h>

int _CRTAPI1 main( int cArgs, char *pArgs[] );

NET_API_STATUS NetSample( LPWSTR lpszDomain,
                          LPWSTR lpszUser,
                          LPWSTR lpszPassword,
                          LPWSTR lpszLocalGroup )
{

    USER_INFO_1                user_info;
    LOCALGROUP_INFO_1          localgroup_info;
    LOCALGROUP_MEMBERS_INFO_3 localgroup_members;
```

```
    LPWSTR                      lpszPrimaryDC = NULL;
    NET_API_STATUS              err = 0;
    DWORD                       parm_err = 0;

// First get the name of the primary domain controller.
// Be sure to free the returned buffer.

    err = NetGetDCName( NULL,                       // Local machine
                        lpszDomain,                 // Domain name
                        (LPBYTE *)&lpszPrimaryDC ); // Returned PDC

    if ( err != 0 )
    {
        printf( "Error getting DC name: %d\n", err );
        return( err );
    }

// Set up the USER_INFO_1 structure.

    user_info.usri1_name = lpszUser;
    user_info.usri1_password = lpszPassword;
    user_info.usri1_priv = USER_PRIV_USER;
    user_info.usri1_home_dir = TEXT("");
    user_info.usri1_comment = TEXT("Sample User");
    user_info.usri1_flags = UF_SCRIPT;
    user_info.usri1_script_path = TEXT("");

    err = NetUserAdd( lpszPrimaryDC,        // PDC name
                      1,                    // Level
                      (LPBYTE)&user_info,   // Input buffer
                      &parm_err );          // Parameter in error

    switch ( err )
    {
    case 0:
        printf("user successfully created.\n");
        break;
    case NERR_UserExists:
        printf("user already exists.\n");
        err = 0;
        break;
    case ERROR_INVALID_PARAMETER:
        printf("Invalid Parameter Error adding user:
        Parameter Index = %d\n",
                parm_err);
```

(continued)

(continued)

```
        NetApiBufferFree( lpszPrimaryDC );
        return( err );
    default:
        printf("Error adding user: %d\n", err);
        NetApiBufferFree( lpszPrimaryDC );
        return( err );
    }

// Set up the LOCALGROUP_INFO_1 structure.

    localgroup_info.lgrpi1_name = lpszLocalGroup;
    localgroup_info.lgrpi1_comment = TEXT("Sample Local group.");

    err = NetLocalGroupAdd( lpszPrimaryDC,    // PDC name
                  1,                          // Level
                  (LPBYTE)&localgroup_info,   // Input buffer
                  &parm_err );                // Parameter in error

    switch ( err )
    {
    case 0:
        printf("Local Group successfully created.\n");
        break;
    case ERROR_ALIAS_EXISTS:
        printf("Local Group already exists.\n");
        err = 0;
        break;
    case ERROR_INVALID_PARAMETER:
        printf("Invalid Parameter Error adding Local Group:
        Parameter Index = %d\n",
                err, parm_err);
        NetApiBufferFree( lpszPrimaryDC );
        return( err );
    default:
        printf("Error adding Local Group: %d\n", err);
        NetApiBufferFree( lpszPrimaryDC );
        return( err );
    }

// Now add the user to the local group.

    localgroup_members.lgrmi3_domainandname = lpszUser;

    err = NetLocalGroupAddMembers( lpszPrimaryDC,      // PDC name
                          lpszLocalGroup,              // Group name
```

```
                        3,                      // Name
                        LPBYTE)&localgroup_members, // Buffer
                        1 );                    // Count

    switch ( err )
    {
    case 0:
        printf("User successfully added to Local Group.\n");
        break;
    case ERROR_MEMBER_IN_ALIAS:
        printf("User already in Local Group.\n");
        err = 0;
        break;
    default:
        printf("Error adding User to Local Group: %d\n", err);
        break;
    }

    NetApiBufferFree( lpszPrimaryDC );
    return( err );
}

int _CRTAPI1 main( int    cArgs,
                   char * pArgs[] )
{
    NET_API_STATUS err = 0;

    printf( "Calling NetSample.\n" );
    err = NetSample( TEXT("SampleDomain"),
                     TEXT("SampleUser"),
                     TEXT("SamplePswd"),
                     TEXT("SampleLG") );
    printf( "NetSample returned %d\n", err );
    return( 0 );
}
```

Determining the Validating Server on Windows 95/98

The code sample in this topic demonstrates how to determine the validating server on Windows 95/98, using the network management functions.

Determining the Windows NT/Windows 2000 domain server that validates a user's logon password from Windows 95/98 is an involved task. On Windows NT/Windows 2000, the 32-bit **NetWkstaUserGetInfo** function determines the validating server. The function uses level 1 to return a **WKSTA_USER_INFO_1** structure. The **wkui1_logon_server** member will contain a pointer to a Unicode string specifying the validating server.

On Windows 95/98, there is no 32-bit function that will return the same information. You must use the 16-bit network management functions to retrieve the same information. The functions are exported from NETAPI.DLL. The link libraries are included with the 16-bit version of Microsoft® Visual C++ (version 1.5*x*).

Use the following basic steps to determine the validating server:

1. Determine the user's logon domain using **NetWkstaGetInfo**.
2. Find the primary domain controller (PDC) using **NetGetDCName**.
3. Get the user information from the PDC for comparison to the backup domain controller (BDC) data using **NetUserGetInfo**.
4. Get a list of BDCs using the **NetServerEnum** function.
5. Loop through the list of BDCs, using **NetUserGetInfo** to retrieve the specific user information, comparing each last logon time, searching for the greatest value.

The largest last logon value will be the latest logon time; it will be associated with the last server to validate the user's logon password.

The following short 16-bit program illustrates how to determine the validating server for a Windows 95 user.

You must use the LAN.H and the NETAPI.LIB files distributed with the SDK for the sample to work. The user must be certain that the directories for the .H and .LIB files are in the search path for the project.

Global Variables:

- **Users**. Pointer to the **USER_INFO_11** structure that contains user information about the *username* derived from the Wksta array. The element of interest:
  usi11_last_logon - seconds since Jan 1, 1970. Time that this server last validated this user's logon password.
- **Wksta**. Pointer to the **WKSTA_INFO_10** structure that contains information about the WFW workstation. This structure is filled first and the information placed therein is used to get additional information. The domain name is used to get a list of BDCs that will be queried for user data. The *username* is the element that will qualify the user data request. The information of interest:
  wki10_username - the current logged-on user.
  wki10_logon_domain - the domain the user logged onto.
- **Servers**. List of backup domain controllers for the user's logon domain. Each server in the array will be queried for information about the current logged-on user as described in the user's variable comments.

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
#include <time.h>
#define INCL_NET
#include <lan.h>

// Constant definitions.

#define LEVEL_01 1
#define LEVEL_11 11
#define LEVEL_10 10
#define LEVEL_00 0

#define SMALL_BUFFER 1024
#define MEDIUM_BUFFER 4*1024
#define SERVER_NAME  50

// Buffer allocation. (Just doing it globally for simplicity.)

LPSTR bdcNames[SMALL_BUFFER];
LPSTR UserData[MEDIUM_BUFFER];
LPSTR WrkSta[SMALL_BUFFER];
LPSTR pdcName[SERVER_NAME];
LPSTR servername[SERVER_NAME];

// Create a structure to hold the current server and
//  the logon time so the values are together.

typedef struct svr_usr {
    LPSTR server;
    long logon_time;
} SVR_USR;

// Create typedefs for the larger structure names to
//  shorten the number of characters to type.

typedef struct user_info_11  USER11;
typedef struct wksta_info_10 WORK10;
typedef struct wksta_info_1 WORK01;
typedef struct server_info_100 SERVER100;
typedef struct tm TIMER;

// Declare pointers so the buffers returned from the system
//  functions can be cast to an appropriate value.

TIMER  *lpTime;
USER11  far *Users;
```

(continued)

(continued)

```
WORK10  far *Wksta;
SERVER0 far *Servers;
WORK01  far *Wksta01;

char * lpszTime;

// Declare values for use with the network management functions.

unsigned short svrEntries;
unsigned short svrRead;
unsigned short usrEntries;
unsigned short wkstaEntries;

//  Set up a temporary variable to use as a loop control variable.

unsigned short i;

char Message[256];

//  Define a global variable to hold the validating server
//  and the time the user last logged on.

SVR_USR validate = { NULL, 0 };

//  Global variable for the return value of the function;
//  used in error checking.

DWORD netRet;

// Bogus WinMain so you can step through the example
//  in a 16-bit debugger.

int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;                // current instance
HANDLE hPrevInstance;            // previous instance
LPSTR lpCmdLine;                 // command line
int nCmdShow;                    // show-window type (open/icon)
{
   //  First, get the workstation information calling
   //  the NetWkstaGetInfo function at level 10.
   //  This will return the domain in which the user logged on.

   netRet = NetWkstaGetInfo( NULL,  // This workstation
                         LEVEL_10,  // Information requested
              (char far *)WrkSta,  // Buffer
```

```
                      SMALL_BUFFER,
                     &wkstaEntries); // Expecting only 1 entry

   if( netRet != NERR_Success )
   {
      //  A network error occurred. Print it to the stdout.

      sprintf(Message,"ERROR: NetWkstaGetInfo API Failed. Error Code: %d\n",
netRet);
      MessageBox( NULL, Message, "NetWkstaGetInfo Error", MB_OK);
      return(0);
   }

   //  Now we must retrieve the backup domain controllers
   //  and the primary domain controller to check
   //  when the last user logon was validated.
   //
   //  The argument list is set as follows:
   //  Execute on the local machine ( NULL )
   //  Pass the domain name of interest (Wksta[0].wki10_logon_domain)
   //  Pass a buffer to receive the domain controller name
   //  Pass the size of the DC name buffer

   Wksta = (WORK10 *)WrkSta;
   netRet = NetGetDCName( NULL,
                          Wksta[0].wki10_logon_domain,
                          (char far *)pdcName,
                          SERVER_NAME);
   if(netRet != NERR_Success )
   {
      //  Could not locate a PDC. Something is wrong, end program.

      sprintf(Message,"ERROR: NetGetDCName API Failed. Error Code: %d\n",
netRet);
      MessageBox( NULL, Message, "NetGetDCName Error", MB_OK);
      return(0);
   }

   //  We have the primary domain controller. Now, query for the
   //  user information, then store it temporarily for comparison
   //  to the backup domain controller's data.

   netRet = NetUserGetInfo( (char far *)pdcName,  // Execute on the PDC
                            Wksta[0].wki10_username, // User's name
                            LEVEL_11,
```

(continued)

(continued)

```
                            (char far *)UserData, // Put structs here
                            SMALL_BUFFER,
                            &usrEntries); // expecting only one entry
   if( netRet != NERR_Success )
   {
      //  No user account on the PDC. End the program.

      sprintf(Message,"ERROR: NetUserGetInfo API Failed on PDC. Error Code:
%d\n", netRet);
      MessageBox( NULL, Message, "NetUserGetInfo Error", MB_OK);
      return(0);
   }

   //  Set the structure so the PDC is the starting validating server.

    validate.server =  (char far *)pdcName;
    Users = (USER11 * )UserData;
    validate.logon_time = Users[0].usri11_last_logon;

    // Now retrieve all of the BDCs.

    netRet = NetServerEnum2(  NULL,
                             LEVEL_00,
                             (char far *)bdcNames,
                             SMALL_BUFFER,
                             &svrRead,
                             &svrEntries,
                             SV_TYPE_DOMAIN_BAKCTRL,
                             Wksta[0].wki10_logon_domain);

    if( netRet != NERR_Success )
    {
      //  OOPS, no BDCs. This could be an error.

      sprintf(Message,"ERROR: NetServerEnum2 API Failed. Error Code: %d",
netRet);
      MessageBox( NULL, Message, "NetServerEnum2", MB_OK);
      return( 0 );
   }

   //  Great, we have a list of BDCs .
   //  if svrEntries > 1
   //  Loop through the list checking the last logon against the
   //  current logon stored in validate.logon_time.
```

```
    Servers =( SERVER0 * )bdcNames;
    for( i = 0; i < svrEntries; i++ )
    {
       // Must add the \\ to the names returned from NetServerEnum.

       _fstrcpy( (LPSTR)servername,"\\\\");
       _fstrcat( (LPSTR)servername,Servers[i].sv0_name);
       netRet = NetUserGetInfo( (LPSTR)servername,  // Execute on a BDC
                          Wksta[0].wki10_username,  // User's name
                          LEVEL_11,
                          (char far *)UserData,
                          SMALL_BUFFER,
                          &usrEntries); // expecting only one entry
       Users = (USER11 *)UserData;
       if( netRet == NERR_Success )
       {
          //  Great, we found a user entry on this BDC. Compare the
          //  last logon time to the stored time. Replace the stored
          //  time if the time is greater.
          //
          //  Replace the servername so the time and the server match.

          if( Users[0].usri11_last_logon > validate.logon_time)
          {
             validate.server = (char far *)&Servers[i];
             validate.logon_time = Users[0].usri11_last_logon;
          }
          else if( Users[0].usri11_last_logon == validate.logon_time )
          {
             //  This could indicate a problem.

             sprintf(Message,"Values are the same. %ls %ls", Servers[i].sv0_name,
validate.server);
             MessageBox( NULL, Message, "HMMMM..........",MB_OK);
          }
       }
    }

   //  Convert the time in seconds to a time structure for display
   //  and build the output string.

    lpTime = gmtime( &validate.logon_time);
    lpszTime = asctime( lpTime );
    _fstrcpy((LPSTR)Message, (LPSTR)"Username: ");
    _fstrcat((LPSTR)Message, (LPSTR)Wksta[0].wki10_username);
```

(continued)

(continued)

```
    _fstrcat((LPSTR)Message, (LPSTR)"\nLast Logon: ");
    _fstrcat((LPSTR)Message, (LPSTR)lpszTime);
    _fstrcat((LPSTR)Message, (LPSTR)"Logon Server: ");
    _fstrcat((LPSTR)Message, (LPSTR)validate.server);

    //  Display the information.

    MessageBox(NULL,Message,"Logon Validation Information", MB_OK);
    return(0);
}
```

Looking Up Text for Error Code Numbers

In Windows NT/Windows 2000, it is sometimes necessary to display error text associated with error codes returned from networking-related functions. You may need to perform this task with the network management functions provided by the system.

The error text for these messages is found in the message table file named Netmsg.dll, which is found in %systemroot%\system32. This file contains error messages in the range NERR_BASE (2100) through MAX_NERR(NERR_BASE+899). These error codes are defined in the SDK header file lmerr.h.

The **LoadLibrary** and **LoadLibraryEx** Win32 functions can load Netmsg.dll. The **FormatMessage** Win32 function maps an error code to message text, given a module handle to the Netmsg.dll file.

The following sample illustrates how to display error text associated with network management functions, in addition to displaying error text associated with system-related error codes. If the supplied error number is in a specific range, the netmsg.dll message module is loaded and used to look up the specified error number with the **FormatMessage** function.

```
#include <windows.h>
#include <stdio.h>

#include <lmerr.h>

void
DisplayErrorText(
    DWORD dwLastError
    );

#define RTN_OK 0
#define RTN_USAGE 1
#define RTN_ERROR 13

int
```

```
__cdecl
main(
    int argc,
    char *argv[]
    )
{
    if(argc != 2) {
        fprintf(stderr,"Usage: %s <error number>\n", argv[0]);
        return RTN_USAGE;
    }

    DisplayErrorText( atoi(argv[1]) );

    return RTN_OK;
}

void
DisplayErrorText(
    DWORD dwLastError
    )
{
    HMODULE hModule = NULL; // default to system source
    LPSTR MessageBuffer;
    DWORD dwBufferLength;

    DWORD dwFormatFlags = FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_IGNORE_INSERTS |
        FORMAT_MESSAGE_FROM_SYSTEM ;

    //
    // If dwLastError is in the network range,
    //  load the message source.
    //

    if(dwLastError >= NERR_BASE && dwLastError <= MAX_NERR) {
        hModule = LoadLibraryEx(
            TEXT("netmsg.dll"),
            NULL,
            LOAD_LIBRARY_AS_DATAFILE
            );

        if(hModule != NULL)
            dwFormatFlags |= FORMAT_MESSAGE_FROM_HMODULE;
    }
```

(continued)

(continued)

```
    //
    // Call FormatMessage() to allow for message
    //  text to be acquired from the system
    //  or from the supplied module handle.
    //

    if(dwBufferLength = FormatMessageA(
        dwFormatFlags,
        hModule, // module to get message from (NULL == system)
        dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // default language
        (LPSTR) &MessageBuffer,
        0,
        NULL
        ))
    {
        DWORD dwBytesWritten;

        //
        // Output message string on stderr.
        //
        WriteFile(
            GetStdHandle(STD_ERROR_HANDLE),
            MessageBuffer,
            dwBufferLength,
            &dwBytesWritten,
            NULL
            );

        //
        // Free the buffer allocated by the system.
        //
        LocalFree(MessageBuffer);
    }

    //
    // If we loaded a message source, unload it.
    //
    if(hModule != NULL)
        FreeLibrary(hModule);
}
```

After you compile this program, you can insert the error code number as an argument and the program will display the text. For example:

```
C:\> netmsg 2453
Could not find domain controller for this domain
```

Windows 95/98 Network Management Code Samples

The following code samples demonstrate how to use the 32-bit network management functions that are supported by Windows 95 and Windows 98. (You can find code samples demonstrating the network management functions supported by Windows NT/Windows 2000 on the appropriate reference page.)

- NetConnectionEnum Sample
- NetFileEnum Sample
- NetSecurityGetInfo Sample
- NetServerGetInfo Sample
- NetSessionDel Sample
- NetSessionEnum Sample
- NetSessionGetInfo Sample
- NetShareAdd Sample
- NetShareDel Sample
- NetShareEnum Sample
- NetShareGetInfo Sample
- NetShareSetInfo Sample

NetConnectionEnum Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates how to list the connections made to a shared resource with a call to the **NetConnectionEnum** function.

The sample allocates the memory required to receive 20 **connection_info_50** structures. If this size is inadequate, the sample warns the caller that there are more entries to enumerate. Finally, the sample frees the allocated memory.

```
#include <stdio.h>
#include <assert.h>
#include <windows.h>
#include <svrapi.h>

const short MAX_ENTRIES = 20;

int main(int argc, char FAR * argv[])
{
    char FAR * pszServerName = NULL;
    char FAR * pszQualifier = NULL;
    short nLevel = 50;
    struct connection_info_50* pBuf = NULL;
    struct connection_info_50* pTmpBuf = NULL;
    short cbBuffer;
    short nEntriesRead = 0;
    short nTotalEntries = 0;
    short nTotalCount = 0;
    int i;
    NET_API_STATUS nStatus;
    //
    // ServerName can be NULL to indicate the local computer; ShareName
    //   is required.
```

(continued)

(continued)

```
   //
   if((argc < 2) || (argc > 3))
   {
      printf("Usage: %s [ServerName] ShareName | \\\\ComputerName\n", argv[0]);
      exit(0);
   }

   if(argc == 3)
      pszServerName=argv[1];

   pszQualifier=argv[argc - 1];
   //
   // Allocate the memory required to receive a maximum of
   //  20 connection_info_50 structures.
   //
   cbBuffer = MAX_ENTRIES * sizeof(struct connection_info_50);

  pBuf = malloc(cbBuffer);

   if (pBuf == NULL)
      printf("No memory\n");

   // Call the NetConnectionEnum function to list the
   //  connections, specifying information level 50.
   //
   nStatus = NetConnectionEnum(pszServerName,
                               pszQualifier,
                               nLevel,
                               (char FAR *)pBuf,
                               cbBuffer,
                               &nEntriesRead,
                               &nTotalEntries);
   //
   // Loop through the entries; process errors.
   //
   if ((nStatus == NERR_Success) || (nStatus == ERROR_MORE_DATA))
   {
      if ((pTmpBuf = pBuf) != NULL)
      {
         for (i = 0; (i < nEntriesRead); i++)
         {
            assert(pTmpBuf != NULL);

            if (pTmpBuf == NULL)
            {
```

```
                fprintf(stderr, "An access violation has occurred\n");
                break;
             }
             //
             // Display information for each entry retrieved.
             //
             printf("\n\tNet Name: %s\n", pTmpBuf->coni50_netname);
             printf("\n\tUser Name: %s\n", pTmpBuf->coni50_username);

             pTmpBuf++;
             nTotalCount++;
          }
       }
    }
    else
       fprintf(stderr, "A system error has occurred: %d\n", nStatus);
    //
    // Display a warning if the buffer was not large enough
    //  to contain all available entries.
    //
    if ((nEntriesRead < nTotalEntries) || (nStatus == ERROR_MORE_DATA))
       fprintf(stderr, "Not all entries have been enumerated\n");
    //
    // Free the allocated memory.
    //
    if (pBuf != NULL)
       free(pBuf);

    fprintf(stderr, "\nTotal of %d entries enumerated\n", nTotalCount);

    return 0;
}
```

NetFileEnum Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates how to list the open files on a server with a call to the **NetFileEnum** function.

The sample allocates the memory required to receive 20 **file_info_50** structures. If this size is inadequate, the sample warns the caller that there are more entries to enumerate. Finally, the sample frees the allocated memory.

```
#include <stdio.h>
#include <assert.h>
#include <windows.h>
#include <svrapi.h>
```

(continued)

(continued)

```
const short MAX_ENTRIES = 20;

int main(int argc, char FAR * argv[])
{
    char FAR * pszServerName = NULL;
    short nLevel = 50;
    struct file_info_50* pBuf = NULL;
    struct file_info_50* pTmpBuf = NULL;
    short cbBuffer;
    short nEntriesRead = 0;
    short nTotalEntries = 0;
    short nTotalCount = 0;
    int i;
    NET_API_STATUS nStatus;
    //
    // ServerName can be NULL to indicate the local computer.
    //
    if (argc > 2)
    {
       printf("Usage: %s [\\\\ServerName]\n", argv[0]);
       exit(1);
    }

    if (argc == 2)
       pszServerName = argv[1];
    //
    // Allocate the memory required to receive a maximum of
    //  20 file_info_50 structures.
    //
    cbBuffer = MAX_ENTRIES * sizeof(struct file_info_50);

    pBuf = malloc(cbBuffer);

    if (pBuf == NULL)
       printf("No memory\n");

    // Call the NetFileEnum function to list the
    //  open files, specifying information level 50.
    //
    nStatus = NetFileEnum(pszServerName,
                          NULL,
                          nLevel,
                          (char FAR *)pBuf,
                          cbBuffer,
                          &nEntriesRead,
```

```
                          &nTotalEntries);
//
// Loop through the entries; process errors.
//
if ((nStatus == NERR_Success) || (nStatus == ERROR_MORE_DATA))
{
   if ((pTmpBuf = pBuf) != NULL)
   {
      for (i = 0; (i < nEntriesRead); i++)
      {
         assert(pTmpBuf != NULL);

         if (pTmpBuf == NULL)
         {
            fprintf(stderr, "An access violation has occurred\n");
            break;
         }
         //
         // Display the information for each entry retrieved.
         //
         printf("\tShare: %s\n", pTmpBuf->fi50_sharename);
         printf("\n\tPath: %s\n", pTmpBuf->fi50_pathname);
         printf("\tUser:   %s\n", pTmpBuf->fi50_username);
         printf("\tID:    %d\n", pTmpBuf->fi50_id);

         pTmpBuf++;
         nTotalCount++;
      }
   }
}
else
   fprintf(stderr, "A system error has occurred: %d\n", nStatus);
//
// Display a warning if the buffer was not large enough
//  to contain all available entries.
//
if ((nEntriesRead < nTotalEntries) || (nStatus == ERROR_MORE_DATA))
   fprintf(stderr, "Not all entries have been enumerated\n");
//
// Free the allocated memory.
//
if (pBuf != NULL)
   free(pBuf);

fprintf(stderr, "\nTotal of %d entries enumerated\n", nTotalCount);
```

(continued)

(continued)

```
    return 0;
}
```

NetSecurityGetInfo Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates use of the **NetSecurityGetInfo** function.

The code sample specifies the **security_info_1** information level. The example allocates and frees the memory required for the information buffer.

```
#include <stdio.h>
#include <assert.h>
#include <windows.h>
#include <svrapi.h>

int main(int argc, char FAR * argv[])
{
    char FAR * pszServerName = NULL;
    short nLevel = 1;
    struct security_info_1* pBuf = NULL;
    unsigned short cbBuffer;
    unsigned short nTotalAvail = 0;
    NET_API_STATUS nStatus;
    //
    // ServerName can be NULL to indicate the local computer.
    //
    if (argc > 2)
    {
       printf("Usage: %s [\\\\ServerName]\n", argv[0]);
       exit(1);
    }

    if (argc == 2)
       pszServerName = argv[1];
    //
    // Allocate the memory for the buffer.
    //
    cbBuffer = sizeof(struct security_info_1);

    pBuf = (struct security_info_1*)malloc(cbBuffer);

    if (pBuf == NULL)
       printf("No memory\n");
    //
    // Call the NetSecurityGetInfo function.
```

```
    //  specifying information level 1.
    //
    nStatus = NetSecurityGetInfo(pszServerName,
                                 nLevel,
                                 (char FAR *)pBuf,
                                 cbBuffer,
                                 &nTotalAvail);
    //
    // If the call is successful, display the data.
    //
    if (nStatus == NERR_Success)
    {
       printf("\n\tContainer: %s\n", pBuf->sec1_container);
       printf("\tAddress book server: %s\n", pBuf->sec1_ab_server);
       printf("\tAddress book provider DLL: %s\n", pBuf->sec1_ab_dll);
        if (pBuf->sec1_security == SEC_SECURITY_SHARE)
        printf("\tSecurity: Share-level\n");
        else // SEC_SECURITY_WINNT, SEC_SECURITY_WINNTAS, or SEC_SECURITY_NETWARE
        printf("\tSecurity: User-level\n");
    }
    else
        fprintf(stderr, "A system error has occurred: %d\n", nStatus);
    //
    // Free the allocated memory.
    //
    if (pBuf != NULL)
       free(pBuf);

    return 0;
}
```

NetServerGetInfo Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates how to retrieve a server's configuration information using a call to the **NetServerGetInfo** function.

The sample calls **NetServerGetInfo** once to determine the size of the buffer needed for the returned data. The code allocates memory for the buffer. Then the sample calls **NetServerGetInfo** again to retrieve the data. Finally, the sample displays the information and frees the allocated memory.

```
#include <stdio.h>
#include <assert.h>
#include <windows.h>
#include <svrapi.h>

int main(int argc, char FAR * argv[])
```

(continued)

(continued)

```
{
    char FAR * pszServerName = NULL;
    short nLevel = 50;
    struct server_info_50* pBuf = NULL;
    unsigned short cbBuffer;
    unsigned short nTotalAvail;
    NET_API_STATUS nStatus;
    //
    // ServerName can be NULL to indicate the local computer.
    //
    if (argc > 2)
    {
      printf("Usage: %s [\\\\ServerName]\n", argv[0]);
       exit(1);
    }

    if (argc == 2)
       pszServerName = argv[1];
    //
    // Call NetServerGetInfo once to determine the
    //   total size needed for the buffer.
    //
    cbBuffer = 0;
    nStatus = NetServerGetInfo(pszServerName,
                               nLevel,
                               (char FAR *)pBuf,
                               cbBuffer,
                               &nTotalAvail);

    if (nStatus != NERR_BufTooSmall)
    {
       fprintf(stderr, "A system error has occurred: %d\n", nStatus);
       exit(1);
    }
    //
    // Allocate the memory required for the buffer.
    //
    cbBuffer = nTotalAvail;

    pBuf = malloc(cbBuffer);

    if (pBuf == NULL)
       printf("No memory\n");
    //
    // Call NetServerGetInfo a second time to retrieve the
```

```
    // information, specifying information level 50.
    //
    nStatus = NetServerGetInfo(pszServerName,
                               nLevel,
                               (char FAR *)pBuf,
                               cbBuffer,
                               &nTotalAvail);
    //
    // If the call is successful, display the data.
    //
    if (nStatus == NERR_Success)
    {
       printf("\tName: %s\n", pBuf->sv50_name);
       printf("\tVersion: %d.%d\n", pBuf->sv50_version_major,pBuf-
>sv50_version_minor);
       printf("\tType: 0x%x\n", pBuf->sv50_type);
    }
    else
       fprintf(stderr, "A system error has occurred: %d\n", nStatus);
    //
    // Free the allocated memory.
    //
    if (pBuf != NULL)
       free(pBuf);

    return 0;
}
```

NetSessionDel Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates how to terminate a session with a call to the **NetSessionDel** function.

```
#include <stdio.h>
#include <windows.h>
#include <svrapi.h>

int main(int argc, char FAR * argv[])
{
    DWORD dwError = 0;
    char FAR * pszServerName = NULL;
    char FAR * pszClientName = NULL;
    short nSessionKey = -1;
    NET_API_STATUS nStatus;
    //
    // ServerName can be NULL to indicate the local computer.
```

(continued)

(continued)

```
    //

    if ((argc < 3) || (argc > 4))
    {
       printf("Usage: %s  ClientName SessionKey [\\\\ServerName]\n", argv[0]);
       printf("Note: ClientName is case sensitive\n");
       exit(1);
    }

    pszClientName = argv[1];
    nSessionKey = atoi(argv[2]);

    if (argc == 4)
       pszServerName = argv[3];
    //
    // Call the NetSessionDel function to terminate the session.
    //
    nStatus = NetSessionDel(pszServerName,
                            pszClientName,
                            nSessionKey);
    //
    // Display the result of the function call.
    //
    if (nStatus == NERR_Success)
       fprintf(stderr, "The specified session(s) has been successfully
deleted\n");
    else
       fprintf(stderr, "A system error has occurred: %d\n", nStatus);

    return 0;
}
```

NetSessionEnum Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates how to list the current sessions on a server with a call to the **NetSessionEnum** function.

The sample allocates the memory required to receive 20 **session_info_50** structures. If this size is inadequate, the sample warns the caller that there are more entries to enumerate. Finally, the sample frees the allocated memory.

```
#include <stdio.h>
#include <assert.h>
#include <windows.h>
#include <svrapi.h>

const short MAX_ENTRIES = 20;
```

```
int main(int argc, char FAR * argv[])
{
    char FAR * pszServerName = NULL;
    short nLevel = 50;
    struct session_info_50* pBuf = NULL;
    struct session_info_50* pTmpBuf = NULL;
    short cbBuffer;
    short nEntriesRead = 0;
    short nTotalEntries = 0;
    short nTotalCount = 0;
    int i;
    NET_API_STATUS nStatus;
    //
    // ServerName can be NULL to indicate the local computer.
    //
    if (argc > 2)
    {
       printf("Usage: %s [\\\\ServerName]\n", argv[0]);
       exit(1);
    }

    if (argc == 2)
       pszServerName = argv[1];

    cbBuffer = MAX_ENTRIES * sizeof(struct session_info_50);
    //
    // Allocate the memory required to receive a maximum of
    //  20 session_info_50 structures.
    //
    pBuf = malloc(cbBuffer);

    if (pBuf == NULL)
       printf("No memory\n");

    // Call the NetSessionEnum function to list the
    //  sessions, specifying information level 50.
    //
    nStatus = NetSessionEnum(pszServerName,
                             nLevel,
                             (char FAR *)pBuf,
                             cbBuffer,
                             &nEntriesRead,
                             &nTotalEntries);
    //
```

(continued)

(continued)

```
    // Loop through the entries; process errors.
    //
    if ((nStatus == NERR_Success) || (nStatus == ERROR_MORE_DATA))
    {
        if ((pTmpBuf = pBuf) != NULL)
        {
            for (i = 0; (i < nEntriesRead); i++)
            {
                assert(pTmpBuf != NULL);

                if (pTmpBuf == NULL)
                {
                    fprintf(stderr, "An access violation has occurred\n");
                    break;
                }
                //
                // Display the information for each entry retrieved.
                //
                printf("\n\tClient: %s\n", pTmpBuf->sesi50_cname);
                printf("\tUser:   %s\n", pTmpBuf->sesi50_username);
                printf("\tActive: %d\n", pTmpBuf->sesi50_time);
                printf("\tIdle:   %d\n", pTmpBuf->sesi50_idle_time);
                printf("\tKey:   %d\n", pTmpBuf->sesi50_key);

                pTmpBuf++;
                nTotalCount++;
            }
        }
    }
    else
        fprintf(stderr, "A system error has occurred: %d\n", nStatus);
    //
    // Display a warning if the buffer was not large enough
    //  to contain all available entries.
    //
    if ((nEntriesRead < nTotalEntries) || (nStatus == ERROR_MORE_DATA))
        fprintf(stderr, "Not all entries have been enumerated\n");
    //
    // Free the allocated memory.
    //
    if (pBuf != NULL)
        free(pBuf);

    fprintf(stderr, "\nTotal of %d entries enumerated\n", nTotalCount);

    return 0;
}
```

NetSessionGetInfo Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates how to retrieve information about a particular session using a call to the **NetSessionGetInfo** function.

The sample calls **NetSessionGetInfo** once to determine the size of the buffer needed for the returned data. The code allocates memory for the buffer. Then the sample calls **NetSessionGetInfo** again to retrieve the data. Finally, the sample displays the information and frees the allocated memory.

```
#include <stdio.h>
#include <windows.h>
#include <svrapi.h>

int main(int argc, char FAR * argv[])
{
    char FAR * pszServerName = NULL;
    short nLevel = 50;
    struct session_info_50* pBuf = NULL;
    unsigned short cbBuffer;
    unsigned short nTotalAvail;
    NET_API_STATUS nStatus;
    //
    // ServerName can be NULL to indicate the local computer.
    //
    if ((argc < 2) || (argc > 3))
    {
       printf("Usage: %s [\\\\ServerName] \\\\ClientName\n", argv[0]);
       exit(1);
    }

    if (argc == 3)
       pszServerName = argv[1];
    //
    // Call NetSessionGetInfo once to determine the
    //   total size needed for the buffer.
    //
    cbBuffer = 0;
    nStatus = NetSessionGetInfo(pszServerName,
                                argv[argc-1],
                                nLevel,
                                (char FAR *)pBuf,
                                cbBuffer,
                                &nTotalAvail);

    if (nStatus != NERR_BufTooSmall)
```

(continued)

(continued)

```
    {
       fprintf(stderr, "A system error has occurred: %d\n", nStatus);
       exit(1);
    }
    //
    // Allocate the memory required for the buffer.
    //
    cbBuffer = nTotalAvail;
    pBuf = malloc(cbBuffer);

    if (pBuf == NULL)
       printf("No memory\n");
    //
    // Call NetSessionGetInfo a second time to retrieve the
    //  information, specifying information level 50.
    //
    nStatus = NetSessionGetInfo(pszServerName,
                                argv[argc-1],
                                nLevel,
                                (char FAR *)pBuf,
                                cbBuffer,
                                &nTotalAvail);
    //
    // If the call is successful, display the data.
    //
    if (nStatus == NERR_Success)
    {
       printf("\n\tComputer: %s\n", pBuf->sesi50_cname);
       printf("\tUser: %s\n", pBuf->sesi50_username);
       printf("\tKey: %d\n", pBuf->sesi50_key);
       printf("\t# Connections: %d\n", pBuf->sesi50_num_conns);
       printf("\t# Opens: %d\n", pBuf->sesi50_num_opens);
    }
    else
       fprintf(stderr, "A system error has occurred: %d\n", nStatus);
    //
    // Free the allocated memory.
    //
    if (pBuf != NULL)
       free(pBuf);

    return 0;
}
```

NetShareAdd Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates how to share a resource on the local computer with a call to the **NetShareAdd** function.

The code sample specifies the **share_info_50** structure and no password on the share. The sample also allocates and frees the memory required for the information buffer.

```
#include <stdio.h>
#include <windows.h>
#include <svrapi.h>

int main(int argc, char FAR * argv[])
{
    char FAR * pszServerName = NULL;
    short nLevel = 50;
    struct share_info_50* pBuf = NULL;
    unsigned short cbBuffer;
    NET_API_STATUS nStatus;
    //
    // ServerName can be NULL to indicate the local computer.
    //
    if ((argc < 3) || (argc > 4))
    {
       printf("Usage: %s [\\\\ServerName] ShareName SharePath\n", argv[0]);
       exit(1);
    }

    if (argc == 4)
       pszServerName = argv[1];
    //
    // Allocate the memory required to specify a
    //   share_info_50 structure.
    //
    cbBuffer = sizeof(struct share_info_50);
    pBuf = malloc(cbBuffer);

    if (pBuf == NULL)
       printf("No memory\n");
    //
    // Assign values to the share_info_50 structure.
    //
    strcpy(pBuf->shi50_netname, argv[argc-2]);
    pBuf->shi50_type = STYPE_DISKTREE;
    pBuf->shi50_flags = SHI50F_FULL;
    pBuf->shi50_remark = NULL;
```

(continued)

(continued)

```
    pBuf->shi50_path = argv[argc-1];
    pBuf->shi50_rw_password[0] = '\0'; // No password
    pBuf->shi50_ro_password[0] = '\0'; // No password
    //
    // Call the NetShareAdd function
    //  specifying information level 50.
    //
    nStatus = NetShareAdd(pszServerName,
                          nLevel,
                          (char FAR *)pBuf,
                          cbBuffer);
    //
    // Display the result of the function call.
    //
    if (nStatus == NERR_Success)
       printf("Share added successfully\n");
    else
       fprintf(stderr, "A system error has occurred: %d\n", nStatus);
    //
    // Free the allocated memory.
    //
    if (pBuf != NULL)
       free(pBuf);

    return 0;
}
```

NetShareDel Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates how to delete a share with a call to the **NetShareDel** function.

```
#include <stdio.h>
#include <windows.h>
#include <svrapi.h>

int main(int argc, char FAR * argv[])
{
    char FAR * pszServerName = NULL;
    NET_API_STATUS nStatus;
    //
    // ServerName can be NULL to indicate the local computer.
    //
    if ((argc < 2) || (argc > 3))
    {
       printf("Usage: %s [\\\\ServerName] ShareName\n", argv[0]);
```

```
        exit(1);
    }

    if (argc == 3)
        pszServerName = argv[1];
    //
    // Call the NetShareDel function to delete the share.
    //
    nStatus = NetShareDel(pszServerName,
                          argv[argc-1],
                               0);
    //
    // Display the result of the function call.
    //
    if (nStatus == NERR_Success)
        printf("Share deleted successfully\n");
    else
        fprintf(stderr, "A system error has occurred: %d\n", nStatus);

    return 0;
}
```

NetShareEnum Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates how to list information about each shared resource on a server with a call to the **NetShareEnum** function.

The sample allocates the memory required to receive 20 **share_info_50** structures. If this size is inadequate, the sample warns the caller that there are more entries to enumerate. Finally, the sample frees the allocated memory.

```
#include <stdio.h>
#include <assert.h>
#include <windows.h>
#include <svrapi.h>

const short MAX_ENTRIES = 20;

int main(int argc, char FAR * argv[])
{
    char FAR * pszServerName = NULL;
    short nLevel = 50;
    struct share_info_50* pBuf = NULL;
    struct share_info_50* pTmpBuf = NULL;
    short cbBuffer;
    short nEntriesRead = 0;
    short nTotalEntries = 0;
```

(continued)

(continued)

```
short nTotalCount = 0;
int i;
NET_API_STATUS nStatus;
//
// ServerName can be NULL to indicate the local computer.
//
if (argc > 2)
{
   printf("Usage: %s [\\\\ServerName]\n", argv[0]);
   exit(1);
}

if (argc == 2)
   pszServerName = argv[1];
//
// Allocate the memory required to receive a maximum of
//  20 share_info_50 structures.
//
cbBuffer = MAX_ENTRIES * sizeof(struct share_info_50);

pBuf = malloc(cbBuffer);

if (pBuf == NULL)
   printf("No memory\n");
//
// Call the NetShareEnum function to list the
//  shares, specifying information level 50.
//
nStatus = NetShareEnum(pszServerName,
                       nLevel,
                       (char FAR *)pBuf,
                       cbBuffer,
                       &nEntriesRead,
                       &nTotalEntries);
//
// Loop through the entries; process errors.
//
if ((nStatus == NERR_Success) || (nStatus == ERROR_MORE_DATA))
{
   if ((pTmpBuf = pBuf) != NULL)
   {
      for (i = 0; (i < nEntriesRead); i++)
      {
         assert(pTmpBuf != NULL);
```

```
            if (pTmpBuf == NULL)
            {
               fprintf(stderr, "An access violation has occurred\n");
               break;
            }
            //
            // Display the information for each entry retrieved.
            //
            printf("\n\tShare: %s\n", pTmpBuf->shi50_netname);
            printf("\tPath: %s\n", pTmpBuf->shi50_path);

            pTmpBuf++;
            nTotalCount++;
         }
      }
   }
   else
      fprintf(stderr, "A system error has occurred: %d\n", nStatus);
   //
   // Display a warning if the buffer was not large enough
   //  to contain all available entries.
   //
   if ((nEntriesRead < nTotalEntries) || (nStatus == ERROR_MORE_DATA))
      fprintf(stderr, "Not all entries have been enumerated\n");
   //
   // Free the allocated memory.
   //
   if (pBuf != NULL)
      free(pBuf);

   fprintf(stderr, "\nTotal of %d entries enumerated\n", nTotalCount);

   return 0;
}
```

NetShareGetInfo Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates how to retrieve information about a shared resource with a call to the **NetShareGetInfo** function.

The sample calls **NetShareGetInfo** once to determine the size of the buffer needed for the returned data. The code allocates memory for the buffer. Then the sample calls **NetShareGetInfo** again to retrieve the data. Finally, the sample displays the information and frees the allocated memory.

```
#include <stdio.h>
#include <assert.h>
```

(continued)

(continued)

```
#include <windows.h>
#include <svrapi.h>

int main(int argc, char FAR * argv[])
{
   char FAR * pszServerName = NULL;
   char FAR * pszNetName = NULL;
   short nLevel = 50;
   struct share_info_50* pBuf = NULL;
   unsigned short cbBuffer;
   unsigned short nTotalAvail;
   NET_API_STATUS nStatus;
   //
   // ServerName can be NULL to indicate the local computer.
   //
   if ((argc < 2) || (argc > 3))
   {
      printf("Usage: %s [\\\\ServerName] ShareName\n", argv[0]);
      exit(1);
   }

   if (argc == 3)
      pszServerName = argv[1];
   //
   // Note that for a Win9x peer server,
   //   the share name needs to be uppercase.
   //
   pszNetName = argv[argc-1];
   //
   // Call NetShareGetInfo once to determine the
   //   total size needed for the buffer.
   //
   cbBuffer = 0;
   nStatus = NetShareGetInfo(pszServerName,
                             pszNetName,
                             nLevel,
                             (char FAR *)pBuf,
                             cbBuffer,
                             &nTotalAvail);

   if (nStatus != NERR_BufTooSmall)
   {
      fprintf(stderr, "A system error has occurred: %d\n", nStatus);
      exit(1);
   }
```

```
    //
    // Allocate the memory required for the buffer.
    //
    cbBuffer = nTotalAvail;
    pBuf = malloc(cbBuffer);

    if (pBuf == NULL)
       printf("No memory\n");
    //
    // Call NetShareGetInfo a second time to retrieve the
    //  information, specifying information level 50.
    //
    nStatus = NetShareGetInfo(pszServerName,
                              pszNetName,
                              nLevel,
                              (char FAR *)pBuf,
                              cbBuffer,
                              &nTotalAvail);
    //
    // If the call is successful, display the data.
    //
    if (nStatus == NERR_Success)
       printf("\n\tPath: %s\n", pBuf->shi50_path);
    else
       fprintf(stderr, "A system error has occurred: %d\n", nStatus);
    //
    // Free the allocated memory.
    //
    if (pBuf != NULL)
       free(pBuf);

    return 0;
}
```

NetShareSetInfo Sample (Windows 95/98)

Windows 95/98: The following code sample demonstrates how to change the parameters associated with a network share using a call to the **NetShareSetInfo** function.

The sample uses the following basic steps to change the remark associated with a network share:

1. Determine the size of the buffer needed to receive share information by calling the **NetShareGetInfo** function.
2. Retrieve share information by calling the **NetShareGetInfo** function a second time, specifying information level 50.

3. Copy the returned information to a second buffer with a call to the **CopyMemory** function. (This is necessary to prevent resetting share information other than the comment.)
4. Modify the remark associated with the share by calling the **NetShareSetInfo** function.

The code also allocates and deallocates the memory required for both buffers.

```
#include <stdio.h>
#include <windows.h>
#include <svrapi.h>

int main(int argc, char FAR * argv[])
{
    char FAR * pszServerName = NULL;
    char FAR * pszNetName = NULL;
    short nLevel = 50;
    struct share_info_50* pBuf = NULL;
    struct share_info_50* pBufNew = NULL;
    unsigned short cbBuffer;
    unsigned short nTotalAvail;
    NET_API_STATUS nStatus;
    //
    // ServerName can be NULL to indicate the local computer.
    //
    if ((argc < 3) || (argc > 4))
    {
       printf("Usage: %s [\\\\ServerName] ShareName Comment\n", argv[0]);
       exit(1);
    }

    if (argc == 4)
       pszServerName = argv[1];

    // Note that for Win9x peer servers,
    //   the share name needs to be uppercase.
    //
    pszNetName = argv[argc-2];

    // Call the NetShareGetInfo function once to determine the
    //   total size needed for the first buffer.
    cbBuffer = 0;
    nStatus = NetShareGetInfo(pszServerName,
                              pszNetName,
                              nLevel,
                              (char FAR *)pBuf,
```

```
                          cbBuffer,
                          &nTotalAvail);
//
// Allocate the memory required for the first buffer.
//
cbBuffer = nTotalAvail;

pBuf = malloc(cbBuffer);

if (pBuf == NULL)
   printf("No memory\n");
//
// Call NetShareGetInfo a second time to retrieve the
//  information, specifying information level 50.
//
nStatus = NetShareGetInfo(pszServerName,
                          pszNetName,
                          nLevel,
                          (char FAR *)pBuf,
                          cbBuffer,
                          &nTotalAvail);
//
// If the call succeeds, allocate the memory required
//   for a second buffer.
//
if (nStatus == NERR_Success)
{
   pBufNew = malloc(cbBuffer);

   if (pBufNew == NULL)
      printf("No memory\n");
  //
  // Copy the first buffer to the second buffer.
  //   This is necessary to prevent resetting other share
  //   information when you call NetShareSetInfo.
  //
  CopyMemory(pBufNew, pBuf, cbBuffer);
  //
  // Assign a new value to the comment
  //  associated with the share.
  //
  strcpy(pBufNew->shi50_remark, argv[argc-1]);
   //
   // Call NetShareSetInfo to make changes, specifying
   //   PARMNUM_ALL to reset all information.
```

(continued)

(continued)

```
        //
        nStatus = NetShareSetInfo(pszServerName,
                                  pszNetName,
                                  nLevel,
                                  (char FAR *)pBufNew,
                                  cbBuffer,
                                  PARMNUM_ALL);
        //
        // Process errors and
        //  free the memory allocated for the second buffer.
        //
        if (nStatus != NERR_Success)
            fprintf(stderr, "A system error has occurred (NetShareSetInfo): %d\n",
nStatus);
        if (pBufNew != NULL)
           free(pBufNew);
    }
    else
        fprintf(stderr, "A system error has occurred (NetShareGetInfo): %d\n",
nStatus);
    //
    // Free the memory allocated for the first buffer.
    //
    if (pBuf != NULL)
        free(pBuf);

    return 0;
}
```

CHAPTER 17

Network Management Reference

Due to the constraints associated with putting network development reference into printed form, the network management reference information isn't provided in its entirety (it's over 550 pages by itself!). However, I've provided a grouping of network management functions in the following sections, and the DVD that accompanies this Network Services Library of course has the entire body of network management reference information as just part of its extensive information.

Network Management Functions

The network management functions can be grouped as follows:

Alert Functions

NetAlertRaise
NetAlertRaiseEx

API Buffer Functions

NetApiBufferAllocate
NetApiBufferFree
NetApiBufferReallocate
NetApiBufferSize

Directory Service Functions

NetGetJoinableOUs
NetGetJoinInformation
NetJoinDomain
NetRenameMachineInDomain
NetUnjoinDomain
NetValidateName

Distributed File System (Dfs) Functions

NetDfsAdd
NetDfsAddFtRoot
NetDfsAddStdRoot
NetDfsAddStdRootForced
NetDfsEnum
NetDfsGetClientInfo
NetDfsGetInfo
NetDfsManagerInitialize
NetDfsRemove
NetDfsRemoveFtRoot
NetDfsRemoveFtRootForced
NetDfsRemoveStdRoot
NetDfsSetClientInfo
NetDfsSetInfo

Get Functions

NetGetAnyDCName
NetGetDCName
NetGetDisplayInformationIndex
NetQueryDisplayInformation

Group Functions

NetGroupAdd
NetGroupAddUser
NetGroupDel
NetGroupDelUser
NetGroupEnum
NetGroupGetInfo
NetGroupGetUsers
NetGroupSetInfo
NetGroupSetUsers

Local Group Functions

NetLocalGroupAdd
NetLocalGroupAddMembers
NetLocalGroupDel
NetLocalGroupDelMembers
NetLocalGroupEnum
NetLocalGroupGetInfo
NetLocalGroupGetMembers
NetLocalGroupSetInfo
NetLocalGroupSetMembers

Message Functions

NetMessageBufferSend
NetMessageNameAdd
NetMessageNameDel
NetMessageNameEnum
NetMessageNameGetInfo

NetFile Functions

NetFileClose
NetFileClose2
NetFileEnum
NetFileGetInfo

Remote Utility Functions

NetRemoteComputerSupports
NetRemoteTOD

Replicator Functions

NetReplExportDirAdd
NetReplExportDirDel
NetReplExportDirEnum
NetReplExportDirGetInfo
NetReplExportDirLock
NetReplExportDirSetInfo
NetReplExportDirUnlock
NetReplGetInfo
NetReplImportDirAdd
NetReplImportDirDel
NetReplImportDirEnum
NetReplImportDirGetInfo
NetReplImportDirLock
NetReplImportDirUnlock
NetReplSetInfo

Schedule Functions

NetScheduleJobAdd
NetScheduleJobDel
NetScheduleJobEnum
NetScheduleJobGetInfo

Server Functions

NetServerDiskEnum
NetServerEnum
NetServerGetInfo
NetServerSetInfo

Server and Workstation Transport Functions

NetServerComputerNameAdd
NetServerComputerNameDel
NetServerTransportAdd
NetServerTransportAddEx
NetServerTransportDel
NetServerTransportEnum
NetWkstaTransportAdd
NetWkstaTransportDel
NetWkstaTransportEnum

Session Functions

NetSessionDel
NetSessionEnum
NetSessionGetInfo

Share Functions

NetConnectionEnum
NetShareAdd
NetShareCheck
NetShareDel
NetShareEnum
NetShareGetInfo
NetShareSetInfo

Statistics Function

NetStatisticsGet

Use Functions

NetUseAdd
NetUseDel
NetUseEnum
NetUseGetInfo

User Functions

NetUserAdd
NetUserChangePassword
NetUserDel
NetUserEnum
NetUserGetGroups
NetUserGetInfo
NetUserGetLocalGroups
NetUserSetGroups
NetUserSetInfo

User Modals Functions

NetUserModalsGet
NetUserModalsSet

Workstation and Workstation User Functions

NetWkstaGetInfo
NetWkstaSetInfo
NetWkstaUserGetInfo
NetWkstaUserSetInfo
NetWkstaUserEnum

Access and Security Functions (Windows 95/98 only)

NetAccessAdd
NetAccessCheck
NetAccessDel
NetAccessEnum
NetAccessGetInfo
NetAccessGetUserPerms
NetAccessSetInfo
NetSecurityGetInfo

Obsolete Functions

NetAuditClear
NetAuditRead
NetAuditWrite
NetConfigGet
NetConfigGetAll
NetConfigSet
NetErrorLogClear
NetErrorLogRead
NetErrorLogWrite
NetLocalGroupAddMember
NetLocalGroupDelMember
NetServiceControl
NetServiceEnum
NetServiceGetInfo
NetServiceInstall

Network Management Structures

The network management structures can be grouped as follows:

Alert Structures

STD_ALERT
ADMIN_OTHER_INFO
ERRLOG_OTHER_INFO
PRINT_OTHER_INFO
USER_OTHER_INFO

Distributed File System (Dfs) Structures

DFS_INFO_1
DFS_INFO_2
DFS_INFO_3
DFS_INFO_4
DFS_INFO_100
DFS_INFO_101
DFS_INFO_102
DFS_INFO_200
DFS_STORAGE_INFO

File Structures

FILE_INFO_2
FILE_INFO_3

Get Structures

NET_DISPLAY_GROUP
NET_DISPLAY_MACHINE
NET_DISPLAY_USER

Group Structures

GROUP_INFO_0
GROUP_INFO_1
GROUP_INFO_2
GROUP_INFO_1002
GROUP_INFO_1005
GROUP_USERS_INFO_0

Local Group Structures

LOCALGROUP_INFO_0
LOCALGROUP_INFO_1
LOCALGROUP_INFO_1002
LOCALGROUP_MEMBERS_INFO_0
LOCALGROUP_MEMBERS_INFO_1
LOCALGROUP_MEMBERS_INFO_2
LOCALGROUP_MEMBERS_INFO_3
LOCALGROUP_USERS_INFO_0

Message Structures

MSG_INFO_0
MSG_INFO_1

Remote Utility Structure

TIME_OF_DAY_INFO

Replicator Structures

REPL_EDIR_INFO_0
REPL_EDIR_INFO_1
REPL_EDIR_INFO_2
REPL_EDIR_INFO_1000
REPL_EDIR_INFO_1001
REPL_IDIR_INFO_0
REPL_IDIR_INFO_1
REPL_INFO_0
REPL_INFO_1000
REPL_INFO_1001
REPL_INFO_1002
REPL_INFO_1003

Schedule Structures

AT_ENUM
AT_INFO

Server Structures

SERVER_INFO_100
SERVER_INFO_101
SERVER_INFO_102
SERVER_INFO_402
SERVER_INFO_403
SERVER_INFO_1005
SERVER_INFO_1010
SERVER_INFO_1016
SERVER_INFO_1017
SERVER_INFO_1018
SERVER_INFO_1107
SERVER_INFO_1501
SERVER_INFO_1502
SERVER_INFO_1503
SERVER_INFO_1506
SERVER_INFO_1509
SERVER_INFO_1510
SERVER_INFO_1511
SERVER_INFO_1512
SERVER_INFO_1513
SERVER_INFO_1515
SERVER_INFO_1516
SERVER_INFO_1518
SERVER_INFO_1523
SERVER_INFO_1528
SERVER_INFO_1529
SERVER_INFO_1530
SERVER_INFO_1533
SERVER_INFO_1536
SERVER_INFO_1538
SERVER_INFO_1539
SERVER_INFO_1540
SERVER_INFO_1541
SERVER_INFO_1542
SERVER_INFO_1544
SERVER_INFO_1550
SERVER_INFO_1552

Server and Workstation Transport Structures

SERVER_TRANSPORT_INFO_0
SERVER_TRANSPORT_INFO_1
SERVER_TRANSPORT_INFO_2
SERVER_TRANSPORT_INFO_3
WKSTA_TRANSPORT_INFO_0

Session Structures

SESSION_INFO_0
SESSION_INFO_1
SESSION_INFO_2
SESSION_INFO_10
SESSION_INFO_502

Share Structures

CONNECTION_INFO_0
CONNECTION_INFO_1
SHARE_INFO_0
SHARE_INFO_1
SHARE_INFO_2
SHARE_INFO_501
SHARE_INFO_502
SHARE_INFO_1004
SHARE_INFO_1005
SHARE_INFO_1006
SHARE_INFO_1501

Statistics Structures

STAT_SERVER_0
STAT_WORKSTATION_0

Use Structures

USE_INFO_0
USE_INFO_1
USE_INFO_2

User Structures

USER_INFO_0
USER_INFO_1
USER_INFO_2
USER_INFO_3
USER_INFO_10
USER_INFO_11
USER_INFO_20
USER_INFO_21
USER_INFO_22
USER_INFO_1003
USER_INFO_1005
USER_INFO_1006
USER_INFO_1007
USER_INFO_1008
USER_INFO_1009
USER_INFO_1010
USER_INFO_1011
USER_INFO_1012
USER_INFO_1013
USER_INFO_1014
USER_INFO_1017
USER_INFO_1018
USER_INFO_1020
USER_INFO_1023
USER_INFO_1024
USER_INFO_1025
USER_INFO_1051
USER_INFO_1052
USER_INFO_1053

User Modals Structures

USER_MODALS_INFO_0
USER_MODALS_INFO_1
USER_MODALS_INFO_2
USER_MODALS_INFO_3
USER_MODALS_INFO_1001
USER_MODALS_INFO_1002
USER_MODALS_INFO_1003
USER_MODALS_INFO_1004
USER_MODALS_INFO_1005
USER_MODALS_INFO_1006
USER_MODALS_INFO_1007

Workstation and Workstation User Structures

WKSTA_INFO_100
WKSTA_INFO_101
WKSTA_INFO_102
WKSTA_USER_INFO_0
WKSTA_USER_INFO_1
WKSTA_USER_INFO_1101

In addition, the following network management structures are supported by Windows 95 and Windows 98.

Windows 95/98 Structures

access_info_0
access_info_1
access_info_2
access_info_12
access_list
connection_info_0
connection_info_1
connection_info_50
file_info_50
security_info_1
server_info_1
server_info_50
session_info_0
session_info_1
session_info_2
session_info_10
session_info_50
share_info_0
share_info_1
share_info_2
share_info_50

Network Management Macros

The following macros can be used with network management alert data buffers:

ALERT_OTHER_INFO
ALERT_VAR_DATA

Mapping ADSI Interfaces to the Network Management Functions

The Active Directory™ Service Interfaces (ADSI) are a set of COM interfaces used to access the capabilities of directory services from different network providers. ADSI presents a single set of directory service interfaces for managing network resources in a distributed computing environment.

If you are programming for Active Directory, you may be able to call certain ADSI interface methods to achieve the same functionality you can achieve by calling certain network management functions.

The following table lists network management functions and function groups, and the ADSI interfaces to which the functions map.

Functions	Interfaces
NetFileEnum, **NetFileGetInfo**	**IADsResource**, **IADsFileServiceOperations**
NetGroup*	**IADsGroup**
NetLocalGroup*	**IADsGroup**
NetServer*	**IADsComputer**
NetSession*	**IADsSession**, **IADsFileServiceOperations**
NetShare*	**IADsFileShare**
NetUser*	**IADsUser**, **IADsComputer**
NetUserModals*	**IADsDomain**

For more information about directory services and programming with ADSI, see ***Active Directory Developer's Reference Library***, also available from Microsoft Press. Information about the custom properties the WinNT provider makes available for the **User** class, and the property methods of the **IADsUser** interface the WinNT provider does not support, are also provided in the ***Active Directory Developer's Reference Library***.

INDEX

Networking Services Programming Elements – Alphabetical Listing

This final part, found in each volume in the Networking Services Library, provides a comprehensive programming element index that has been designed to make your life easier.

Rather than cluttering the TOCs of each individual volume in this library with the names of programming elements, I've relegated such per-element information to a central location: the back of each volume. This index points you to the volume that has the information you need, and organizes the information in a way that lends itself to easy use.

Also, to keep you as informed and up-to-date as possible about Microsoft technologies, I've created (and maintain) a live Web-based document that maps Microsoft technologies to the locations where you can get more information about them. The following link gets you to the live index of technologies:

www.iseminger.com/winprs/technologies

The format of this index is in a constant state of improvement. I've designed it to be as useful as possible, but the real test comes when you put it to use. If you can think of ways to make improvements, send me feedback at *winprs@microsoft.com*. While I can't guarantee a reply, I'll read the input, and if others can benefit, I will incorporate the idea into future libraries.

Locators are arranged by Volume Number followed by Page Number.

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

Y

Network Protocols and Interfaces

This essential reference book is part of the five-volume NETWORKING SERVICES DEVELOPER'S REFERENCE LIBRARY. In its printed form, this material is portable, easy to use, and easy to browse—a highly condensed, completely indexed, intelligently organized complement to the information available on line and through the Microsoft Developer Network (MSDN™). Each book includes an overview of the five-volume library, an appendix of programming elements, an index of referenced Microsoft® technologies, and tips on how and where to find other Microsoft developer reference resources you may need.

Network Protocols and Interfaces

This volume provides concise reference materials about how to use important Windows® network interfaces, protocols, and services. It discusses the Domain Name System (DNS), the Dynamic Host Configuration Protocol (DHCP), the Multicast Address Dynamic Client Allocation Protocol (MADCAP), Network Basic Input/Output System (NetBIOS), the Simple Network Management Protocol (SNMP) and the WinSNMP API, the Internet Protocol Helper (IP Helper), the Synchronization Manager, the System Event Notification Service (SENS), and the Internet Authentication Service (IAS).

Part No. 097-0002784