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

Routing

BASED ON
msdn™ library

Microsoft®

David Iseminger
Series Editor

Routing

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.

Acquisitions Editor: Ben Ryan
Project Editor: Wendy Zucker

Part No. 097-0002787

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

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 5 of the *Networking Services Developer's Reference Library* is a complete treatment of the routing capabilities built into RRAS.

The routing components of RRAS make it possible for a computer running Windows NT Server 4.0 or Windows 2000 Server to function as a network router. (RRAS also provides the next generation of server functionality for the Remote Access Service (RAS) for Windows. See Volume 4 for more information about the remote access capabilities of RRAS.)

This volume also has 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 get 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...because that useful developer resource information is included in this library, as well.
- 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 refer to 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 steered this library's content and choices of included technologies; I hope you find its information is useful, portable, a good value, and as accessible as it can be.

Part 2 of this volume provides the following routing information.

Router Administration

The router administration API enables developers to create applications that manage the router service on a computer running Microsoft Windows 2000 or running Microsoft Windows NT 4.0 with the RRAS add-on installed. (Note that not all API functions are supported on both of these platforms.) The following topics are covered in this volume, and provide detailed information about router administration:

- Components of the Router Architecture
- Router Initialization
- Router Management Functions
- Router Interface Functions
- Router Manager (**Transport**) Functions
- Router Manager Client (**InterfaceTransport**) Functions
- **MprInfo** Functions and Information Headers
- Managing Router Clients and Interfaces

Message Information Base (MIB)

The routing capabilities built into RRAS include the Management Information Base (MIB) API, which makes it possible to query and set the values of MIB variables exported by one of the router managers or any of the routing protocols that the router managers service. By using this API, the router supports the Simple Network Management Protocol (SNMP).

Packet Filtering

Packet Filtering enables the developer to create and manage input and output filters for IP packets. Each IP adapter interface can be associated with one or more filters. Filters can include source and destination addresses, address mask and port, and protocol identifiers.

Routing Protocol Interface

This chapter that describes the Routing Protocol Interface and explains how the integration of third-party routing protocols into RRAS is possible. RRAS defines the interface between the router manager and the Dynamic-Link Library (DLL) for routing protocols, and exposes that interface through routing protocol interface programming capabilities.

Routing Table Manager (RTM) v1

The Routing Table Manager (RTM) is a central repository of routing information for all routing protocols that operate under RRAS. The RTM provides routing information to all interested components, such as routing protocols, management agents, and monitoring agents. The RTM also determines the best route to each destination network known to the routing protocols. It determines this route based on routing protocol priorities and on metrics associated with the routes, then passes the best-route information on to the forwarders and back to the routing protocols.

Routing Table Manager (RTM) v2

The Routing Table Manager Version 2 (RTMv2) API is a feature of Windows 2000 that you can use to write routing protocols that interact with the routing table managers. RTMv2 is not available for Windows NT 4.0. Additionally, RTMv2 cannot be used for IPX routing protocols that run on Windows NT 4.0 or Windows 2000. If you are using IPX or writing routing protocols for Windows NT 4.0, you must use the Routing Table Manager Version 1 (RTMv1) API.

Multicast Group Manager

The Multicast Group Manager (MGM) API enables developers to use the multicast routing capabilities of Windows 2000 Server. Developers can write routing protocols that join and leave multicast groups, as well as administrative applications that track group membership. Routing protocol developers can use MGM to develop callback functions to communicate group membership information directly to the routing protocol.

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.

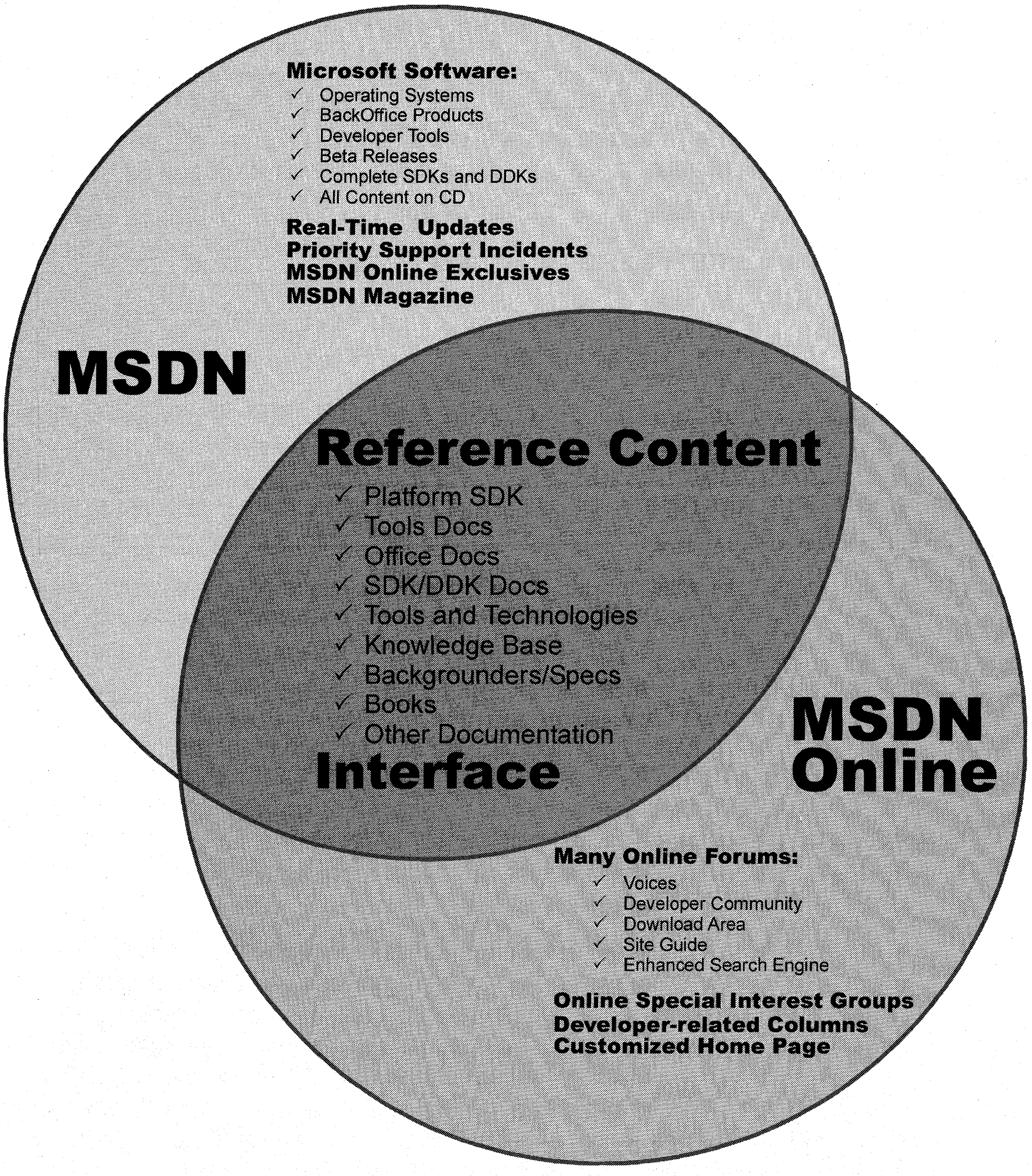


Figure 3-1: The similarities and differences in coverage between MSDN and MSDN Online.

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

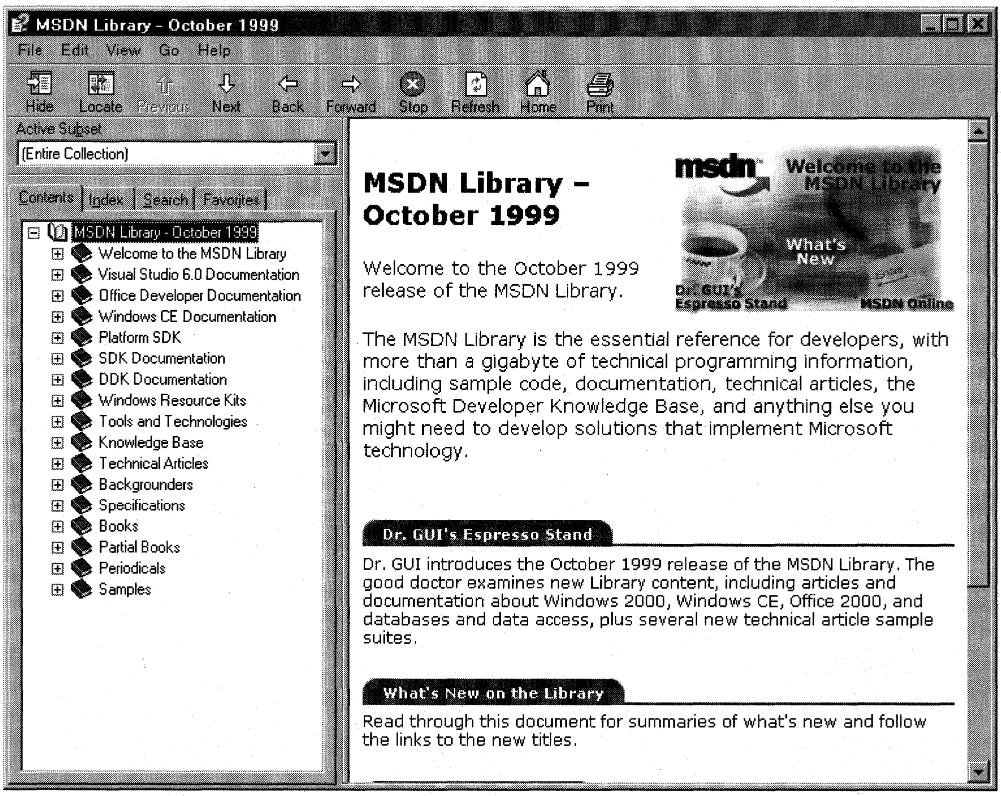
MSDN subscriptions come with an installable interface, and the Professional and Universal subscriptions also come with a bunch of Microsoft product software such as Windows platform versions and BackOffice applications. There's no need to tell you how to use Microsoft product software, but there's a 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.



MSDN Library – October 1999

Welcome to the October 1999 release of the MSDN Library.

The MSDN Library is the essential reference for developers, with more than a gigabyte of technical programming information, including sample code, documentation, technical articles, the Microsoft Developer Knowledge Base, and anything else you might need to develop solutions that implement Microsoft technology.

Dr. GUI's Espresso Stand

Dr. GUI introduces the October 1999 release of the MSDN Library. The good doctor examines new Library content, including articles and documentation about Windows 2000, Windows CE, Office 2000, and databases and data access, plus several new technical article sample suites.

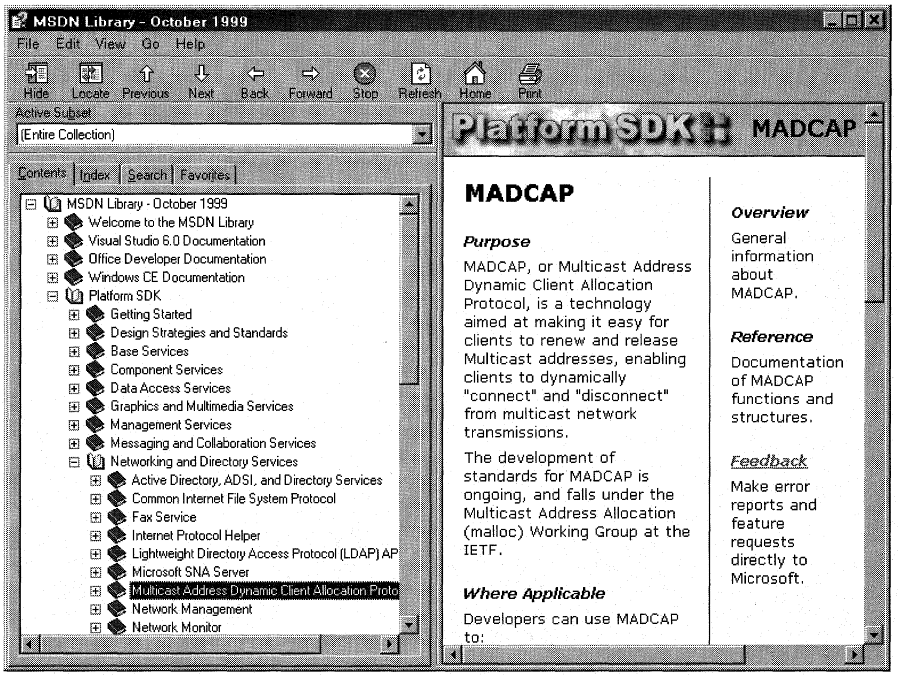
What's New on the Library

Read through this document for summaries of what's new and follow the links to the new titles.

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.


MSDN Library - October 1999
File Edit View Go Help
Hide Locate Previous Next Back Forward Stop Refresh Home Print
Active Subset
(Entire Collection)
Contents Index Search Favorites
Platform SDK MADCAP

MSDN Library - October 1999
- Welcome to the MSDN Library
- Visual Studio 6.0 Documentation
- Office Developer Documentation
- Windows CE Documentation
- Platform SDK
  - Getting Started
  - Design Strategies and Standards
  - Base Services
  - Component Services
  - Data Access Services
  - Graphics and Multimedia Services
  - Management Services
  - Messaging and Collaboration Services
  - Networking and Directory Services
    - Active Directory, ADSI, and Directory Services
    - Common Internet File System Protocol
    - Fax Service
    - Internet Protocol Helper
    - Lightweight Directory Access Protocol (LDAP) AP
    - Microsoft SNA Server
    - Multicast Address Dynamic Client Allocation Proto
    - Network Management
    - Network Monitor

MADCAP

Purpose

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 IETF.

Where Applicable

Developers can use MADCAP to:

Overview

General information about MADCAP.

Reference

Documentation of MADCAP functions and structures.

Feedback

Make error reports and feature requests directly to Microsoft.

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.

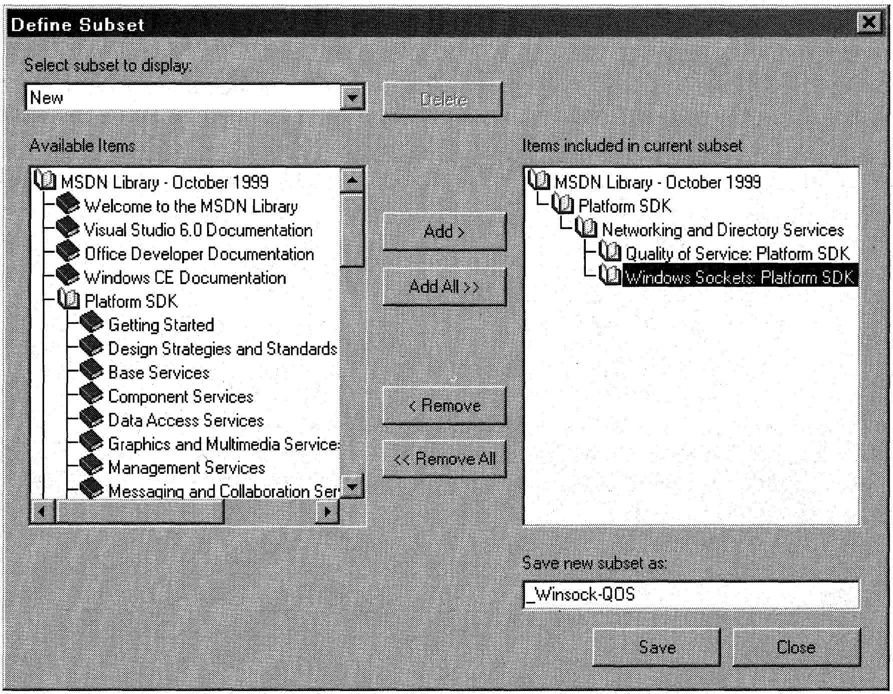

Define Subset
Select subset to display:
New
Delete
Available Items
MSDN Library - October 1999
Welcome to the MSDN Library
Visual Studio 6.0 Documentation
Office Developer Documentation
Windows CE Documentation
Platform SDK
Getting Started
Design Strategies and Standards
Base Services
Component Services
Data Access Services
Graphics and Multimedia Service
Management Services
Messaging and Collaboration Ser
Add >
Add All >>
< Remove
<< Remove All
Items included in current subset
MSDN Library - October 1999
Platform SDK
Networking and Directory Services
Quality of Service: Platform SDK
Windows Sockets: Platform SDK
Save new subset as:
_Winsock-QOS
Save
Close

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.

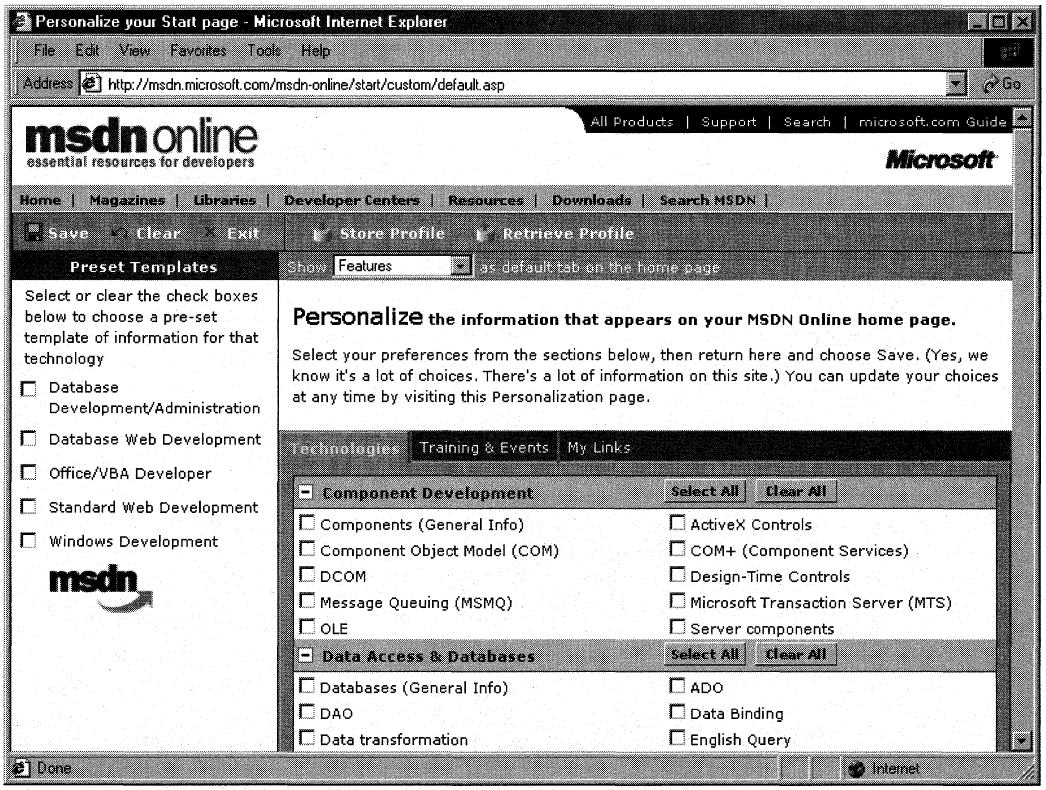
If you've used MSN's home page before (*www.msn.com*), you're familiar with the fact that you can customize the page to your liking; choose from an assortment of available national news, computer news, local news, local weather, stock quotes, and other collections of information or news that suit your 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


Personalize your Start page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Go
All Products | Support | Search | microsoft.com Guide
msdn online
essential resources for developers
Microsoft
Home | Magazines | Libraries | Developer Centers | Resources | Downloads | Search MSDN |
Save Clear Exit Store Profile Retrieve Profile
Preset Templates
Technologies Training & Events My Links
Select All Clear All
Done
Internet

http://msdn.microsoft.com/msdn-online/start/custom/default.asp

Show Features as default tab on the home page

Select or clear the check boxes below to choose a pre-set template of information for that technology

- ☐ Database Development/Administration
- ☐ Database Web Development
- ☐ Office/VBA Developer
- ☐ Standard Web Development
- ☐ Windows Development

Personalize the information that appears on your MSDN Online home page.

Select your preferences from the sections below, then return here and choose Save. (Yes, we know it's a lot of choices. There's a lot of information on this site.) You can update your choices at any time by visiting this Personalization page.

Component Development

☐ Components (General Info)	☐ ActiveX Controls
☐ Component Object Model (COM)	☐ COM+ (Component Services)
☐ DCOM	☐ Design-Time Controls
☐ Message Queuing (MSMQ)	☐ Microsoft Transaction Server (MTS)
☐ OLE	☐ Server components

Data Access & Databases

☐ Databases (General Info)	☐ ADO
☐ DAO	☐ Data Binding
☐ Data transformation	☐ English Query

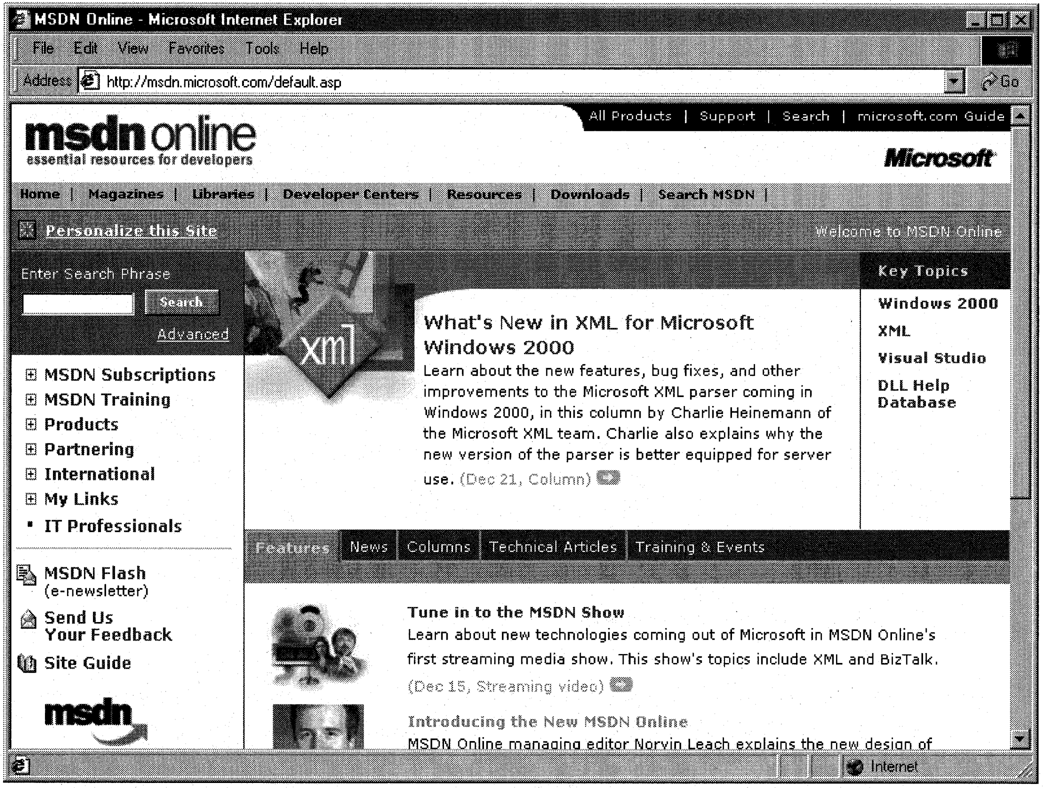
Figure 3-5: The MSDN Online Personalize Page.

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.


MSDN Online - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://msdn.microsoft.com/default.asp
Go
All Products | Support | Search | microsoft.com Guide
msdn online
essential resources for developers
Microsoft
Home | Magazines | Libraries | Developer Centers | Resources | Downloads | Search MSDN |
Personalize this Site
Welcome to MSDN Online
Enter Search Phrase
Search
Advanced
MSDN Subscriptions
MSDN Training
Products
Partnering
International
My Links
IT Professionals
MSDN Flash (e-newsletter)
Send Us Your Feedback
Site Guide
msdn
Key Topics
Windows 2000
XML
Visual Studio
DLL Help Database
Features News Columns Technical Articles Training & Events
Internet

What's New in XML for Microsoft Windows 2000

Learn about the new features, bug fixes, and other improvements to the Microsoft XML parser coming in Windows 2000, in this column by Charlie Heinemann of the Microsoft XML team. Charlie also explains why the new version of the parser is better equipped for server use. (Dec 21, Column)

Tune in to the MSDN Show

Learn about new technologies coming out of Microsoft in MSDN Online's first streaming media show. This show's topics include XML and BizTalk. (Dec 15, Streaming video)

Introducing the New MSDN Online

MSDN Online managing editor Norvin Leach explains the new design of

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	Resources
Magazines	Downloads
Libraries	Search MSDN
Developer Centers	

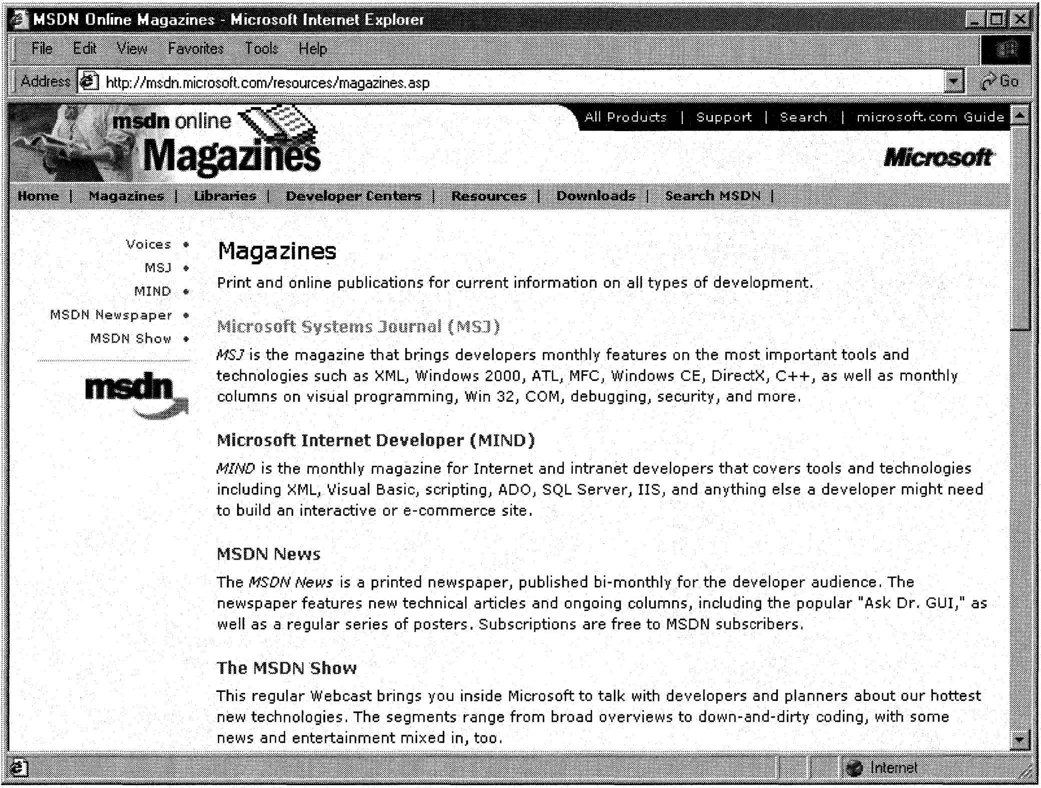
The navigation bar is available regardless of where you are in MSDN Online, so the capability to navigate the site from this familiar menu is always available, leaving you a click away from any area on MSDN Online. These menu categories create a functional and logical grouping of MSDN Online's feature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


MSDN Online Magazines - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://msdn.microsoft.com/resources/magazines.asp
Go
msdn online Magazines
All Products | Support | Search | microsoft.com Guide
Microsoft
Home | Magazines | Libraries | Developer Centers | Resources | Downloads | Search MSDN |
Voices
MSJ
MIND
MSDN Newspaper
MSDN Show
msdn
Internet

Magazines

Print and online publications for current information on all types of development.

Microsoft Systems Journal (MSJ)

MSJ is the magazine that brings developers monthly features on the most important tools and technologies such as XML, Windows 2000, ATL, MFC, Windows CE, DirectX, C++, as well as monthly columns on visual programming, Win 32, COM, debugging, security, and more.

Microsoft Internet Developer (MIND)

MIND is the monthly magazine for Internet and intranet developers that covers tools and technologies including XML, Visual Basic, scripting, ADO, SQL Server, IIS, and anything else a developer might need to build an interactive or e-commerce site.

MSDN News

The *MSDN News* is a printed newspaper, published bi-monthly for the developer audience. The newspaper features new technical articles and ongoing columns, including the popular "Ask Dr. GUI," as well as a regular series of posters. Subscriptions are free to MSDN subscribers.

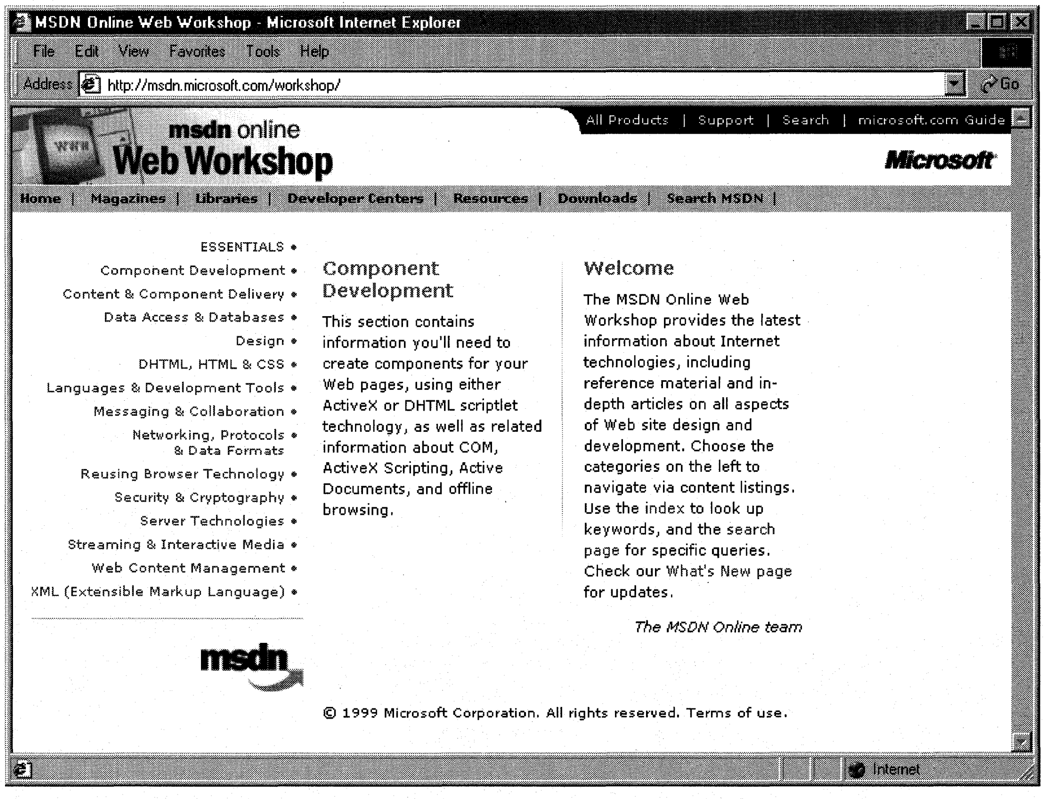
The MSDN Show

This regular Webcast brings you inside Microsoft to talk with developers and planners about our hottest new technologies. The segments range from broad overviews to down-and-dirty coding, with some news and entertainment mixed in, too.

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*.


MSDN Online Web Workshop - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://msdn.microsoft.com/workshop/
msdn online Web Workshop
All Products | Support | Search | microsoft.com Guide
Microsoft
Home | Magazines | Libraries | Developer Centers | Resources | Downloads | Search MSDN |
ESSENTIALS
Component Development
Content & Component Delivery
Data Access & Databases
Design
DHTML, HTML & CSS
Languages & Development Tools
Messaging & Collaboration
Networking, Protocols & Data Formats
Reusing Browser Technology
Security & Cryptography
Server Technologies
Streaming & Interactive Media
Web Content Management
XML (Extensible Markup Language)
Internet

Component Development

This section contains information you'll need to create components for your Web pages, using either ActiveX or DHTML scriptlet technology, as well as related information about COM, ActiveX Scripting, Active Documents, and offline browsing.

Welcome

The MSDN Online Web Workshop provides the latest information about Internet technologies, including reference material and in-depth articles on all aspects of Web site design and development. Choose the categories on the left to navigate via content listings. Use the index to look up keywords, and the search page for specific queries. Check our What's New page for updates.

The MSDN Online team

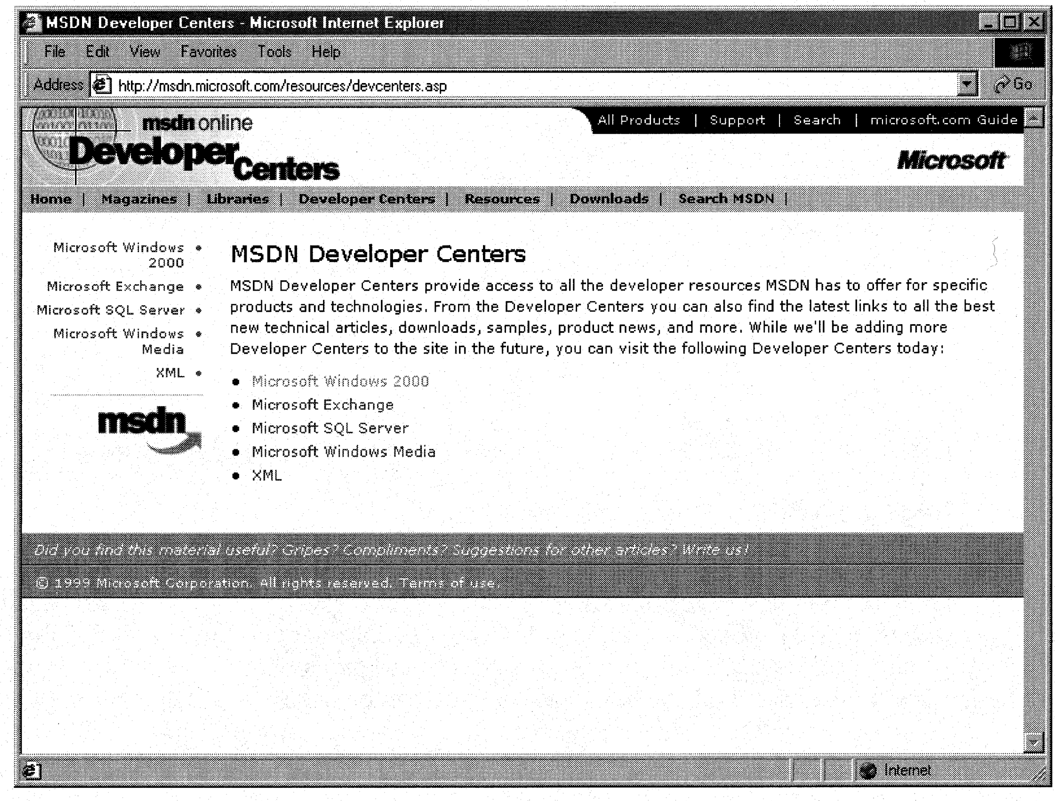
© 1999 Microsoft Corporation. All rights reserved. Terms of use.

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.


MSDN Developer Centers - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://msdn.microsoft.com/resources/devcenters.asp
Go
msdn online Developer Centers
All Products | Support | Search | microsoft.com Guide
Microsoft
Home | Magazines | Libraries | Developer Centers | Resources | Downloads | Search MSDN |
Microsoft Windows 2000
Microsoft Exchange
Microsoft SQL Server
Microsoft Windows Media
XML
msdn
Internet

MSDN Developer Centers

MSDN Developer Centers provide access to all the developer resources MSDN has to offer for specific products and technologies. From the Developer Centers you can also find the latest links to all the best new technical articles, downloads, samples, product news, and more. While we'll be adding more Developer Centers to the site in the future, you can visit the following Developer Centers today:

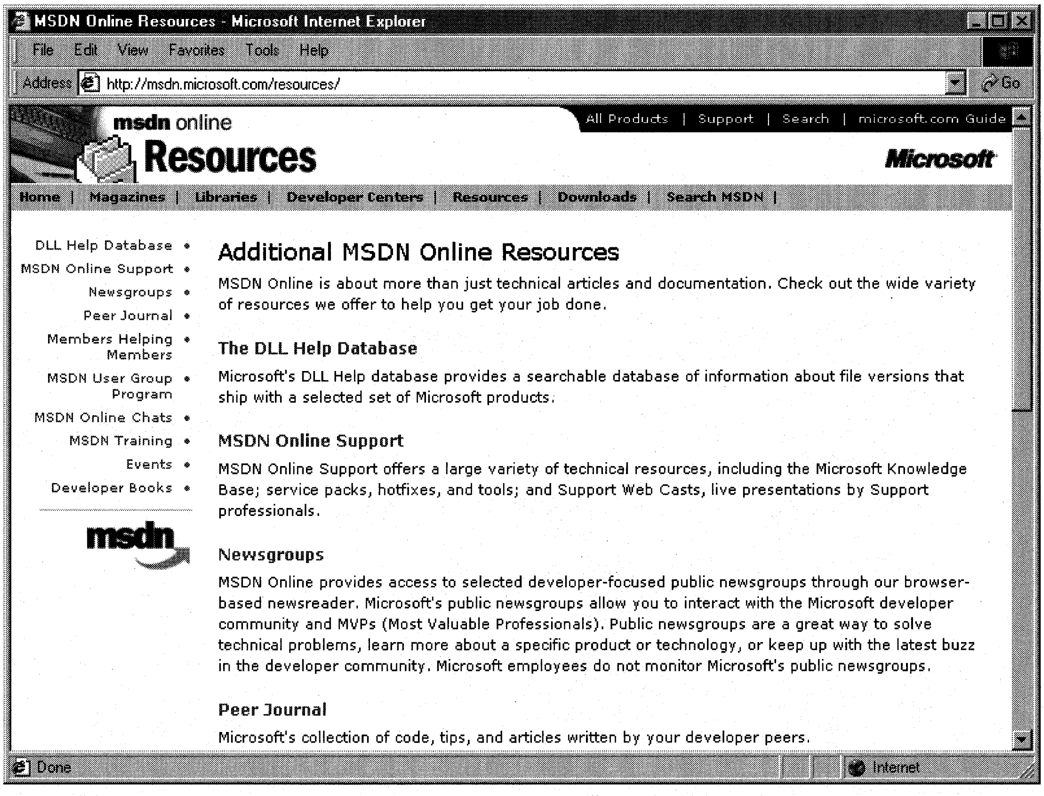
- Microsoft Windows 2000
- Microsoft Exchange
- Microsoft SQL Server
- Microsoft Windows Media
- XML

Did you find this material useful? Gripes? Compliments? Suggestions for other articles? Write us!

© 1999 Microsoft Corporation. All rights reserved. Terms of use.

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.


MSDN Online Resources - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://msdn.microsoft.com/resources/
Go
msdn online
All Products | Support | Search | microsoft.com Guide
Resources
Microsoft
Home | Magazines | Libraries | Developer Centers | Resources | Downloads | Search MSDN |
DLL Help Database
MSDN Online Support
Newsgroups
Peer Journal
Members Helping Members
MSDN User Group Program
MSDN Online Chats
MSDN Training
Events
Developer Books
msdn
Done
Internet

Additional MSDN Online Resources

MSDN Online is about more than just technical articles and documentation. Check out the wide variety of resources we offer to help you get your job done.

The DLL Help Database

Microsoft's DLL Help database provides a searchable database of information about file versions that ship with a selected set of Microsoft products.

MSDN Online Support

MSDN Online Support offers a large variety of technical resources, including the Microsoft Knowledge Base; service packs, hotfixes, and tools; and Support Web Casts, live presentations by Support professionals.

Newsgroups

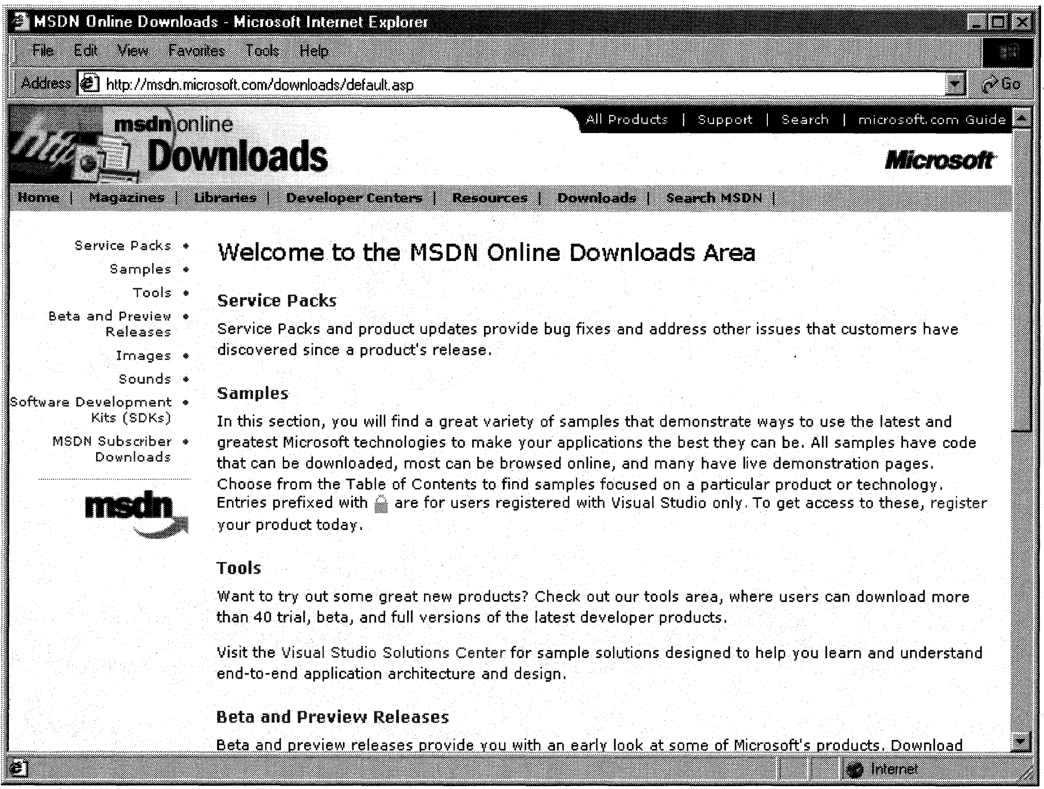
MSDN Online provides access to selected developer-focused public newsgroups through our browser-based newsreader. Microsoft's public newsgroups allow you to interact with the Microsoft developer community and MVPs (Most Valuable Professionals). Public newsgroups are a great way to solve technical problems, learn more about a specific product or technology, or keep up with the latest buzz in the developer community. Microsoft employees do not monitor Microsoft's public newsgroups.

Peer Journal

Microsoft's collection of code, tips, and articles written by your developer peers.

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.


MSDN Online Downloads - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://msdn.microsoft.com/downloads/default.asp
Go
msdn online Downloads
All Products | Support | Search | microsoft.com Guide
Microsoft
Home | Magazines | Libraries | Developer Centers | Resources | Downloads | Search MSDN |
Service Packs
Samples
Tools
Beta and Preview Releases
Images
Sounds
Software Development Kits (SDKs)
MSDN Subscriber Downloads
msdn
Internet

Welcome to the MSDN Online Downloads Area

Service Packs

Service Packs and product updates provide bug fixes and address other issues that customers have discovered since a product's release.

Samples

In this section, you will find a great variety of samples that demonstrate ways to use the latest and greatest Microsoft technologies to make your applications the best they can be. All samples have code that can be downloaded, most can be browsed online, and many have live demonstration pages. Choose from the Table of Contents to find samples focused on a particular product or technology. Entries prefixed with 🔒 are for users registered with Visual Studio only. To get access to these, register your product today.

Tools

Want to try out some great new products? Check out our tools area, where users can download more than 40 trial, beta, and full versions of the latest developer products.

Visit the Visual Studio Solutions Center for sample solutions designed to help you learn and understand end-to-end application architecture and design.

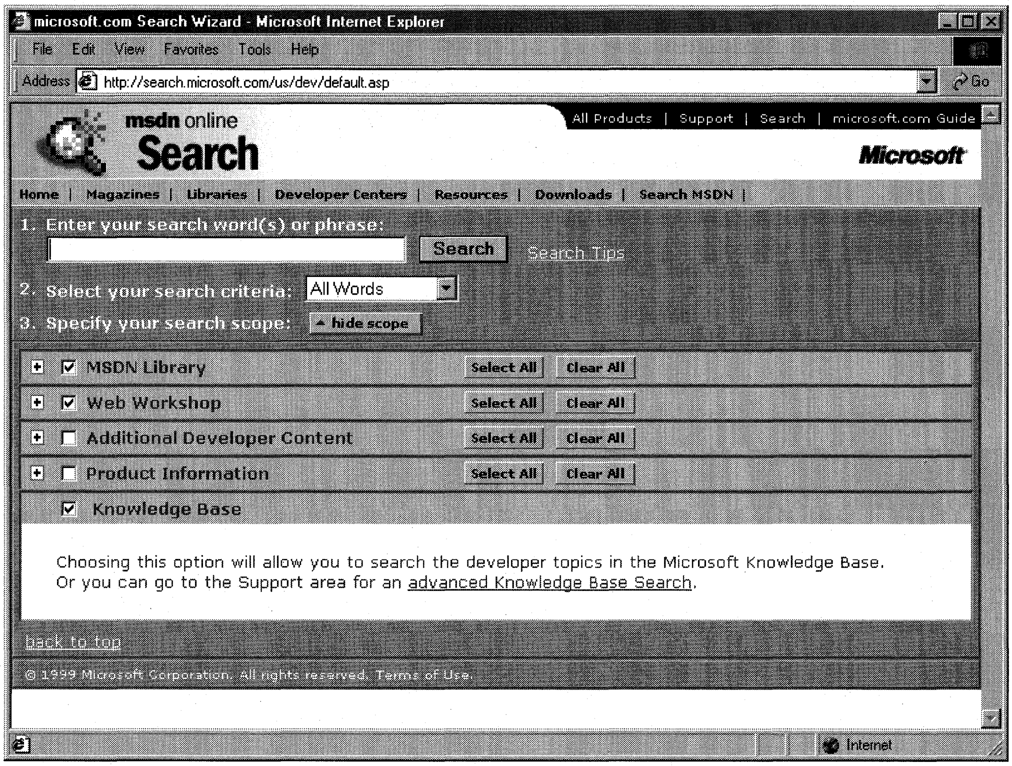
Beta and Preview Releases

Beta and preview releases provide you with an early look at some of Microsoft's products. Download

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**.


microsoft.com Search Wizard - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://search.microsoft.com/us/dev/default.asp
msdn online
Search
All Products | Support | Search | microsoft.com Guide
Microsoft
Home | Magazines | Libraries | Developer Centers | Resources | Downloads | Search MSDN |
1. Enter your search word(s) or phrase:
Search
Search Tips
2. Select your search criteria: All Words
3. Specify your search scope: hide scope
MSDN Library Select All Clear All
Web Workshop Select All Clear All
Additional Developer Content Select All Clear All
Product Information Select All Clear All
Knowledge Base
back to top
Internet

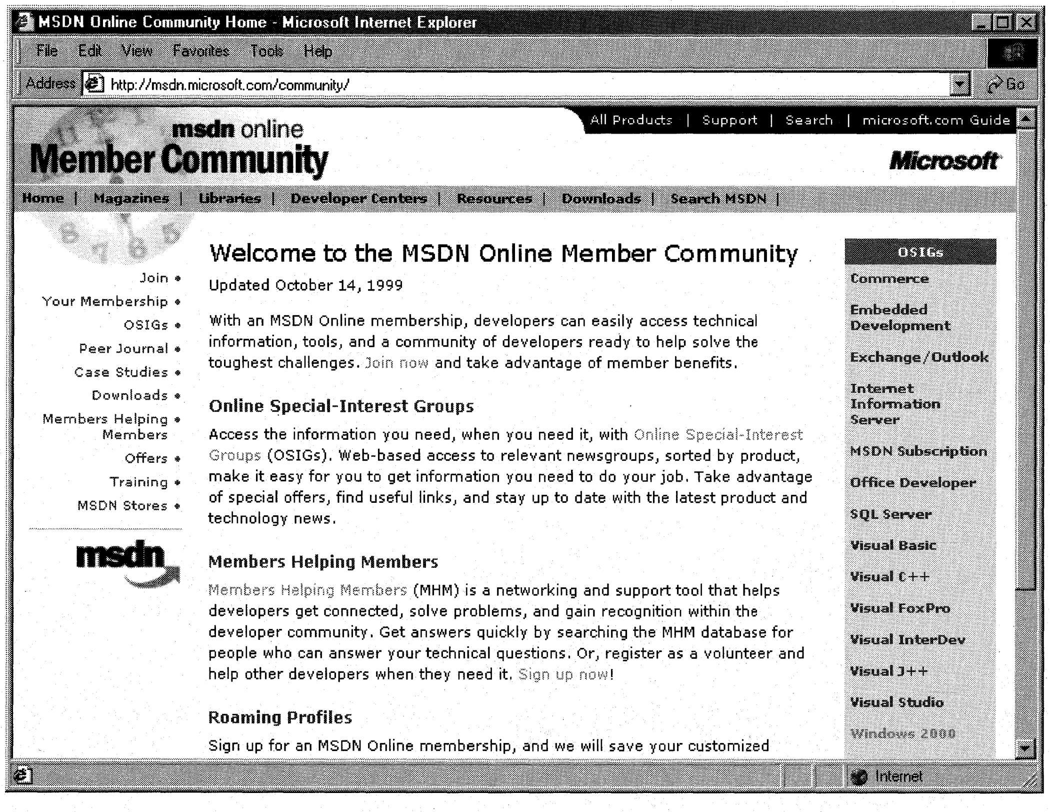
Choosing this option will allow you to search the developer topics in the Microsoft Knowledge Base.
Or you can go to the Support area for an advanced Knowledge Base Search.

© 1999 Microsoft Corporation. All rights reserved. Terms of Use.

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*.



Welcome to the MSDN Online Member Community

Updated October 14, 1999

With an MSDN Online membership, developers can easily access technical information, tools, and a community of developers ready to help solve the toughest challenges. Join now and take advantage of member benefits.

Online Special-Interest Groups

Access the information you need, when you need it, with Online Special-Interest Groups (OSIGs). Web-based access to relevant newsgroups, sorted by product, make it easy for you to get information you need to do your job. Take advantage of special offers, find useful links, and stay up to date with the latest product and technology news.

Members Helping Members

Members Helping Members (MHM) is a networking and support tool that helps developers get connected, solve problems, and gain recognition within the developer community. Get answers quickly by searching the MHM database for people who can answer your technical questions. Or, register as a volunteer and help other developers when they need it. Sign up now!

Roaming Profiles

Sign up for an MSDN Online membership, and we will save your customized

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.

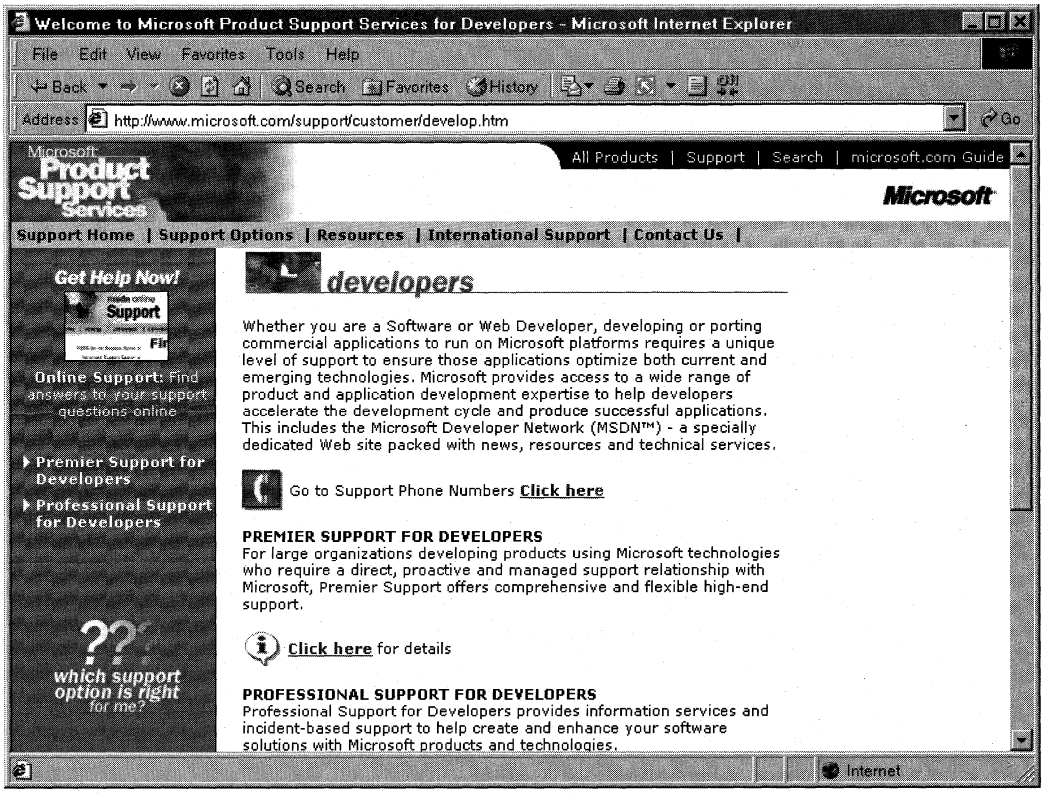
Microsoft provides plenty of resource material through MSDN and MSDN Online, and the WPRS provides a great filtered version of focused reference material and development knowledge. However, there is a lot more information to be had. Some of that information comes from Microsoft, some of it from the general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


Welcome to Microsoft Product Support Services for Developers - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.microsoft.com/support/customer/develop.htm
Go
Microsoft Product Support Services
All Products | Support | Search | microsoft.com Guide
Microsoft
Support Home | Support Options | Resources | International Support | Contact Us |
Get Help Now!
Online Support: Find answers to your support questions online
Premier Support for Developers
Professional Support for Developers
??? which support option is right for me?
developers
Internet

Whether you are a Software or Web Developer, developing or porting commercial applications to run on Microsoft platforms requires a unique level of support to ensure those applications optimize both current and emerging technologies. Microsoft provides access to a wide range of product and application development expertise to help developers accelerate the development cycle and produce successful applications. This includes the Microsoft Developer Network (MSDN™) - a specially dedicated Web site packed with news, resources and technical services.

Go to Support Phone Numbers Click here

PREMIER SUPPORT FOR DEVELOPERS
For large organizations developing products using Microsoft technologies who require a direct, proactive and managed support relationship with Microsoft, Premier Support offers comprehensive and flexible high-end support.

Click here for details

PROFESSIONAL SUPPORT FOR DEVELOPERS
Professional Support for Developers provides information services and incident-based support to help create and enhance your software solutions with Microsoft products and technologies.

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 listed on the next page.

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

Understanding Routing Technologies

Networking knowledge is like being able to tie your shoes; in today's Internet society, each step without that knowledge increases your chances of tripping up. Routing is at the heart of networking.

Many programmers who develop solutions with RRAS have extensive knowledge of routers and routing technologies. They know everything there is to know about packets and how they are constructed, can name five different routing protocols, and in general could give lectures on routing technologies at the drop of a hat. This chapter isn't for them.

This chapter is for the rest (pronounced "majority") of the developers out there creating routing solutions with Microsoft's RRAS APIs; developers who perhaps have a reasonable familiarity with routing but are fuzzy around the edges, or developers who are familiar with basic routing but aren't quite sure how packets are constructed and how such construction can have a bearing on routing. This chapter is also for the other large group of developers who might be interested in routing technologies or simply want to find out more about routing for their own edification. Whether a refresher or an introduction, this chapter provides an explanation of routing and router technologies, and presents the information in a concise and readable fashion.

This chapter is divided into three sections, each of which builds on the previous section to create a foundation of knowledge on which you can better understand routing and its various protocols:

- Routing Basics
- Routable Protocols
- Routing Protocols

Routing Basics

The idea behind routing is fundamental: to pass data between LAN segments so that a collection of LAN segments functions as one big network. The implementation is a bit more involved, however, especially when you have multiple routers and multiple subnets. For routers to function at all they must have some prior knowledge of the network to enable a routing decision to be made. Such decisions might be simple, such as knowing to send everything not destined for the local network to an alternative network. Or they might be much more complex. Simply put, routers need to know where to send data that comes to them, and figuring out how to do so requires configuration—whether that be static (done by an administrator) or dynamic (updated often and accomplished through communication with other routers on the network).

Routers make decisions on where to send data based on information they keep in their *routing table*. A router's routing table is its bible, atlas, and calculator all rolled into one; it dictates behavior and treatment of neighbors, determines distances from the "you are here" sign to where they need to send their data, and calculates path costs in penny-pinching router terms. Routers are also egotistical; the universe revolves around them, and every routing table starts with a "you are here" sign (the center of its known universe)—its own address.

In networks with multiple routers and numerous subnets, routers can be configured to talk to one another about the roadmap of the network. Such communication enables routers to determine how to send data to any destination in the network, and is achieved through a *routing protocol*.

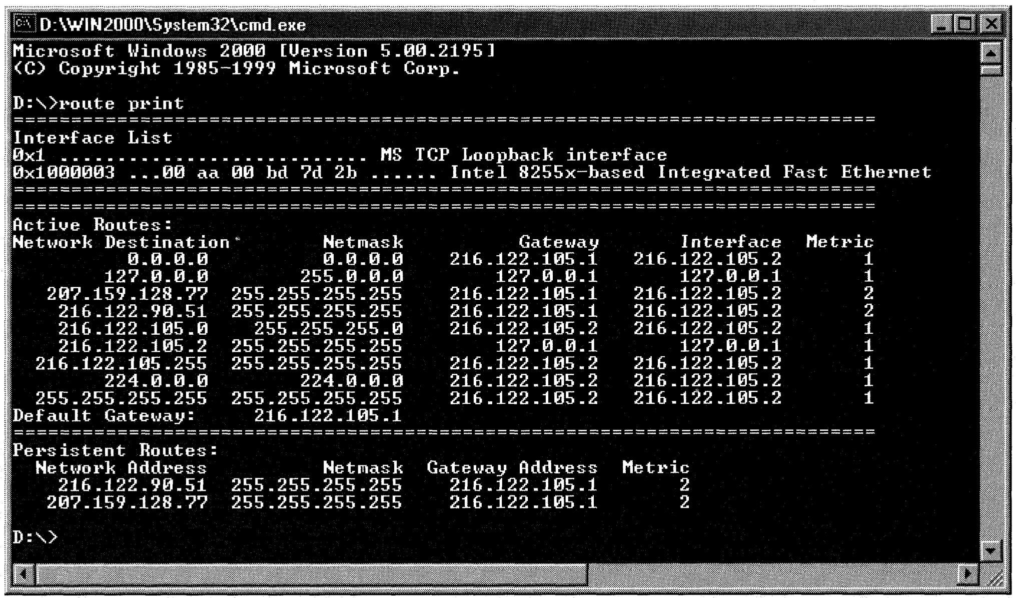
To begin with, however, I'm going to start with a reasonably short list of routing terms whose definitions should help you understand routing issues in general, and should also help you understand some of the discussions presented later in this chapter.

Basic Routing Terminology

The following sections provide definitions for some basic, often-seen routing terms whose definitions can help you better understand routing and the various protocols, components, or algorithms associated with routing.

Routing Table

A routing table is a list of available routes to network destinations. Routing tables often have associated metrics—a means by which routers measure the "expense" of reaching a given route—for each available route. Figure 5-1 provides an example of a simple routing table as seen from a Microsoft Windows 2000 command prompt window when the route print command is issued. Notice that there are a few additional entries in the routing table that I've personally set to enable particular routing capabilities I use in my day to day (Internet-connected) activities.



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-1999 Microsoft Corp.

D:\>route print
===========================================================================
Interface List
0x1 ........................... MS TCP Loopback interface
0x1000003 ...00 aa 00 bd 7d 2b ...... Intel 8255x-based Integrated Fast Ethernet
===========================================================================
===========================================================================
Active Routes:
Network Destination        Netmask          Gateway       Interface  Metric
          0.0.0.0          0.0.0.0    216.122.105.1   216.122.105.2       1
        127.0.0.0        255.0.0.0        127.0.0.1       127.0.0.1       1
   207.159.128.77  255.255.255.255    216.122.105.1   216.122.105.2       2
     216.122.90.51  255.255.255.255    216.122.105.1   216.122.105.2       2
     216.122.105.0    255.255.255.0    216.122.105.2   216.122.105.2       1
     216.122.105.2  255.255.255.255        127.0.0.1       127.0.0.1       1
   216.122.105.255  255.255.255.255    216.122.105.2   216.122.105.2       1
        224.0.0.0        224.0.0.0    216.122.105.2   216.122.105.2       1
  255.255.255.255  255.255.255.255    216.122.105.2   216.122.105.2       1
Default Gateway:     216.122.105.1
===========================================================================
Persistent Routes:
  Network Address          Netmask  Gateway Address  Metric
    216.122.90.51  255.255.255.255    216.122.105.1       2
   207.159.128.77  255.255.255.255    216.122.105.1       2

D:\>
```

Figure 5-1: The Routing Table Presented When a Route Print Command is Issued.

Hop Count

Hop count is the number of routers that must be traversed to reach a given destination. For example, a destination with a hop count of three would have gone to the default router (hop 1); upon determining that the destination host was not connected to any of the router's local subnets, the router would have forwarded the packet (after checking its routing table for the appropriate router to which to send the packet) to the next router (hop 2); again, upon determining that the destination host wasn't directly connected to one of the router's subnet, it would have forwarded the packet (after checking its routing table for the appropriate router) to the next router (hop 3), which would determine that the destination host was on a subnet to which it was directly attached, and then would send it to the destination host. Three routers crossed/traversed = three hops.

Most routable protocols, such as IP and IPX, have a maximum hop count. For IPX the maximum hop count is 16; this designation of maximum hop count means that if the packet is being sent through a network and reaches its 17th router on the way to its destination, it is dropped (discarded, deleted). Hop count is modified each time a packet passes through a router to enable such detection. If it weren't for hop counts, packets that were misrouted could zoom around the network forever, eventually filling your network bandwidth with random, wandering (useless) packets. As it is, such lost (misguided and misaddressed) packets eventually are discarded by routers to ensure the availability of bandwidth for non-vagrant packets.

Default Gateway

The *default gateway*, also referred to as *default route*, is an address that is used by routers (and computers) when no other means (in the routing table) of reaching a destination is available. The default gateway is not necessarily a last resort, as the definition might at first sound; rather, it is a means of differentiating between locally available or defined routes (the routes or host of which the router has specific knowledge) and "all others."

Convergence

The time it takes routers to *converge* after a change in network topology (which could be an added or downed router, or changes in the metric of a given link) can be considered the determining factor of the stability and performance of your network. Convergence is the process of updating the routing tables of routers in an intranetwork to reflect changes in the network routing topology. For example, say you have an intranetwork with 17 routers, and one of those routers has a power supply that blows up and renders the router useless and dead. Segments to which that router were attached are no longer reachable through the dead router, and because routers communicate with one another through routing protocols, other routers on the intranetwork will learn of this failed router and adjust their routing tables accordingly. (One of the aspects of a routing protocol is the question, "Are you still there?" If the asking router doesn't get a reply after a specific period of time, it presumes that the other router is dead.) When such an event occurs, routers that have new information in their routing table (news of this dead router is novel) share that information with every other router on the intranetwork. The time it takes for all routers on the intranetwork to hear about the change and adjust their routing tables accordingly is considered *convergence time*. Once all routers share the same information in their individual routing tables (this does not mean that their routing tables are identical; remember that all routers' routing tables are from a "you are here" perspective), the network is considered converged. The importance of a short convergence time is the following: So long as the network is not converged, the system incurs packet loss attributable to routers passing packets to the dead router (and as you might suspect, that dead router not routing those packets). The shorter the convergence time, the better the solution.

This is where the advantage of dynamic routing over static routing can really be seen. If static routing is used and one of your routers goes down (remember that static routing requires an administrator to manually create the routing table for each router in the intranetwork), any network traffic that goes through that router will be undeliverable and network connections will be dead (for traffic going through that router). The situation will remain as such until an administrator troubleshoots the problem, and either modifies the routing table or replaces the router. With dynamic routing, the routing protocol itself detects and troubleshoots the loss of the router, and if possible, packets will be routed around the failed router without intervention from the administrator.

Flooding

Flooding is the means by which routers advertise routing table changes to the rest of the routers on their internetwork. Much as it sounds, flooding involves a saturation of the entire routed internetwork with packets containing information about the sending router's routing table. Transmissions of such packets have special information included in them that enables them to be broadcast only over subnets that haven't received the specific "version" of the routing update; if they did not have such special information, flooding could cause such a barrage of packets (as they were sent and resent over subnets from multiple connected routers) that the network would be brought to a standstill. To avoid that situation, routing protocols that implement flooding ensure that flooding crosses a subnet only once. By properly flooding the network with router change information, each router can be assured that updates to its routing table are reflected in every other router in the internetwork communicating with the sending router's routing protocol.

Routing Loops

Let's say your machine is on Network A and you're trying to send data to a machine on Network D. Suppose your local router (the one you use to get beyond your local subnet) has a routing table that says, "To get to Network D, use the router on Network B," and the router on Network B says, "To get to Network D, use the router on Network C." Further, the router on Network C says, "To get to Network D, use the router on Network A." You're in a *routing loop*. Network A—trying to get to Network D—has sent you to Network B, then to Network C, and back to Network A, which will then send you back to Network B and start that vicious cycle all over again. This situation is yet another drawback to static routing tables; dynamic routing and the implied use of a routing protocol is designed to avoid routing loops. Fortunately every packet that gets routed (IP and IPX) has a hop count (see its definition earlier in this chapter) to keep packets from circling around the network and taking up bandwidth until the end of time; eventually the packets will reach their maximum hop count and be discarded.

Black Holes

Routes that end in a dead end are considered *black holes*. In its simplest terms, a black hole is a routing table entry that has no listening router on the destination end. So if you have Network A trying to get to Network C, and the router attached to Network A says it must go through Network B to get there, but Network B isn't forwarding packets, you've run into a black hole. Black holes are, in this terminology as well as in real life, good things to avoid.

Static Routing vs. Dynamic Routing

When a packet arrives at a router, the router looks at its routing table and determines where to send that packet next. But how does it determine what to put in its routing table? A router's routing table is determined and configured either statically or dynamically.

With static routing, routing table entries are manually input and updated by someone such as a network administrator. No information is exchanged between routers on an internetwork that implements static routing, and therefore, routers that are dead, down, or otherwise unreachable are not detected. Thus if you're in a network environment where static routers are used and one of them fails, the portion of your network that depends on that router for connection for routing becomes unavailable. Static routing can work just fine in small networks; the routing capability that comes out of the box with Windows NT 4.0 or Windows 2000 (without specifically implementing RRAS routing features) is an example of static routing. However, static routing does not scale well to anything other than small networks due to the overhead associated with building, troubleshooting, and administrating static routing tables.

Dynamic routing enables the dynamic updating of routing tables. Routers that implement dynamic routing are capable of communicating with other routers on the network, and with such interrouter communication, can detect downed routers, determine the best route to take to get to each interconnected subnet, and modify routing tables based on new information (such as a downed router or a newly added router). Furthermore, such routing information in a dynamic routing environment can be propagated to the rest of the network's routers. Once initially configured, dynamic routers don't require administrative intervention to adjust to changes in their routing environment. Routing Information Protocol (RIP), RIP II and Open Shortest Path First (OSPF) are examples of protocol implementations of dynamic routing.

Construction of a Frame

Have you ever wondered how a 17 MB file gets from the file server in the central of your firm's network to your computer, and how those pieces of data get transmitted over the network wire? It all has to do with routing. Even before that, the transmission requires that the file be chopped up into pieces before it hits the network, and then reconstructed once it reaches its destination. Understanding how this occurs is important in understanding how routing works.

Let's make an everyday comparison to see how frames are constructed. When you want to send a gift in the mail, you might start by wrapping the gift in some nice tissue, then you might place it in a box, wrap the box, put a ribbon on it, place the wrapped-and-ribboned box into a shipping box, put an address on the box, and finally, attach postage appropriate to the means by which it is shipped. If you're sending it with United Postal Service (UPS), you must have someone affix an appropriate UPS sticker on the box; if you're sending it via Federal Express, you'll need their sticker, and finally if you're sending it through regular mail, you'll need the right sticker or stamp. All you really want to do is send the gift to someone half-way across town (or all the way across the world), but to get it there you have to jump through the appropriate hoops. Let's break this down into parts:

The gift: The point of this whole ordeal is to send the gift where you want it to go.

Tissue: You have to properly present the gift, so you wrap it in gift tissue.

Gift box: You can't have that gift flopping around. Put it in a box.

Ribbon: A gift isn't a gift without a ribbon. It will get smashed on the way, and no one really pays much attention to the ribbon, but custom is custom.

Shipping box: You can't ship the gift box as is, so you use a shipping box.

Address: The guy in the uniform and matching truck must know where to shuffle this package to, so you give him the address. The address will have a state, city, and street address (general part of the address), and a name (specific, unique part of the address).

Appropriate shipping sticker: Put a UPS sticker on something going through the regular mail and you'll get nowhere (and in fact, it might just disappear without a trace), so you need to make sure you get the right sticker on it. Maybe, however, you're fortunate enough to have a butler or a secretary who does all these things for you, so when you want to send something off you don't worry about the sticker, and instead just hand it off to someone else who affixes the appropriate sticker for you.

On the other side of the delivery, the series of events plays itself in reverse: the driver who delivers to the address looks at the sticker (the sticker has been changed because it's going to Germany—a different language, but still addressed to the same person) and delivers it to the address. The person to whom the package was addressed receives the package and checks the sticker (to ensure it is for them), opens the shipping box, unties the ribbon, opens the gift box, removes the gift tissue, and takes out the gift.

This is almost exactly how a network frame is created and sent across the network.

You start with the data, and as you go through the OSI model, information is appended to the data until you reach the final layer (physical), where the frame is complete and can be sent onto the wire and reach the intended destination. Figure 5-2 illustrates this.

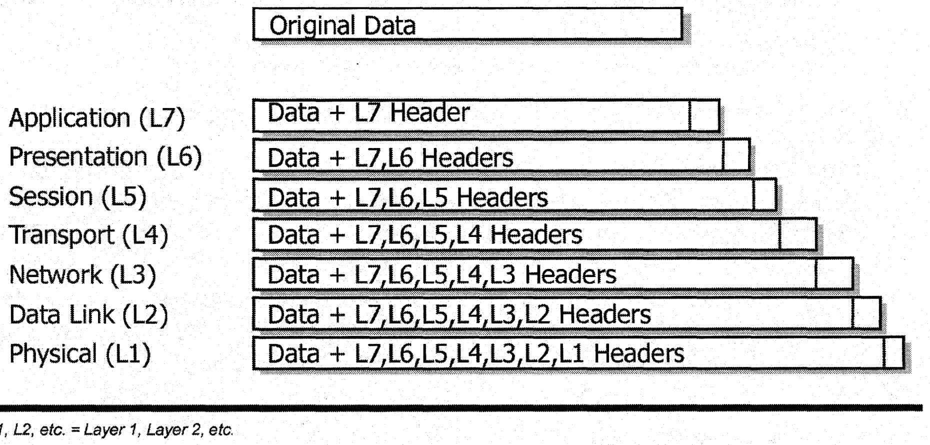


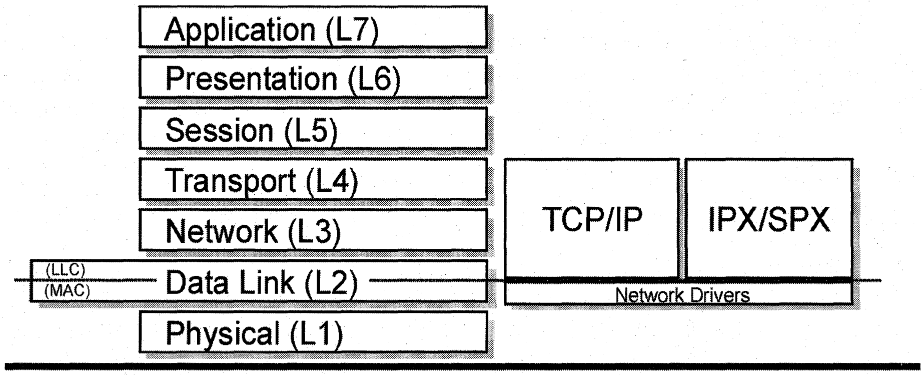
Figure 5-2: Appending Headers at Each Layer to Produce a Packet.

Starting with the data (the gift), information is appended at each theoretical layer of the OSI model to enable the data to get from one computer on the network to another. The reason for doing this is that each layer in the OSI model—for example, the transport layer—expects information specific to its layer to be there when it receives the data. An obvious example in this case would be the address of your Germany-bound package. If there's no address, the delivery driver doesn't know what to do with the package.

But in our frame-constructing model, every appended piece of information is just as important as the address is to the delivery guy—the data can't be delivered if even one is missing or incorrect.

Routable Protocols

IP and IPX match up with the OSI model (see Figure 5-3) in a way that enables packets transmitted with their protocols to be routed.



L1, L2, etc. = Layer 1, Layer 2, etc.
LLC = Logical Link Control - part of the further division of the Data Link Layer, per the 802 Project
MAC = Media Access Control - part of the further division of the Data Link Layer, per the 802 Project

Figure 5-3: The OSI Model and How Its Layers Match Up with IPX and IP.

Notice that the IP part of the TCP/IP suite has a break right between the network layer and the transport layer. This is the all-important, routing-enabled break point that makes TCP/IP a protocol suite that can be routed across multiple subnets. This break point enables a subnet/segment to be identified by a special number, and by doing so enables a way to differentiate between such subnets. Also notice that IPX has a break between the network and transport Layers—again, this enables IPX to be routed between subnets. The process of doing so goes something like this:

When a local host looks at its network information (this is actually a mini-version of a routing table, similar to what a router maintains to track the topology of the internetwork), it determines whether the data it's trying to send is going to a machine on the local subnet or to a machine outside its local subnet. If it's going to a machine on the local subnet, no router is involved and the client simply sends the data to the machine. If the local host determines it's trying to send data to a machine outside the local subnet, it sends the data to the router for delivery.

When the router receives the frame, it looks at the frame's destination information and checks its routing table for the appropriate route the frame should take to get to the destination host. When the router determines the appropriate route, it strips the Layer 2 header information (data link) and replaces it with its own information—information that will either take it to the destined host (if the host is connected to a subnet to which the router has a direct connection), or forward it to the next appropriate router. Note that despite this stripping, information about the source and destination IP addresses is retained. Figure 5-4 puts this process into an illustration.

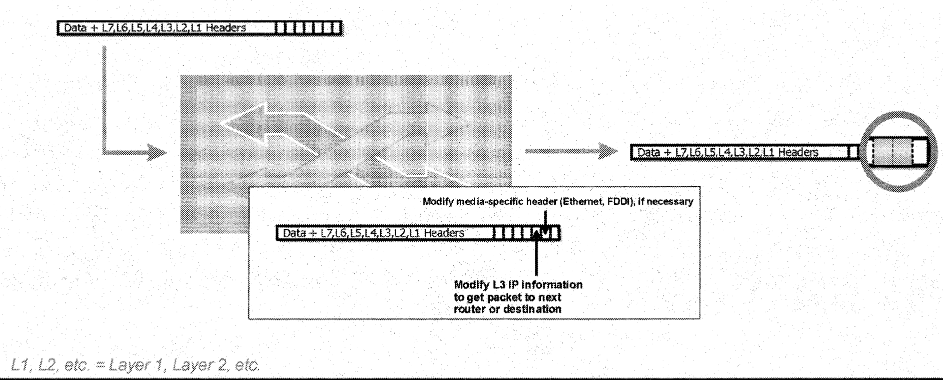
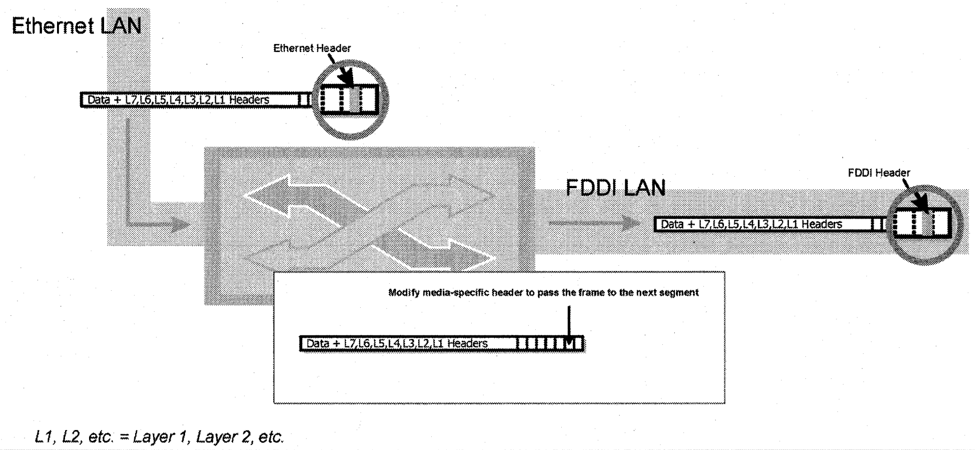

Data + L7,L6,L5,L4,L3,L2,L1 Headers
Data + L7,L6,L5,L4,L3,L2,L1 Headers
Modify media-specific header (Ethernet, FDDI), if necessary
Data + L7,L6,L5,L4,L3,L2,L1 Headers
Modify L3 IP information to get packet to next router or destination
L1, L2, etc. = Layer 1, Layer 2, etc.

Figure 5-4: A Router Stripping Network Information and Replacing It With Its Own.

The router does this for a number of reasons. First let's take another look at our data frame in Figure 5-5, this time putting in the appropriate comparisons to the OSI model and the network media.


Ethernet LAN
Ethernet Header
Data + L7,L6,L5,L4,L3,L2,L1 Headers
FDDI LAN
FDDI Header
Data + L7,L6,L5,L4,L3,L2,L1 Headers
Modify media-specific header to pass the frame to the next segment
Data + L7,L6,L5,L4,L3,L2,L1 Headers
L1, L2, etc. = Layer 1, Layer 2, etc.

__Note:__ Though Layer 1 is being graphically represented here as a 'header,' it is more accurately a governed standard for transmitting the data across its medium.

Figure 5-5: The OSI Layer with IPX or IP, Plus the Appropriate Frame Type.

Notice that the MAC header includes information about the type of media over which the frame is going to travel. So what happens if, on the way to the destination machine, the router has to send your frame over a network different than yours—say a Token Ring network or an FDDI network? In that case, the router, by replacing the information contained on Layer 2 (the Data Link Layer) with its own, enables the packet to travel over any kind of network using any kind of medium to reach its final destination. The router (or routers) along the way is thus acting as a sort of "media gateway" (by definition, a bridge) by manipulating Layer 2.

The end result is that two of the general purpose protocols shipped with Windows NT and Windows 2000 (the fourth, DLC, isn't general purpose)—IP and IPX—are routable because those protocol specifications have built-in mechanisms that enable individual subnets to be uniquely identified and have means (available routers) to forward their frames between subnets. NetBEUI has no means by which its local subnet can be uniquely identified (no means of segregating devices between logical segments) and is thus not routable.

Routing Protocols

What's the difference between *routable* protocols and *routing* protocols? Routable protocols (such as IP and IPX) enable computers or devices on different subnets to communicate with each other. Routing protocols (such as RIP, RIP II, or OSPF) enable routers—the devices that connect individual subnets—to exchange information about routing tables in order to create one big happy virtual network out of all of the interconnected individual segments in a given network. Routing protocols go from not too terribly difficult to grasp (RIP) to something out of an anesthesiology textbook (OSPF). As with many subjects on computing, however, routing protocols are made easier or more difficult to understand based on how they are presented and explained.

IGPs vs. EGPs

Let's make some broad distinctions. Networks (and their routers) that require routing protocols vary greatly in size (from somewhat small to the size of the Internet). Large networks are often broken into smaller units to allow for easier administration and more reasonable routing solutions. As a result, we have two kinds of communication between routers: Interior Gateway Protocols (IGP) and Exterior Gateway Protocols (EGP).

IGPs are used for communication between routers that are in the same interconnected network. IGPs enable the sharing of routing table information among routers that are considered a part of the same interconnected network. If a network is divided into different areas, IGPs share routing information with members of the same area.

IGPs do a couple of things. First, IGPs enable routers to create a complete routing table—a routing table that includes information on how to reach all routers (and thus all subnets) in the internetwork. Second, IGPs provide a means by which a router can determine the best way to send data to another router, or more specifically, the best way to send a packet destined for any given computer (or other network device) to its destination.

The methods by which IGPs disseminate their routing information and determine the best route by which they should reach a given subnet or router fall into two categories: Distance Vector and Link State. *Distance Vector* is the simpler of the two and is the easiest to understand and implement. RIP and RIP II are examples of Distance Vector protocols. *Link State* is more complex to understand and implement, but makes up for its complexities by solving the many shortcomings inherent with Distance Vector routing solutions. For large networks, Link State IGPs are almost required. OSPF is an example of a Link State protocol. Figure 5-6 illustrates the scope of coverage of IGPs.

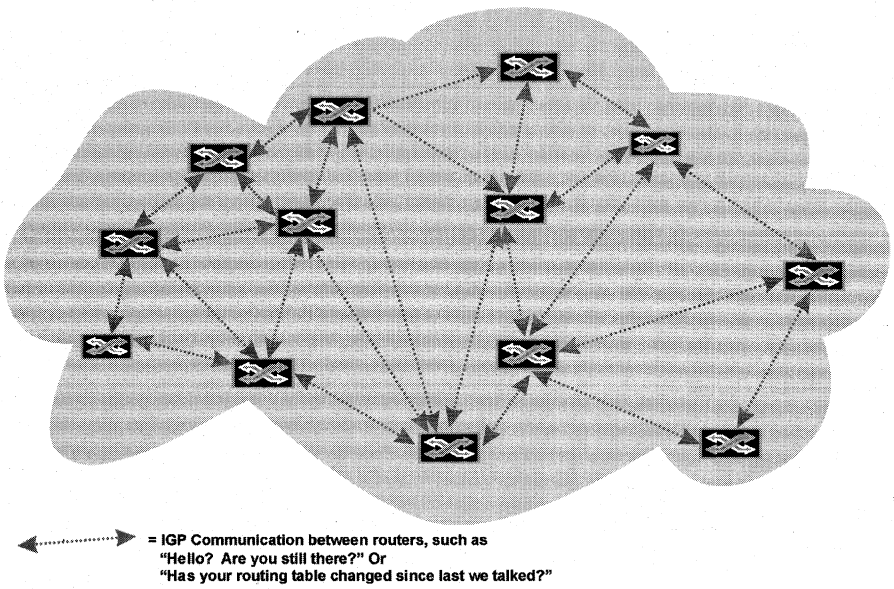

= IGP Communication between routers, such as
"Hello? Are you still there?" Or
"Has your routing table changed since last we talked?"

Figure 5-6: Where IGP Fits Into the Picture.

In contrast, EGPs are a means of communication between routers that are not a part of the same interconnected network or area. EGPs are used to communicate information (such as how their interconnected networks are advertised to the outside world) outside their area. EGP is unusual in that it is a classification for a type of routing protocol, and also the name of a particular protocol (EGP). (Kind of like the doctor who is named Doctor.) EGP (the protocol) was used on the Internet to connect its multitude of interconnected networks. It was replaced a number of years ago with Border Gateway Protocol (BGP), which is itself an EGP. Figure 5-7 illustrates the difference between an IGP and an EGP.

If you have an Autonomous System (AS) that needs to communicate with the outside world—say the Internet, for example—you will need to implement an EGP on your AS Boundary Router (or have your ISP do it for you). As mentioned earlier, a protocol called EGP was first used as the Exterior Gateway Protocol on the Internet that enabled Autonomous Systems to communicate. The successor to EGP is BGP, and it builds and improves on the experience gathered with the use of the EGP protocol. BGP-4 is in use on the Internet today. An in-depth explanation of BGP is not provided here; if you want information on BGP standards or recommendations, check out the IETF web site at *www.ietf.org*.

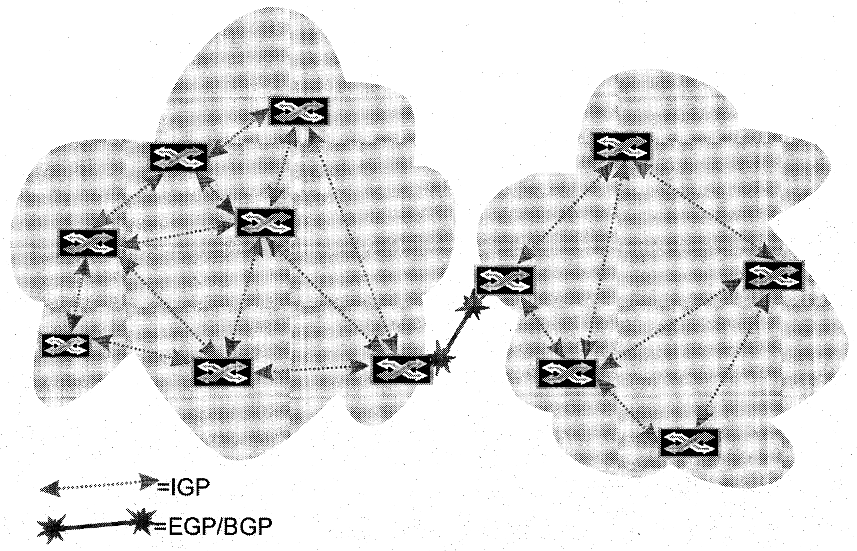


Figure 5-7: Differentiating Between IGPs and EGPs.

The following sections focus on the various IGPs available with most routers, including Windows NT and Windows 2000 RRAS.

RIP and RIP II for IP

This section provides an overview of RIP and RIP II for IP, and discusses some of its drawbacks. This discussion is intended to familiarize you with the issues you might face if developing solutions for RIP, and conversely, provides you with information that the people who administer routers and networks must consider when determining the advantages and drawbacks associated with RIP and RIP II. Such information is intended to provide you with more context and insight into the administrative side of routing and routing protocols.

RIP for IP is a relatively simple—though useful and widely deployed—implementation of a Distance Vector protocol. RIP provides the most basic information required to create and maintain routing protocols throughout an internetwork, and does so by using a simple metric for calculating the "cost" of a given route. Consider Figure 5-8.

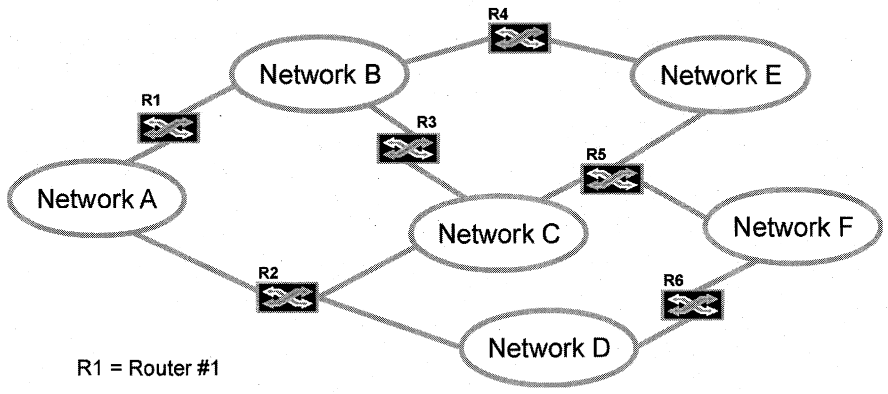


Figure 5-8: A Handful of Routers and Subnets with Sufficient Redundancy.

Notice that there is more than one way to reach each of the attached subnets. Some of those routes are longer than the others because they have more hop counts (routers to cross) than other routes. In a network with "costs" that directly associate with the number of hops to each network, the routing table for ROUTER1 would have a very simple routing table.

The primary advantage of RIP and RIP II is that both of them are much easier to implement and administer than OSPF; this can be a significant determining factor in choosing which routing protocol to use in a given implementation. Administrators in small network environments often do not have the bandwidth (in terms of time or attention) to dedicate themselves to understanding the implementation details of more elaborate and complex routing protocols, and therefore, are justified (and correct) in implementing RIP in their networks.

RIP creates a complete routing table for all routers speaking RIP on the internetwork. Best of all, the routing table is done automatically by periodic updates that are traded between routers throughout the network. This periodic update is one of RIP's shortcomings. The timeframe for these intervals is thirty seconds (and failures are concluded only after many of these intervals pass), and they are sent out across the network using broadcasts. This method has two drawbacks.

The first drawback: Because RIP waits three minutes before considering a router down (and sends out challenges to find out whether the router is truly down), several minutes can pass before data destined for a network with a dead router link is rerouted. During that period, all data is dropped into the black hole of the dead router...never escaping, never being forwarded, and causing bad things to happen. In networks, several minutes is an awfully long time; such a lengthy recovery interval is considered slow convergence.

The second drawback: Using broadcasts creates unwanted network traffic. In large networks this is a significant drawback, but an even worse situation is using RIP over WAN links. The bandwidth available on a WAN link is generally precious, and having to share such limited bandwidth with a chatty routing protocol that sends out broadcasts every 30 seconds is considered expensive. Certain routers can modify their behavior to dull this effect if the router knows it is using a WAN link, but such configurations must be made manually. RIP is therefore not a good choice where precious WAN links are in place within the network, or in large networks.

There are some other problems with RIP—it uses a hop count that is independent of the TTL field of an IP packet and has a maximum of 15 hops, after which the packet is dropped. In large networks there could certainly be more than 15 hops; if there were a destination IP address sitting 16 hops from the source in a RIP network, and even if there were appropriate router connections between the source and destination machines, RIP would drop the packets on the 16th hop, resulting in an error that said something like “destination host unreachable.” Another disadvantage of RIP is that, although it stores multiple entries for equal-cost routes to a destination, RIP uses only the first route in its list, which results in a lack of load balancing between like-cost routes.

RIP II addresses some of the glaring deficiencies of RIP. Most significant among those deficiencies were RIP’s inability to identify a subnet mask, the load it placed on the network by the use of multicasts, and the ability to use simple password authentication. RIP II, however, is still a Distance Vector protocol and does not have some of the rich routing features available in Link State protocols available today. RIP II retains the ease of use and implementation, and for that reason RIP II can still be a viable solution for small networks. RIP II, though attractive for some, is still not as attractive from a feature standpoint as Link State protocols.

RIP for IPX

IPX is a different animal altogether than IP (but you knew this already). The way IPX implements some of its features, and the way it maintains information about its features across the network, necessitates a different approach to routing than the simple RIP versions we saw earlier for IP. A little bit of background information on IPX and the way NetWare servers advertise their services is in order.

IPX uses something called Service Advertising Protocol (SAP) to maintain a list of available services being offered by Novell NetWare Servers on a given subnet. For simple (non-routed) network deployments, such advertisements are easy; the server itself can maintain and respond to service requests by sending or responding to broadcasts sent over the local network. But in a routed network, management of such advertisements (these server-related services, such as file server services, can be shared throughout a routed network) must be done at the router because broadcasts don’t go beyond the local subnet (meaning that routers don’t forward broadcasts). These facts—that IPX uses broadcasts to advertise their services, and that routers don’t forward broadcasts—require that any IPX routing protocol manage and appropriately forward SAP broadcasts to servers (responding to requests) and clients (making requests). RIP for IPX manages such IPX-related issues within its protocol.

Neither Windows NT nor Windows 2000 uses SAP, except where NetWare-like services are provided, such as File and Print Services for NetWare (FPNW) or services like Microsoft SQL Server, where IPX clients would only be aware of and find services with help from SAP.

Other than this distinct difference (and the obvious difference in network addresses), RIP for IP and RIP for IPX work in similar ways; both use flooding, both maintain and share routing tables, and both update using broadcasts on specific intervals.

OSPF

Open Shortest Path First (OSPF) is a routing protocol that takes things like bandwidth availability and network congestion into consideration when determining the best route to send packets across the network. OSPF works in IP networks and is a Link State protocol, making it an attractive solution for large networks or networks that incorporate WAN links into their topology. OSPF is the most complex of the routing protocols, but don't let anyone fool you: When taken in bite-sized pieces, OSPF is straight forward.

OSPF differs from RIP in a number of ways. The first and perhaps most obvious distinction is that RIP is Distance Vector–based, while OSPF is Link State–based. This distinction is important; it means that OSPF can react to changes in network utilization on given links and reroute around the increased traffic—dynamically. Without manual, administrative intervention, RIP has no way of doing this. Such on-the-fly modifications based on network traffic are often called *load balancing*.

OSPF is a hierarchical protocol, just like IP, DNS, and Windows 2000 domains are hierarchical. A hierarchical protocol enables groups of subnetworks to be addressed from a top down perspective, with a "top" network responsible for addressing a group of subnetworks, and each of those subnetworks capable of having subnetworks within themselves. For example, I could have a network with a "top" address of 122.0.0.0 and have routing outside that network send anything destined for 122.x.x.x subnetworks sent to the router servicing that group. Within the group of 122.x.x.x networks I could have a subnetwork of 122.46.x.x and have all subnetworks planned therein (122.46.17.x, for example) reachable through that router. Figure 5-9 provides a visual representation of a hierarchical protocol.

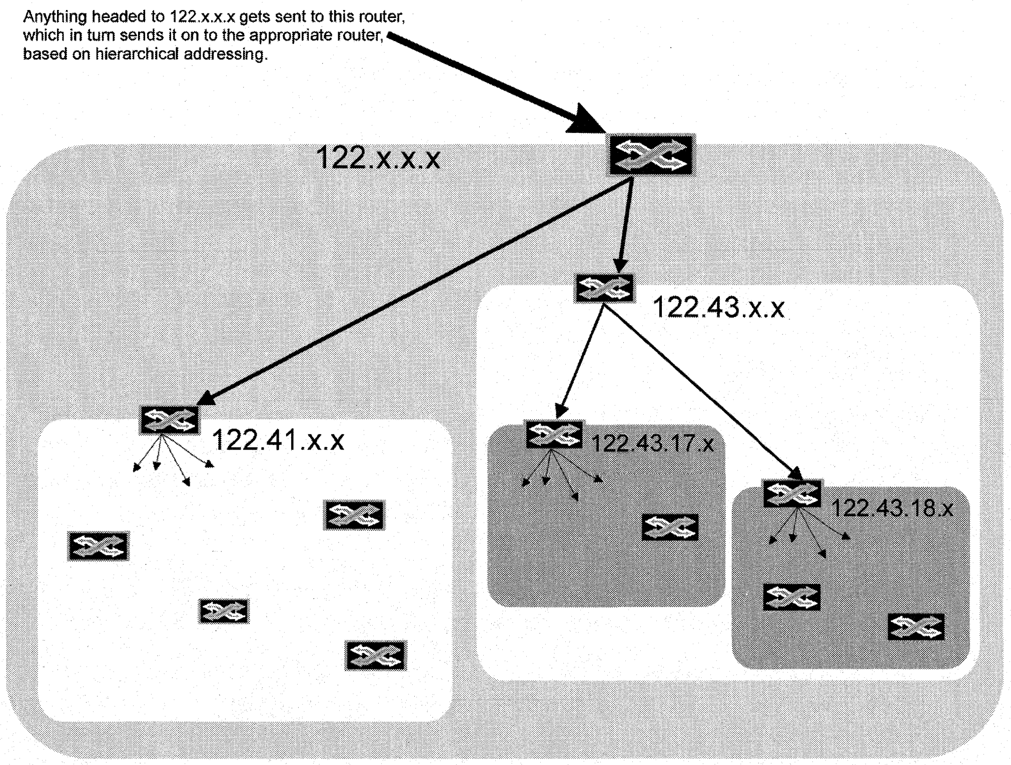

Anything headed to 122.x.x.x gets sent to this router, which in turn sends it on to the appropriate router, based on hierarchical addressing.
122.x.x.x
122.43.x.x
122.41.x.x
122.43.17.x
122.43.18.x

Figure 5-9: Representation of a Hierarchical Protocol.

Because OSPF is a widely used protocol, knowledge of how it functions and operates is important to routing development (and routing administration, for that matter). As such, I'm going to go into more detail about OSPF in this chapter so that you can have reasonable (if not introductory) understanding of how OSPF operates.

Let's start with an overview of how OSPF works on a system-wide level, then move into its interaction on a local level with neighboring routers and explain how such interaction makes OSPF such an attractive large-network routing protocol.

The Overall View of OSPF

OSPF works under the premise of Autonomous Systems (AS), and can further segment this network-wide organizational unit into smaller, easier to manage groups called *Areas*. An AS, as the name suggests, is the highest level of organization for an independent network; Areas are groups of networks within an AS that work as one administrative, routing-area unit. In every OSPF AS there must be one area called the Backbone—the mother of all Areas and the administrative and data-passing center of the networking universe for the overall AS. The backbone is the central nervous system of your AS, and all routes (if possible) should converge on or stem from your OSPF Backbone. The reasoning behind this is that, ideally, when routing data between areas the network will use the Backbone.

Once Areas are established—which by definition are separate entities within the AS—there must be a means by which these Areas can communicate with one another, with the Backbone, and with the outside world. These means are accomplished through designating different routers with different roles. There are a few types of router roles in an OSPF network:

Internal Routers: These are routers that function within an Area and do not have interfaces to segments or networks outside the area in which they reside.

Area Border Routers: Communication between Areas is done through an Area Border Router, which is a router that is attached to two Areas; it keeps information about each Area to which it has an interface, and communicates that information to the Backbone.

Backbone Routers: Similar to Area Border Routers are Backbone Routers. Backbone Routers are, quite simply, routers that have at least one interface on the Backbone.

Autonomous System/AS Boundary Router: Finally, there often are instances when hosts on a network need access to areas outside the AS, such as the Internet. In these situations, the network administrator designates one router as an AS Boundary Router, which connects your AS to networks that are outside your AS. It could be that your company has more than one AS, and thus you will have at least one—and much more responsibly and likely will have more than one—router in your network that will be designated an AS Boundary Router. Figure 5-10 illustrates the various types of routers.

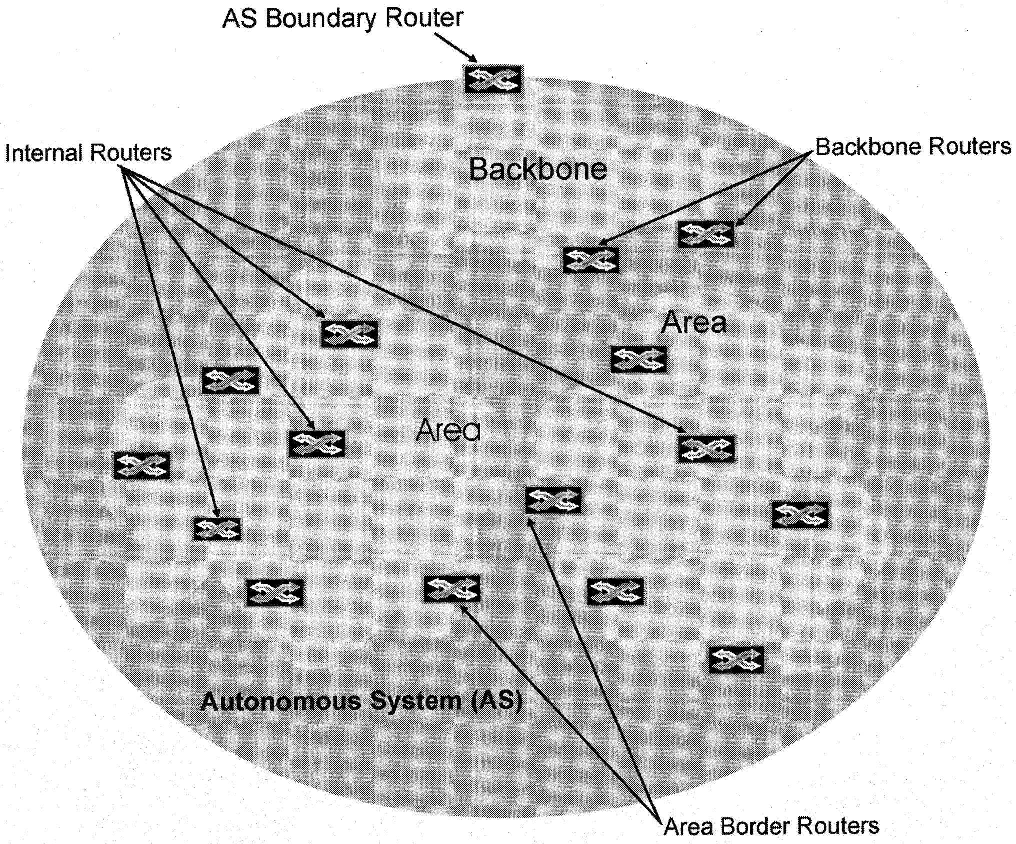


Figure 5-10: An AS with Areas and Router Role-Holders Identified.

OSPF uses a database called a Link State Database (LSDB) to maintain information about its network. The LSDB is a "map" of the entire network, and being thus, needs information from every other router in the internetwork in order to make it complete. The internetwork is an Area if the AS has been segmented; if it has not been segmented into areas, the LSDB will need information about every router in the AS. This is one reason segmenting medium or large networks into Areas is such an advantage; groups of networks placed into manageable groups cuts down on the requisite processing and traffic overhead associated with too many routers residing in the same management unit (AS or Area).

From this LSDB, routers calculate the Shortest Path First (SPF) Tree, which is a map of how to get to every router on the network (and thus every segment) relative to the location of the calculating router. OSPF better utilizes multiple routes than RIP, and has a more complete picture of the internetwork. Throughout the internetwork (whether that is an Area if so segmented, or an AS if not), the LSDB is identical on every router, and the SPF Tree for each router is unique.

The Local View of OSPF

OSPF has a logical, traffic-sensitive and redundancy-sensitive way of going about its operation. This method doesn't change once you scrutinize to the local router-to-router operation of OSPF (what I'm calling the local view of OSPF). The way OSPF routers exchange information, govern their behavior with adjacent routers, and ensure that communication between routers is kept to a minimum is fairly detailed and beyond the scope of this chapter. Instead, this section focuses on providing a good explanation of how these communications and relationships work. We've seen how OSPF segments responsibilities and processing requirements in the overview—the same applies at the local level. Consider Figure 5-11.

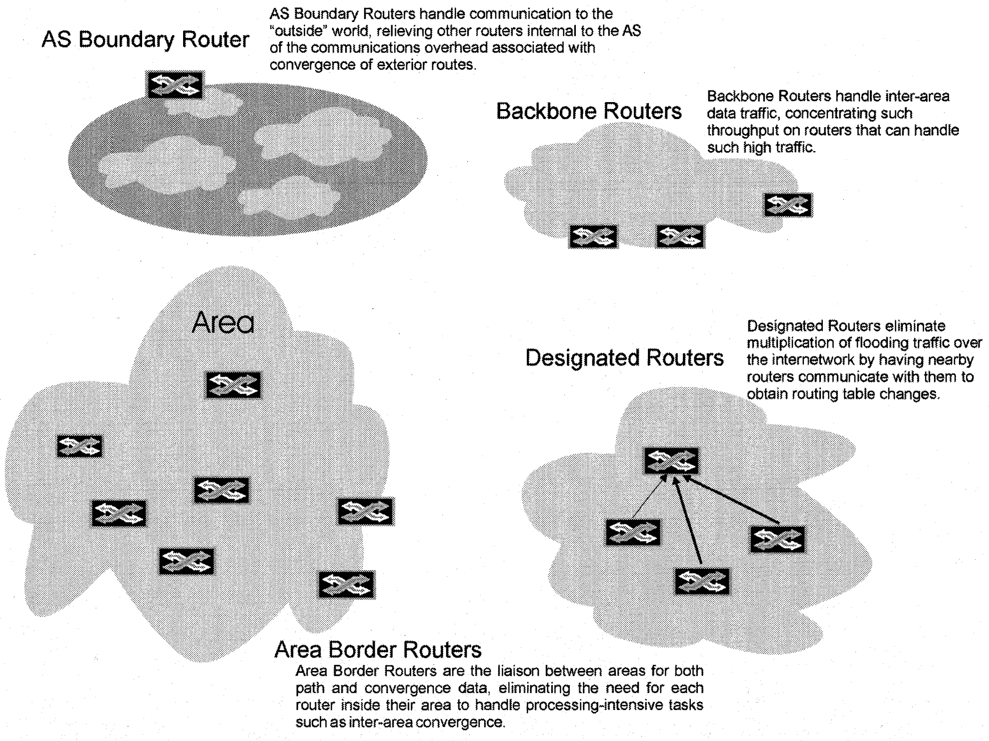

AS Boundary Router
AS Boundary Routers handle communication to the "outside" world, relieving other routers internal to the AS of the communications overhead associated with convergence of exterior routes.
Backbone Routers
Backbone Routers handle inter-area data traffic, concentrating such throughput on routers that can handle such high traffic.
Area
Designated Routers
Designated Routers eliminate multiplication of flooding traffic over the internetwork by having nearby routers communicate with them to obtain routing table changes.
Area Border Routers
Area Border Routers are the liaison between areas for both path and convergence data, eliminating the need for each router inside their area to handle processing-intensive tasks such as inter-area convergence.

Figure 5-11: OSPF's Tree-Like Segmentation of Responsibilities.

To understand the smarts behind OSPF's local policies, we need to compare what probably seems obvious: the definition of a routed network.

In a contiguous network environment, every subnet is in some way connected to the internetwork, which is the basic definition of an internetwork. To attain such connections, routers connect these subnets, so by virtue of their inclusion, all routers are also interconnected through one link or another. Thus, every router in the network is connected in some way to at least one other router (unless, of course, you only have one router in your network, in which case you shouldn't be using OSPF). To put this another way, every router on the internetwork has at least one "neighbor" router. Most networks have some sort of redundancy built into them, which means that routers on a given wire (network connection media) often have more than one neighbor. Regardless of how many neighbors a router has, every router on the network has at least one neighbor. Think of it this way: If you have a single-file line of people, and you instruct them to shake hands with either the person in front or behind them (shaking hands with more than one person is okay, so long as everyone shakes at least one other person's hand), everyone in the line will have shaken hands with someone.

Here is the point at which the logic built into OSPF shines and where some of the complexity tends to make people shy away from OSPF.

Routers in an OSPF environment form an *adjacency* with neighboring routers that share certain common criteria (specifically authentication, passwords, Hello and Dead intervals, Area IDs and Stub/non-Stub status). In fact, an OSPF router *must* form an adjacency with one of its neighbors in order to be considered part of the Area/AS. With this adjacency-forming requirement, OSPF ensures at least one adjacency for every router participating in its network; OSPF routers need only synchronize their LSDBs with adjacent routers. If all routers on the network are synchronized with their adjacent routers, the entire network is synchronized—or more accurately, the entire network has converged. Imagine the network traffic and computational overhead avoided by having only adjacent routers synchronize with each other, instead of having every router in the network synchronize with every other router in the network (it's a lot).

I'll provide a quick example to clarify why this is so cool. Remember the line of people who shook hands with each of their neighbors? That was a pretty easy example, because they're all in a line and it's easy to visualize that everyone would have someone with which they could shake hands. Let's take that example a step further and say that this line of people is in a huge room (a gym, perhaps). Now tell them they are to spread out across the gym, but that they must stay within an arm's reach of at least one person (being within an arm's reach of more than one person is okay as well). Chaos ensues for a few minutes while everyone shuffles until everyone is spread out until the gym is full and everyone has complied and is within arm's reach of at least one person. Now you (the instructor) walk into the middle of this gathering and step between two people who are within arm's reach and tell them, "I have a word I want you to whisper to everyone within arm's reach of you, and everyone you tell is to whisper that word to everyone within arm's reach of them." You tell them the word—watermelon—and the whispering begins. Eventually everyone in the gym knows the word. You can step between any two other people or any one person (remember, they only have to be within arm's reach of one person) and do the same thing with a new word; eventually everyone in the gym knows the new word. This is exactly how needing to synchronize only with adjacent routers ensures synchronization of the entire network's LSDB.

This is an excellent system, but it has one drawback: Routers on broadcast networks (such as Ethernet) will create adjacencies with more than one or two routers, potentially a whole bunch of routers. This goes crazy on broadcast networks. OSPF gets around this problem with *Designated Routers*.

A Designated Router is the router on a multiple-access network that instructs all other routers on the multiple-access network to create adjacencies—and therefore synchronize their LSDB—only with it. The Designated Router is determined by comparing Router Priorities (router priority is a defined term in OSPF); routers with a Router Priority of zero never become a Designated Router. Among those routers with a Router Priority greater than zero, the router with a higher priority becomes the Designated Router for that multiple-access network. There is also a Backup Designated Router, just in case the Designated Router goes down. Figure 5-12 depicts how the use of Designated Routers might appear in a network deployment.

OSPF routers form adjacencies with neighboring routers upon startup or initialization. Designated Routers must have a Router Priority of one or greater.

Communication Between OSPF Routers

OSPF routers maintain communication between one another through the use of *Hello packets*. Hello packets are small and have default settings generally around 10 to 15 seconds that facilitate continuous communication—and thus knowledge of ongoing availability—between routers. There are two time intervals associated with the exchange of Hello packets: *Hello intervals* and *Dead intervals*. Hello intervals are equivalent to "are you still there?" communication. Dead intervals are equivalent to "if I don't hear from you within my Dead interval time, I'm going to assume you're no longer up and modify my LSDB (and thus my SPF Tree) appropriately" communication.

Hello intervals and Dead intervals are configurable in OSPF. If routers are on a network that is particularly busy, or there is a router that's doing so much routing that setting a Hello interval at 10 or 15 seconds would create unwanted stress on the router or network, these intervals can be increased. Be careful, though: The idea behind making these intervals relatively short is to keep convergence time short, and a short convergence time is one of the greatest advantages of OSPF. If you increase the Hello and Dead intervals too much, you'll be undoing one of OSPF's best features and the network won't be able to react quickly to changes in router topology.

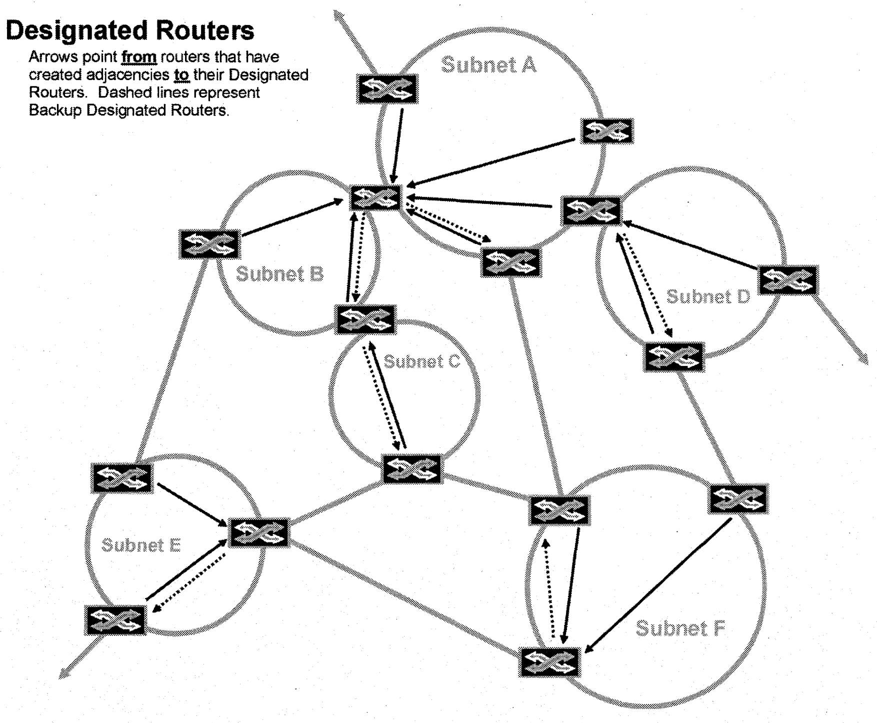

Designated Routers
Arrows point from routers that have created adjacencies to their Designated Routers. Dashed lines represent Backup Designated Routers.
Subnet A
Subnet B
Subnet C
Subnet D
Subnet E
Subnet F

Figure 5-12: Designated Routers and Backup Designated Routers in Multiple-Access Networks.

Once the Dead interval expires and an OSPF router determines that one of its neighbors is dead, the router communicates this fact to all of its Adjacent routers. Remember that OSPF routers synchronize their LSDBs with Adjacent routers; if a dead router is detected, the detecting Adjacent router changes its LSDB and its LSDB falls out of sync with its neighboring routers' LSDBs. Because the changed LSDB of the detecting router is more recent, Adjacent routers update their LSDBs with this new information and a chain reaction occurs—all neighbors within the AS or Area resynchronize (converge) and within a certain amount of time (dependent upon the size of the network and the speed of your routers' CPUs, but something like 30 seconds or so), the LSDB for the internetwork once again is converged and the downed router's impact on the network is taken into consideration.

CHAPTER 6

Router Administration

Router Administration Overview

The router administration API allows developers to create applications to manage the router service on a computer running Microsoft® Windows® 2000, or running Microsoft® Windows NT® 4.0 with the Routing and RAS (RRAS) add-on installed. Not all API functions are supported on both of these platforms. The Requirements section on the reference page for each router administration function specifies which platforms support that function.

Components of the Router Architecture

The router administration documentation makes frequent reference to the following components of the router.

Dynamic Interface Manager (DIM)

All router administration functions pass through DIM. Depending on the nature of the function, DIM may pass the call on to one of the router managers. Functions that deal only with interfaces are handled by DIM. If the function affects routing protocols, DIM will call into the router manager for the transport corresponding to that protocol. For example, if the function affects the Open Shortest Path First (OSPF) protocol, DIM will call into the IP Router Manager, since OSPF is an IP routing protocol.

Router Managers

Each routable transport has its own router manager. Currently router managers exist for the IP and IPX transports. The router managers manage router protocols and other types of routing clients that run on interfaces on the local computer.

Routing Protocols and other Clients

Clients are service providers that function within the framework of the router architecture. Routing protocols are one type of client that is supported by the router.

Clients are specific to a particular routable transport: either IP or IPX. Routing protocols for the IP transport include Open Shortest Path First (OSPF) and Routing Information Protocol (RIP). Examples of IPX routing protocols are Service Advertising Protocol (SAP) and RIP for IPX. An example of a client that is not a routing protocol is Network Address Translation (NAT) for IP.

The interface between the router manager and a client is described in the section **Routing Protocol Interface**. All clients conform to this interface. Using this interface, vendors can implement their own clients that are compatible with the router.

Interfaces

An interface is a connection to an external network. Each interface is identified by a unique interface *index*. Interfaces are logical entities; router clients such at NAT or OSPF deal with all types of interfaces similarly. In terms of implementation however, an interface can represent a dedicated connection (such as to a Local Area Network (LAN)) or a non-dedicated, dial up connection (such as a PPP connection to a Wide Area Network (WAN)).

In the case of a LAN interface, the interface corresponds to an actual physical device in the computer, a LAN adapter. In the case of a WAN interface, the interface is mapped to a port at the time a connection is established. The port could be a COM port, a parallel port or a virtual port (for tunnels such as PPTP and L2TP).

WAN interfaces have the additional quality that they typically receive a network address only at the time that a connection is established. For example, a WAN interface using PPP receives its network layer address from the remote peer during the connection process. Receiving a network address as part of the connection process is sometimes referred to as "late-binding."

Router Initialization

Configuration information for the router, the router managers and the routing protocols/clients is divided into global information and per interface information and is stored in the registry and the router's phonebook file, router.pbk.

When the router process starts, DIM (Dynamic Interface Manager) reads the router configuration from the registry. DIM creates the interfaces that are specified by the interface information.

DIM also retrieves the global router manager information. DIM starts the router managers corresponding to this information, and passes them the information. For example, if DIM finds global information for the IP router manager in the registry, DIM will start the IP router manager and pass it the global information. If no global information is present in the registry for a particular router manager, DIM will not start that router manager.

The router managers examine the global information received from DIM. If the router manager finds information specific to a particular client within the global information, the router manager will load the DLL for the client (for example ipNAT.dll) and initialize the client by calling the client's **RegisterProtocol** and **StartProtocol** functions. The router manager passes the client-specific global information to the client in the call to **StartProtocol**.

At each stage, the information being passed to the next entity is opaque to the entity above it. That is, DIM does not interpret the global information for the IP Router Manager, beyond the fact that the information is meant for the IP Router Manager. Similarly, the IP Router Manager does not interpret the OSPF specific information beyond the fact that it is OSPF information.

Router Management Functions

The following sections discuss the different types of router management functions and what you should know to use them effectively.

All router management functions require administrator privilege. A user in the Power User group will not have sufficient privilege to use the router management functions.

The Different Classes of Router Management Functions

The router management functions can be divided up into the administration functions and the configuration functions. The administration functions have a prefix of **MprAdmin** and the configuration functions have a prefix of **MprConfig**. Despite the naming, both sets of functions are used for router management. The **MprAdmin** functions operate directly on the running router. The **MprConfig** functions have similar functionality, but operate on the router configuration stored in the registry. Both types of functions pass information blocks.

The router management functions can also be divided up based on what components of the router they manage: interfaces, router managers, or router manager clients.

The router interface functions have a prefix of either **MprAdminInterface** or **MprConfigInterface**. Use these functions to access interfaces. The router manager functions have a prefix of **MprAdminTransport** or **MprConfigTransport**. Use these functions to access the router managers. Lastly, the router manager client functions have a prefix of **MprAdminInterfaceTransport** or **MprConfigInterfaceTransport**. Use these functions to access the clients running on the router.

A subset of MprAdmin functions is the MprAdminMib functions. These also operate on the running route alone. However, these functions do not pass information blocks. These functions provide additional flexibility to the protocol designer, especially for retrieving non-configuration information, such as statistics.

Ensuring that Changes Occur Immediately and are Persistent

A developer can make changes to the router configuration directly using the router configuration functions. However, any changes made to the configuration will not take effect until the router is restarted, since this is the only time that DIM reads the configuration from the registry.

A developer can make changes to the running router by using the router administration functions. However, these changes are not persistent: since they haven't been written to the registry, they will be lost if the router is restarted.

In order to make changes that are both immediate and persistent, a developer will need to use both the router administration and the router configuration functions. If the router is not running, the developer need only call the appropriate router configuration functions.

For **querying** information from the running router, use the router administration functions. If the router is not running, query information using the router configuration functions.

Using Router Administration and Configuration Functions Remotely

Most of the router administration and configuration functions can be called on a computer other than the one being administered. These functions take as a parameter, a handle to the router service or configuration to administer. The administration functions use RPC (Remote Procedure Call) to communicate with the routing service specified by the handle. The configuration functions write to and read from the registry of the computer specified by the handle.

To administer the routing service on a remote machine first call **MprAdminIsServiceRunning** to verify that the service is running. Then call **MprdminServerConnect** to obtain the handle. If the router service is not running on the remote machine, all router administration ("MprAdmin") calls will fail.

To make changes to the router configuration on a remote machine obtain a handle by calling the **MprConfigServerConnect** function.

Router Interface Functions

Use the following functions to administer interfaces on the router.

Administration Function	Configuration Function
MprAdminInterfaceCreate	**MprConfigInterfaceCreate**
MprAdminInterfaceDelete	**MprConfigInterfaceDelete**
MprAdminInterfaceEnum	**MprConfigInterfaceEnum**
MprAdminInterfaceGetHandle	**MprConfigInterfaceGetHandle**
MprAdminInterfaceGetInfo	**MprConfigInterfaceGetInfo**
MprAdminInterfaceSetInfo	**MprConfigInterfaceSetInfo**

These functions affect the interfaces themselves, not clients running on the interfaces. For this reason, none of the functions require the caller to specify a particular transport (IP or IPX); although clients (such as routing protocols) are associated with particular transports, the interfaces themselves are not.

These functions are handled directly by DIM. They do not utilize the router managers.

The **MprAdminInterfaceCreate** and **MprAdminInterfaceDelete** functions cannot create or delete LAN interfaces. They can only create or delete demand-dial interfaces. See **ROUTER_INTERFACE_TYPE** for a list of interface types.

Router Manager (Transport) Functions

Use the following functions to administer the router managers. These functions also allow a developer to read and write the global information for the router managers, and the global information for router clients (such as routing protocols).

Administration function	Configuration function
MprAdminTransportCreate	**MprConfigTransportCreate**
No administration function	**MprConfigTransportDelete**
MprAdminTransportGetInfo	**MprConfigTransportGetInfo**
MprAdminTransportSetInfo	**MprConfigTransportSetInfo**
No administration function	**MprConfigTransportEnum**
No administration function	**MprConfigTransportGetHandle**

Router Manager Client (InterfaceTransport) Functions

Use the following functions to administer clients (such as routing protocols) on particular interfaces. These functions also allow a developer to read and write interface-specific information for router clients (such as routing protocols).

Administration function	Configuration function
MprAdminInterfaceTransportAdd	**MprConfigInterfaceTransportAdd**
MprAdminInterfaceTransportRemove	**MprConfigInterfaceTransportRemove**
MprAdminInterfaceTransportGetInfo	**MprConfigInterfaceTransportGetInfo**
MprAdminInterfaceTransportSetInfo	**MprConfigInterfaceTransportSetInfo**
No administration function	**MprConfigInterfaceTransportEnum**
No administration function	**MprConfigInterfaceTransportGetHandle**

MprInfo Functions and Information Headers

The following functions require that the caller pass an information structure or *header* as one of the parameters.

Administration function	Configuration function
No administration function	**MprConfigTransportCreate**
MprAdminTransportSetInfo	**MprConfigTransportSetInfo**
MprAdminInterfaceTransportSetInfo	**MprConfigInterfaceTransportSetInfo**
MprAdminInterfaceTransportAdd	**MprConfigInterfaceTransportAdd**

Similarly, the following functions return information headers.

Administration function	Configuration function
MprAdminTransportGetInfo	**MprConfigTransportGetInfo**
MprAdminInterfaceTransportGetInfo	**MprConfigInterfaceTransportGetInfo**

For the transport functions, the information header contains global information for the transport. For the client ("InterfaceTransport") functions, the header contains information specific to the client (for example, OSPF) being administered.

The information headers and their contents should be manipulated only by using the MprInfo functions. Developers should not attempt to manipulate the contents of the information headers directly.

The "interface-only" functions such as **MprAdminInterfaceSetInfo** do not require the use of MprInfo functions. The information that is passed and returned with these functions is always in the form of an **MPR_INTERFACE** structure.

Managing Router Clients and Interfaces

The following topics describe how to perform typical management tasks using the MprAdmin and MprConfig and MprInfo functions:

- **Changing Interface-Specific and Global Information for Clients**
- **Deleting a Client from an Interface**

Changing Interface-Specific and Global Information for Clients

To change the interface information for a specific client, for example NAT, first use the appropriate "GetInfo" function to retreive the current information. If the router is running, use **MprAdminInterfaceTransportGetInfo**. If the router is not running, use the **MprConfigInterfaceTransportGetInfo**. This call will retrieve the information for all the clients running on the specified interface. For example, if both OSPF and RIP are running on a particular interface, this call will retrieve the interface information for both. Use the **MprInfoBlockFind** function to locate the information block corresponding to the client you want to modify. Then use the **MprInfoBlockSet** functions to perform the modifications. Lastly, use either **MprAdminInterfaceTransportSetInfo** or **MprConfigInterfaceSetInfo** to make the changes either to the running router or the router configuration in the registry.

Global client information is information that is not specific to any particular interface on which the client is running. Use a similar procedure to modify global information for a specific client. First retrieve the global information for all the clients using **MprAdminTransportGetInfo** or **MprConfigTransportGetInfo**. Then use the MprInfo functions to modify the information. Lastly, use the **MprAdminTransportSetInfo** or **MprConfigTransportSetInfo** functions to "save" the modified information back to either the running router or the registry.

Calls to the above administration functions go through the Dynamic Interface Manager (DIM), and eventually translate into calls from the router manager to the clients themselves. All clients, whether or not they are routing protocols, must conform to the interface described in the section **Router Protocol Interface**. As part of this interface, the routing protocol must support the following functions (among others):

- **GetGlobalInfo**
- **SetGlobalInfo**
- **GetInterfaceInfo**
- **SetInterfaceInfo**

The router manager calls the **GetInterfaceInfo** functions for each of the clients to gather the information that is returned from a call to **MprAdminInterfaceTransportGetInfo**. Similarly, when the router manager receives updated information via **MprAdminInterfaceTransportSetInfo** call, it uses the **SetInterfaceInfo** functions to update the interface information for each of the clients.

Deleting a Client from an Interface

To delete a client, such as a routing protocol, from a particular interface, use either **MprAdminInterfaceTransportGetInfo** or **MprConfigInterfaceTransportGetInfo** to retrieve all the client information for the interface. Use **MprInfoBlockRemove** to remove the information block for the client to be deleted. Then use **MprInfoBlockAdd** to add a zero-length block for the client to be deleted. Finally, use **MprAdminInterfaceTransportSetInfo** or **MprConfigInterfaceTransportSetInfo** to "save" the information back to either the running router or the registry.

If the router manager receives a zero-length interface information block for a client, it knows to delete that client from the interface. The router manager will delete the client by calling the client's implementation of **DeleteInterface**. Note the important distinction between passing an information header that doesn't contain an information block for a client, and passing an information header that contains a zero-length information block for the client. In the first case, the router manager will take no action with respect to the client. In the second case, the router manager will delete the client from the interface.

Router Administration Reference

Use the following functions, structures, and enumerated types when developing software to administer Microsoft® Windows® 2000 routers:

Router Administration Functions

Use the functions on the following page when developing software to administer Microsoft® Windows® 2000 routers.

MprAdminBufferFree
MprAdminDeregisterConnectionNotification
MprAdminGetErrorString
MprAdminInterfaceConnect
MprAdminInterfaceCreate
MprAdminInterfaceDelete
MprAdminInterfaceDisconnect
MprAdminInterfaceEnum
MprAdminInterfaceGetCredentials
MprAdminInterfaceGetCredentialsEx
MprAdminInterfaceGetHandle
MprAdminInterfaceGetInfo
MprAdminInterfaceQueryUpdateResult
MprAdminInterfaceSetCredentials
MprAdminInterfaceSetCredentialsEx
MprAdminInterfaceSetInfo
MprAdminInterfaceTransportAdd
MprAdminInterfaceTransportGetInfo
MprAdminInterfaceTransportRemove
MprAdminInterfaceTransportSetInfo
MprAdminInterfaceUpdatePhonebookInfo
MprAdminInterfaceUpdateRoutes
MprAdminIsServiceRunning
MprAdminRegisterConnectionNotification
MprAdminServerConnect
MprAdminServerDisconnect
MprAdminServerGetInfo
MprAdminTransportCreate
MprAdminTransportGetInfo
MprAdminTransportSetInfo

MprAdminBufferFree

The **MprAdminBufferFree** function frees memory buffers returned by: **MprAdminInterfaceGetInfo**, **MprAdminInterfaceEnum**, **MprAdminServerGetInfo**, **MprAdminInterfaceTransportGetInfo**, and **MprAdminTransportGetInfo**.

```
DWORD MprAdminBufferFree(
  PVOID pBuffer    // memory buffer to free
);
```

Parameters

pBuffer
Pointer to the memory buffer to free.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is the following error code.

Value	Meaning
ERROR_INVALID_PARAMETER	The *pBuffer* parameter is NULL.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MprAdminInterfaceGetInfo**, **MprAdminInterfaceEnum**, **MprAdminServerGetInfo**, **MprAdminInterfaceTransportGetInfo**, **MprAdminTransportGetInfo**

MprAdminDeregisterConnectionNotification

The **MprAdminDeregisterConnectionNotification** function deregisters an event object that was previously registered using **MprAdminRegisterConnectionNotification**. Once deregistered, this event is longer signaled when an interface connects or disconnects.

```
DWORD MprAdminDeregisterConnectionNotification(
  MPR_SERVER_HANDLE hMprServer,    // handle to server
  HANDLE hEventNotification        // handle to event
);
```

Parameters

hMprServer
Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

hEventNotification
Handle to an event object to deregister. This event will no longer be signaled when an interface connects or disconnects.

Return Values

If the function is successful, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes.

Value	Meaning
ERROR_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_DDM_NOT_RUNNING	The Demand Dial Manager (DDM) is not running.
ERROR_INVALID_PARAMETER	The *hEventNotification* parameter is NULL or is an invalid handle.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **FormatMessage**, **MprAdminRegisterConnectionNotification**

MprAdminGetErrorString

The **MprAdminGetErrorString** function returns the string associated with a router error from mprerror.h.

```
DWORD MprAdminGetErrorString(
  DWORD dwError,                // router error code
  LPWSTR * lplpwsErrorString   // descriptive text for error
);
```

Parameters

dwError
The error code for a Windows 2000 router error.

lplpwsErrorString
Pointer to a **LPWSTR** variable that will point to the text associated with the *dwError* code on successful return. Free this memory by calling **LocalFree**.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_PARAMETER	The error code in *dwError* is unknown.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in Mprapi.h.
Library: Use Mprapi.lib.

See Also

Router Administration Reference, Router Administration Functions, **LocalFree**

MprAdminInterfaceConnect

The **MprAdminInterfaceConnect** function creates a connection to the specified WAN interface.

```
DWORD MprAdminInterfaceConnect(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  HANDLE hInterface,               // handle to interface
  HANDLE hEvent,                   // event to signal when
                                   // connection attempt is
                                   // complete
  BOOL fBlocking                   // flag to control
                                   // synchronous vs.
                                   // asychronous operation
);
```

Parameters

hMprServer
Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

hInterface
Handle to the interface. This handle is obtained from a previous call to **MprAdminInterfaceCreate**.

hEvent
Handle to an event that will be signaled after the attempt to connect the interface has completed. The function initiates the connection attempt and returns immediately. After the event is signaled, you can obtain the result of the connection attempt by calling **MprAdminInterfaceGetInfo**.

If this parameter is NULL, and *fBlocking* is TRUE, then this call is synchronous, that is, the function will not return until the connection attempt has completed.

The caller must specify NULL for this parameter, if *hMprserver* specifies a remote router.

fBlocking
If *hEvent* is NULL and this parameter is set to TRUE, the function will not return until the connection attempt has completed.

If *hEvent* is NULL, and this parameter is set to FALSE, the function will return immediately. A return value of **PENDING** indicates that the connection attempt was initiated successfully.

If *hEvent* is not NULL, this parameter is ignored.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_ALREADY_CONNECTING	A connection is already in progress on this interface.
ERROR_DDM_NOT_RUNNING	The Demand Dial Manager (DDM) is not running.
ERROR_INTERFACE_DISABLED	The interface is currently disabled.
ERROR_INTERFACE_HAS_NO_DEVICES	No adapters are available for this interface.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.
ERROR_SERVICE_IS_PAUSED	The Demand Dial service is currently paused.
PENDING	The interface is in the process of connecting. The caller should wait on the *hEvent* handle, if one was specified. After the event is signaled, you can obtain the state of the connection and any associated error can by calling **MprAdminInterfaceGetInfo**.

Remarks

The following table summarizes the relationship between *hEvent* and *fBlocking*.

hEvent	fBlocking	Result
Event Handle	Ignored	The call returns immediately. A return value of PENDING indicates that the attempt was initiated successfully. Wait on *hEvent*. When *hEvent* is signalled, use **MprAdminInterfaceGetInfo** to determine the success or failure of the connection attempt.
NULL	TRUE	The call will not return until connection attempt has completed.
NULL	FALSE	The call returns immediately. A return value of PENDING indicates that the attempt was initiated successfully.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MprAdminInterfaceCreate**, **MprAdminInterfaceDisconnect**, **MprAdminInterfaceGetInfo**, **MprAdminServerConnect**

MprAdminInterfaceCreate

The **MprAdminInterfaceCreate** function creates an interface on a specified server.

```
DWORD MprAdminInterfaceCreate(
  MPR_SERVER_HANDLE hMprServer,   // handle to router
  DWORD dwLevel,                  // level of information provided
  LPBYTE lpBuffer,                // info that describes interface
  HANDLE * phInterface            // handle to the interface
);
```

Parameters

hMprServer
Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

dwLevel
Level of the information passed in *lpBuffer*. Must be either zero.

Windows 2000 and later: This parameter may have a value of one. A value of one indicates that the *lpBuffer* parameter points to an **MPR_INTERFACE_1** structure.

lpBuffer
Pointer to an **MPR_INTERFACE_0** structure that contains the information to create the interface. The *hInterface* member of this structure is ignored.

Windows 2000 and later: The *lpBuffer* parameter may point to either an **MPR_INTERFACE_0** or **MPR_INTERFACE_1** structure. The type of structure should be indicated by the value of the *dwLevel* parameter.

phInterface
Pointer to a **HANDLE** variable. On successful return, the variable contains a handle to use in all subsequent calls to manage this interface.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_DDM_NOT_RUNNING	The router interface type specified in the **MPR_INTERFACE_0** or **MPR_INTERFACE_1** structure is not supported because the Dynamic Interface Manager is configured to run only on a LAN.
ERROR_INTERFACE_ALREADY_EXISTS	An interface with the same name already exists.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NOT_SUPPORTED	The *dwLevel* value is invalid.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MPR_INTERFACE_0**, **MPR_INTERFACE_1**, **MprAdminInterfaceDelete**, **MprAdminServerConnect**

MprAdminInterfaceDelete

The **MprAdminInterfaceDelete** function deletes an interface on a specified server.

```
DWORD MprAdminInterfaceDelete(
  MPR_SERVER_HANDLE hMprServer,   // handle to the router
  HANDLE hInterface               // handle to the interface
);
```

Parameters

hMprServer
A handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

hInterface
Handle to the interface to delete. Obtain this handle by calling **MprAdminInterfaceCreate**.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INTERFACE_CONNECTED	The interface specified is a demand dial interface and is currently connected.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MprAdminInterfaceCreate**, **MprAdminServerConnect**

MprAdminInterfaceDisconnect

The **MprAdminInterfaceDisconnect** function disconnects a connected WAN interface.

```
DWORD MprAdminInterfaceDisconnect(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  HANDLE hInterface                // handle to interface
);
```

Parameters

hMprServer
Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

hInterface
Handle to the interface. This handle is obtained from a previous call to **MprAdminInterfaceCreate**.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_DDM_NOT_RUNNING	The Demand Dial Manager (DDM) is not running.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.
ERROR_INTERFACE_NOT_CONNECTED	This interface is not connected. Therefore, it cannot be disconnected.

! Requirements

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

+ See Also

Router Administration Reference, Router Administration Functions, **MprAdminInterfaceCreate**, **MprAdminInterfaceConnect**, **MprAdminServerConnect**

MprAdminInterfaceEnum

The **MprAdminInterfaceEnum** function enumerates all the interfaces on a specified server.

```
DWORD MprAdminInterfaceEnum(
  MPR_SERVER_HANDLE hMprServer,     // handle to the router
  DWORD dwLevel,                    // level of information
                                    // provided
  LPBYTE * lplpbBuffer,             // array of information
                                    // structures
  DWORD dwPrefMaxLen,               // maximum length of data
                                    // to return
  HANDLE lpdwEntriesRead,           // number of interfaces
                                    // enumerated
  HANDLE lpdwTotalEntries,          // number of interfaces
                                    // that could
                                    // have been enumerated
  LPDWORD lpdwResumeHandle          // handle for continuing
                                    // the enumeration
);
```

Parameters

hMprServer

Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

dwLevel

Level of the information passed in *lpBuffer*. Must be zero.

Windows 2000 and later: This parameter may have a value of one. A value of one indicates that the *lplpBuffer* parameter points to an array of **MPR_INTERFACE_1** structures.

lplpbBuffer

Pointer to a pointer variable that will point to an array of **MPR_INTERFACE_0** structures on successful return. This memory should be freed by the **MprAdminBufferFree** call.

Windows 2000 and later: The pointer variable may point to an array of either **MPR_INTERFACE_0** or **MPR_INTERFACE_1** structures. The type of the structures should be indicated by the value of the *dwLevel* parameter.

dwPrefMaxLen

Specifies the preferred maximum length of returned data (in 8-bit bytes). If this parameter is –1, the buffer returned will be large enough to hold all available information.

lpdwEntriesRead

Pointer to a **DWORD** variable. On successful return, this variable contains the total number of interfaces that were enumerated from the current resume position.

lpdwTotalEntries

Pointer to a **DWORD** variable. On successful return, this variable contains the total number of interfaces that could have been enumerated from the current resume position.

lpdwResumeHandle

Pointer to a **DWORD** variable. On successful return, this variable contains a resume handle that can be used to continue the enumeration. The handle should be zero on the first call, and left unchanged on subsequent calls. If the return code is **ERROR_MORE_DATA** then the call may be re-issued with the handle to retrieve more data. If on return, the handle is NULL, the enumeration cannot be continued. For other types of error returns, this handle is invalid.

This parameter is optional. If the caller specifies NULL for this parameter, the function will not return a resume handle.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_MORE_DATA	More information is available; the enumeration can be continued.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NOT_SUPPORTED	The value of *dwLevel* is invalid.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MPR_INTERFACE_1**, **MprAdminBufferFree**, **MprAdminServerConnect**

MprAdminInterfaceGetCredentials

Use the **MprAdminInterfaceGetCredentials** function to retrieve the domain, username, and password for dialing out on the specified demand-dial interface.

```
DWORD MprAdminInterfaceGetCredentials(
  LPWSTR lpwsServer,           // string containing name
                               // of router
  LPWSTR lpwsInterfaceName,    // string containing name
                               // of interface
  LPWSTR lpwsUserName,         // string to receive username
  LPWSTR lpwsPassword,         // string to receive password
  LPWSTR lpwsDomainName        // string to receive domain name
);
```

Parameters

lpwsServer

Pointer to a Unicode string containing the name of the Windows 2000 router on which to execute this call.

This parameter is optional. If the caller specifies NULL for this parameter, the call is executed on the local machine.

lpwsInterfaceName
Pointer to a Unicode string containing the name of the demand-dial interface. Use **MprAdminInterfaceGetInfo** to obtain the interface name.

lpwsUserName
Pointer to a Unicode string to receive the name of the user. This string should be UNLEN+1 long.

This parameter is optional. If the caller specifies NULL for this parameter, the function will not return the user name.

lpwsPassword
Pointer to a Unicode string to receive the password. This string should be PWLEN+1 long.

This parameter is optional. If the caller specifies NULL for this parameter, the function will not return the password.

lpwsDomainName
Pointer to a Unicode string to receive the domain name. This string should be DNLEN+1 long.

This parameter is optional. If the caller specifies NULL for this parameter, the function will not return the domain name.

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_CANNOT_FIND_PHONEBOOK_ENTRY	The specified interface doesn't have any demand dial parameters associated with it.
ERROR_INVALID_PARAMETER	At least, one of the following is true: The *lpwsInterfaceName* parameter is NULL. All three of the *lpwsUserName*, *lpwsPassword*, and *lpwsDomainName* parameters are NULL.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Remarks

The *lpwsUserName*, *lpwsPassword*, and *lpwsDomainName* parameters are optional. However, if the caller specifies NULL for all three parameters, **MprAdminInterfaceGetCredentials** will return ERROR_INVALID_PARAMETER.

The constants UNLEN, PWLEN, and DNLEN are the maximum lengths for the username, password, and domain name. These constants are defined in lmcons.h.

Note that the order of the parameters in **MprAdminInterfaceGetCredentials** is different from **MprAdminInterfaceSetCredentials**.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **FormatMessage**, **MprAdminInterfaceSetCredentials**

MprAdminInterfaceGetCredentialsEx

Use the **MprAdminInterfaceGetCredentialsEx** function to retrieve extended credentials information for the specified interface. Use this function to retrieve credentials information used for Extensible Authentication Protocols (EAPs).

```
DWORD MprAdminInterfaceGetCredentialsEx(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  HANDLE hInterface,               // handle to interface
  DWORD dwLevel,                   // format of credentials
  LPBYTE * lplpbBuffer             // retreived credentials
);
```

Parameters

hMprServer
A handle to a Windows 2000 router. This handle is obtained from a previous call to **MprAdminServerConnect**.

hInterface
A handle to the interface. This handle is obtained from a previous call to **MprAdminInterfaceCreate**.

dwLevel
Specifies the format of credentials information retrieved. This parameter must be zero, which indicates that the information is formatted as an **MPR_CREDENTIALSEX_0** structure.

lplpbBuffer
Pointer to a pointer to an **MPR_CREDENTIALSEX_0** structure to receive the extended credentials information. Free the memory occupied by this structure with **MprAdminBufferFree**.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.
ERROR_INVALID_PARAMETER	The *lplpbBuffer* parameter is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NOT_SUPPORTED	The *dwLevel* value is invalid.

Requirements

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

See Also

MprAdminInterfaceCreate, **MprAdminInterfaceGetCredentials**, **MprAdminInterfaceSetCredentialsEx**, **MprAdminServerConnect**

MprAdminInterfaceGetHandle

The **MprAdminInterfaceGetHandle** function retrieves a handle to a specified interface.

```
DWORD MprAdminInterfaceGetHandle(
  MPR_SERVER_HANDLE hMprServer,     // handle to router
  LPWSTR lpwsInterfaceName,         // name of interface
  HANDLE * phInterface,             // handle to interface
  BOOL fIncludeClientInterfaces     // toggles inclusion of
                                    // client interfaces
);
```

Parameters

hMprServer
  Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

lpwsInterfaceName
  Pointer to a Unicode string that contains the name of the interface to be retrieved.

phInterface
Pointer to a **HANDLE** variable that, on successful return, will contain a handle to the interface specified by *lpwsInterfaceName*.

fIncludeClientInterfaces
If this parameter is FALSE, interfaces of type **ROUTER_IF_TYPE_CLIENT** will be ignored in the search for the interface with the name specified by *lpwsInterfaceName*. If this parameter is TRUE, a handle to an interface of type **ROUTER_IF_TYPE_CLIENT** may be returned. Since it is possible that there are several interfaces of type **ROUTER_IF_TYPE_CLIENT**, the handle returned will be for the first interface found with the name specified by *lpwsInterfaceName*.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_PARAMETER	*lpwsInterfaceName* is NULL.
ERROR_NO_SUCH_INTERFACE	No interface exists with the name specified by *lpwsInterfaceName*.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **ROUTER_INTERFACE_TYPE**, **MprAdminServerConnect**

MprAdminInterfaceGetInfo

The **MprAdminInterfaceGetInfo** function retrieves information for a specified interface on a specified server.

```
DWORD MprAdminInterfaceGetInfo(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  HANDLE hInterface,               // handle to interface
  DWORD dwLevel,                   // level of information
  LPBYTE * lplpbBuffer             // buffer for information
);
```

Parameters

hMprServer

A handle to the Windows 2000 router to query. This handle is obtained from a previous call to **MprAdminServerConnect**.

hInterface

A handle to the interface obtained by a previous call to **MprAdminInterfaceCreate**.

dwLevel

Specifies the type of structure returned through the *lplpbBuffer* parameter. Must be zero.

Windows 2000 and later: This parameter may have a value of one. A value of one indicates that the *lpBuffer* parameter points to an **MPR_INTERFACE_1** structure.

lplpbBuffer

Pointer to a pointer variable. On successful return, this variable will point to an **MPR_INTERFACE_0** structure. Free this memory by calling **MprAdminBufferFree**.

Windows 2000 and later: The pointer variable may point to either an **MPR_INTERFACE_0** or **MPR_INTERFACE_1** structure. The type of the structure should be indicated by the value of the *dwLevel* parameter.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.
ERROR_INVALID_PARAMETER	The *lplpbBuffer* parameter is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NOT_SUPPORTED	The *dwLevel* value is invalid.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MPR_INTERFACE_0**, **MPR_INTERFACE_1**, **MprAdminBufferFree**, **MprAdminInterfaceCreate**, **MprAdminServerConnect**

MprAdminInterfaceQueryUpdateResult

The **MprAdminInterfaceQueryUpdateResult** function returns the result of the last request to a specified router manager to update its routes for a specified interface. For more information, see **MprAdminInterfaceUpdateRoutes**.

```
DWORD MprAdminInterfaceQueryUpdateResult(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  HANDLE hInterface,               // handle to interface
  DWORD dwTransportId,             // transport/router
                                   // manager ID
  LPDWORD lpdwUpdateResult         // result of last route
                                   // update
);
```

Parameters

hMprServer
Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

hInterface
Handle to the interface. This handle is obtained from a previous call to **MprAdminInterfaceCreate**.

dwTransportId
A **DWORD** variable containing the transport identifier. This parameter identifies the router manager that updated its routing information.

lpdwUpdateResult
A pointer to a **DWORD** variable. On successful return, this variable will contain the result of the last call to **MprAdminInterfaceUpdateRoutes**.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INTERFACE_NOT_CONNECTED	The specified interface is not connected; the result of the last update is no longer available.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.
ERROR_INVALID_PARAMETER	The *lpdwUpdateResult* parameter is NULL.

Value	Meaning
ERROR_NO_SUCH_INTERFACE	The specified transport is not running on the specified interface.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed router manager.

Remarks

The *dwTransportId* parameter specifies both a transport and a router manager, since Windows 2000 router maintains a router manager for each transport.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MprAdminInterfaceCreate**, **MprAdminInterfaceUpdateRoutes**, **MprAdminServerConnect**

MprAdminInterfaceSetCredentials

Use **MprAdminInterfaceSetCredentials** function to set the domain, username, and password that will be used for dialing out on the specified demand-dial interface.

```
DWORD MprAdminInterfaceSetCredentials(
  LPWSTR lpwsServer,        // string containing name
                            // of router
  LPWSTR lpwsInterfaceName, // string containing name
                            // of interface
  LPWSTR lpwsUserName,      // string containing username
  LPWSTR lpwsDomainName,    // string containing domain name
  LPWSTR lpwsPassword       // string containing password
);
```

Parameters

lpwsServer

Pointer to a Unicode string containing the name of the Windows 2000 router on which to execute this call.

This parameter is optional. If the caller specifies NULL for this parameter, the call is executed on the local machine.

lpwsInterfaceName
Pointer to a Unicode string containing the name of the demand-dial interface. Use **MprAdminInterfaceGetInfo** to obtain the interface name.

lpwsUserName
Pointer to a Unicode string containing the username.

This parameter is optional. If the caller specifies NULL for this parameter, the function will not change the username associated with this interface.

lpwsDomainName
Pointer to a Unicode string containing the domain name.

This parameter is optional. If the caller specifies NULL for this parameter, the function will not change the domain name associated with this interface.

lpwsPassword
Pointer to a Unicode string containing the password.

This parameter is optional. If the caller specifies NULL for this parameter, the function will not change the password associated with this interface.

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	At least, one of the following is true: The *lpwsInterfaceName* parameter is NULL, or it is longer than MAX_INTERFACE_NAME_LEN. At least one of the *lpwsUserName*, *lpwsPassword*, and *lpwsDomainName* parameters is too long, and therefore invalid. See **Remarks** section for more information.
ERROR_NOT_ENOUGH_MEMORY	Insufficient memory to create a new data structure to contain the credentials.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Remarks

The *lpwsUserName*, *lpwsPassword*, and *lpwsDomainName* parameters are optional. If the caller specifies NULL for all three parameters, **MprAdminInterfaceSetCredentials** will remove all credential information for this interface.

The constants UNLEN, PWLEN, and DNLEN are the maximum lengths for the username, password, and domain name. These constants are defined in lmcons.h.

Note that the order of the parameters in **MprAdminInterfaceSetCredentials** is different from **MprAdminInterfaceGetCredentials**.

! Requirements

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

+ See Also

Router Administration Reference, Router Administration Functions, **FormatMessage**, **MprAdminInterfaceGetCredentials**, **MprAdminInterfaceGetInfo**

MprAdminInterfaceSetCredentialsEx

Use the **MprAdminInterfaceSetCredentialsEx** function to set extended credentials information for an interface. Use this function to set credentials information used for Extensible Authentication Protocols (EAPs).

```
DWORD MprAdminInterfaceSetCredentialsEx(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  HANDLE hInterface,               // handle to interface
  DWORD dwLevel,                   // format of credentials
  LPBYTE lpbBuffer                 // new credentials
);
```

Parameters

hMprServer
A handle to a Windows 2000 router. This handle is obtained from a previous call to **MprAdminServerConnect**.

hInterface
A handle to the interface. This handle is obtained from a previous call to **MprAdminInterfaceCreate**.

dwLevel
Specifies the format of the credentials information. This parameter must be zero, which indicates that the information is formatted as an **MPR_CREDENTIALSEX_0** structure.

lplpbBuffer
Pointer to an **MPR_CREDENTIALSEX_0** structure containing the new extended credentials information for the interface.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.
ERROR_INVALID_PARAMETER	The *lplpbBuffer* parameter is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NOT_SUPPORTED	The *dwLevel* value is invalid.

! Requirements

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

+ See Also

MprAdminInterfaceCreate, **MprAdminInterfaceSetCredentials**, **MprAdminInterfaceSetCredentialsEx**, **MprAdminServerConnect**

MprAdminInterfaceSetInfo

The **MprAdminInterfaceSetInfo** function set information for a specified interface on a specified server.

```
DWORD MprAdminInterfaceSetInfo(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  HANDLE hInterface,               // handle to interface
  DWORD dwLevel,                   // level of information
  LPBYTE lpbBuffer                 // buffer of information
);
```

Parameters

hMprServer
A handle to the Windows 2000 router to query. This handle is obtained from a previous call to **MprAdminServerConnect**.

hInterface
A handle to the interface obtained by a previous call to **MprAdminInterfaceCreate**.

dwLevel
Specifies the type of structure returned through the *lplpbBuffer* parameter. Must be zero, one, or two.

lpbBuffer
Pointer to a an **MPR_INTERFACE_0**, **MPR_INTERFACE_1**, or **MPR_INTERFACE_2** structure. The type of the structure should be indicated by the value of the *dwLevel* parameter. Free this memory by calling **MprAdminBufferFree**.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.
ERROR_INVALID_PARAMETER	The *lplpbBuffer* parameter is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NOT_SUPPORTED	The *dwLevel* value is invalid.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MPR_INTERFACE_0**, **MPR_INTERFACE_1**, **MPR_INTERFACE_2**, **MprAdminBufferFree**, **MprAdminInterfaceCreate**, **MprAdminInterfaceGetInfo**, **MprAdminServerConnect**

MprAdminInterfaceTransportAdd

The **MprAdminInterfaceTransportAdd** function adds a transport (for example, IP or IPX) to a specified interface.

```
DWORD MprAdminInterfaceTransportAdd(
  MPR_SERVER_HANDLE hMprServer,    // handle to the router
  HANDLE hInterface,          // handle to the interface
  DWORD dwTransportId,        // transport/router manager ID
  LPBYTE pInterfaceInfo,      // interface information
  DWORD dwInterfaceInfoSize   // size of interface information
);
```

Parameters

hMprServer
Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

hInterface
Handle to the interface to which to add the transport. This handle is obtained by a previous call to **MprAdminInterfaceCreate**.

dwTransportId
Value that identifies the transport to add to the interface.

pInterfaceInfo
Pointer to an information header containing interface information for this transport. Use the Information Header Functions to manipulate information headers.

dwInterfaceInfoSize
Size, in bytes, of the information pointed to by *pInterfaceInfo*.

Remarks

The *dwTransportId* parameter also specifies the router manager because a Windows 2000 router uses a different router manager for each transport.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.
ERROR_INVALID_PARAMETER	The *pInterfaceInfo* parameter is NULL.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed transport or router manager.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MprAdminInterfaceCreate**, **MprAdminInterfaceTransportRemove**, **MprAdminServerConnect**

MprAdminInterfaceTransportGetInfo

The **MprAdminInterfaceTransportGetInfo** function retrieves information about a transport running on a specified interface.

```
DWORD MprAdminInterfaceTransportGetInfo(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  HANDLE hInterface,               // handle to interface
  DWORD dwTransportId,             // transport/router
                                   // manager ID
  LPBYTE * ppInterfaceInfo,        // buffer to receive
                                   // interface information
  LPDWORD lpdwInterfaceInfoSize    // size of interface
                                   // information returned
);
```

Parameters

hMprServer

Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

hInterface

Handle to the interface. This handle is obtained from a previous call to **MprAdminInterfaceCreate**.

dwTransportId

Value that identifies the transport/router manager for which information is requested.

ppInterfaceInfo

Pointer to a pointer variable. On successful return, the pointer variable points to an information header containing information for the specified interface and transport. Use the Information Header Functions to manipulate information headers.

lpdwInterfaceInfoSize

Pointer to a **DWORD** variable. On successful return, this variable contains the size in bytes of the interface information returned through the *ppInterfaceInfo* parameter.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not return the size of the interface information.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid, or if the interface specified is administratively disabled.
ERROR_NO_SUCH_INTERFACE	The specified transport is not running on the specified interface.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed transport or router manager.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MprAdminInterfaceCreate**, **MprAdminInterfaceTransportSetInfo**, **MprAdminServerConnect**

MprAdminInterfaceTransportRemove

The **MprAdminInterfaceTransportRemove** function removes a transport (for example, IP or IPX) from a specified interface.

```
DWORD MprAdminInterfaceTransportRemove(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  HANDLE hInterface,               // handle to interface
  DWORD dwTransportId              // transport/router
                                   // manager ID
);
```

Parameters

hMprServer
Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

hInterface
Handle to the interface from which to remove the transport. Obtain this handle by alling **MprAdminInterfaceCreate**.

dwTransportId
Identifies the transport to remove from the interface.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INTERFACE_CONNECTED	The interface specified is a demand dial interface and is currently connected.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.
ERROR_NO_SUCH_INTERFACE	The specified transport is not running on the specified interface.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed transport.

Remarks

The *dwTransportId* parameter specifies a router manager because a Windows 2000 router uses a different router manager for each routable transport.

! Requirements

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

+ See Also

Router Administration Reference, Router Administration Functions, **MprAdminInterfaceCreate**, **MprAdminInterfaceTransportAdd**, **MprAdminServerConnect**

MprAdminInterfaceTransportSetInfo

The **MprAdminInterfaceTransportSetInfo** function sets information for a transport running on a specified interface.

```
DWORD MprAdminInterfaceTransportSetInfo(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  HANDLE hInterface,               // handle to interface
  DWORD dwTransportId,             // identifier for the
```

(continued)

(continued)

```
                                  // transport
  LPBYTE pInterfaceInfo,          // interface information
  DWORD dwInterfaceInfoSize       // size of interface
                                  // information
);
```

Parameters

hMprServer
: Handle to the Windows 2000 router on which to execute this call. Obtain the handle by calling **MprAdminServerConnect**.

hInterface
: Handle to the interface. Obtain this handle by calling **MprAdminInterfaceCreate**.

dwTransportId
: Value that identifies the transport for which information is set.

pInterfaceInfo
: Pointer to an information header containing information for the specified interface and transport. Use the Information Header Functions to manipulate information headers.

dwInterfaceInfoSize
: Size, in bytes, of the information pointed to by *pInterfaceInfo*.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.
ERROR_INVALID_PARAMETER	The *pInterfaceInfo* parameter is NULL.
ERROR_NO_SUCH_INTERFACE	The specified transport is not running on the specified interface.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed transport.

! Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MprAdminInterfaceCreate**, **MprAdminInterfaceTransportGetInfo**, **MprAdminServerConnect**

MprAdminInterfaceUpdatePhonebookInfo

Call the **MprAdminInterfaceUpdatePhonebookInfo** function after making any changes to the phonebook entry for the specified demand dial interface. This function forces the router to pick up the changes for that interface.

```
DWORD MprAdminInterfaceUpdatePhonebookInfo(
  MPR_SERVER_HANDLE hMprServer,   // handle to the router
  HANDLE hInterface               // handle to the interface
);
```

Parameters

hMprServer
Handle to the Windows 2000 router on which to execute this call. Obtain the handle by calling **MprAdminServerConnect**.

hInterface
Handle to a demand-dial interface. Obtain this handle by calling **MprAdminInterfaceCreate**.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_CANNOT_LOAD_PHONEBOOK	The function could not load the phonebook into memory.
ERROR_CANNOT_OPEN_PHONEBOOK	The function could not find the phonebook file.
ERROR_DDM_NOT_RUNNING	The Demand Dial Manager (DDM) is not running.
ERROR_INTERFACE_HAS_NO_DEVICES	No device is currently associated with this interface.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.

(continued)

(continued)

Value	Meaning
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **FormatMessage**, **MprAdminInterfaceCreate**, **MprAdminServerConnect**

MprAdminInterfaceUpdateRoutes

The **MprAdminInterfaceUpdateRoutes** function requests that a specified router manager update its routing information for a specified interface.

```
DWORD MprAdminInterfaceUpdateRoutes(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  HANDLE hInterface,               // handle to interface
  DWORD dwTransportId,             // identifies the router
                                   // manager
  HANDLE hEvent                    // event to signal when
                                   // update complete
);
```

Parameters

hMprServer
Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

hInterface
Handle to the interface. Obtain this handle by calling **MprAdminInterfaceCreate**.

dwTransportId
Identifies the router manager that should update its routing information. (Windows 2000 router uses a different router manager for each transport.)

hEvent
Handle to an event that will be signaled when the attempt to update routing information for this interface has completed. If this value is NULL, then the function is synchronous. If *hMprServer* specifies a remote router, the caller must specify NULL for this parameter.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INTERFACE_NOT_CONNECTED	The specified interface is not connected. Therefore, routes cannot be updated.
ERROR_INVALID_HANDLE	The *hInterface* value is invalid.
ERROR_NO_SUCH_INTERFACE	The specified transport is not running on the specified interface.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any of the router managers.
ERROR_UPDATE_IN_PROGRESS	A routing information update operation is already in progress on this interface.
PENDING	The interface is in the process of updating routing information. The caller should wait on the event object specified by *hEvent*. After the event is signaled, the status of the update operation can be obtained by calling **MprAdminInterfaceQueryUpdateResult**.

Remarks

The *dwTransportId* parameter specifies both a transport (for example, IP or IPX) and a unique router manager because Windows 2000 router uses a different router manager for each transport.

! Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MprAdminInterfaceCreate**, **MprAdminInterfaceQueryUpdateResult**, **MprAdminServerConnect**

MprAdminIsServiceRunning

The **MprAdminIsServiceRunning** function checks if the Routing and Remote Access Service is running on a specified machine. If not, none of the **MprAdmin**XXX calls will succeed.

```
BOOL MprAdminIsServiceRunning(
  LPWSTR lpwsServerName    // name of machine to query
);
```

Parameters

lpwsServerName

Pointer to a Unicode string containing the name of the server to query.

Return Values

The return value is one of the following Boolean values:

Value	Meaning
TRUE	The service is running on the specified server.
FALSE	The service is not running on the specified server.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in Mprapi.h.
Library: Use Mprapi.lib.

See Also

Router Administration Reference, Router Administration Functions

MprAdminRegisterConnectionNotification

The **MprAdminRegisterConnectionNotification** function registers an event object with the Demand Dial Manager (DDM) so that, if an interface connects or disconnects, the event is signaled.

```
DWORD MprAdminRegisterConnectionNotification(
  MPR_SERVER_HANDLE hMprServer,    // handle to server
  HANDLE hEventNotification        // handle to event
);
```

Parameters

hMprServer

Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminServerConnect**.

hEventNotification

Handle to an event object. This event is signaled whenever an interface connects or disconnects.

Return Values

If the function is successful, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes.

Value	Meaning
ERROR_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_DDM_NOT_RUNNING	The DDM is not running.
ERROR_INVALID_PARAMETER	The *hEventNotification* parameter is NULL or is an invalid handle.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Remarks

If the event is signaled, indicating that an interface has connected or disconnected, the calling application can then use a function such as **MprAdminConnectionEnum** or **MprAdminInterfaceEnum** to determine which interface was affected.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **FormatMessage**, **MprAdminConnectionEnum**, **MprAdminDeregisterConnectionNotification**, **MprAdminInterfaceEnum**

MprAdminServerConnect

Call the **MprAdminServerConnect** function to connect to the Windows 2000 router to administer. Call this function before making any other calls to the server. Use the handle returned in subsequent calls to administer interfaces on the server.

```
DWORD MprAdminServerConnect(
  LPWSTR lpwsServerName,              // name of router
  MPR_SERVER_HANDLE * phMprServer     // handle to router
);
```

Parameters

lpwsServerName
: Pointer to a Unicode string that contains the name of the remote server.

phMprServer
: Pointer to a **HANDLE** variable that, on successful return, contains a handle to the server. Use this handle in subsequent calls to administer the server.

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_ACCESS_DENIED	The caller does not have sufficient privilege.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in Mprapi.h.
Library: Use Mprapi.lib.

See Also

Router Administration Reference, Router Administration Functions, **MprAdminServerDisconnect**

MprAdminServerDisconnect

The **MprAdminServerDisconnect** function disconnects the connection made by a previous call to **MprAdminServerConnect**.

```
VOID MprAdminServerDisconnect(
  MPR_SERVER_HANDLE hMprServer    // handle to router
);
```

Parameters

hMprServer
Handle to the Windows 2000 router from which to disconnect. Obtain this handle by calling **MprAdminServerConnect**.

Return Values

None.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.
Header: Declared in Mprapi.h.
Library: Use Mprapi.lib.

See Also

Router Administration Reference, Router Administration Functions, **MprAdminServerConnect**

MprAdminServerGetInfo

The **MprAdminServerGetInfo** function retrieves routing and RAS information from a specified server.

```
DWORD MprAdminServerGetInfo(
  MPR_SERVER_HANDLE hMprServer,    // handle to router
  DWORD dwLevel,                   // level of information
                                   // requested
  LPBYTE * lplpbBuffer             // MPR_SERVER_0 structure
);
```

Parameters

hMprServer
Handle to the Windows 2000 router to query. Obtain this handle by calling **MprAdminMIBServerConnect.**

dwLevel
Level of the information returned in *lplpbBuffer.* Must be zero.

lplpbBuffer
Pointer to a pointer variable. On successful return, this pointer variable will point to an **MPR_SERVER_0** structure. Free this memory by calling **MprAdminBufferFree.**

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_PARAMETER	The *lplpbBuffer* parameter is NULL.

Requirements

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

See Also

Router Administration Reference, Router Administration Functions, **MprAdminBufferFree**, **MPR_SERVER_0**, **MprAdminServerConnect**

MprAdminTransportCreate

The **MprAdminTransportCreate** function loads a new transport, and starts the router manager for the transport.

```
DWORD MprAdminTransportCreate(
  MPR_SERVER_HANDLE hMprServer,
  DWORD dwTransportId,
  LPWSTR lpwsTransportName,
  LPBYTE pGlobalInfo,
  DWORD dwGlobalInfoSize,
  LPBYTE pClientInterfaceInfo,
  DWORD dwClientInterfaceInfoSize,
  LPWSTR lpwsDLLPath
);
```

Parameters

hMprServer
Handle to the Windows 2000 router on which to set the information. Obtain this handle by calling **MprAdminServerConnect**.

dwTransportId
Value that identifies the transport for which to set information.

lpwsTransportName
Pointer to a null-terminated Unicode string that contains the name of the transport.

pGlobalInfo
Pointer to a buffer containing global information for the transport. Use the Information Header Functions to manipulate information headers.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not set the global information.

dwGlobalInfoSize
Size, in bytes, of the buffer pointed to by the *pGlobalInfo* parameter.

pClientInterfaceInfo
Pointer to a buffer containing default client interface information for the transport.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not set the default client interface information.

dwClientInterfaceInfoSize
Size, in bytes, of the buffer pointed to by the *pClientInterfaceInfo* parameter.

lpwsDLLPath
Pointer to a null-terminated Unicode string that contains the path to the DLL for the transport.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_PARAMETER	The *pGlobalInfo* parameter and the *pClientInterfaceInfo* parameter are both NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_PROTOCOL_ALREADY_INSTALLED	The specified transport is already running on the specified router.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed transport.

Requirements

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

See Also

MprAdminTransportGetInfo, **MprAdminTransportSetInfo**

MprAdminTransportGetInfo

The **MprAdminTransportGetInfo** function retrieves global information, default client interface information, or both, for a specified transport.

```
DWORD MprAdminTransportGetInfo(
  MPR_SERVER_HANDLE hMprServer,         // handle to router
  DWORD dwTransportId,                  // identifies the transport
  LPBYTE * ppGlobalInfo,                // buffer to receive global
                                        // information for transport
  LPDWORD lpdwGlobalInfoSize,           // size of global
                                        // information returned
  LPBYTE * ppClientInterfaceInfo,       // buffer to receive
                                        // client interface
                                        // information
  LPDWORD lpdwClientInterfaceInfoSize   // size of client
                                        // interface
                                        // information
                                        // returned
);
```

Parameters

hMprServer

Handle to the Windows 2000 router to query. This handle is obtained from a previous call to **MprAdminServerConnect**.

dwTransportId

Value that identifies the transport about which to retrieve information.

ppGlobalInfo

Pointer to a pointer variable. On successful return, this variable points to an information header containing global information for this transport. Use the Information Header Functions to manipulate information headers.

Free this memory by calling **MprAdminBufferFree**.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not retrieve the global information.

lpdwGlobalInfoSize

Pointer to a **DWORD** variable. On successful return, this variable contains the size in bytes of the global information for the transport.

ppClientInterfaceInfo

Pointer to a pointer variable. On successful return, this variable points to default client interface information for this transport. Free this memory by calling **MprAdminBufferFree**.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not retrieve the client interface information.

lpdwClientInterfaceInfoSize

Pointer to a **DWORD** variable. On successful return, this variable contains the size in bytes of the client interface information.

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_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_PARAMETER	One of the following is true: The *ppGlobalInfo* parameter and the *ppClientInterfaceInfo* parameter are both NULL. The *ppGlobalInfo* parameter does not point to valid memory. The *ppClientInterfaceInfo* parameter does not point to valid memory.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed transport.

Remarks

The *ppGlobalInfo* and *ppClientInterfaceInfo* parameters cannot both be NULL.

! Requirements

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

+ See Also

Router Administration Reference, Router Administration Functions, **MprAdminBufferFree**, **MprAdminServerConnect**, **MprAdminTransportSetInfo**

MprAdminTransportSetInfo

The **MprAdminTransportSetInfo** function sets global information, or default client interface information, or both, for a specified transport.

```
DWORD MprAdminTransportSetInfo(
  MPR_SERVER_HANDLE hMprServer,     // handle to router
  DWORD dwTransportId,              // identifies the transport
  LPBYTE pGlobalInfo,               // global information for
                                    // transport
  DWORD dwGlobalInfoSize,           // size of global
                                    // information
  LPBYTE pClientInterfaceInfo,      // information for client
                                    // interfaces
  DWORD dwClientInterfaceInfoSize   // size of information
                                    // for client interfaces
);
```

Parameters

hMprServer
Handle to the Windows 2000 router on which to set the information. Obtain this handle by calling **MprAdminServerConnect**.

dwTransportId
Value that identifies the transport for which to set information.

pGlobalInfo
Pointer to a buffer containing global information for the transport. Use the Information Header Functions to manipulate information headers.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not set the global information.

dwGlobalInfoSize
Size, in bytes, of the buffer pointed to by the *pGlobalInfo* parameter.

pClientInterfaceInfo
Pointer to a buffer containing default client interface information for the transport.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not set the default client interface information.

dwClientInterfaceInfoSize
Size, in bytes, of the buffer pointed to by the *pClientInterfaceInfo* parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the error codes on the following page.

Value	Meaning
ERROR_ACCESS_DENIED	The caller does not have sufficient privilege.
ERROR_INVALID_PARAMETER	The *pGlobalInfo* parameter and the *pClientInterfaceInfo* parameter are both NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed transport.

! Requirements

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

+ See Also

Router Administration Reference, Router Administration Functions, **MprAdminServerConnect**, **MprAdminTransportGetInfo**

Router Configuration Functions

Use the following functions when developing software to configure Microsoft® Windows NT®/Windows® 2000 routers:

MprConfigBufferFree
MprConfigGetFriendlyName
MprConfigGetGuidName
MprConfigInterfaceCreate
MprConfigInterfaceDelete
MprConfigInterfaceEnum
MprConfigInterfaceGetHandle
MprConfigInterfaceGetInfo
MprConfigInterfaceSetInfo
MprConfigInterfaceTransportAdd
MprConfigInterfaceTransportEnum
MprConfigInterfaceTransportGetHandle
MprConfigInterfaceTransportGetInfo
MprConfigInterfaceTransportRemove
MprConfigInterfaceTransportSetInfo
MprConfigServerBackup
MprConfigServerConnect
MprConfigServerDisconnect
MprConfigServerGetInfo
MprConfigServerInstall
MprConfigServerRestore
MprConfigTransportCreate
MprConfigTransportDelete
MprConfigTransportEnum
MprConfigTransportGetHandle
MprConfigTransportGetInfo
MprConfigTransportSetInfo

MprConfigBufferFree

The **MprConfigBufferFree** function frees buffers allocated by calls to the following functions:

MprConfigXEnum
MprConfigXGetInfo

where X stands for **Server**, **Interface, Transport,** or **InterfaceTransport**.

```
DWORD MprConfigBufferFree(
  LPVOID pBuffer    // address of memory buffer to free
);
```

Parameters

pBuffer
Pointer to a memory buffer allocated by a previous call to:
MprConfigXEnum
MprConfigXGetInfo
where X stands for **Server**, **Interface, Transport,** or **InterfaceTransport**.

Return Values

If the function succeeds, the return value is NO_ERROR.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **MprConfigInterfaceEnum**, **MprConfigInterfaceTransportEnum**, **MprConfigTransportEnum**, **MprConfigInterfaceGetInfo**, **MprConfigInterfaceTransportGetInfo**, **MprConfigServerGetInfo**, **MprConfigTransportGetInfo**

MprConfigGetFriendlyName

The **MprConfigGetFriendlyName** function returns the user-friendly name for an interface that corresponds to the specified GUID name.

```
DWORD MprConfigGetFriendlyName(
  HANDLE hMprConfig,      // handle to router configuration
  PWCHAR pszGuidName,     // GUID name for the interface
  PWCHAR pszBuffer,       // buffer to receive user-friendly
                          // name
  DWORD dwBufferSize      // size of buffer passed in
);
```

Parameters

hMprConfig
Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

pszGuidName
Pointer to a Unicode string containing the GUID name for the interface.

pszBuffer
Pointer to a buffer to receive the user-friendly name for the interface.

dwBufferSize
Size, in bytes, of the buffer pointed to by *pszBuffer*.

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_BUFFER_OVERFLOW	The buffer pointed to by *pszBuffer* is not large enough to hold the returned GUID name.
ERROR_INVALID_PARAMETER	At least one of the parameters *hMprConfig*, *pszGuidName*, or *pszBuffer* is NULL.
ERROR_NOT_FOUND	No GUID name was found that corresponds to the specified user-friendly name.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **MprConfigGetGuidName**, **MprConfigServerConnect**

MprConfigGetGuidName

The **MprConfigGetGuidName** function returns the GUID name for an interface that corresponds to the specified user-friendly name.

```
DWORD MprConfigGetGuidName(
  HANDLE hMprConfig,       // handle to router configuration
  PWCHAR pszFriendlyName,  // user-friendly name for the
                           // interface
  PWCHAR pszBuffer,        // buffer to receive GUID name
  DWORD dwBufferSize       // size of buffer passed in
);
```

Parameters

hMprConfig
Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

pszFriendlyName
Pointer to a Unicode string containing the user-friendly name for the interface.

pszBuffer
Pointer to a buffer to receive the GUID name for the interface.

dwBufferSize
Size, in bytes, of the buffer pointed to by *pszBuffer*.

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_BUFFER_OVERFLOW	The buffer pointed to by *pszBuffer* is not large enough to hold the returned GUID name.
ERROR_INVALID_PARAMETER	At least one of the parameters *hMprConfig*, *pszFriendlyName*, or *pszBuffer* is NULL.
ERROR_NOT_FOUND	No GUID name was found that corresponds to the specified user-friendly name.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **MprConfigGetFriendlyName**, **MprConfigServerConnect**

MprConfigServerInstall

The **MprConfigServerInstall** function configures Routing and Remote Access Service with a default configuration.

```
DWORD MprConfigServerInstall(
    DWORD    dwLevel;
    PVOID    pBuffer;
);
```

Parameters

dwLevel
  This parameter is reserved for future use, and should be zero.

pBuffer
  This parameter is reserved for future use, and should be NULL.

Return Values

If the functions succeeds, the return value is ERROR_SUCCESS.

Remarks

The **MprConfigServerInstall** function performs the following tasks:

- Resets the current RouterManager and Interface keys.
- Initializes RAS structures for IP.
- Sets the router type to include:
  - ROUTER_TYPE_RAS
  - ROUTER_TYPE_WAN
  - ROUTER_TYPE_LAN
- Sets the error logging level and authorization settings to defaults.
- Sets the devices for Routing and RAS.
- Adds the RRAS snapin to the computer management console.
- Deletes the router phonebook.
- Registers the router in the domain.
- Writes out the "router is configured" registry key.

The **MprConfigServerInstall** function does not start Routing and RAS. Nor does it set the service start type for Routing and RAS.

Requirements

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

See Also

Windows 2000 Registry Layout

MprConfigInterfaceCreate

The **MprConfigInterfaceCreate** function creates a router interface in the specified router configuration.

```
DWORD MprConfigInterfaceCreate(
  HANDLE hMprConfig,         // handle to the router configuration
  DWORD dwLevel,             // level of information requested
  LPBYTE lpbBuffer,          // MPR_INTERFACE_0 structure
  HANDLE *phRouterInterface  // handle to the interface
                             // configuration
);
```

Parameters

hMprConfig
Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

dwLevel
Level of information requested. This parameter must be zero.

lpbBuffer
Pointer to an **MPR_INTERFACE_0** structure.

phRouterInterface
Pointer to a handle variable. On successful return, this variable will contain a handle to the interface configuration.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the error codes on the following page.

Value	Meaning
ERROR_INVALID_PARAMETER	At least one of the following is true: *hMprConfig* is NULL *dwLevel* is not zero. *lpbBuffer* is NULL *phRouterInterface* is NULL
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

! Requirements

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

+ See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigInterfaceDelete**, **MprConfigServerConnect**

MprConfigInterfaceDelete

The **MprConfigInterfaceDelete** function removes a router interface from the router configuration. All transport information associated with this interface is also removed.

```
DWORD MprConfigInterfaceDelete(
  HANDLE hMprConfig,         // handle to the router
                             // configuration
  HANDLE hRouterInterface    // handle to the interface
                             // configuration
);
```

Parameters

hMprConfig

Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

hRouterInterface

Handle to the interface configuration. Obtain this handle by calling **prConfigInterfaceCreate**, **MprConfigInterfaceGetHandle**, or **MprConfigInterfaceEnum**.

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 *hMprConfig* parameter is NULL, or the *hRouterInterface* parameter is NULL, or both parameters are NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigInterfaceCreate**, **MprConfigInterfaceEnum**, **MprConfigInterfaceGetHandle**, **MprConfigServerConnect**

MprConfigInterfaceEnum

The **MprConfigInterfaceEnum** function enumerates the interfaces that are configured for the router.

```
DWORD MprConfigInterfaceEnum(
  HANDLE hMprConfig,          // handle to the router
                              // configuration
  DWORD dwLevel,              // level of information
                              // requested
  LPBYTE *lplpBuffer,         // array of MPR_INTERFACE_0
                              // structures
  DWORD dwPrefMaxLen,         // maximum length of data to
                              // return
  LPDWORD lpdwEntriesRead,    // number of entries
                              // enumerated
  LPDWORD lpdwTotalEntries,   // number of entries that
                              // could've been enumerated
  LPDWORD lpdwResumeHandle      // handle for continuing
                                // the enumeration
);
```

Parameters

hMprConfig
: Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

dwLevel
: Level of the information returned through the *lplpBuffer* parameter. This parameter must be zero.

lplpBuffer
: Pointer to a pointer variable. On successful return, this pointer variable will point to an array of **MPR_INTERFACE_0** structures. Free this memory by calling **MprConfigBufferFree**.

dwPrefMaxLen
: Specifies the preferred maximum length of returned data (in 8-bit bytes). If this parameter is –1, the buffer returned will be large enough to hold all available information.

lpdwEntriesRead
: Pointer to a **DWORD** variable. On successful return, this variable contains the total number of entries that were enumerated from the current resume position.

lpdwTotalEntries
: Pointer to a **DWORD** variable. On successful return, this variable contains the total number of entries that could have been enumerated from the current resume position.

lpdwResumeHandle
: Pointer to a **DWORD** variable. On successful return, this variable contains a resume handle that can be used to continue the enumeration. The handle should be zero on the first call, and left unchanged on subsequent calls. If on return, the handle is NULL, the enumeration cannot be continued. For other types of error returns, this handle is invalid.

  This parameter is optional. If the caller specifies NULL for this parameter, the function will not return a resume handle.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the error codes on the following page.

Value	Meaning
ERROR_INVALID_PARAMETER	One of the following is true: *hMprConfig* is NULL. *dwLevel* is not zero. *lplpBuffer* is NULL. *dwPrefMaxLen* is smaller than the size of a single **MPR_INTERFACE_0** structure. *lpdwEntriesRead* is NULL. *lpdwTotalEntries* is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NO_MORE_ITEMS	No more entries available from the current resume position.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigBufferFree**, **MprConfigServerConnect**

MprConfigInterfaceGetHandle

The **MprConfigInterfaceGetHandle** function retrieves a handle to the specified interface's configuration in the specified router configuration.

```
DWORD MprConfigInterfaceGetHandle(
  HANDLE hMprConfig,          // handle to the router
                              // configuration
  LPWSTR lpwsInterfaceName,   // Unicode string containing
                              // name of interface
  HANDLE *phRouterInterface   // handle to the specified
                              // interface
);
```

Parameters

hMprConfig
Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

lpwsInterfaceName
Pointer to a Unicode string containing the name of the interface for which the configuration handle is requested.

phRouterInterface
Pointer to a handle variable. On successful return, this variable contains a handle to the interface configuration.

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 *hMprConfig* parameter is NULL, or the *lpwsInterfaceName* parameter is NULL, or both parameters are NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NO_SUCH_INTERFACE	The specified interface was not found in the router configuration.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigServerConnect**

MprConfigInterfaceGetInfo

The **MprConfigInterfaceGetInfo** function retrieves the configuration for the specified interface from the router.

```
DWORD MprConfigInterfaceGetInfo(
  HANDLE hMprConfig,          // handle to the router
                              // configuration
  HANDLE hRouterInterface,    // handle to the interface
                              // configuration
  DWORD dwLevel,              // level of information
                              // requested
  LPBYTE *lplpBuffer,         // MPR_INTERFACE_0 structure
  LPDWORD lpdwBufferSize      // size of returned data
);
```

Parameters

hMprConfig
Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

hRouterInterface
Handle to the interface configuration for which to retrieve information. Obtain this handle by calling **MprConfigInterfaceCreate**, **MprConfigInterfaceGetHandle**, or **MprConfigInterfaceEnum**.

dwLevel
Level of the information returned in the *lplpBuffer* parameter. This parameter must be zero.

lplpBuffer
Pointer to a pointer variable. On successful return, this pointer variable points to an **MPR_INTERFACE_0** structure. Free this buffer by calling **MprConfigBufferFree**.

lpdwBufferSize
Pointer to a **DWORD** variable. On successful return, this variable will contain the size, in bytes, of the data returned through *lplpBuffer*.

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	At least one of the following is true: *hMprConfig* is NULL *hRouterInterface* is NULL *dwLevel* is not zero. *lplpBuffer* is NULL *lpdwBufferSize* is NULL

Value	Meaning
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NO_SUCH_INTERFACE	The interface corresponding to *hRouterInterface* is not present in the router configuration.

! Requirements

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

+ See Also

Router Administration Reference, Router Configuration Functions, **MprConfigBufferFree**, **MprConfigInterfaceCreate**, **MprConfigInterfaceEnum**, **MprConfigInterfaceGetHandle**, **MprConfigServerConnect**

MprConfigInterfaceSetInfo

The **MprConfigInterfaceSetInfo** function sets the configuration for the specified interface.

```
DWORD MprConfigInterfaceSetInfo(
  HANDLE hMprConfig,           // handle to the router
                               // configuration
  HANDLE hRouterInterface,     // handle to the interface
                               // configuration
  DWORD dwLevel,               // level of the information
                               // requested
  LPBYTE lpBuffer              // MPR_INTERFACE_0 structure
);
```

Parameters

hMprConfig
Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

hRouterInterface
Handle to the interface configuration being updated. Obtain this handle by calling **MprConfigInterfaceCreate**, **MprConfigInterfaceGetHandle**, or **MprConfigInterfaceEnum**.

dwLevel
Level of the information in the *lpBuffer* parameter. This parameter must be zero.

lpBuffer
Pointer to a buffer containing an **MPR_INTERFACE_0** structure. The information in this buffer is used to update the interface configuration.

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	At least one of the following is true: *hMprConfig* is NULL. *HRouterInterface* is NULL. *dwLevel* is not zero. *lpBuffer* is NULL.
ERROR_NO_SUCH_INTERFACE	The interface corresponding to *hRouterInterface* is not present in the router configuration.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigInterfaceCreate**, **MprConfigInterfaceEnum**, **MprConfigInterfaceGetHandle**, **MprConfigServerConnect**

MprConfigInterfaceTransportAdd

The **MprConfigInterfaceTransportAdd** function adds the specified transport to the specified interface configuration on the router.

```
DWORD MprConfigInterfaceTransportAdd(
  HANDLE hMprConfig,            // handle to the router
                                // configuration
  HANDLE hRouterInterface,      // handle to the interface
                                // configuration
  DWORD dwTransportId,          // identifier of the
                                // transport/router manager
  LPWSTR lpwsTransportName,     // transport name
  LPBYTE pInterfaceInfo,        // interface information
  DWORD dwInterfaceInfoSize,    // size of interface
                                // information
  HANDLE *phRouterIfTransport   // handle to transport
                                // configuration
);
```

Parameters

hMprConfig
Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

hRouterInterface
Handle to the interface configuration to which to add the specified transport. Obtain this handle by calling **MprConfigInterfaceCreate**, **MprConfigInterfaceGetHandle**, or **MprConfigInterfaceEnum**.

dwTransportId
Value that identifies the transport. This parameter also identifies the router manager for the transport.

lpwsTransportName
Pointer to a Unicode string containing the name for the transport being added. If this parameter is not specified and the transport is IP or IPX, **MprConfigInterfaceTransportAdd** uses "IP" or "IPX". If this parameter is not specified and the transport is other than IP or IPX, **MprConfigInterfaceTransportAdd** converts the *dwTransportId* parameter into a string and uses that as the transport name.

pInterfaceInfo
Pointer to an information header containing information for the specified interface and transport. The router manager for the specified transport interprets this information. Use the Information Header Functions to manipulate information headers.

dwInterfaceInfoSize
Size, in bytes, of the data pointed to by *pInterfaceInfo*.

phRouterIfTransport
Pointer to a handle variable. On successful return, this variable contains a handle to the transport configuration for this interface.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the error codes on the following page.

Value	Meaning
ERROR_INVALID_PARAMETER	One of the following is true: *hMprConfig* is NULL. *hRouterInterface* is NULL. *phRouterIfTransport* is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Remarks

In addition to specifying a transport, the *dwTransportId* parameter also specifies a router manager, because a Windows 2000 router maintains a unique router manager for each transport.

If the specified transport already exists, **MprConfigInterfaceTransportAdd** does the equivalent of an **MprConfigInterfaceTransportSetInfo** call using the specified parameter values.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MPR_IFTRANSPORT_0**, **MprConfigInterfaceCreate**, **MprConfigInterfaceEnum**, **MprConfigInterfaceGetHandle**, **MprConfigServerConnect**

MprConfigInterfaceTransportEnum

The **MprConfigInterfaceTransportEnum** function enumerates the transports configured on the specified interface.

```
DWORD MprConfigInterfaceTransportEnum(
  HANDLE hMprConfig,          // handle to the router
                              // configuration
  HANDLE hRouterInterface,    // handle to the interface
                              // configuration
  DWORD dwLevel,              // level of information
                              // requested
  LPBYTE *lplpBuffer,          // array of MPR_IFTRANSPORT_0
                               // structures
  DWORD dwPrefMaxLen,          // maximum length of data to
                               // return
  LPDWORD lpdwEntriesRead,     // number of entries
                               // enumerated
  LPDWORD lpdwTotalEntries,    // number of entries that
                               // could've been enumerated
  LPDWORD lpdwResumeHandle     // handle for continuing the
                               // enumeration
);
```

Parameters

hMprConfig

Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

hRouterInterface

Handle to the interface configuration from which to enumerate the transports. Obtain this handle by calling **MprConfigInterfaceCreate**, or **MprConfigInterfaceEnum**.

dwLevel

Level of the information returned in the *lplpBuffer* parameter. This parameter must be zero.

lplpBuffer

Pointer to pointer variable. On successful return, this pointer variable will point to an array of **MPR_IFTRANSPORT_0** structures This memory should be freed by calling **MprConfigBufferFree**.

dwPrefMaxLen

Specifies the preferred maximum length of returned data (in 8-bit bytes). If this parameter is –1, the buffer returned will be large enough to hold all available information.

lpdwEntriesRead

Pointer to a **DWORD** variable. On successful return, this variable contains the total number of entries that were enumerated from the current resume position.

lpdwTotalEntries

Pointer to a **DWORD** variable. On successful return, this variable contains the total number of entries that could have been enumerated from the current resume position.

lpdwResumeHandle

Pointer to a **DWORD** variable. On successful return, this variable contains a resume handle that can be used to continue the enumeration. The handle should be zero on the first call, and left unchanged on subsequent calls. If on return, the handle is NULL, the enumeration cannot be continued. For other types of error returns, this handle is invalid.

This parameter is optional. If the caller specifies NULL for this parameter, the function will not return a resume handle.

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	One of the following is true: *hMprConfig* is NULL. *HRouterInterface* is NULL. *dwLevel* is not zero. *lplpBuffer* is NULL. *dwPrefMaxLen* is smaller than the size of a single **MPR_TRANSPORT_0** structure. *lpdwEntriesRead* is NULL. *lpdwTotalEntries* is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NO_MORE_ITEMS	No more entries available from the current resume position.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MPR_IFTRANSPORT_0**, **MprConfigBufferFree**, **MprConfigInterfaceCreate**, **MprConfigInterfaceEnum**, **MprConfigInterfaceGetHandle**

MprConfigInterfaceTransportGetHandle

The **MprConfigInterfaceTransportGetHandle** function retrieves a handle to the specified transport configuration on the specified interface in the specified router configuration.

```
DWORD MprConfigInterfaceTransportGetHandle(
  HANDLE hMprConfig,           // handle to the router
                               // configuration
  HANDLE hRouterInterface,     // handle to the interface
                               // configuration
  DWORD dwTransportId,         // identifier of the
                               // transport configuration
  HANDLE *phRouterIfTransport  // handle to the transport
                               // configuration
);
```

Parameters

hMprConfig
: Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

hRouterInterface
: Handle to the interface configuration. Obtain this handle by calling **MprConfigInterfaceCreate**, **MprConfigInterfaceGetHandle**, or **MprConfigInterfaceEnum**.

dwTransportId
: Identifies the transport configuration.

phRouterIfTranport
: Pointer to a handle variable. On successful return, this variable contains a handle to the transport configuration for this interface.

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	At least one of the following is true: *hMprConfig* is NULL. *hRouterInterface* is NULL. *phRouterIfTransport* is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NO_SUCH_INTERFACE	The interface specified by *hRouterInterface* was not found in the router configuration, or the transport specified by *dwTransportId* was not enabled on the specified interface.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigInterfaceCreate**, **MprConfigInterfaceEnum**, **MprConfigInterfaceGetHandle**, **MprConfigServerConnect**

MprConfigInterfaceTransportGetInfo

The **MprConfigInterfaceTransportGetInfo** function retrieves the configuration information for the specified transport and interface.

```
DWORD MprConfigInterfaceTransportGetInfo(
  HANDLE hMprConfig,             // handle to the router
                                 // configuration
  HANDLE hRouterInterface,       // handle to the
                                 // interface configuration
  HANDLE hRouterIfTransport,     // handle to the transport
                                 // configuration
  LPBYTE *ppInterfaceInfo,       // configuration
                                 // information for
                                 // transport and interface
  LPDWORD lpdwInterfaceInfoSize  // size of interface
                                 // transport information
);
```

Parameters

hMprConfig
Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

hRouterInterface
Handle to the interface configuration from which to retrieve the specified interface information. Obtain this handle by calling **MprConfigInterfaceCreate**, **MprConfigInterfaceGetHandle**, or **MprConfigInterfaceEnum**.

hRouterIfTransport
Handle to the transport configuration from which to retrieve the specified transport information. Obtain this handle by calling **MprConfigInterfaceTransportGetHandle** or **MprConfigInterfaceTransportEnum**.

ppInterfaceInfo

Pointer to a pointer variable. On successful return, this pointer variable points to an information header containing configuration information for the transport and interface. Use the Information Header Functions to manipulate information headers. Free this memory by calling **MprConfigBufferFree**.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not return the configuration information.

lpdwInterfaceInfoSize

Pointer to a **DWORD** variable. On successful return, this variable contains the size, in bytes, of the data pointed to by *ppInterfaceInfo*.

This parameter is optional; the calling application may specify NULL for this parameter. However, if *ppInterfaceInfo* is not NULL, this parameter cannot be NULL. For more information, see the Remarks section later in this topic.

Return Values

If the function succeeds, the return value is NO_ERROR. For more information, see the Remarks section later in this topic.

If the function fails, the return value is one of the following error codes.

Value	Meaning
ERROR_INVALID_PARAMETER	One of the following is true: *hMprConfig* is NULL. *hRouterInterface* is NULL. *hRouterIfTransport* is NULL. *ppInterfaceInfo* is not NULL, but *lpdwInterfaceInfoSize* is NULL.
ERROR_NO_SUCH_INTERFACE	The interface specified by *hRouterInterface* was not found in the router configuration, or the transport specified by *hRouterIfTransport* was not enabled on the specified interface.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Remarks

If the *ppInterfaceInfo* parameter is NULL, **MprConfigInterfaceTransportGetInfo** does nothing and returns immediately with a value of NO_ERROR.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MPR_IFTRANSPORT_0**, **MprConfigBufferFree**, **MprConfigInterfaceCreate**, **MprConfigInterfaceEnum**, **MprConfigInterfaceGetHandle**, **MprConfigInterfaceTransportEnum**, **MprConfigInterfaceTransportGetHandle**, **MprConfigServerConnect**

MprConfigInterfaceTransportRemove

The **MprConfigInterfaceTransportRemove** function removes the specified transport from the specified interface configuration on the router.

```
DWORD MprConfigInterfaceTransportRemove(
  HANDLE hMprConfig,           // handle to the router
                               // configuration
  HANDLE hRouterInterface,     // handle to the interface
                               // configuration
  HANDLE hRouterIfTransport    // handle to the transport
                               // configuration
);
```

Parameters

hMprConfig
Handle to the router configuration. The handle is obtained from a previous call to **MprConfigServerConnect**.

hRouterInterface
Handle to the interface configuration from which to delete the specified transport. Obtain this handle by calling **MprConfigInterfaceCreate**, **MprConfigInterfaceGetHandle**, or **MprConfigInterfaceEnum**.

hRouterIfTransport
Handle to the interface transport configuration to be deleted. Obtain this handle by calling **MprConfigInterfaceTransportAdd**, **MprConfigInterfaceTransportGetHandle**, or **MprConfigInterfaceTransportEnum**.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the error codes on the following page.

Value	Meaning
ERROR_INVALID_PARAMETER	One of the following is true: *hMprConfig* is NULL. *hRouterInterface* is NULL. *phRouterIfTransport* is NULL.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigInterfaceCreate**, **MprConfigInterfaceEnum**, **MprConfigInterfaceGetHandle**, **MprConfigInterfaceTransportAdd**, **MprConfigInterfaceTransportEnum**, **MprConfigInterfaceTransportGetHandle**, **MprConfigServerConnect**

MprConfigInterfaceTransportSetInfo

The **MprConfigInterfaceTransportSetInfo** function updates the configuration information for the specified transport and interface.

```
DWORD MprConfigInterfaceTransportSetInfo(
  HANDLE hMprConfig,          // handle to the router
                              // configuration
  HANDLE hRouterInterface,    // handle to the interface
                              // configuration
  HANDLE hRouterIfTransport,  // handle to the transport
                              // configuration
  LPBYTE pInterfaceInfo,      // configuration information
                              // for transport and
                              // interface
  DWORD dwInterfaceInfoSize   // size of configuration
                              // information
);
```

Parameters

hMprConfig

Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

hRouterInterface

Handle to the interface configuration in which to update the information. Obtain this handle by calling **MprConfigInterfaceCreate** or **MprConfigInterfaceEnum**.

hRouterIfTransport

Handle to the transport configuration in which to update the information. Obtain this handle by calling **MprConfigInterfaceTransportGetHandle** or **MprConfigInterfaceTransportEnum**.

pInterfaceInfo

Pointer to an information header containing configuration information for the specified interface and transport. The router manager for the specified transport interprets this information. Use the Information Header Functions to manipulate information headers.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not update the configuration information.

dwInterfaceInfoSize

Size, in bytes, of the data pointed to by *pInterfaceInfo*.

This parameter is optional; the calling application may specify zero for this parameter. However, if *pInterfaceInfo* is not NULL, this parameter cannot be zero. For more information, see the Remarks section later in this topic.

Return Values

If the function succeeds, the return value is NO_ERROR. For more information, see the Remarks section later in this topic.

If the function fails, the return value is one of the following error codes.

Value	Meaning
ERROR_INVALID_PARAMETER	At least one of the following is true: *hMprConfig* is NULL. *hRouterInterface* is NULL. *hRouterIfTransport* is NULL.
ERROR_NO_SUCH_INTERFACE	The interface specified by *hRouterInterface* is no longer present in the router configuration, or the transport specified by *hRouterIfTransport* is no longer present on the interface.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Remarks

If the *pInterfaceInfo* parameter is NULL, **MprConfigInterfaceTransportSetInfo** does nothing and returns immediately with a value of NO_ERROR.

! Requirements

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

+ See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigInterfaceCreate**, **MprConfigInterfaceEnum**, **MprConfigInterfaceGetHandle**, **MprConfigInterfaceTransportEnum**, **MprConfigInterfaceTransportGetHandle**, **MprConfigServerConnect**

MprConfigServerBackup

The **MprConfigServerBackup** function creates a backup of the router-manager, interface, and phonebook configuration for the router.

```
DWORD MprConfigServerBackup (
  HANDLE hMprConfig,   // handle to the router configuration
  LPWSTR lpwsPath      // path to backup directory
)
```

Parameters

hMprConfig
Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

lpwsPath
Pointer to a Unicode string that contains the path to the directory where **MprConfigServerBackup** to write the backup files. This path should end with a trailing backslash.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the error codes on the following page.

Value	Meaning
ERROR_INVALID_PARAMETER	The *hMprConfig* parameter is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigServerConnect**, **MprConfigServerRestore**

MprConfigServerConnect

The **MprConfigServerConnect** function connects to the Windows 2000 router to be configured. Call this function before making any other calls to the server. The handle returned by this function is used in subsequent calls to configure interfaces and transports on the server.

```
DWORD MprConfigServerConnect(
  LPWSTR lpwsServerName,    // address of server name
  HANDLE *phMprConfig       // address of router
                            // configuration handle
);
```

Parameters

lpwsServername
Pointer to a Unicode string containing the name of the remote server to configure. If this parameter is NULL, the function returns a handle to the router configuration on the local machine .

phMprConfig
Pointer to a handle variable. On successful return, this variable contains a handle to the router configuration.

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 *phMprConfig* parameter is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigServerDisconnect**

MprConfigServerDisconnect

The **MprConfigServerDisconnect** function disconnects a connection made by a previous call to **MprConfigServerConnect**.

```
VOID MprConfigServerDisconnect(
  HANDLE hMprConfig    // handle to router configuration
);
```

Parameters

hMprConfig
Handle to a router configuration obtained from a previous call to **MprConfigServerConnect**.

Return Values

None.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigServerConnect**

MprConfigServerGetInfo

The **MprConfigServerGetInfo** function retrieves server-level configuration information for the specified router.

```
DWORD MprConfigServerGetInfo(
  HANDLE hMprConfig,    // handle to the router configuration
  DWORD dwLevel,        // level of information
  LPBYTE *lplpBuffer    // buffer to receive information
);
```

Parameters

hMprConfig
Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

dwLevel
Specifies the level of the information requested. This parameter must be zero.

lplpBuffer
Pointer to a pointer variable. On successful return, this variable will point to a buffer containing the retrieved information. Free the memory for this buffer using **MprConfigBufferFree**.

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	At least one of the following is true: The *hMprConfig* parameter is NULL. The *dwLevel* parameter is not zero. The *lplpBuffer* parameter is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Remarks

Currently, the only information returned by **MprConfigServerGetInfo** is the **fLanOnlyMode** flag.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MPR_SERVER_0**, **MprConfigBufferFree**, **MprConfigServerConnect**

MprConfigServerRestore

The **MprConfigServerRestore** function restores the router-manager, interface, and phonebook configuration from a backup created by a previous call to **MprConfigServerBackup**.

```
DWORD MprConfigServerRestore(
  HANDLE hMprConfig,    // handle to the router configuration
  LPWSTR lpwsPath       // path to backup directory
);
```

Parameters

hMprConfig,
Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

lpwsPath
Pointer to a Unicode string that contains the path to the directory where **MprConfigServerBackup** to write the backup files. This path should end with a trailing backslash.

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 *hMprConfig* parameter is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.

(continued)

(continued)

Value	Meaning
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigServerBackup**, **MprConfigServerConnect**

MprConfigTransportCreate

The **MprConfigTransportCreate** function adds the specified transport to the list of transports present in the specified router configuration.

```
DWORD MprConfigTransportCreate(
  HANDLE hMprConfig,              // handle to the router
                                  // configuration
  DWORD dwTransportId,            // identifier of the
                                  // transport/router manager
  LPWSTR lpwsTransportName        // address of the transport
                                  // name
  LPBYTE pGlobalInfo              // address of the global
                                  // info
  DWORD dwGlobalInfoSize,         // size of the global info
                                  // buffer
  LPBYTE pClientInterfaceInfo     // address of info for
                                  // client routers
  DWORD dwClientInterfaceInfoSize, // size of the client
                                  // info buffer
  LPWSTR lpwsDLLPath              // address of the router
                                  // manager DLL name
  HANDLE phRouterTransport        // handle to the transport
                                  // configuration
);
```

Parameters

hMprConfig
Handle to the router configuration to which to add the transport. Obtain this handle by calling **MprConfigServerConnect**.

dwTransportId
Value that identifies the transport to add to the configuration. This parameter also identifies the router manager for the transport.

lpwsTransportName
Pointer to a Unicode string containing the name of the transport being added. If this parameter is not specified, the *dwTransportId* parameter is converted into a string and used as the transport name.

pGlobalInfo
Pointer to an information header containing global information for the transport. The router manager for the transport interprets this information. Use the Information Header Functions to manipulate information headers.

dwGlobalInfoSize
Size, in bytes, of the data pointed to by the *pGlobalInfo* parameter.

pClientInterfaceInfo
Pointer to an information header containing default interface information for client routers. This information is used to configure dynamic interfaces for client routers for this transport. Use the Information Header Functions to manipulate information headers.

This parameter is optional; the calling application may specify NULL for this parameter.

dwClientInterfaceInfoSize
Size, in bytes, of the data pointed to by the *pClientInterfaceInfo* parameter. If the calling application specifies NULL for *pClientInterfaceInfo*, the calling application should specify zero for this parameter.

lpwsDLLPath
Pointer to a Unicode string containing the name of the router manager DLL for the specified transport. If this name is specified, the function sets the DLL path for this transport to this name.

This parameter is optional; the calling application may specify NULL for this parameter.

phRouterTransport
Pointer to a handle variable. On successful return, this variable contains a handle to the transport configuration.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the error codes on the following page.

Value	Meaning
ERROR_INVALID_PARAMETER	The *hMprConfig* parameter is NULL, or the *phRouterTransport* parameter is NULL, or both are NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Remarks

If the specified transport already exists, **MprConfigTransportCreate** does the equivalent of an **MprConfigTransportSetInfo** call using the supplied parameter values.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigServerConnect**

MprConfigTransportDelete

The **MprConfigTransportDelete** function removes the specified transport from the list of transports present in the specified router configuration.

```
DWORD MprConfigTransportDelete(
  HANDLE hMprConfig,       // handle to the router configuration
  HANDLE hRouterTransport  // handle to the transport
                           // configuration
);
```

Parameters

hMprConfig
Handle to the router configuration from which to remove the transport. Obtain this handle by calling **MprConfigServerConnect**.

hRouterTransport
Handle to the configuration for the transport being deleted. Obtain this handle by calling **MprConfigTransportCreate** or **MprConfigTransportGetHandle**.

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 *hMprConfig* parameter is NULL, or the *hRouterTransport* parameter is NULL, or both are NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigServerConnect**, **MprConfigTransportCreate**, **MprConfigTransportGetHandle**

MprConfigTransportEnum

The **MprConfigTransportEnum** function enumerates the transports configured on the router.

```
DWORD MprConfigTransportEnum(
  HANDLE hMprConfig,         // handle to the router configuration
  DWORD dwLevel,             // level of information requested
  LPBYTE *lplpBuffer,        // array of MPR_TRANSPORT_0
                             // structures
  DWORD dwPrefMaxLen,        // maximum length of data to return
  LPDWORD lpdwEntriesRead,   // number of entries enumerated
  LPDWORD lpdwTotalEntries,  // number of entries that
                             // could've been enumerated
  LPDWORD lpdwResumeHandle   // handle for continuing the
                             // enumeration
);
```

Parameters

hMprConfig

Handle to the router configuration for the transports. Obtain this handle by calling **MprConfigServerConnect**.

dwLevel

Level of the information returned through the *lplpBuffer* parameter. This parameter must be zero.

lplpBuffer

Pointer to a pointer variable. On successful return, this pointer will point to an array of **MPR_TRANSPORT_0** structures. Free this memory buffer by calling **MprConfigBufferFree**.

dwPrefMaxLen

Specifies the preferred maximum length of returned data (in 8-bit bytes). If this parameter is –1, the buffer returned will be large enough to hold all available information.

lpdwEntriesRead

Pointer to a **DWORD** variable. On successful return, this variable contains the total number of entries that were enumerated from the current resume position.

lpdwTotalEntries

Pointer to a **DWORD** variable. On successful return, this variable contains the total number of entries that could have been enumerated from the current resume position.

lpdwResumeHandle

Pointer to a **DWORD** variable. On successful return, this variable contains a resume handle that can be used to continue the enumeration. The handle should be zero on the first call, and left unchanged on subsequent calls. If on return, the handle is NULL, the enumeration cannot be continued. For other types of error returns, this handle is invalid.

This parameter is optional. If the caller specifies NULL for this parameter, the function will not return a resume handle.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the error codes on the following page.

Value	Meaning
ERROR_INVALID_PARAMETER	At least one of the following is true: *hMprConfig* is NULL. *dwLevel* is not zero. *lplpBuffer* is NULL. *dwPrefMaxLen* is smaller than the size of a single **MPR_TRANSPORT_0** structure. *lpdwEntriesRead* is NULL. *lpdwTotalEntries* is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_NO_MORE_ITEMS	No more entries available from the current resume position.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigBufferFree**, **MprConfigServerConnect**

MprConfigTransportGetHandle

The **MprConfigTransportGetHandle** function retrieves a handle to the specified transport's configuration in the specified router configuration.

```
DWORD MprConfigTransportGetHandle(
  HANDLE hMprConfig,            // handle to the router
                                // configuration
  DWORD dwTransportId,          // transport identifier
  HANDLE *phRouterTransport     // handle to the transport's
                                // configuration
);
```

Parameters

hMprConfig
Handle to the router configuration. The handle is obtained from a previous call to **MprConfigServerConnect**.

dwTransportId
Identifies the transport for which to retrieve the configuration.

phRouterTransport
Pointer to a handle variable. On successful return, this variable will contain a handle to the specified transport's configuration.

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 *hMprConfig* parameter and/or the *phRouterTransport* parameter is NULL.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
ERROR_UNKNOWN_PROTOCOL_ID	The transport specified by *dwTransportId* was not found in the router configuration.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

! Requirements

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

+ See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigServerConnect**

MprConfigTransportGetInfo

The **MprConfigTransportGetInfo** function retrieves the configuration for the specified transport from the router.

```
DWORD MprConfigTransportGetInfo(
  HANDLE hMprConfig,                    // handle to the router
                                        // configuration
  HANDLE hRouterTransport,              // handle to the transport
                                        // configuration
  LPBYTE *ppGlobalInfo,                 // global information for
                                        // the transport
  LPDWORD lpdwGlobalInfoSize,           // size of global
                                        // information
  LPBYTE *ppClientInterfaceInfo,        // interface information
                                        // for client routers
  LPDWORD lpdwClientInterfaceInfoSize,  // size of interface
                                        // information
  LPWSTR *lplpwsDLLPath                 // name of router manager DLL
);
```

Parameters

hMprConfig

Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

hRouterTransport

Handle to the transport configuration being retrieved. Obtain this handle by calling **MprConfigTransportCreate**, **MprConfigTransportGetHandle**, or **MprConfigTransportEnum**.

ppGlobalInfo

Pointer to a pointer variable. On successful return, this pointer variable points to an information header that contains global information for the transport. Use the Information Header Functions to manipulate information headers. Free this buffer by calling **MprConfigBufferFree**.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not retrieve the global information.

lpdwGlobalInfoSize

Pointer to a **DWORD** variable. On successful return, this variable contains the size, in bytes, of the buffer returned through the *ppGlobalInfo* parameter.

This parameter is optional; the calling application may specify NULL for this parameter. However, if *ppGlobalInfo* is not NULL, this parameter cannot be NULL.

ppClientInterfaceInfo

Pointer to a pointer variable. On successful return, this pointer points to an information header containing default interface information for client routers for this transport. Use the Information Header Functions to manipulate information headers. Free the buffer by calling **MprConfigBufferFree**.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not retrieve the interface information.

lpdwClientInterfaceInfoSize

Pointer to a **DWORD** variable. On successful return, this variable contains the size, in bytes, of the buffer returned through the *ppClientInterfaceInfo* parameter.

This parameter is optional; the calling application may specify NULL for this parameter. However, if *ppClientInterfaceInfo* is not NULL, this parameter cannot be NULL.

lplpwsDLLPath

Pointer to a pointer to a Unicode string. On successful return, the Unicode string contains the name of the router manager DLL for the specified transport.

This parameter is optional. If the calling application specifies NULL for this parameter, the function does not retrieve the name of the router manager DLL.

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	At least one of the following is true: *hMprConfig* is NULL *hRouterTransport* is NULL *ppGlobalInfo* is not NULL, but *lpdwGlobalInfoSize* is NULL. *ppClientInterfaceInfo* is not NULL, but *lpdwClientInterfaceInfo* is NULL.
ERROR_UNKNOWN_PROTOCOL_ID	The transport configuration corresponding to *hRouterTransport* was not found in the router configuration.
ERROR_NOT_ENOUGH_MEMORY	Insufficient resources to complete the operation.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Remarks

If the *pGlobalInfo*, *pClientInterfaceInfo*, and *lpwsDLLPath* parameters are all NULL, the function does nothing and returns a value of NO_ERROR.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigBufferFree**, **MprConfigServerConnect**, **MprConfigTransportCreate**, **MprConfigTransportEnum**, **MprConfigTransportGetHandle**

MprConfigTransportSetInfo

The **MprConfigTransportSetInfo** function changes the configuration for the specified transport in the specified router configuration.

```
DWORD MprConfigTransportSetInfo(
  HANDLE hMprConfig,              // handle to the router configuration
  HANDLE hRouterTransport,        // handle to the transport
                                  // configuration
  LPBYTE pGlobalInfo,             // global information for the
                                  // transport
  DWORD dwGlobalInfoSize,         // size of global information
  LPBYTE pClientInterfaceInfo,    // interface information
                                  // for client routers
  DWORD dwClientInterfaceInfoSize, // size of interface
                                  // information
  LPWSTR lpwsDLLPath              // name of router manager DLL
);
```

Parameters

hMprConfig

Handle to the router configuration. Obtain this handle by calling **MprConfigServerConnect**.

hRouterTransport

Handle to the transport configuration being updated. Obtain this handle by calling **MprConfigTransportCreate**, **MprConfigTransportGetHandle**, or **MprConfigTransportEnum**.

pGlobalInfo

Pointer to an information header containing global information for the transport. The router manager for the transport interprets this information. Use the Information Header Functions to manipulate information headers.

This parameter is optional; the calling application may specify NULL for this parameter.

dwGlobalInfoSize

Size, in bytes, of the data pointed to by *pGlobalInfo*. If the calling application specifies NULL for *pGlobalInfo*, the calling application should specify zero for this parameter.

pClientInterfaceInfo
Pointer to an information header containing default interface information for client routers. The information is used to configure dynamic interfaces for client routers for this transport. Use the Information Header Functions to manipulate information headers.

This parameter is optional; the calling application may specify NULL for this parameter.

dwClientInterfaceInfoSize
Size, in bytes, of the data pointed to by *pClientInterfaceInfo*. If the calling application specifies NULL for *pClientInterfaceInfo*, the calling application should specify zero for this parameter.

lpwsDLLPath
Name of the router manager DLL for the specified transport.

This parameter is optional; the calling application may specify NULL for this parameter.

Return Values

If the function succeeds, the return value is NO_ERROR. For more information, see the *Remarks* section later in this topic.

If the function fails, the return value is one of the following error codes.

Value	Meaning
ERROR_INVALID_PARAMETER	The *hMprConfig* parameter is NULL, the *hRouterTransport* parameter is NULL, or both are NULL.
ERROR_UNKNOWN_PROTOCOL_ID	The transport configuration corresponding to *hRouterTransport* was not found in the router configuration.
Other	Use **FormatMessage** to retrieve the system error message corresponding to the error code returned.

Remarks

Use **MprConfigTransportSetInfo** to set the transport's global information, default interface information, or the name of the router manager DLL for the transport.

MprConfigTransportSetInfo attempts to set the items in the order in which they appear in the parameter list:

1. Global information
2. Default interface information for client routers
3. Router manager DLL name

If **MprConfigTransportSetInfo** is unable to set any one of the items, it returns immediately without attempting to set the remaining items.

If the *pGlobalInfo*, *pClientInterfaceInfo*, and *lpwsDLLPath* parameters are all NULL, the function does nothing, returning a value of NO_ERROR.

Requirements

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

See Also

Router Administration Reference, Router Configuration Functions, **FormatMessage**, **MprConfigServerConnect**, **MprConfigTransportCreate**, **MprConfigTransportEnum**, **MprConfigTransportGetHandle**

Router Administration Structures

The Router Administration Functions and the Router Configuration Functions use the following structures:

IP_ADAPTER_BINDING_INFO
IP_LOCAL_BINDING
IPX_ADAPTER_BINDING_INFO
MPR_CREDENTIALSEX_0
MPR_IFTRANSPORT_0
MPR_INTERFACE_0
MPR_INTERFACE_1
MPR_INTERFACE_2
MPR_SERVER_0
MPR_TRANSPORT_0

IP_ADAPTER_BINDING_INFO

The **IP_ADAPTER_BINDING_INFO** structure contains IP-specific information for a particular network adapter.

```
typedef struct IP_ADAPTER_BINDING_INFO {
  DWORD              NumAddresses;
  DWORD              RemoteAddress;
  IP_LOCAL_BINDING   Address[1];
}IP_ADAPTER_BINDING_INFO, *PIP_ADAPTER_BINDING_INFO;
```

Members

NumAddresses
The number of IP addresses associated with this adapter.

RemoteAddress
This member is for WAN interfaces. It contains the address of the machine at the other end of a dial-up link.

Address
Pointer to an array of **IP_LOCAL_BINDING** structures. The array will contain a structure for each of the IP addresses associated with this adapter.

Remarks

Since an adapter may have more than one IP address, the **IP_ADAPTER_BINDING_INFO** structure maintains an array of **IP_LOCAL_BINDING** structures.

Requirements

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

See Also

Router Administration Reference, Router Administration Structures, **IP_LOCAL_BINDING**, **IPX_ADAPTER_BINDING_INFO**

IP_LOCAL_BINDING

The **IP_LOCAL_BINDING** structure contains IP address information for an adapter.

```
typedef struct IP_LOCAL_BINDING {
  DWORD    IPAddress;
  DWORD    Mask;
}IP_LOCAL_BINDING, * PIP_LOCAL_BINDING;
```

Members

IPAddress
An IP address for the adapter.

Mask
The network mask for the IP address.

Remarks

Since an adapter may have more than one IP address, the **IP_ADAPTER_BINDING_INFO** structure maintains an array of **IP_LOCAL_BINDING** structures.

Requirements

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

See Also

Router Administration Reference, Router Administration Structures, **IP_ADAPTER_BINDING_INFO**

IPX_ADAPTER_BINDING_INFO

The **IPX_ADAPTER_BINDING_INFO** structure contains IPX-specific information for a particular network adapter.

```
typedef struct IPX_ADAPTER_BINDING_INFO {
  ULONG    AdapterIndex;
  UCHAR    Network[4];
  UCHAR    LocalNode[6];
  UCHAR    RemoteNode[6];
  ULONG    MaxPacketSize;
  ULONG    LinkSpeed;
}IPX_ADAPTER_BINDING_INFO, * PIPX_ADAPTER_BINDING_INFO;
```

Members

AdapterIndex
Identifies the adapter that has been allocated for the interface.

Network[4]
The network number to which the adapter is bound.

LocalNode[6]
The node number to which the adapter is bound.

RemoteNode[6]
The node number of a peer router or client for demand dial point-to-point connections (for LAN connections this field will be set to the broadcast node address: i.e. 0xFFFFFFFFFFFF).

MaxPacketSize
The maximum packet size that can be transmitted over a connection established on the adapter.

LinkSpeed
The speed of the connection in 100 baud, for example, for 9600 baud connection this value is 96.

Requirements

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

See Also

Router Administration Reference, Router Administration Structures, **IP_ADAPTER_BINDING_INFO**

MPR_CREDENTIALSEX_0

The **MPR_CREDENTIALSEX_0** structure contains extended credentials information such as the information used by Extensible Authentication Protocols (EAPs).

```
typedef struct _MPR_CREDENTIALSEX_0 {
  DWORD    dwSize;               // size of info
  LPBYTE   lpbCredentialsInfo;   // extended credentials info
} MPR_CREDENTIALSEX_0, *PMPR_CREDENTIALSEX_0;
```

Members

dwSize
Specifies the size of the data pointed to by the **lpbCredentialsInfo** member.

lpbCredentialsInfo
Pointer to the extended credentials information.

Requirements

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

See Also

MprAdminInterfaceGetCredentialsEx, **MprAdminInterfaceSetCredentialsEx**

MPR_IFTRANSPORT_0

The **MPR_IFTRANSPORT_0** structure contains information for a particular interface transport.

```
typedef struct _MPR_IFTRANSPORT_0 {
  DWORD     dwTransportId;
  HANDLE    hIfTransport;
  WCHAR     wszIfTransportName[MAX_TRANSPORT_NAME_LEN+1];
} MPR_IFTRANSPORT_0, *PMPR_IFTRANSPORT_0;
```

Members

dwTransportId
The transport identifier.

hIfTransport
Handle to the interface transport.

wszIfTransportName
Unicode string containing the name of the interface transport.

Requirements

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

See Also

Router Administration Reference, Router Administration Structures, **MPR_TRANSPORT_0**, **MprConfigInterfaceTransportEnum**

MPR_INTERFACE_0

The **MPR_INTERFACE_0** structure contains information for a particular router interface.

```
typedef struct _MPR_INTERFACE_0 {
  WCHAR                    wszInterfaceName[MAX_INTERFACE_NAME_LEN+1];
  HANDLE                   hInterface;
  BOOL                     fEnabled;
  ROUTER_INTERFACE_TYPE    IfType;
  ROUTER_CONNECTION_STATE  dwConnectionState;
  DWORD                    fUnReachabilityReasons;
  DWORD                    dwLastError;
} MPR_INTERFACE_0, * PMPR_INTERFACE_0;
```

Members

wszInterfaceName
Pointer to a Unicode string containing the name of the interface.

hInterface
Handle to the interface.

fEnabled
TRUE if the interface is enabled. FALSE if the interface is administratively disabled.

IfType
Specifies the type of interface.

dwConnectionState
Current state of the interface, for example connected, disconnected, or unreachable. For a list of possible states, see **ROUTER_CONNECTION_STATE**.

fUnReachabilityReasons
Reason value. If the interface is unreachable, this member stores the reason. See Unreachability Reasons for a list of possible values.

dwLastError
Value that is set to nonzero if the interface fails to connect.

! Requirements

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

+ See Also

Router Administration Reference, Router Administration Structures, **MprAdminInterfaceEnum**, **MprAdminInterfaceGetInfo**, **ROUTER_CONNECTION_STATE**, **ROUTER_INTERFACE_TYPE**, Unreachability Reasons

MPR_INTERFACE_1

The **MPR_INTERFACE_1** structure contains information for a particular router interface.

```
typedef struct _MPR_INTERFACE_1 {
  WCHAR                  wszInterfaceName[MAX_INTERFACE_NAME_LEN+1];
  HANDLE                     hInterface;
  BOOL                       fEnabled;
  ROUTER_INTERFACE_TYPE      IfType;
  ROUTER_CONNECTION_STATE    dwConnectionState;
  DWORD                      fUnReachabilityReasons;
  DWORD                      dwLastError;
  LPWSTR                     lpwsDialoutHoursRestriction;
} MPR_INTERFACE_1, * PMPR_INTERFACE_1;
```

Members

wszInterfaceName
Pointer to a Unicode string containing the name of the interface.

hInterface
Handle to the interface.

fEnabled
TRUE if the interface is enabled. FALSE if the interface is administratively disabled.

IfType
Specifies the type of interface.

dwConnectionState
Current state of the interface, for example connected, disconnected, or unreachable. For a list of possible states, see **ROUTER_CONNECTION_STATE**.

fUnReachabilityReasons

Reason value. If the interface is unreachable, this member stores the reason. See Unreachability Reasons for a list of possible values.

dwLastError

Value that is set to nonzero if the interface fails to connect.

lpwsDialoutHoursRestriction

Pointer to a Unicode string specifying the times during which dial-out is restricted. The format for this string is:

```
<day><space><time range><space><time range> . . . <NULL><day>. . .
<NULL><NULL>
```

Where day is a numeral corresponding to a day of the week.

Numeral	Day
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

Time range is of the form HH:MM-HH:MM, using 24-hour notation.

The string <space> in the preceding syntax denotes a space character. The string <NULL> denotes a null character.

The restriction string is terminated by two consecutive null characters.

Example:

```
2 09:00-12:00 13:00-17:30<NULL>4 09:00-12:00 13:00-17:30<NULL><NULL>
```

The preceding string restricts dialout to Tuesdays and Thursdays from 9:00 AM to 12:00 PM and from 1:00 PM to 5:30 PM.

! Requirements

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

+ See Also

Router Administration Reference, Router Administration Structures, **MprAdminInterfaceCreate**, **MprAdminInterfaceEnum**, **MprAdminInterfaceGetInfo**, **ROUTER_CONNECTION_STATE**, **ROUTER_INTERFACE_TYPE**, Unreachability Reasons

MPR_INTERFACE_2

The **MPR_INTERFACE_2** structure contains information for a particular router interface.

```
typedef struct _MPR_INTERFACE_2 {
 WCHAR                    wszInterfaceName[MAX_INTERFACE_NAME_LEN+1];
 HANDLE                     hInterface;
 BOOL                       fEnabled;
 ROUTER_INTERFACE_TYPE      dwIfType;
 ROUTER_CONNECTION_STATE dwConnectionState;
 DWORD                      fUnReachabilityReasons;
 DWORD                      dwLastError;
 //
 // Demand dial-specific properties
 //
 DWORD         dwfOptions;
 //
 // Location/phone number
 //
 WCHAR         szLocalPhoneNumber[ RAS_MaxPhoneNumber + 1 ];
 PWCHAR        szAlternates;
 //
 // PPP/Ip
 //
 DWORD         ipaddr;
 DWORD         ipaddrDns;
 DWORD         ipaddrDnsAlt;
 DWORD         ipaddrWins;
 DWORD         ipaddrWinsAlt;
 //
 // Framing
 //
 DWORD         dwFrameSize;
 DWORD         dwfNetProtocols;
 //
 // Device
 //
 WCHAR         szDeviceType[ MPR_MaxDeviceType + 1 ];
 WCHAR         szDeviceName[ MPR_MaxDeviceName + 1 ];
 //
 // X.25
 //
 WCHAR         szX25PadType[ MPR_MaxPadType + 1 ];
 WCHAR         szX25Address[ MPR_MaxX25Address + 1 ];
 WCHAR         szX25Facilities[ MPR_MaxFacilities + 1 ];
```

```
    WCHAR       szX25UserData[ MPR_MaxUserData + 1 ];
    DWORD       dwChannels;
    //
    // Multilink
    //
    DWORD       dwSubEntries;
    DWORD       dwDialMode;
    DWORD       dwDialExtraPercent;
    DWORD       dwDialExtraSampleSeconds;
    DWORD       dwHangUpExtraPercent;
    DWORD       dwHangUpExtraSampleSeconds;
    //
    // Idle timeout
    //
    DWORD       dwIdleDisconnectSeconds;
    //
    // Entry Type
    //
    DWORD       dwType;
    //
    // EncryptionType
    //
    DWORD       dwEncryptionType;
    //
    // EAP information
    //
    DWORD       dwCustomAuthKey;
    DWORD       dwCustomAuthDataSize;
    LPBYTE      lpbCustomAuthData;
    //
    // Guid of the connection
    //
    GUID        guidId;
    //
    // Vpn Strategy
    //
    DWORD       dwVpnStrategy;
} MPR_INTERFACE_2, *PMPR_INTERFACE_2;
```

Members

wszInterfaceName

Pointer to a Unicode string containing the name of the interface.

hInterface

Handle to the interface.

fEnabled
TRUE if the interface is enabled. FALSE if the interface is administratively disabled.

IfType
Specifies the type of interface.

dwConnectionState
Current state of the interface, for example connected, disconnected, or unreachable. For a list of possible states, see **ROUTER_CONNECTION_STATE**.

fUnReachabilityReasons
Reason value. If the interface is unreachable, this member stores the reason. See Unreachability Reasons for a list of possible values.

dwLastError
Value that is set to nonzero if the interface fails to connect.

dwfOptions
A set of bit flags that specify connection options. You can set one or more of the following flags.

Flag	Description
MPRIO_SpecificIpAddr	If this flag is set, RRAS tries to use the IP address specified by **ipaddr** as the IP address for the dial-up connection. If this flag is not set, the value of the **ipaddr** member is ignored. Setting the MPRIO_SpecificIpAddr flag corresponds to selecting the Specify an IP Address setting in the TCP/IP settings dialog box. Clearing the MPRIO_SpecificIpAddr flag corresponds to selecting the Server Assigned IP Address setting in the TCP/IP settings dialog box. Currently, an IP address set in the phonebook entry properties or retrieved from a server overrides the IP address set in the network control panel.
MPRIO_SpecificNameServers	If this flag is set, RRAS uses the **ipaddrDns, ipaddrDnsAlt, ipaddrWins**, and **ipaddrWinsAlt** members to specify the name server addresses for the dial-up connection. If this flag is not set, RRAS ignores these members. Setting the MPRIO_SpecificNameServers flag corresponds to selecting the Specify Name Server Addresses setting in the TCP/IP Settings dialog box. Clearing the MPRIO_SpecificNameServers flag corresponds to selecting the Server Assigned Name Server Addresses setting in the TCP/IP Settings dialog box.

Flag	Description
MPRIO_IpHeaderCompression	If this flag is set, RRAS negotiates to use IP header compression on PPP connections. If this flag is not set, IP header compression is not negotiated. This flag corresponds to the Use IP Header Compression check box in the TCP/IP settings dialog box. It is generally advisable to set this flag because IP header compression significantly improves performance. The flag should be cleared only when connecting to a server that does not correctly negotiate IP header compression.
MPRIO_RemoteDefaultGateway	If this flag is set, the default route for IP packets is through the dial-up adapter when the connection is active. If this flag is clear, the default route is not modified. This flag corresponds to the Use Default Gateway on Remote Network check box in the TCP/IP settings dialog box.
MPRIO_DisableLcpExtensions	If this flag is set, RRAS disables the PPP LCP extensions defined in RFC 1570. This may be necessary to connect to certain older PPP implementations, but interferes with features such as server callback. Do not set this flag unless specifically required.
MPRIO_SwCompression	If this flag is set, software compression is negotiated on the link. Setting this flag causes the PPP driver to attempt to negotiate Compression Control Protocol (CCP) with the server. This flag should be set by default, but clearing it can reduce the negotiation period if the server does not support a compatible compression protocol.
MPRIO_RequireEncryptedPw	If this flag is set, only secure password schemes can be used to authenticate the client with the server. This prevents the PPP driver from using the PAP plain-text authentication protocol to authenticate the client. However, the MS-CHAP, MD5-CHAP and SPAP authentication protocols are supported. Clear this flag for increased interoperability, and set it for increased security. This flag corresponds to the Require Encrypted Password check box in the Security dialog box. See also MPRIO_RequireMsEncryptedPw.

(continued)

(continued)

Flag	Description
MPRIO_RequireMsEncryptedPw	If this flag is set, only the Microsoft secure password schemes can be used to authenticate the client with the server. This prevents the PPP driver from using the PAP plain-text authentication protocol, MD5-CHAP, or SPAP. The flag should be cleared for maximum interoperability and should be set for maximum security. This flag takes precedence over MPRIO_RequireEncryptedPw. This flag corresponds to the Require Microsoft Encrypted Password check box in the Security dialog box. See also MPRIO_RequireDataEncryption.
MPRIO_RequireDataEncryption	If this flag is set, data encryption must be negotiated successfully or the connection should be dropped. This flag is ignored unless MPRIO_RequireMsEncryptedPw is also set. This flag corresponds to the Require Data Encryption check box in the Security dialog box.
MPRIO_NetworkLogon	If this flag is set, RRAS logs on to the network after the point-to-point connection is established. **Windows NT/2000:** This flag currently has no effect under Windows NT.
MPRIO_UseLogonCredentials	If this flag is set, RRAS uses the user name, password, and domain of the currently logged-on user when dialing this entry. This flag is ignored unless MPRIO_RequireMsEncryptedPw is also set. Note that this setting is ignored by the **RasDial** function, where specifying empty strings for the **szUserName** and **szPassword** members of the **RASDIALPARAMS** structure gives the same result. This flag corresponds to the Use Current Username and Password check box in the Security dialog box.
MPRIO_PromoteAlternates	This flag has an effect when alternate phone numbers are defined by the **dwAlternateOffset** member. If this flag is set, an alternate phone number that connects successfully becomes the primary phone number, and the current primary phone number is moved to the alternate list. This flag corresponds to the check box in the Alternate Numbers dialog box.

Flag	Description
MPRIO_SecureLocalFiles	If this flag is set, RRAS checks for existing remote file system and remote printer bindings before making a connection with this entry. Typically, you set this flag on phonebook entries for public networks to remind users to break connections to their private network before connecting to a public network.
MPRIO_RequireEAP	If this flag is set, an Extensible Authentication Protocol (EAP) must be supported for authentication.
MPRIO_RequirePAP	If this flag is set, Password Authentication Protocol must be supported for authentication.
MPRIO_RequireSPAP	If this flag is set, Shiva's Password Authentication Protocol must be supported for authentication.
MPRIO_SharedPhoneNumbers	If this flag is set, phone numbers are shared.
MPRIO_RequireCHAP	If this flag is set, the Challenge Handshake Protocol must be supported for authentication.
MPRIO_RequireMsCHAP	If this flag is set, the Microsoft Challenge Handshake Protocol must be supported for authentication.

szLocalPhoneNumber
Specifies a null-terminated string containing a telephone number.

dwAlternates
Specifies the offset, in bytes, from the beginning of the structure to a list of consecutive null-terminated strings. The last string is terminated by two consecutive null characters. The strings are alternate phone numbers that RRAS dials in the order listed if the primary number (see **szLocalPhoneNumber**) fails to connect.

ipaddr
Specifies the IP address to be used while this connection is active. This member is ignored unless **dwfOptions** specifies the MPRIO_SpecificIpAddr flag.

ipaddrDns
Specifies the IP address of the DNS server to be used while this connection is active. This member is ignored unless **dwfOptions** specifies the MPRIO_SpecificNameServers flag.

ipaddrDnsAlt
Specifies the IP address of a secondary or backup DNS server to be used while this connection is active. This member is ignored unless **dwfOptions** specifies the MPRIO_SpecificNameServers flag.

ipaddrWins
Specifies the IP address of the WINS server to be used while this connection is active. This member is ignored unless **dwfOptions** specifies the MPRIO_SpecificNameServers flag.

ipaddrWinsAlt
Specifies the IP address of a secondary WINS server to be used while this connection is active. This member is ignored unless **dwfOptions** specifies the MPRIO_SpecificNameServers flag.

dwFrameSize
Specifies the network protocol frame size. The value should be either 1006 or 1500.

dwfNetProtocols
Specifies the network protocols to negotiate. This member can be a combination of the following flags.

Flag	Description
MPRNP_Ipx	Negotiate the IPX protocol.
MPRNP_Ip	Negotiate the TCP/IP protocol.

szDeviceType
Specifies a null-terminated string indicating the RRAS device type referenced by **szDeviceName**. This member can be one of the following string constants.

String	Description
MPRDT_Modem	A modem accessed through a COM port.
MPRDT_Isdn	An ISDN card with corresponding NDISWAN driver installed.
MPRDT_X25	An X.25 card with corresponding NDISWAN driver installed.
MPRDT_Vpn	A virtual private network connection.
MPRDT_Pad	A Packet Assembler/Disassembler.
MPRDT_Generic	Generic
MPRDT_Serial	Direct serial connection through a serial port.
MPRDT_FrameRelay	Frame Relay
MPRDT_Atm	Asynchronous Transfer Mode
MPRDT_Sonet	Sonet
MPRDT_SW56	Switched 56K Access
MPRDT_Irda	Infrared Data Association (IrDA) compliant device.
MPRDT_Parallel	Direct parallel connection through a parallel port.

szDeviceName
Contains a null-terminated string containing the name of a TAPI device to use with this phone-book entry, for example, "XYZ Corp 28800 External". To enumerate all available RAS-capable devices, use the **RasEnumDevices** function.

szX25PadType

Contains a null-terminated string that identifies the X.25 PAD type. Set this member to "" unless the entry should dial using an X.25 PAD.

Under Windows NT/Windows 2000, the **szX25PadType** string maps to a section name in PAD.INF.

szX25Address

Contains a null-terminated string that identifies the X.25 address to connect to. Set this member to "" unless the entry should dial using an X.25 PAD or native X.25 device.

szX25Facilities

Contains a null-terminated string that specifies the facilities to request from the X.25 host at connection. This member is ignored if **szX25Address** is an empty string ("").

szX25UserData

Contains a null-terminated string that specifies additional connection information supplied to the X.25 host at connection. This member is ignored if **szX25Address** is an empty string ("").

dwChannels

dwSubEntries

Specifies the number of multilink subentries associated with this entry. When calling **RasSetEntryProperties**, set this member to zero. To add subentries to a phone-book entry, use the **RasSetSubEntryProperties** function.

dwDialMode

Indicates whether RRAS should dial all of this entry's multilink subentries when the entry is first connected. This member can be one of the following values.

Value	Meaning
MPRDM_DialAll	Dial all subentries initially.
MPRDM_DialAsNeeded	Adjust the number of subentries as bandwidth is needed. RRAS uses the **dwDialExtraPercent, dwDialExtraSampleSeconds, dwDialHangUpExtraPercent,** and **dwHangUpExtraSampleSeconds** members to determine when to dial or disconnect a subentry.

dwDialExtraPercent

Specifies a percent of the total bandwidth available from the currently connected subentries. RRAS dials an additional subentry when the total bandwidth used exceeds **dwDialExtraPercent** percent of the available bandwidth for at least **dwDialExtraSampleSeconds** seconds.

This member is ignored unless the **dwDialMode** member specifies the MPRDM_DialAsNeeded flag.

dwDialExtraSampleSeconds

Specifies the number of seconds that current bandwidth usage must exceed the threshold specified by **dwDialExtraPercent** before RRAS dials an additional subentry.

This member is ignored unless the **dwDialMode** member specifies the MPRDM_DialAsNeeded flag.

dwHangUpExtraPercent

Specifies a percent of the total bandwidth available from the currently connected subentries. RRAS terminates (hangs up) an existing subentry connection when total bandwidth used is less than **dwHangUpExtraPercent** percent of the available bandwidth for at least **dwHangUpExtraSampleSeconds** seconds.

This member is ignored unless the **dwDialMode** member specifies the MPRDM_DialAsNeeded flag.

dwHangUpExtraSampleSeconds

Specifies the number of seconds that current bandwidth usage must be less than the threshold specified by **dwHangUpExtraPercent** before RRAS terminates an existing subentry connection.

This member is ignored unless the **dwDialMode** member specifies the MPRDM_DialAsNeeded flag.

dwIdleDisconnectSeconds

Specifies the number of seconds after which the connection is terminated due to inactivity. Note that unless the idle timeout is disabled, the entire connection is terminated if the connection is idle for the specified interval. This member can specify a number of seconds, or one of the following values.

Value	Meaning
MPRIDS_Disabled	There is no idle timeout for this connection.
MPRIDS_UseGlobalValue	Use the user preference value as the default.

dwType

The type of phone-book entry. This member can be one of the following types

Type	Description
MPRET_Phone	Phone line, for example, modem, ISDN, X.25.
MPRET_Vpn	Virtual Private Network
MPRET_Direct	Direct serial or parallel connection

dwEncryptionType

The type of encryption to use for Microsoft Point to Point Encryption (MPPE) with the connection. This member can be one of the values on the following page.

Value	Meaning
MPR_ET_None	No encryption
MPR_ET_Require	Require encryption
MPR_ET_RequireMax	Require maximum-strength encryption.
MPR_ET_Optional	Do encryption if possible. No encryption is okay.

The value of **dwEncryptionType** doesn't affect how passwords are encrypted. Whether passwords are encrypted and how passwords are encrypted is determined by the authentication protocol, e.g. PAP, MS-CHAP, EAP.

dwCustomAuthKey
This member is used for Extensible Authentication Protocol (EAP). This member contains the authentication key provided to the EAP vendor.

dwCustomAuthDataSize
Size of the data pointed to by **lpbCustomAuthData** member.

lpbCustomAuthData
Pointer to authentication data to use with Extensible Authentication Protocol (EAP)

guidId
The GUID (Globally Unique IDentifier) that represents this phone-book entry. This member is not settable.

dwVpnStrategy
The VPN strategy to use when dialing a VPN connection. This member can have one of the following values.

Value	Meaning
MPR_VS_Default	With this strategy, RRAS dials PPTP first. If PPTP fails, L2TP is attempted. Whichever protocol succeeds is tried first in subsequent dialing for this entry.
MPR_VS_PptpOnly	RAS will dial only PPTP.
MPR_VS_PptpFirst	RAS will always dial PPTP first.
MPR_VS_L2tpOnly	RAS will dial only L2TP.
MPR_VS_L2tpFirst	RAS will always dial L2TP first.

Remarks

The **MPR_INTERFACE_2** structure has a number of fields that are similar to fields the **RASENTRY** structure. The following fields from **RASENTRY** have no counterpart in **MPR_INTERFACE_2**:

dwCountryID
dwCountryCode
szAreaCode
dwFramingProtocol

Requirements

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

See Also

MPR_INTERFACE_0, **MPR_INTERFACE_1**, **MprAdminInterfaceGetInfo**, **MprAdminInterfaceSetInfo**

MPR_SERVER_0

The **MPR_SERVER_0** structure contains information for a particular Windows 2000 router.

```
typedef struct MPR_SERVER_0 {
  BOOL     fLanOnlyMode;
  DWORD    dwUptime;
  DWORD    dwTotalPorts;
  DWORD    dwPortsInUse;
} MPR_SERVER_0; *PMPR_SERVER_0;
```

Members

fLanOnlyMode
If TRUE, the Demand Dial Manager (DDM) is not running on the Windows 2000 router. If FALSE, the DDM is running on the Windows 2000 router.

dwUptime
The elapsed time (in seconds) since the router was started.

dwTotalPorts
The number of ports on the system.

dwPortsInUse
The number of ports currently in use.

Requirements

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

See Also

Router Administration Reference, Router Administration Structures, **MprAdminServerGetInfo**, **MprConfigServerGetInfo**

MPR_TRANSPORT_0

The **MPR_TRANSPORT_0** structure contains information for a particular transport.

```
typedef struct _MPR_TRANSPORT_0 {
  DWORD     dwTransportId;
  HANDLE    hTransport;
  WCHAR     wszTransportName[MAX_TRANSPORT_NAME_LEN+1];
} MPR_TRANSPORT_0, *PMPR_TRANSPORT_0;
```

Members

dwTransportId
The transport identifier.

hTransport
Handle to the transport.

wszTransportName
Unicode string containing the name of the transport.

Requirements

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

See Also

Router Administration Reference, Router Administration Structures, **MPR_IFTRANSPORT_0**

Router Administration Enumerated Types

The Router Administration Functions and the Router Configuration Functions use the following enumerated types:

ROUTER_CONNECTION_STATE
ROUTER_INTERFACE_TYPE

ROUTER_CONNECTION_STATE

The **ROUTER_CONNECTION_STATE** type enumerates the possible states of an interface on a Windows 2000 router.

```
typdef enum _ROUTER_CONNECTION_STATE {
  ROUTER_IF_STATE_UNREACHABLE,
  ROUTER_IF_STATE_DISCONNECTED,
```

(continued)

(continued)

```
  ROUTER_IF_STATE_CONNECTING,
  ROUTER_IF_STATE_CONNECTED
}ROUTER_CONNECTION_STATE
```

Values

ROUTER_IF_STATE_UNREACHABLE
  The interface is unreachable. See Unreachability Reasons for a list of possible reasons.

ROUTER_IF_STATE_DISCONNECTED
  The interface is reachable but disconnected.

ROUTER_IF_STATE_CONNECTING
  The interface is in the process of connecting

ROUTER_IF_STATE_CONNECTED
  The interface is connected.

Remarks

These states are sometimes referred to as "operational states."

! Requirements

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

+ See Also

Router Administration Reference, Router Administration Enumerated Types, **MPR_INTERFACE_0**, **MPR_INTERFACE_1**, Unreachability Reasons

ROUTER_INTERFACE_TYPE

The **ROUTER_INTERFACE_TYPE** type enumerates the different kinds of interfaces on a Windows 2000 router.

```
typedef enum _ROUTER_INTERFACE_TYPE {
  ROUTER_IF_TYPE_CLIENT,
  ROUTER_IF_TYPE_HOME_ROUTER,
  ROUTER_IF_TYPE_FULL_ROUTER,
  ROUTER_IF_TYPE_DEDICATED,
  ROUTER_IF_TYPE_INTERNAL,
  ROUTER_IF_TYPE_LOOPBACK,
  ROUTER_IF_TYPE_TUNNEL1,
  ROUTER_IF_TYPE_DIALOUT
} ROUTER_INTERFACE_TYPE;
```

Values

ROUTER_IF_TYPE_CLIENT
The interface is for a remote client.

ROUTER_IF_TYPE_HOME_ROUTER
The interface is for a home router.

ROUTER_IF_TYPE_FULL_ROUTER
The interface is for a full router.

ROUTER_IF_TYPE_DEDICATED
The interface is always connected. It is a LAN interface, or the interface is connected over a leased line.

ROUTER_IF_TYPE_INTERNAL
The interface is an internal-only interface.

ROUTER_IF_TYPE_LOOPBACK
The interface is a loopback interface.

ROUTER_IF_TYPE_TUNNEL1
The interface is a connected over a virtual private network (VPN).

ROUTER_IF_TYPE_DIALOUT
The interface is a dial-on-demand (DOD) interface.

! Requirements

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

+ See Also

Router Administration Reference, Router Administration Enumerated Types, **MPR_INTERFACE_0**, **MPR_INTERFACE_1**, **RAS_CONNECTION_2**

Unreachability Reasons

The following table lists constant values that indicate why an interface is currently unreachable.

Value	Meaning
MPR_INTERFACE_ADMIN_DISABLED	The administrator has disabled the interface.
MPR_INTERFACE_CONNECTION_FAILURE	The previous connection attempt failed. Look at the **dwLastError** member for the error code.
MPR_INTERFACE_DIALOUT_HOURS_RESTRICTION	Dial-out is not allowed at the current time.

(continued)

(continued)

Value	Meaning
MPR_INTERFACE_OUT_OF_RESOURCES	No ports or devices are available for use.
MPR_INTERFACE_SERVICE_PAUSED	The service is paused.
MPR_INTERFACE_NO_MEDIA_SENSE	The network cable is disconnected from the network card.
MPR_INTERFACE_NO_DEVICE	The netword card has been removed from the machine.

See Also

MPR_INTERFACE_0, MPR_INTERFACE_1, MIB_IFROW, MIB_IFSTATUS

Information Header Functions

Use the following functions to manipulate router information headers and blocks. An information header is composed of private meta-data and information blocks. Information blocks are arrays of information structures of various types.

The following functions manipulate information headers:

MprInfoCreate
MprInfoDelete
MprInfoDuplicate
MprInfoRemoveAll

The following functions manipulate information blocks within an information header:

MprInfoBlockAdd
MprInfoBlockFind
MprInfoBlockQuerySize
MprInfoBlockRemove
MprInfoBlockSet

Many of the router administration and configuration functions use information headers.

MprInfoBlockAdd

The **MprInfoBlockAdd** function creates a new header that is identical to an existing header with the addition of a new block.

```
DWORD MprInfoBlockAdd(
  LPVOID lpHeader,          // pointer to existing header
  DWORD dwInfoType,         // info type of new block
  DWORD dwItemSize,          // size of items in new block
  DWORD dwItemCount,         // number of items in new block
  LPBYTE lpItemData,         // data for new block
  LPVOID * lplpNewHeader     // new header (incl. new block)
);
```

Parameters

lpHeader
Pointer to the header to which to add the new block.

dwInfoType
Specifies the type of block to add. The types available depend on the transport: **IP** or **IPX**.

dwItemSize
Specifies the size of each item in the block to be added.

dwItemCount
Specifies the number of items of size *dwItemSize* to be copied as data for the new block.

lpItemData
Pointer to the data for the new block. The size in bytes of this buffer should be equal to the product of *dwItemSize* and *dwItemCount*.

lplpNewHeader
Pointer to a pointer variable that, on successful return, points to the new header.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_INVALID_PARAMETER	The *lpHeader*, *lplpNewHeader*, or *lpItemData* parameter is NULL, or a block of type *dwInfoType* already exists in the header.
Other	The call failed. Use **FormatMessage** to retrieve the error message corresponding to the returned error code.

Remarks

After adding an information block, obtain the new size of the information header by call **MprInfoBlockQuerySize**.

Requirements

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

See Also

FormatMessage, **MprInfoBlockRemove**, **MprInfoDuplicate**, **MprInfoBlockQuerySize**

MprInfoBlockFind

The **MprInfoBlockFind** function locates a specified block in an information header, and retrieves information about the block.

```
DWORD MprInfoBlockFind(
  LPVOID lpHeader,          // info header to search
  DWORD dwInfoType,         // block type to search for
  LPDWORD lpdwItemSize,     // size of data items in found
                            // block
  LPDWORD lpdwItemCount,    // number of data items in found
                            // block
  LPBYTE * lplpItemData     // data in found block
);
```

Parameters

lpHeader
Specifies the header in which to locate the block.

dwInfoType
Specifies the type of block to locate. The types available depend on the transport: **IP** or **IPX**.

lpdwItemSize
Pointer to a **DWORD** variable that, on successful return, specifies the size of each item in the located block's data. This parameter is optional. If this parameter is NULL, the item size will not be returned.

lpdwItemCount
Pointer to a **DWORD** variable that, on successful return, specifies the number of items of size *dwItemSize* contained in the block's data. This parameter is optional. If this parameter is NULL, the item count will not be returned.

lplpItemData
Pointer to a pointer that, on successful return, points to the data for the located block. This parameter is optional. If this parameter is NULL, the data will not be returned.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_INVALID_PARAMETER	The *lpInfoHeader* parameter is NULL.
ERROR_NOT_FOUND	No block of type *dwInfoType* exists in the header.
Other	The call failed. Use **FormatMessage** to retrieve the error message corresponding to the returned error code.

Requirements

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

See Also

FormatMessage

MprInfoBlockQuerySize

The **MprInfoBlockQuerySize** function returns the returns the size of the information header.

```
DWORD APIENTRY
MprInfoBlockQuerySize(
    IN    LPVOID          lpHeader
);
```

Parameters

lpHeader
  Pointer to the information header for which to return the size.

Return Values

MprInfoBlockQuerySize returns the size of the information header.

Requirements

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

See Also

MprInfoBlockAdd, **MprInfoBlockRemove**

MprInfoBlockRemove

The **MprInfoBlockRemove** function creates a new header that is identical to an existing header with a specified block removed.

```
DWORD MprInfoBlockRemove(
  LPVOID lpHeader,          // pointer to existing header
  DWORD dwInfoType,         // info type of block to remove
  LPVOID * lplpNewHeader    // new header with block removed
);
```

Parameters

lpHeader
Pointer to the header from which the block should be removed.

dwInfoType
Specifies the type of block to be removed. The types available depend on the transport: **IP** or **IPX**.

lplpNewHeader
Pointer to a pointer variable that, on successful, return, points to the new header.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_INVALID_PARAMETER	The *lpHeader* parameter is NULL, or no block of type *dwInfoType* exists in the header.
ERROR_NOT_ENOUGH_MEMORY	The memory allocation required for successful execution of **MprInfoBlockRemove** could not be completed.
Other	The call failed. Use **FormatMessage** to retrieve the error message corresponding to the returned error code.

Remarks

After removing an information block, obtain the new size of the information header by call **MprInfoBlockQuerySize**.

Requirements

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

See Also

FormatMessage, **MprInfoBlockAdd**, **MprInfoBlockQuerySize**

MprInfoBlockSet

The **MprInfoBlockSet** creates a new header that is identical to an existing header with a specified block modified.

```
DWORD MprInfoBlockSet(
  LPVOID lpHeader,          // pointer to existing header
  DWORD dwInfoType,         // info type of block to modify
  DWORD dwItemSize,         // new size of items in the
                            // modified block
  DWORD dwItemCount,        // new number of items in the
                            // modified block
  LPBYTE lpItemData,        // new data for the modified
                            // block
  LPVOID * lplpNewHeader    // new header (incl. modified
                            // block)
);
```

Parameters

lpHeader
Pointer to the header in which to modify the specified block.

dwInfoType
Specifies the type of block to change. The types available depend on the transport: **IP** or **IPX**.

dwItemSize
Specifies the size of each item in the block's new data.

dwItemCount
Specifies the number of items of size *dwItemSize* to be copied as the new data for the block.

lpItemData
Pointer to the new data for the block. This should point to a buffer with a size (in bytes) equal to the product of *dwItemSize* and *dwItemCount*.

lplpNewHeader
Pointer to a pointer variable that, on successful return, points to the new header.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_INVALID_PARAMETER	One (or more) required parameters is NULL, or no block of type *dwInfoType* exists in the header.
Other	The call failed. Use **FormatMessage** to retrieve the error message corresponding to the returned error code.

Requirements

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

See Also

FormatMessage, **MprInfoBlockAdd**, **MprInfoBlockRemove**, **MprInfoBlockSet**

MprInfoCreate

The **MprInfoCreate** function creates a new information header.

```
DWORD MprInfoCreate(
  DWORD dwVersion,          // version of info header
  LPVOID * lplpNewHeader    // pointer to new info header
);
```

Parameters

dwVersion
Specifies the version of the information header structure to be created. The only value currently supported is **RTR_INFO_BLOCK_VERSION**, as declared in Mprapi.h.

lplpNewHeader
Pointer to the allocated and initialized header.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the values on the following page.

Value	Description
ERROR_INVALID_PARAMETER	The *lplpNewHeader* parameter is NULL.
ERROR_NOT_ENOUGH_MEMORY	The requested memory allocation could not be completed.
Other	The call failed. Use FormatMessage to retrieve the error message corresponding to the returned error code.

Requirements

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

See Also

FormatMessage

MprInfoDelete

The **MprInfoDelete** function deletes an information header created using **MprInfoCreate**, or retrieved by **MprInfoBlockAdd**, **MprInfoBlockRemove**, or **MprInfoBlockSet**.

```
DWORD MprInfoDelete(
  LPVOID lpHeader    // pointer to header to delete
);
```

Parameters

lpHeader
Pointer to the header to be deallocated.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails the return value is one of the following values.

Value	Description
ERROR_INVALID_PARAMETER	The *lpHeader* parameter is NULL.
Other	The call failed. Use FormatMessage to retrieve the error message corresponding to the returned error code.

! Requirements

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

+ See Also

MprInfoBlockAdd, **MprInfoBlockRemove**, **MprInfoBlockSet**, **FormatMessage**

MprInfoDuplicate

The **MprInfoDuplicate** function duplicates an existing information header.

```
DWORD MprInfoDuplicate(
  LPVOID lpHeader,        // pointer to existing info header
  LPVOID * lplpNewHeader  // pointer to pointer to duplicate
);
```

Parameters

lpHeader
Pointer to the information header to duplicate.

lplpNewHeader
Pointer to a pointer variable. On successful return, this variable points to the new (duplicate) information header.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_INVALID_PARAMETER	The *lplpNewHeader* parameter is NULL.
ERROR_NOT_ENOUGH_MEMORY	The requested memory allocation could not be completed.
Other	The call failed. Use **FormatMessage** to retrieve the error message corresponding to the returned error code.

! Requirements

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

See Also

FormatMessage, **MprInfoCreate**

MprInfoRemoveAll

The **MprInfoRemoveAll** function removes all information blocks from the specified header.

```
DWORD APIENTRY
MprInfoRemoveAll(
    IN      LPVOID          lpHeader,
    OUT     LPVOID *        lplpNewHeader
);
```

Parameters

lpHeader
Pointer to the information header from which to remove all information blocks.

lplpNewHeader
Pointer to a pointer variable. On successful return, this variable points to the information header with all information blocks removed.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values:

ERROR_INVALID_PARAMETER
Either the *lpHeader* parameter is NULL or the *lplpNewHeader* parameter is NULL.

Other
Use FormatMessage to retrieve the error message corresponding to the returned error code.

Requirements

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

See Also

MprInfoBlockRemove

Functions That Use Information Blocks

The following table lists functions that use information blocks. Each of these functions has a parameter that points to either a global information block or an interface information block.

Function	Info block parameter
MprAdminInterfaceTransportAdd	pInterfaceInfo
MprAdminInterfaceTransportGetInfo	ppInterfaceInfo
MprAdminInterfaceTransportSetInfo	pInterfaceInfo
MprAdminTransportGetInfo	ppGlobalInfo
MprAdminTransportSetInfo	pGlobalInfo
MprConfigInterfaceTransportAdd	pInterfaceInfo
MprConfigInterfaceTransportGetInfo	ppInterfaceInfo
MprConfigInterfaceTransportSetInfo	pInterfaceInfo
MprConfigTransportCreate	pGlobalInfo
MprConfigTransportGetInfo	ppGlobalInfo
MprConfigTransportSetInfo	pGlobalInfo

Router Information Structures

The following reference pages describe the router information structures for the IP and IPX transports:

- IP Information Structures
- IPX Information Structures

IP Information Structures

The **MIB_IPFORWARDROW** is used for the IP transport. This structure is defined in Iprtrmib.h.

MIB_IPFORWARDROW

IPX Information Structures

The following information structures are for the IPX transport. The structures **IPX_TRAFFIC_FILTER_INFO** and **IPX_TRAFFIC_FILTER_GLOBAL_INFO** are defined in Ipxtfflt.h. The remaining structures are defined in Ipxrtdef.h.

IPX_ADAPTER_INFO
IPX_GLOBAL_INFO
IPX_IF_INFO
IPX_SERVER_ENTRY
IPX_STATIC_NETBIOS_NAME_INFO
IPX_STATIC_ROUTE_INFO
IPX_STATIC_SERVICE_INFO
IPX_TRAFFIC_FILTER_GLOBAL_INFO
IPX_TRAFFIC_FILTER_INFO
IPXWAN_IF_INFO

IPX_IF_INFO

The **IPX_IF_INFO** structure stores information for an IPX interface.

```
typedef struct _IPX_IF_INFO {
  ULONG    AdminState;       // admin state of the interface
  ULONG    NetbiosAccept;    // accept Netbios broadcast
                             // packets
  ULONG    NetbiosDeliver;   // deliver Netbios broadcast
                             // packets
} IPX_IF_INFO, *PIPX_IF_INFO;
```

Members

AdminState
Specifies the administrative state of the interface.

NetbiosAccept
Specifies whether to accept NetBIOS broadcast packets.

NetbiosDeliver
Specifies whether to deliver NetBIOS broadcast packets

Requirements

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

See Also

IPXWAN_IF_INFO

IPX_STATIC_SERVICE_INFO

The **IPX_STATIC_SERVICE_INFO** structure describes a particular static IPX service.

```
typedef struct _IPX_SERVER_ENTRY {
  USHORT    Type;
  UCHAR     Name[48];
  UCHAR     Network[4];
  UCHAR     Node[6];
  UCHAR     Socket[2];
  USHORT    HopCount;
  } IPX_SERVER_ENTRY, * PIPX_SERVER_ENTRY;

typedef IPX_SERVER_ENTRY IPX_STATIC_SERVICE_INFO;
```

Members

Type
Specifies the service type as defined by the SAP specification.

Name[48]
Specifies the service name as defined by SAP specifications.

Network[4]
Specifies the network number portion of the service address.

Node[6]
Specifies the node number portion of the service address.

Socket[2]
Specifies the socket number portion of service address.

HopCount
Specifies the service hop count.

Remarks

The **IPX_STATIC_SERVICE_INFO** structure is a typedef of the **IPX_SERVER_ENTRY** structure. The typedef is in Ipxrtdef.h.

Requirements

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

See Also

IPX_SERVER_ENTRY

IPXWAN_IF_INFO

The **IPXWAN_IF_INFO** structure stores the administrative state for an IPX WAN interface.

```
typedef struct _IPXWAN_IF_INFO {
  ULONG    AdminState;  // enable/disable IPXWAN negotiation
} IPXWAN_IF_INFO, *PIPXWAN_IF_INFO;
```

Members

AdminState

Specifies the administrative state of the interface. This member can be one of the following values. These value are defined in Ipxconst.h.

ADMIN_STATE_DISABLED
ADMIN_STATE_ENABLED
ADMIN_STATE_ENABLED_ONLY_FOR_NETBIOS_STATIC_ROUTING
ADMIN_STATE_ENABLED_ONLY_FOR_OPER_STATE_UP

! Requirements

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

+ See Also

Functions That Use Information Blocks, **IPX_IF_INFO**

Router Information Enumeration Types

Use the following information types when calling information header functions:

- **IP Info Types for Router Information Blocks**
- **IPX Info Types for Router Information Blocks**

IP Info Types for Router Information Blocks

The following information types are listed in Ipinfoid.h. Use these information types with the Information Header functions when running the IP transport.

IP_DEMAND_DIAL_FILTER_INFO
IP_IN_FILTER_INFO
IP_OUT_FILTER_INFO
IP_GLOBAL_INFO
IP_IFFILTER_INFO
IP_INTERFACE_STATUS_INFO
IP_MCAST_HEARBEAT_INFO
IP_MCAST_BOUNDARY_INFO
IP_PROT_PRIORITY_INFO
IP_ROUTE_INFO
IP_ROUTER_DISC_INFO

IPX Info Types for Router Information Blocks

The following information types are listed in Ipxrtdef.h. Use these information types with the router information block functions when running the IPX transport.

Info type	Info structure
IPX_ADAPTER_INFO_TYPE	**IPX_ADAPTER_INFO**
IPX_GLOBAL_INFO_TYPE	**IPX_GLOBAL_INFO**
IPX_INTERFACE_INFO_TYPE	**IPX_IF_INFO**
IPX_IN_TRAFFIC_FILTER_GLOBAL_INFO_TYPE	**IPX_TRAFFIC_FILTER_GLOBAL_INFO**
IPX_OUT_TRAFFIC_FILTER_GLOBAL_INFO_TYPE	**IPX_TRAFFIC_FILTER_GLOBAL_INFO**
IPX_IN_TRAFFIC_FILTER_INFO_TYPE	**IPX_TRAFFIC_FILTER_INFO**
IPX_OUT_TRAFFIC_FILTER_INFO_TYPE	**IPX_TRAFFIC_FILTER_INFO**
IPX_STATIC_NETBIOS_NAME_INFO_TYPE	**IPX_STATIC_NETBIOS_NAME_INFO**
IPX_STATIC_ROUTE_INFO_TYPE	**IPX_STATIC_ROUTE_INFO**
IPX_STATIC_SERVICE_INFO_TYPE	**IPX_STATIC_SERVICE_INFO**
IPXWAN_INTERFACE_INFO_TYPE	**IPXWAN_IF_INFO**

CHAPTER 7

Management Information Base (MIB)

MIB Overview

The Management Information Base (MIB) API makes it possible to query and set the values of MIB variables exported by one of the router managers or any of the routing protocols that the router managers service. By using this API, the router supports the Simple Network Management Protocol (SNMP).

In the SNMP framework, each manageable object in the router is represented by a variable that has a unique Object Identifier (OID). Corresponding to each OID is a value that represents the current state of the object. The collection of OIDs and values is referred to as a Management Information Base (MIB). The **MprAdminMib** calls allow a developer to specify an object by its OID and either query or write ("Set") the object's value.

To query and set MIB variables, the module that services the calls must define a set of data structures. These data structures include structures to use as Object Identifiers and structures that hold the values of the MIB variables being accessed. These data structures are opaque to all but the caller of the MIB function and the module servicing the call.

The module servicing the MIB call will be a router manager or one of the routing protocols. The caller must specify a router manager even if the call will be handled by one of the routing protocols. In such a case, the caller should specify the router manager that corresponds to the protocol family for that routing protocol. For example, if the Open Shortest Path First (OSPF) routing protocol were handling the MIB call, the caller would need to specify the IP Router Manager, since OSPF belongs to the IP protocol family. In each of the MIB functions, the *dwTransportId* parameter specifies a router manager, and the *RoutingPid* parameter specifies the routing protocol. The router manager also has a unique *RoutingPid*, since some of the MIB variables may be handled by the router manager itself.

The **MprAdminMib** functions can be called on a computer other than the one being administered. The **MprAdminMIB** functions that query or write values, take as a parameter a handle to the computer to administer. Use the **MprAdminMIBServerConnect** function to establish the connection to the remote computer and obtain this handle. After calling the necessary **MprAdminMIB** functions to accomplish a particular administrative task, call the **MprAdminMIBServerDisconnect** function to sever the connection to the remote computer.

The **MprAdminMIBEntryCreate** and **MprAdminMIBEntrySet** functions take as parameters an OID and a buffer which contains the new value for the object.

The **MprAdminMIBEntryGet**, **MprAdminMIBEntryGetFirst** and **MprAdminMIBEntryGetNext** functions take as parameters an OID and the address of a pointer variable. On successful return, the pointer variable points to a buffer that contains the value for the object. The caller should free the memory for this buffer by calling the **MprAdminMIBBufferFree** function.

The **MprAdminMIBEntryGetFirst** and **MprAdminMIBEntryGetNext** functions enable a developer to perform an "SNMP walk". Because the OIDs are ordered, each OID (and therefore each manageable object) has a "next" OID. An SNMP-Walk refers to traversing a portion of the MIB by reading (or writing) a sequence of OIDs.

All **MprAdminMib** calls pass through the Dynamic Interface Manager (DIM). Depending on the OID, DIM passes the call to one of the router managers. (Both IP and IPX support SNMP). Again, depending on the OID, the router manager may handle the call itself, or pass the call to one of its clients. All router clients are required to implement and export the following functions which correspond to the similarly named **MprAdminMIB** functions:

- MibCreate
- MibDelete
- MibSet
- MibGet
- MibGetFirst
- MibGetNext
- MibGetTrapInfo
- MibSetTrapInfo

Using the MIB API

This section contains examples that show how to use the MIB API to query and set variables:

Obtaining the MIB II Interfaces Table

The following code uses **MprAdminMIBEntryGet** to obtain the MIB II interfaces table.

```
HANDLE            _hMibSrv;
MIB_OPAQUE_QUERY  MibOpaqueQuery;
PMIB_OPAQUE_INFO  pMibOpaqueInfo = NULL;
DWORD  dwInSize, dwOutSize, dwResult;
PMIB_IFTABLE  pIntfTable;
```

```
MibOpaqueQuery.dwVarId = IF_TABLE;
dwInSize = sizeof( MIB_OPAQUE_QUERY );
dwOutSize = 0;

dwResult = MprAdminMIBEntryGet ( _hMibSrv,
                                  PID_IP,
                                  IPRTRMGR_PID,
                                  (PVOID)&MibOpaqueQuery,
                                  dwInSize,
                                  (PVOID *)&pMibOpaqueInfo,
                                  &dwOutSize );

if ( dwResult != NO_ERROR )
        return;

if ( pMibOpaqueInfo == NULL )
    return;

pIntfTable = ( PMIB_IFTABLE ) pMibOpaqueInfo -> rgbyData;
```

See Also

MIB_OPAQUE_INFO, **MIB_OPAQUE_QUERY**, **MprAdminMIBEntryGet**

MIB Reference

This section describes the reference elements used in the Management Information Base (MIB) API. Use these reference elements to query and set the values of the MIB variables exported by one of the router managers, or any of the routing protocols that the router manager services:

- MIB Functions
- MIB Structures
- Transport and Protocol Constants

MIB Functions

Use the following functions to query and set MIB variables:

MprAdminMIBBufferFree
MprAdminMIBEntryCreate
MprAdminMIBEntryDelete
MprAdminMIBEntryGet
MprAdminMIBEntryGetFirst
MprAdminMIBEntryGetNext
MprAdminMIBEntrySet
MprAdminMIBGetTrapInfo
MprAdminMIBServerConnect
MprAdminMIBServerDisconnect
MprAdminMIBSetTrapInfo

MprAdminMIBBufferFree

The **MprAdminMIBBufferFree** function frees buffers returned by the following functions:

- **MprAdminMIBEntryGet**
- **MprAdminMIBEntryGetFirst**
- **MprAdminMIBEntryGetNext**

```
VOID MprAdminMIBBufferFree(
  LPVOID pBuffer    // address of memory to free
);
```

Parameters

pBuffer
Pointer to a memory buffer to free.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_INVALID_PARAMETER	The *pBuffer* parameter is NULL.

Requirements

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

See Also

MIB Reference, MIB Functions, **MprAdminMIBEntryGet**, **MprAdminMIBEntryGetFirst**, **MprAdminMIBEntryGetNext**

MprAdminMIBEntryCreate

The **MprAdminMIBEntryCreate** function creates an entry for one of the variables exported by a routing protocol or router manager.

```
DWORD MprAdminMIBEntryCreate(
  MIB_SERVER_HANDLE hMibServer,    // handle to router
  DWORD dwTransportId,             // transport/router
                                   // manager ID
  DWORD dwRoutingPid,            // routing protocol ID
  LPVOID lpEntry,                // opaque data structure
  DWORD dwEntrySize              // size of opaque data
                                 // structure
);
```

Parameters

hMibServer
Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminMIBServerConnect**.

dwTransportId
Specifies the router manager that exported the variable.

dwRoutingPid
Specifies the routing protocol that exported the variable.

lpEntry
Pointer to an opaque data structure. The data structure's format is determined by the module servicing the call. The data structure should contain information that identifies the variable being created and the value to assign to the variable.

dwEntrySize
Specifies the size, in bytes, of the data pointed to by the *lpEntry* parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_ACCESS_DENIED	The caller does not have sufficient privileges.
ERROR_CANNOT_COMPLETE	The *dwRoutingPid* variable does not match any installed routing protocol.
ERROR_NOT_ENOUGH_MEMORY	There are insufficient resources to complete the operation.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed router manager.

Remarks

Do not pass in NULL for the *lpEntry* parameter because the resulting behavior is undefined.

! Requirements

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

+ See Also

MIB Reference, MIB Functions, MIB Structures, **MprAdminMIBServerConnect**, **MprAdminMIBEntryDelete**, Protocol Identifiers, Transport Identifiers

MprAdminMIBEntryDelete

The **MprAdminMIBEntryDelete** function deletes an entry for one of the variables that was exported by a routing protocol or router manager.

```
DWORD MprAdminMIBEntryDelete(
  MIB_SERVER_HANDLE hMibServer,    // handle to router
  DWORD dwTransportId,             // transport/router
                                   // manager ID
  DWORD dwRoutingPid,              // routing protocol ID
  LPVOID lpEntry,                  // opaque data structure
  DWORD dwEntrySize                // size of opaque data
                                   // structure
);
```

Parameters

hMibServer
Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminMIBServerConnect**.

dwTransportId
Specifies the router manager that exported the variable.

dwRoutingPid
Specifies the routing protocol that exported the variable.

lpEntry
Pointer to an opaque data structure. The data structure's format is determined by the module servicing the call. The data structure should contain information that identifies the variable to be deleted.

dwEntrySize
Specifies the size, in bytes, of the data pointed to by *lpEntry* parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_ACCESS_DENIED	The caller does not have sufficient privileges.
ERROR_CANNOT_COMPLETE	The *dwRoutingPid* variable does not match any installed routing protocol.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed router manager.

Remarks

Do not pass in NULL for the *lpEntry* parameter because the resulting behavior is undefined.

! Requirements

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

+ See Also

MIB Reference, MIB Functions, MIB Structures, **MprAdminMIBServerConnect**, **MprAdminMIBEntryCreate**, Protocol Identifiers, Transport Identifiers

MprAdminMIBEntryGet

The **MprAdminMIBEntryGet** function retrieves the value of one of the variables exported by a routing protocol or router manager.

```
DWORD MprAdminMIBEntryGet(
  MIB_SERVER_HANDLE hMibServer,    // handle to router
  DWORD dwTransportId,             // transport/router
                                   // manager ID
  DWORD dwRoutingPid,              // routing protocol ID
  LPVOID lpInEntry,                // address of data to
                                   // identify variable
  DWORD dwInEntrySize,             // size of data to
                                   // identify variable
  LPVOID * lplpOutEntry,           // address of output data
  LPDWORD lpdwOutEntrySize         // size of output data
);
```

Parameters

hMibServer
: Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminMIBServerConnect**.

dwTransportId
: Specifies the router manager that exported the variable.

dwRoutingPid
: Specifies the routing protocol that exported the variable.

lpInEntry
: Pointer to an opaque data structure. The data structure's format is determined by the module servicing the call. The data structure should contain information that identifies the variable being queried.

dwInEntrySize
: Specifies the size, in bytes, of the data structure pointed to by *lpInEntry*.

lplpOutEntry
: Pointer to a pointer variable. On successful return, this pointer variable points to an opaque data structure. The data structure's format is determined by the module servicing the call. The data structure contains the value of the variable that was queried. Free this memory by calling **MprAdminMIBBufferFree**.

lpdwOutEntrySize
: Pointer to a **DWORD** variable that, on successful return, contains the size in bytes of the data structure returned through the *lplpOutEntry* parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_ACCESS_DENIED	The caller does not have sufficient privileges.
ERROR_CANNOT_COMPLETE	The *dwRoutingPid* variable does not match any installed routing protocol.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed router manager.
ERROR_NOT_ENOUGH_MEMORY	There are insufficient resources to complete the operation.

Remarks

Do not pass in NULL for the *lpInEntry* parameter because the resulting behavior is undefined.

Requirements

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

See Also

MIB Reference, MIB Functions, MIB Structures, **MprAdminMIBBufferFree**, **MprAdminMIBServerConnect**, **MprAdminMIBEntrySet**, **MprAdminMIBEntryGetFirst**, **MprAdminMIBEntryGetNext**, Obtaining the MIB II Interfaces Table, Protocol Identifiers, Transport Identifiers

MprAdminMIBEntryGetFirst

The **MprAdminMIBEntryGetFirst** function retrieves the first variable of some set of variables exported by a protocol or router manager. The module servicing the call defines "first".

```
DWORD MprAdminMIBEntryGetFirst(
  MIB_SERVER_HANDLE hMibServer,    // handle to router
  DWORD dwTransportId,             // transport/router
                                   // manager ID
  DWORD dwRoutingPid,              // routing protocol ID
  LPVOID lpInEntry,                // address of data to
                                   // identify variable
  DWORD dwInEntrySize,             // size of data to
                                   // identify variable
  LPVOID * lplpOutEntry,           // address of output data
  LPDWORD lpdwOutEntrySize         // size of output data
);
```

Parameters

hMibServer
Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminMIBServerConnect**.

dwTransportId
Specifies the router manager that exported the variable.

dwRoutingPid
Specifies the routing protocol that exported the variable.

lpInEntry
Pointer to an opaque data structure. The data structure's format is determined by the module servicing the call. The data structure should contain information that identifies the variable being queried.

dwInEntrySize
Specifies the size in bytes of the data pointed to by *lpInEntry*.

lplpOutEntry
Pointer to a pointer variable. On successful return, this pointer variable points to an opaque data structure. The data structure's format is determined by the module servicing the call. The data structure contains the value of the first variable from the set of variables exported. Free this memory by calling **MprAdminMIBBufferFree**.

lpdwoutEntrySize
Pointer to a **DWORD** variable. On successful return, this variable contains the size, in bytes, of the data structure that was returned through the *lplpOutEntry* parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_ACCESS_DENIED	The caller does not have sufficient privileges.
ERROR_CANNOT_COMPLETE	The dwRoutingPid variable does not match any installed routing protocol.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed transport/router manager.
ERROR_NOT_ENOUGH_MEMORY	There are insufficient resources to complete the operation.

Remarks

Do not pass in NULL for the *lpInEntry* parameter because the resulting behavior is undefined.

Requirements

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

See Also

MIB Reference, MIB Functions, MIB Structures, **MprAdminMIBBufferFree**, **MprAdminMIBServerConnect**, **MprAdminMIBEntryGet**, **MprAdminMIBEntryGetNext**, Protocol Identifiers, Transport Identifiers

MprAdminMIBEntryGetNext

The **MprAdminMIBEntryGetNext** function retrieves the next variable of some set of variables exported by a protocol or router manager. The module servicing the call defines "next".

```
DWORD MprAdminMIBEntryGetNext(
  MIB_SERVER_HANDLE hMibServer,    // handle to router
  DWORD dwTransportId,             // transport/router
                                   // manager ID
  DWORD dwRoutingPid,              // routing protocol ID
  LPVOID lpInEntry,                // address of data to
                                   // identify variable
  DWORD dwInEntrySize,             // size of data to
                                   // identify variable
  LPVOID * lplpOutEntry,           // address of output data
  LPDWORD lpdwOutEntrySize         // size of output data
);
```

Parameters

hMibServer
Handle to the Windows 2000 router on which to execute this call. This handle is obtained from a previous call to **MprAdminMIBServerConnect**.

dwTransportId
Specifies the router manager that exported the variable.

dwRoutingPid
Specifies the routing protocol that exported the variable.

lpInEntry
Pointer to an opaque data structure. The data structure's format is determined by the module servicing the call. The data structure should contain information that identifies the variable being queried.

dwInEntrySize
Specifies the size, in bytes, of the data structure pointed to by *lpInEntry*.

lplpOutEntry
Pointer to a pointer variable. On successful return, this pointer variable points to an opaque data structure. The data structure's format is determined by the module servicing the call. The data structure contains the value of the ***next*** variable from the set of variables exported. Free this memory by calling **MprAdminMIBBufferFree**.

lpdwoutEntrySize
Pointer to a **DWORD** variable. On a successful return, this variable contains the size in bytes of the data structure returned through the *lplpOutEntry* parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_ACCESS_DENIED	The caller does not have sufficient privileges.
ERROR_CANNOT_COMPLETE	The *dwRoutingPid* variable does not match any installed routing protocol.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed router manager.
ERROR_NOT_ENOUGH_MEMORY	There are insufficient resources to complete the operation.

Remarks

Do not pass in NULL for the *lpInEntry* parameter because the resulting behavior is undefined.

Requirements

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

See Also

MIB Reference, MIB Functions, MIB Structures, **MprAdminMIBBufferFree**, **MprAdminMIBServerConnect**, **MprAdminMIBEntryGet**, **MprAdminMIBEntryGetFirst**, Protocol Identifiers, Transport Identifiers

MprAdminMIBEntrySet

The **MprAdminMIBEntrySet** function sets the value of one of the variables exported by a routing protocol or router manager.

```
DWORD MprAdminMIBEntrySet(
  MIB_SERVER_HANDLE hMibServer,    // handle to router
  DWORD dwTransportId,             // transport/router
                                   // manager ID
  DWORD dwRoutingPid,              // routing protocol ID
  LPVOID lpEntry,                  // opaque data structure
  DWORD dwEntrySize                // size of opaque data
                                   // structure
);
```

Parameters

hMibServer

Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminMIBServerConnect**.

dwTransportId

Specifies the router manager that exported the variable.

dwRoutingPid

Specifies the routing protocol that exported the variable.

lpEntry

Pointer to an opaque data structure. The data structure's format is determined by the module servicing the call. The data structure should contain information that identifies the variable being set and the value to be assigned to the variable.

dwEntrySize

Specifies the size, in bytes, of the data pointed to by the *lpEntry* parameter.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_ACCESS_DENIED	The caller does not have sufficient privileges.
ERROR_CANNOT_COMPLETE	The *dwRoutingPid* variable does not match any installed routing protocol.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed router manager.

Remarks

Do not pass in NULL for the *lpEntry* parameter because the resulting behavior is undefined.

! Requirements

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

+ See Also

MIB Reference, MIB Functions, MIB Structures, **MprAdminMIBServerConnect**, **MprAdminMIBEntryGet**, Protocol Identifiers, Transport Identifiers

MprAdminMIBGetTrapInfo

The **MprAdminMIBGetTrapInfo** function queries the module that set a trap event for more information about the trap.

```
DWORD MprAdminMIBGetTrapInfo(
  MIB_SERVER_HANDLE hMibServer,
  DWORD dwTransportId,
  DWORD dwRoutingPid,
  LPVOID lpInData,
  DWORD dwInDataSize,
  LPVOID * lplpOutData,
  LPDWORD lpdwOutDataSize
);
```

Parameters

hMibServer
[in] Handle to the Windows 2000 router on which to execute this call. Obtain this handle by calling **MprAdminMIBServerConnect**.

dwTransportId
[in] Specifies a **DWORD** variable that contains the protocol family identifier.

dwRoutingPid
[in] Specifies a **DWORD** variable that contains the identifier of the routing protocol.

lpInData
[in] Specifies the address of the input data.

dwInDataSize
[in] Specifies a **DWORD** variable that contains the size, in, bytes of the data pointed to by *lpInData*.

lplpOutData
[out] Specifies on successful return the address of a pointer to the output data.

lpdwOutDataSize
[in, out] Specifies on successful return, the address of a **DWORD** variable that contains the size, in bytes, of the data pointed to by **lplpOutData*.

Return Values

If the functions succeeds, the return value is NO_ERROR

If the function fails, the return value is one of the following error codes.

Value	Description
ERROR_ACCESS_DENIED	The caller does not have sufficient privileges.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed router manager.
ERROR_NOT_ENOUGH_MEMORY	There are insufficient resources to complete the operation.

! Requirements

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

+ See Also

MIB Reference, MIB Functions, **MprAdminMIBSetTrapInfo**

MprAdminMIBServerConnect

Call the **MprAdminMIBServerConnect** function to connect to the Windows 2000 router being administered. This call should be made before any other calls to the server. The handle returned by this function is used in subsequent MIB calls.

```
DWORD MprAdminMIBServerConnect(
  LPWSTR lpwsServerName,           // name of router
  MIB_SERVER_HANDLE * phMibServer   // handle to router
);
```

Parameters

lpwsServerNamer

Pointer to a Unicode string that contains the name of the remote server. If the caller specifies NULL for this parameter, the function returns a handle to the local server.

phMibServer

Pointer to a handle variable. On successful return, this variable contains a handle to the server.

Return Values

If the function succeeds, the return value is NO_ERROR.

! Requirements

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

See Also

MIB Reference, MIB Functions, **MprAdminMIBServerDisconnect**

MprAdminMIBServerDisconnect

The **MprAdminMIBServerDisconnect** function disconnects the connection made by a previous call to **MprAdminMIBServerConnect**.

```
DWORD MprAdminMIBServerDisconnect(
  MIB_SERVER_HANDLE hMibServer    // handle to router
);
```

Parameters

hMibServer
Handle to the Windows 2000 router from which to disconnect. Obtain this handle by calling **MprAdminMIBServerConnect**.

Return Values

If the function succeeds, the return value is NO_ERROR.

Requirements

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

See Also

MIB Reference, MIB Functions, **MprAdminMIBServerConnect**

MprAdminMIBSetTrapInfo

The **MprAdminMIBSetTrapInfo** function passes in a handle to an event which is signaled whenever a TRAP needs to be issued.

```
DWORD MprAdminMIBSetTrapInfo(
  DWORD dwTransportId,
  DWORD dwRoutingPid,
  HANDLE hEvent,
  LPVOID lpInData,
  DWORD dwInDataSize,
  LPVOID * lplpOutData,
  LPDWORD lpdwOutDataSize
);
```

Parameters

dwTransportId
[in] Specifies a **DWORD** variable that contains the protocol family identifier.

dwRoutingPid
[in] Specifies a **DWORD** variable that contains the identifier of the routing protocol.

hEvent
[in] Handle to an event that is signaled when a TRAP needs to be issued.

lpInData
[in] Pointer to the input data.

dwInDataSize
[in] Specifies a **DWORD** variable that contains the size in bytes of the data pointed to by *lpInData*.

lplpOutData
[out] Specifies on successful return, the address of a pointer to the output data.

lpdwOutDataSize
[in, out] Pointer to a **DWORD** variable that contains the size in bytes of the data pointed to by **lplpOutData*.

Return Values

If the functions succeeds, the return value is NO_ERROR

If the function fails, the return value is one of the following error codes.

Value	Description
ERROR_ACCESS_DENIED	The caller does not have sufficient privileges.
ERROR_UNKNOWN_PROTOCOL_ID	The *dwTransportId* value does not match any installed router manager.
ERROR_NOT_ENOUGH_MEMORY	There are insufficient resources to complete the operation.

Requirements

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

See Also

MIB Reference, Transport and Protocol Constants, **MprAdminMIBGetTrapInfo**

MIB Structures

Use the following structures with the MIB functions to get and set MIB variables. These structures are defined in Iprtrmib.h.

MIB_BEST_IF
MIB_ICMP
MIB_IFNUMBER
MIB_IFROW
MIB_IFSTATUS
MIB_IFTABLE
MIB_IPADDRROW
MIB_IPADDRTABLE
MIB_IPFORWARDNUMBER
MIB_IPFORWARDROW
MIB_IPFORWARDTABLE
MIB_IPMCAST_GLOBAL
MIB_IPMCAST_IF_ENTRY
MIB_IPMCAST_IF_TABLE
MIB_IPMCAST_MFE
MIB_IPMCAST_MFE_STATS
MIB_IPMCAST_OIF
MIB_IPMCAST_OIF_STATS
MIB_IPNETROW
MIB_IPNETTABLE
MIB_IPSTATS
MIB_MFE_STATS_TABLE
MIB_MFE_TABLE
MIB_OPAQUE_INFO
MIB_OPAQUE_QUERY
MIB_PROXYARP
MIB_TCPROW
MIB_TCPSTATS
MIB_TCPTABLE
MIB_UDPROW
MIB_UDPSTATS
MIB_UDPTABLE
MIBICMPINFO
MIBICMPSTATS

MIB_BEST_IF

The **MIB_BEST_IF** structure stores the index of the interface that has the best route to a particular destination address.

```
typedef struct _MIB_BEST_IF {
  DWORD   dwDestAddr;    // destination address
  DWORD   dwIfIndex;     // best interface for that dest addr
} MIB_BEST_IF, *PMIB_BEST_IF;
```

Members

dwDestAddr
Specifies the IP address of the destination.

dwIfIndex
Specifies the index of the interface that has the best route to the destination address specified by the **dwDestAddr** member.

Requirements

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

See Also

GetBestInterface, **GetBestRoute**

MIB_ICMP

The **MIB_ICMP** structure contains the Internet Control Message Protocol (ICMP) statistics for a particular computer.

```
typedef struct _MIB_ICMP {
  MIBICMPINFO   stats;    // contains ICMP stats
} MIB_ICMP,*PMIB_ICMP;
```

Members

stats

Specifies a **MIBICMPINFO** structure that contains the ICMP statistics for the computer.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

+ See Also

GetIcmpStatistics, **MIBICMPINFO**

MIB_IFNUMBER

The **MIB_IFNUMBER** structure stores the number of interfaces on a particular computer.

```
typedef struct _MIB_IFNUMBER {
  DWORD   dwValue;    // number of interfaces
} MIB_IFNUMBER, *PMIB_IFNUMBER;
```

Members

dwValue

Specifies the number of interfaces on the computer.

! Requirements

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

+ See Also

MIB_IFTABLE

MIB_IFROW

The **MIB_IFROW** structure stores information about a particular interface.

```
typedef struct _MIB_IFROW {
  WCHAR   wszName[MAX_INTERFACE_NAME_LEN];
  DWORD   dwIndex;    // index of the interface
  DWORD   dwType;     // type of interface
  DWORD   dwMtu;      // max transmission unit
  DWORD   dwSpeed;    // speed of the interface
  DWORD   dwPhysAddrLen;      // length of physical address
  BYTE    bPhysAddr[MAXLEN_PHYSADDR]; // physical address of adapter
  DWORD   dwAdminStatus;      // administrative status
  DWORD   dwOperStatus;       // operational status
  DWORD   dwLastChange;       // last time operational
                              // status changed
  DWORD   dwInOctets;         // octets received
  DWORD   dwInUcastPkts;      // unicast packets received
  DWORD   dwInNUcastPkts;     // non-unicast packets received
  DWORD   dwInDiscards;       // received packets discarded
  DWORD   dwInErrors;         // erroneous packets received
  DWORD   dwInUnknownProtos;  // unknown protocol
                              // packets received
  DWORD   dwOutOctets;        // octets sent
  DWORD   dwOutUcastPkts;     // unicast packets sent
  DWORD   dwOutNUcastPkts;    // non-unicast packets sent
  DWORD   dwOutDiscards;      // outgoing packets discarded
  DWORD   dwOutErrors;        // erroneous packets sent
  DWORD   dwOutQLen;          // output queue length
  DWORD   dwDescrLen;         // length of bDescr member
  BYTE    bDescr[MAXLEN_IFDESCR];  // interface description
} MIB_IFROW,*PMIB_IFROW;
```

Members

wszName[MAX_INTERFACE_NAME_LEN]
Pointer to a Unicode string that contains the name of the interface.

dwIndex
Specifies the index that identifies the interface.

dwType
Specifies the type of interface.

dwMtu
Specifies the Maximum Transmission Unit (MTU).

dwSpeed
Specifies the speed of the interface in bits per second.

dwPhysAddrLen
Specifies the length of the physical address specified by the **bPhysAddr** member.

bPhysAddr[MAXLEN_PHYSADDR]
Specifies the physical address of the adapter for this interface.

dwAdminStatus
Specifies the interface is administratively enabled or disabled.

dwOperStatus
Specifies the operational status of the interface. This member can be one of the following values:

MIB_IF_OPER_STATUS_NON_OPERATIONAL
MIB_IF_OPER_STATUS_UNREACHABLE
MIB_IF_OPER_STATUS_DISCONNECTED
MIB_IF_OPER_STATUS_CONNECTING
MIB_IF_OPER_STATUS_CONNECTED
MIB_IF_OPER_STATUS_OPERATIONAL

dwLastChange
Specifies the last time the operational status changed.

dwInOctets
Specifies the number of octets of data received through this interface.

dwInUcastPkts
Specifies the number of unicast packets received through this interface.

dwInNUcastPkts
Specifies the number of non-unicast packets received through this interface. This includes broadcast and multicast packets.

dwInDiscards
Specifies the number of incoming packets that were discarded even though they did not have errors.

dwInErrors
Specifies the number of incoming packets that were discarded because of errors.

dwInUnknownProtos
Specifies the number of incoming packets that were discarded because the protocol was unknown.

dwOutOctets
Specifies the number of octets of data sent through this interface.

dwOutUcastPkts
Specifies the number of unicast packets sent through this interface.

dwOutNUcastPkts
Specifies the number of non-unicast packets sent through this interface. This includes broadcast and multicast packets.

dwOutDiscards
Specifies the number of outgoing packets that were discarded even though they did not have errors.

dwOutErrors
Specifies the number of outgoing packets that were discarded because of errors.

dwOutQLen
Specifies the output queue length.

dwDescrLen
Specifies the length of the **bDescr** member.

bDescr[MAXLEN_IFDESCR]
Contains a description of the interface.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

See Also

GetIfEntry, **MIB_IFSTATUS**, **MIB_IFTABLE**, **MPR_INTERFACE_0**

MIB_IFSTATUS

The **MIB_IFSTATUS** structure stores status information for a particular interface.

```
typedef struct _MIB_IFSTATUS {
  DWORD   dwIfIndex;
  DWORD   dwAdminStatus;
  DWORD   dwOperationalStatus;
  BOOL    bMHbeatActive;
  BOOL    bMHbeatAlive;
} MIB_IFSTATUS, *PMIB_IFSTATUS;
```

Members

dwIfIndex
Specifies the index that identifies the interface.

dwAdminStatus
Specifies the administrative status of the interface, that is, whether the interface is administratively enabled or disabled.

dwOperationalStatus
Specifies the operational status of the interface. See ***ROUTER_CONNECTION_STATE*** for a list of the possible operational states.

bMHbeatActive
Specifies whether multicast heartbeat detection is enabled. A value of TRUE indicates that heartbeat detection is enabled. A value of FALSE indicates that heartbeat detection is disabled.

bMHbeatAlive

Specifies whether the multicast heartbeat dead interval has been exceeded. A value of TRUE indicates that the interval has been exceeded. A value of FALSE indicates that the interval has not been exceeded.

! Requirements

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

+ See Also

MIB_IFROW, **MPR_INTERFACE_0**, **MPR_INTERFACE_1**

MIB_IFTABLE

The **MIB_IFTABLE** structure contains a table of interface entries.

```
typedef struct _MIB_IFTABLE {
  DWORD     dwNumEntries;      // number of entries in table
  MIB_IFROW table[ANY_SIZE];   // array of interface entries
} MIB_IFTABLE, *PMIB_IFTABLE;
```

Members

dwNumEntries

Specifies the number of interface entries in the array.

table[ANY_SIZE]

Pointer to a table of interface entries implemented as an array of **MIB_IFROW** structures.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

+ See Also

GetIfTable, **MIB_IFNUMBER**, **MIB_IFROW**

MIB_IPADDRROW

The **MIB_IPADDRROW** specifies information for a particular IP address.

```
typedef struct _MIB_IPADDRROW {
  DWORD   dwAddr;              // IP address
  DWORD   dwIndex;             // interface index
  DWORD   dwMask;              // subnet mask
  DWORD   dwBCastAddr;         // broadcast address
  DWORD   dwReasmSize;         // rassembly size
  unsigned short   unused1;    // not currently used
  unsigned short   unused2;    // not currently used
} MIB_IPADDRROW, *PMIB_IPADDRROW;
```

Members

dwAddr
Specifies the IP address.

dwIndex
Specifies the index of the interface associated with this IP address.

dwMask
Specifies the subnet mask for the IP address.

dwBCastAddr
Specifies the broadcast address. A broadcast address is typically the IP address with the host portion set to either all zeros or all ones.

dwReasmSize
Specifies the maximum reassembly size for received datagrams

unused1
This member is not currently used.

unused2
This member is not currently used.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

See Also

MIB_IPADDRTABLE

MIB_IPADDRTABLE

The **MIB_IPADDRTABLE** structure contains a table of IP address entries.

```
typedef struct _MIB_IPADDRTABLE {
  DWORD          dwNumEntries;    // number of entries in
```

```
                              // the table
  MIB_IPADDRROW table[ANY_SIZE]; // array of IP address
                              // entries
} MIB_IPADDRTABLE, *PMIB_IPADDRTABLE;
```

Members

dwNumEntries

Specifies the number of IP address entries in the table.

table[ANY_SIZE]

Pointer to a table of IP address entries implemented as an array of **MIB_IPADDRROW** structures.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

+ See Also

GetIpAddrTable, **MIB_IPADDRROW**

MIB_IPFORWARDNUMBER

The **MIB_IPFORWARDNUMBER** stores the number of routes in a particular IP routing table.

```
typedef struct _MIB_IPFORWARDNUMBER {
  DWORD   dwValue;
} MIB_IPFORWARDNUMBER, *PMIB_IPFORWARDNUMBER;
```

Members

dwValue

Specifies the number of routes in the IP routing table.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

+ See Also

GetIpForwardTable, **MIB_IPFORWARDROW**, **MIB_IPFORWARDTABLE**

MIB_IPFORWARDROW

The **MIB_IPFORWARDROW** structure contains information that describes an IP network route.

```
typedef struct _MIB_IPFORWARDROW {
  DWORD   dwForwardDest;       // IP addr of destination
  DWORD   dwForwardMask;       // subnetwork mask of
                               // destination
  DWORD   dwForwardPolicy;     // conditions for multi-path
                               // route
  DWORD   dwForwardNextHop;    // IP address of next hop
  DWORD   dwForwardIfIndex;    // index of interface
  DWORD   dwForwardType;       // route type
  DWORD   dwForwardProto;      // protocol that generated
                               // route
  DWORD   dwForwardAge;        // age of route
  DWORD   dwForwardNextHopAS;  // autonomous system number
                               // of next hop
  DWORD   dwForwardMetric1;    // protocol-specific metric
  DWORD   dwForwardMetric2;    // protocol-specific metric
  DWORD   dwForwardMetric3;    // protocol-specific metric
  DWORD   dwForwardMetric4;    // protocol-specific metric
  DWORD   dwForwardMetric5;    // protocol-specific metric
} MIB_IPFORWARDROW, *PMIB_IPFORWARDROW;
```

Members

dwForwardDest
The IP address of the destination host.

dwForwardMask
The subnet mask of the destination host.

dwForwardPolicy
Specifies the set of conditions that would cause the selection of a multi-path route. This member is typically in IP TOS format. For more information, see *RFC 1354.*

dwForwardNextHop
Specifies the IP address of the next hop in the route.

dwForwardIfIndex
Specifies the index of the interface for this route.

dwForwardType
Specifies the route type as defined in RFC 1354. The list on the following page shows the possible values for this member.

Value	Meaning
4	The next hop is not the final destination (remote route).
3	The next hop is the final destination (local route).
2	The route is invalid.
1	Other.

dwForwardProto
Specifies the protocol that generated the route. See *Protocol Identifiers* for a list of possible protocols.

dwForwardAge
Specifies the age of the route in seconds.

dwForwardNextHopAS
Specifies the autonomous system number of the next hop.

dwForwardMetric1
Specifies a routing-protocol-specific metric value. This metric value is documented in RFC 1354.

dwForwardMetric2
Specifies a routing-protocol-specific metric value. This metric value is documented in RFC 1354.

dwForwardMetric3
Specifies a routing-protocol-specific metric value. This metric value is documented in RFC 1354.

dwForwardMetric4
Specifies a routing-protocol-specific metric value. This metric value is documented in RFC 1354.

dwForwardMetric5
Specifies a routing-protocol-specific metric value. This metric value is documented in RFC 1354.

! Requirements

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

+ See Also

CreateIpForwardEntry, **DeleteIpForwardEntry**, **MIB_IPFORWARDTABLE**, **SetIpForwardEntry**

MIB_IPFORWARDTABLE

The **MIB_IPFORWARDTABLE** structure contains a table of IP route entries.

```
typedef struct _MIB_IPFORWARDTABLE {
  DWORD               dwNumEntries;     // number of entries
                                        // in the table
  MIB_IPFORWARDROW    table[ANY_SIZE]; // array of route
                                        // entries
} MIB_IPFORWARDTABLE, *PMIB_IPFORWARDTABLE;
```

Members

dwNumEntries
Specifies the number of route entries in the table.

table[ANY_SIZE]
Pointer to a table of route entries implemented as an array of **MIB_IPFORWARDROW** structures.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

See Also

GetIpForwardTable, **MIB_IPFORWARDNUMBER**, **MIB_IPFORWARDROW**

MIB_IPMCAST_GLOBAL

The **MIB_IPMCAST_GLOBAL** structure stores global information for IP multicast on a particular computer.

```
typedef struct _MIB_IPMCAST_GLOBAL {
  DWORD   dwEnable;
} MIB_IPMCAST_GLOBAL, *PMIB_IPMCAST_GLOBAL;
```

Members

dwEnable
Specifies whether IP multicast is enabled on the computer.

Requirements

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

See Also

MIB_IPMCAST_IF_ENTRY, MIB_IPMCAST_MFE, MIB_IPMCAST_OIF

MIB_IPMCAST_IF_ENTRY

The **MIB_IPMCAST_IF_ENTRY** stores information about an IP multicast interface.

```
typedef struct _MIB_IPMCAST_IF_ENTRY {
  DWORD   dwIfIndex;
  DWORD   dwTtl;
  DWORD   dwProtocol;
  DWORD   dwRateLimit;
  ULONG   ulInMcastOctets;
  ULONG   ulOutMcastOctets;
} MIB_IPMCAST_IF_ENTRY, *PMIB_IPMCAST_IF_ENTRY;
```

Members

dwIfIndex
Specifies the index of this interface.

dwTtl
Specifies the time-to-live value for this interface.

dwProtocol
Specifies the multicast routing protocol that owns this interface.

dwRateLimit
Specifies the rate limit of this interface.

ulInMcastOctets
Specifies the number of octets of multicast data received through this interface.

ulOutMcastOctets
Specifies the number of octets of multicast data sent through this interface.

Requirements

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

See Also

MIB_IPMCAST_IF_TABLE

MIB_IPMCAST_IF_TABLE

The **MIB_IPMCAST_IF_TABLE** structure contains a table of IP multicast interface entries.

```
typedef struct _MIB_IPMCAST_IF_TABLE {
  DWORD                 dwNumEntries;     // number of
                                          // entries
  MIB_IPMCAST_IF_ENTRY  table[ANY_SIZE];  // table of
                                          // interfaces
} MIB_IPMCAST_IF_TABLE, *PMIB_IPMCAST_IF_TABLE;
```

Members

dwNumEntries
Specifies the number of interface entries in the table.

table[ANY_SIZE]
Pointer to a table of interface entries implemented as an array of **MIB_IPMCAST_IF_TABLE** structures.

! Requirements

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

+ See Also

MIB_IPMCAST_IF_ENTRY

MIB_IPMCAST_MFE

The **MIB_IPMCAST_MFE** structure stores the information for an IP MFE.

```
typedef struct _MIB_IPMCAST_MFE {
  DWORD   dwGroup;
  DWORD   dwSource;
  DWORD   dwSrcMask;
  DWORD   dwUpStrmNgbr;
  DWORD   dwInIfIndex;
  DWORD   dwInIfProtocol;
  DWORD   dwRouteProtocol;
  DWORD   dwRouteNetwork;
  DWORD   dwRouteMask;
  ULONG   ulUpTime;
  ULONG   ulExpiryTime;
  ULONG   ulTimeOut;
  ULONG   ulNumOutIf;
  DWORD   fFlags;
  DWORD   dwReserved;
  MIB_IPMCAST_OIF rgmioOutInfo[ANY_SIZE];
} MIB_IPMCAST_MFE, *PMIB_IPMCAST_MFE;
```

Members

dwGroup
Specifies the range of multicast groups for this MFE. Zero indicates a wildcard group.

dwSource
Specifies the range of source addresses for this MFE. Zero indicates a wildcard source.

dwSrcMask
Specifies the subnet mask that corresponds to **dwSourceAddr**. The **dwSourceAddr** and **dwSourceMask** members are used together to define a range of sources.

dwUpStrmNgbr
Specifies the upstream neighbor that is related to this MFE.

dwInIfIndex
Specifies the index of the interface to which this MFE is related.

dwInIfProtocol
Specifies the routing protocol that owns the incoming interface to which this MFE is related.

dwRouteProtocol
Specifies the client that created the route.

dwRouteNetwork
Specifies the address associated with the route referred to by **dwRouteProtocol**.

dwRouteMask
Specifies the mask associated with the route referred to by **dwRouteProtocol**.

ulUpTime
Specifies how long, in seconds, this MFE has been valid. This value starts from zero and is incremented until it reaches the **ulTimeOut** value, at which time the MFE is deleted.

ulExpiryTime
Specifies the time, in seconds, that remains before the MFE expires and is deleted. This value starts from **ulTimeOut** and is decremented until it reaches zero, at which time the MFE is deleted.

ulTimeOut
Specifies the total length of time that this MFE should remain valid. After the time-out value is exceeded, the MFE is deleted. This value is static.

ulNumOutIf
Specifies the number of outgoing interfaces that are associated with this MFE.

fFlags
This member is reserved for future use and should be NULL.

dwReserved
This member is reserved and should be NULL.

rgmioOutInfo[ANY_SIZE]
Pointer to a table of outgoing interface statistics that are implemented as an array of **MIB_IPMCAST_OIF** structures.

Remarks

The **MIB_IPMCAST_MFE** structure does not have a fixed size. Use the **SIZEOF_MIB_MFE(X)** macro to determine the size of this structure. This macro is defined in the Iprtrmib.h header file.

The **dwRouteProtocol**, **dwRouteNetwork**, and **dwRouteMask** members uniquely identify the route to which this MFE is related.

Requirements

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

See Also

MIB_MFE_TABLE, **MIB_IPMCAST_OIF**

MIB_IPMCAST_MFE_STATS

The **MIB_IPMCAST_MFE_STATS** structure stores the statistics associated with an MFE.

```
typedef struct _MIB_IPMCAST_MFE_STATS {
  DWORD   dwGroup;
  DWORD   dwSource;
  DWORD   dwSrcMask;
  DWORD   dwUpStrmNgbr;
  DWORD   dwInIfIndex;
  DWORD   dwInIfProtocol;
  DWORD   dwRouteProtocol;
  DWORD   dwRouteNetwork;
  DWORD   dwRouteMask;
  ULONG   ulUpTime;
  ULONG   ulExpiryTime;
  ULONG   ulNumOutIf;
  ULONG   ulInPkts;
```

```
    ULONG    ulInOctets;
    ULONG    ulPktsDifferentIf;
    ULONG    ulQueueOverflow;
    MIB_IPMCAST_OIF_STATS    rgmiosOutStats[ANY_SIZE];
} MIB_IPMCAST_MFE_STATS, *PMIB_IPMCAST_MFE_STATS;
```

Members

dwGroup
Specifies the range of multicast groups for this MFE. Zero indicates a wildcard group.

dwSource
Specifies the range of source addresses for this MFE. Zero indicates a wildcard source.

dwSrcMask
Specifies the subnet mask that corresponds to **dwSourceAddr**. The **dwSourceAddr** and **dwSourceMask** members are used together to define a range of sources.

dwUpStrmNgbr
Specifies the upstream neighbor related to this MFE.

dwInIfIndex
Specifies the index of the interface to which this MFE is related.

dwInIfProtocol
Specifies the routing protocol that owns the incoming interface to which this MFE is related.

dwRouteProtocol
Specifies the client that created the route.

dwRouteNetwork
Specifies the address associated with the route referred to by **dwRouteProtocol**.

dwRouteMask
Specifies the mask associated with the route referred to by **dwRouteProtocol**.

ulUpTime
Specifies the time, in seconds, since the MFE was created.

ulExpiryTime
Specifies the time, in seconds, before the MFE expires and is deleted.

ulNumOutIf
Specifies the number of outgoing interfaces in the outgoing interface list for this MFE.

ulInPkts
Specifies the number of multicast packets received that matched this MFE.

ulInOctets
Specifies the number of octets of data received on the incoming interface.

ulPktsDifferentIf
Specifies the number packets that were received on interfaces other than the incoming interface.

ulQueueOverflow
Specifies the number of packets that were discarded because the buffer queue for this MFE overflowed.

rgmiosOutStats[ANY_SIZE]
Pointer to a table of outgoing interface statistics that are implemented as an array of **MIB_IPMCAST_OIF_STATS** structures.

Remarks

The **MIB_IPMCAST_MFE_STATS** structure does not have a fixed size. Use the macro **SIZEOF_MIB_MFE_STATS(X)** to determine the size of this structure. This macro is defined in the Iprtrmib.h header file.

The **dwRouteProtocol**, **dwRouteNetwork**, and **dwRouteMask** members uniquely identify the route to which this MFE is related.

! Requirements

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

+ See Also

MIB_IPMCAST_OIF_STATS

MIB_IPMCAST_OIF

The **MIB_IPMCAST_OIF** structure stores the information required to send an outgoing IP multcast packet.

```
typedef struct _MIB_IPMCAST_OIF {
  DWORD   dwOutIfIndex;      // index of outgoing interface
  DWORD   dwNextHopAddr;     // dest addr for packet
  PVOID   pvReserved;        // reserved
  DWORD   dwReserved;        // reserved
} MIB_IPMCAST_OIF, *PMIB_IPMCAST_OIF;
```

Members

dwOutIfIndex
Specifies the index of the interface on which to send the outgoing IP multicast packet.

dwNextHopAddr
Specifies the destination address for the outgoing IP multicast packet.

pvReserved
This member is reserved and should be NULL.

dwReserved

This member is reserved and should be zero.

Requirements

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

See Also

MIB_IPMCAST_MFE, **MIB_IPMCAST_OIF_STATS**

MIB_IPMCAST_OIF_STATS

The **MIB_IPMCAST_OIF_STATS** structure stores the statistics that are associated with an outgoing multicast interface.

```
typedef struct _MIB_IPMCAST_OIF_STATS {
  DWORD   dwOutIfIndex;
  DWORD   dwNextHopAddr;
  PVOID   pvDialContext;
  ULONG   ulTtlTooLow;
  ULONG   ulFragNeeded;
  ULONG   ulOutPackets;
  ULONG   ulOutDiscards;
} MIB_IPMCAST_OIF_STATS, *PMIB_IPMCAST_OIF_STATS;
```

Members

dwOutIfIndex

Specifies the outgoing interface to which these statistics are related.

dwNextHopAddr

Specifies the address of the next hop that corresponds to *dwOutIfIndex*. The *dwOutIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks. Examples of point-to-multipoint interfaces include Non-Broadcast Multiple-Access (NBMA) interfaces, and the internal interface on which all dial-up clients connect.

For Ethernet and other broadcast interfaces, specify zero. Also specify zero for point-to-point interfaces, which are identified by only *dwOutIfIndex*.

pvDialContext

This member is reserved for future use and should be NULL.

ulTtlTooLow

Specifies the number of packets on this outgoing interface that were discarded because the packet's time-to-live value was too low.

ulFragNeeded
Specifies the number of packets that required fragmentation when they were forwarded on this interface.

ulOutPackets
Specifies the number of packets that were forwarded out this interface.

ulOutDiscards
Specifies the number of packets that were discarded on this interface.

Requirements

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

See Also

MIB_IPMCAST_OIF

MIB_IPNETROW

The **MIB_IPNETROW** structure contains information for an Address Resolution Protocol (ARP) table entry.

```
typedef struct _MIB_IPNETROW {
  DWORD   dwIndex;        // adapter index
  DWORD   dwPhysAddrLen;  // physical address length
  BYTE    bPhysAddr[MAXLEN_PHYSADDR];  // physical address
  DWORD   dwAddr;         // IP address
  DWORD   dwType;         // ARP entry type
} MIB_IPNETROW, *PMIB_IPNETROW;
```

Members

dwIndex
Specifies the index of the adapter.

dwPhysAddrLen
Specifies the length of the physical address.

bPhysAddr[MAXLEN_PHYSADDR]
Specifies the physical address.

dwAddr
Specifies the IP address.

dwType
Specifies the type of ARP entry. This type can be one of the values on the following page.

Value	Meaning
4	Static
3	Dynamic
2	Invalid
1	Other

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

+ See Also

CreateIpNetEntry, **DeleteIpNetEntry**, **MIB_IPNETTABLE**, **SetIpNetEntry**

MIB_IPNETTABLE

The **MIB_IPNETTABLE** contains a table of Address Resolution Protocol (ARP) entries.

```
typedef struct _MIB_IPNETTABLE {
  DWORD          dwNumEntries;    // number of entries in
                                  // table
  MIB_IPNETROW   table[ANY_SIZE]; // array of ARP entries
} MIB_IPNETTABLE, *PMIB_IPNETTABLE;
```

Members

dwNumEntries
Specifies the number of ARP entries in the table.

table[ANY_SIZE]
Pointer to a table of ARP entries implemented as an array of **MIB_IPNETROW** structures.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

+ See Also

GetIpNetTable, **MIB_IPNETROW**

MIB_IPSTATS

The **MIB_IPSTATS** structure stores information about the IP protocol running on a particular computer.

```
typedef struct _MIB_IPSTATS {
  DWORD   dwForwarding;        // IP forwarding enabled or disabled
  DWORD   dwDefaultTTL;        // default time-to-live
  DWORD   dwInReceives;        // datagrams received
  DWORD   dwInHdrErrors;       // received header errors
  DWORD   dwInAddrErrors;      // received address errors
  DWORD   dwForwDatagrams;     // datagrams forwarded
  DWORD   dwInUnknownProtos;   // datagrams with unknown
                               // protocol
  DWORD   dwInDiscards;        // received datagrams
                               // discarded
  DWORD   dwInDelivers;        // received datagrams
                               // delivered
  DWORD   dwOutRequests;       //
  DWORD   dwRoutingDiscards;   //
  DWORD   dwOutDiscards;       // sent datagrams discarded
  DWORD   dwOutNoRoutes;       // datagrams for which no
                               // route
  DWORD   dwReasmTimeout;      // datagrams for which all
                               // frags didn't arrive
  DWORD   dwReasmReqds;        // datagrams requiring
                               // reassembly
  DWORD   dwReasmOks;          // successful reassemblies
  DWORD   dwReasmFails;        // failed reassemblies
  DWORD   dwFragOks;           // successful fragmentations
  DWORD   dwFragFails;         // failed fragmentations
  DWORD   dwFragCreates;       // datagrams fragmented
  DWORD   dwNumIf;             // number of interfaces on
                               // computer
  DWORD   dwNumAddr;           // number of IP address on
                               // computer
  DWORD   dwNumRoutes;         // number of routes in routing
                               // table
} MIB_IPSTATS, *PMIB_IPSTATS;
```

Members

dwForwarding

Specifies whether IP forwarding is enabled or disabled.

dwDefaultTTL

Specifies the default initial Time To Live (TTL) for datagrams originating on a particular computer.

dwInReceives
Specifies the number of datagrams received.

dwInHdrErrors
Specifies the number of datagrams received that have header errors.

dwInAddrErrors
Specifies the number of datagrams received that have address errors.

dwForwDatagrams
Specifies the number of datagrams forwarded.

dwInUnknownProtos
Specifies the number of datagrams received that have an unknown protocol.

dwInDiscards
Specifies the number of received datagrams discarded.

dwInDelivers
Specifies the number of received datagrams delivered.

dwOutRequests
Specifies the number of outgoing datagrams that IP is requested to transmit. This number does not include forwarded datagrams.

dwRoutingDiscards
Specifies the number of outgoing datagrams discarded.

dwOutDiscards
Specifies the number of transmitted datagrams discarded.

dwOutNoRoutes
Specifies the number of datagrams for which this computer did not have a route to the destination IP address. These datagrams were discarded.

dwReasmTimeout
Specifies the amount of time allowed for all pieces of a fragmented datagram to arrive. If all pieces do not arrive within this time, the datagram is discarded.

dwReasmReqds
Specifies the number of datagrams requiring reassembly.

dwReasmOks
Specifies the number of datagrams successfully reassembled.

dwReasmFails
Specifies the number of datagrams that cannot be reassembled.

dwFragOks
Specifies the number of datagrams that were fragmented successfully.

dwFragFails
Specifies the number of datagrams that cannot be fragmented because the "don't fragment" bit in the IP header is set. These datagrams are discarded.

dwFragCreates
Specifies the number of fragments created.

dwNumIf
Specifies the number of interfaces.

dwNumAddr
Specifies the number of IP addresses associated with this computer.

dwNumRoutes
Specifies the number of routes in the IP routing table.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

MIB_MFE_TABLE

The **MIB_MFE_TABLE** structure contains a table of Multicast Forwarding Entries (MFEs).

```
typedef struct _MIB_MFE_TABLE {
  DWORD           dwNumEntries;      // number of entries
  MIB_IPMCAST_MFE table[ANY_SIZE];   // table of MFEs
} MIB_MFE_TABLE, *PMIB_MFE_TABLE;
```

Members

dwNumEntries
Specifies the number of multicast forwarding entries in the table.

table[ANY_SIZE]
Pointer to a table of multicast forwarding entries implemented as an array of **MIB_IPMCAST_MFE** structures.

! Requirements

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

+ See Also

MIB_IPMCAST_MFE

MIB_MFE_STATS_TABLE

The **MIB_MFE_STATS_TABLE** structure stores statistics for a group of MFEs.

```
typedef struct _MIB_MFE_STATS_TABLE {
  DWORD   dwNumEntries;
  MIB_IPMCAST_MFE_STATS table[ANY_SIZE];
} MIB_MFE_STATS_TABLE, *PMIB_MFE_STATS_TABLE;
```

Members

dwNumEntries
Specifies the number of MFE entries in the array.

table[ANY_SIZE]
Pointer to a table of MFEs that are implemented as an array of **MIB_IPMCAST_MFE_STATS** structures.

Requirements

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

See Also

MIB_IPMCAST_MFE_STATS

MIB_OPAQUE_INFO

The **MIB_OPAQUE_INFO** structure contains information returned from a management information base opaque query.

```
typedef struct _MIB_OPAQUE_INFO {
  DWORD  dwId;

  union
  {
    ULONGLONG   ullAlign;
    BYTE        rgbyData[1];
  };

} MIB_OPAQUE_INFO, *PMIB_OPAQUE_INFO;
```

MIB_OPAQUE_QUERY

The **MIB_OPAQUE_QUERY** structure contains information for a management information base opaque query.

```
typedef struct _MIB_OPAQUE_QUERY {
  DWORD   dwVarId;
  DWORD   rgdwVarIndex[ANY_SIZE];
} MIB_OPAQUE_QUERY, *PMIB_OPAQUE_QUERY;
```

Members

dwVarId
Specifies the ID of the MIB object to query.

rgdwVarIndex[ANY_SIZE]
Specifies the index of the MIB object to query.

Requirements

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

See Also

MIB_OPAQUE_INFO

MIB_PROXYARP

The **MIB_PROXYARP** structure stores information for a Proxy Address Resolution Protocol (PARP) entry.

```
typedef struct _MIB_PROXYARP {
  DWORD   dwAddress;    // address for which to proxy
  DWORD   dwMask;       // subnet mask for the address
  DWORD   dwIfIndex;    // interface on which to proxy
} MIB_PROXYARP, *PMIB_PROXYARP;
```

Members

dwAddress
Specifies the IP address for which to act as a proxy.

dwMask
Specifies the subnet mask for the IP address specified by the **dwAddress** member.

dwIfIndex
Specifies the index of the interface on which to act as proxy for the address specified by the **dwAddress** member.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

See Also

CreateProxyArpEntry, **DeleteProxyArpEntry**

MIB_TCPROW

The **MIB_TCPROW** structure contains information for a TCP connection.

```
typedef struct _MIB_TCPROW {
  DWORD   dwState;        // state of the connection
  DWORD   dwLocalAddr;    // address on local computer
  DWORD   dwLocalPort;    // port number on local computer
  DWORD   dwRemoteAddr;   // address on remote computer
  DWORD   dwRemotePort;   // port number on remote computer
} MIB_TCPROW, *PMIB_TCPROW;
```

Members

dwState

Specifies the state of the TCP connection. This member can have one of the following values.

Value	Description
MIB_TCP_STATE_CLOSED	To be determined.
MIB_TCP_STATE_LISTEN	To be determined.
MIB_TCP_STATE_SYN_SENT	To be determined.
MIB_TCP_STATE_SYN_RCVD	To be determined.
MIB_TCP_STATE_ESTAB	To be determined.
MIB_TCP_STATE_FIN_WAIT1	To be determined.
MIB_TCP_STATE_FIN_WAIT2	To be determined.
MIB_TCP_STATE_CLOSE_WAIT	To be determined.
MIB_TCP_STATE_CLOSING	To be determined.
MIB_TCP_STATE_LAST_ACK	To be determined.
MIB_TCP_STATE_TIME_WAIT	To be determined.
MIB_TCP_STATE_DELETE_TCB	Transmission Control Block (TCB) deleted.

dwLocalAddr

Specifies the address for the connection on the local computer.

dwLocalPort

Specifies the port number for the connection on the local computer.

dwRemoteAddr

Specifies the address for the connection on the remote computer.

dwRemotePort

Specifies the port number the connection on the remote computer.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

See Also

MIB_TCPTABLE, **SetTcpEntry**

MIB_TCPSTATS

The **MIB_TCPSTATS** structure contains statistics for the TCP protocol running on the local computer.

```
typedef struct _MIB_TCPSTATS {
  DWORD   dwRtoAlgorithm;     // time-out algorithm
  DWORD   dwRtoMin;           // minimum time-out
  DWORD   dwRtoMax;           // maximum time-out
  DWORD   dwMaxConn;          // maximum connections
  DWORD   dwActiveOpens;      // active opens
  DWORD   dwPassiveOpens;     // passive opens
  DWORD   dwAttemptFails;     // failed attempts
  DWORD   dwEstabResets;      // established connections
                              // reset
  DWORD   dwCurrEstab;        // established connections
  DWORD   dwInSegs;           // segments received
  DWORD   dwOutSegs;          // segment sent
  DWORD   dwRetransSegs;      // segments retransmitted
  DWORD   dwInErrs;           // incoming errors
  DWORD   dwOutRsts;          // outgoing resets
  DWORD   dwNumConns;         // cumulative connections
} MIB_TCPSTATS, *PMIB_TCPSTATS;
```

Members

dwRtoAlgorithm

Specifies the retransmission time-out algorithm in use. This member can be one of the following values.

Value	Description
MIB_TCP_RTO_CONSTANT	Constant Time-out
MIB_TCP_RTO_RSRE	MIL-STD-1778 Appendix B
MIB_TCP_RTO_VANJ	Van Jacobson's Algorithm
MIB_TCP_RTO_OTHER	Other

dwRtoMin
Specifies the minimum retransmission time-out value in milliseconds.

dwRtoMax
Specifies the maximum retransmission time-out value in milliseconds.

dwMaxConn
Specifies the maximum number of connections. If this member is –1, the maximum number of connections is dynamic.

dwActiveOpens
Specifies the number of active opens. In an active open, the client is initiating a connection with the server.

dwPassiveOpens
Specifies the number of passive opens. In a passive open, the server is listening for a connection request from a client.

dwAttemptFails
Specifies the number of failed connection attempts.

dwEstabResets
Specifies the number of established connections that have been reset.

dwCurrEstab
Specifies the number of currently established connections.

dwInSegs
Specifies the number of segments received or transmitted.

dwOutSegs
Specifies the number of segments transmitted. This number does not include retransmitted segments.

dwRetransSegs
Specifies the number of segments retransmitted.

dwInErrs
Specifies the number of errors received.

dwOutRsts
Specifies the number of segments transmitted with the reset flag set.

dwNumConns
Specifies the cumulative number of connections.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

+ See Also

GetTcpStatistics

MIB_TCPTABLE

The **MIB_TCPTABLE** structure contains a table of TCP connections.

```
typedef struct _MIB_TCPTABLE {
  DWORD      dwNumEntries; // number of entries in the table
  MIB_TCPROW table[ANY_SIZE]; // array of TCP connections
} MIB_TCPTABLE, *PMIB_TCPTABLE;
```

Members

dwNumEntries
Specifies the number of entries in the table.

table[ANY_SIZE]
Pointer to a table of TCP connections implemented as an array of **MIB_TCPROW** structures.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

+ See Also

GetTcpTable, **MIB_TCPROW**

MIB_UDPROW

The **MIB_UDPROW** structure contains address information for sending and receiving User Datagram Protocol (UDP) datagrams.

```
typedef struct _MIB_UDPROW {
  DWORD   dwLocalAddr;    // IP address on local computer
  DWORD   dwLocalPort;    // port number on local computer
} MIB_UDPROW, * PMIB_UDPROW;
```

Members

dwLocalAddr
Specifies the IP address on the local computer.

dwLocalPort
Specifies the port number on the local computer.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

See Also

MIB_UDPSTATS, MIB_UDPTABLE

MIB_UDPSTATS

The **MIB_UDPSTATS** structure contains statistics for the User Datagram Protocol (UDP) running on the local computer.

```
typedef struct _MIB_UDPSTATS {
  DWORD   dwInDatagrams;    // received datagrams
  DWORD   dwNoPorts;        // datagrams for which no port
  DWORD   dwInErrors;       // errors on received datagrams
  DWORD   dwOutDatagrams;   // sent datagrams
  DWORD   dwNumAddrs;       // number of entries in
                            // UDP listener table
} MIB_UDPSTATS,*PMIB_UDPSTATS;
```

Members

dwInDatagrams
Specifies the number of datagrams received.

dwNoPorts
Specifies the number of received datagrams that were discarded because the port specified was invalid.

dwInErrors
Specifies the number of erroneous datagrams that were received. This number does not include the value contained by the **dwNoPorts** member.

dwOutDatagrams
Specifies the number of datagrams transmitted.

dwNumAddrs
Specifies the number of entries in the UDP listener table.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

See Also

GetUdpStatistics, **MIB_UDPROW**

MIB_UDPTABLE

The **MIB_UDPTABLE** structure contains a table of **MIB_UDPROW** structures.

```
typedef struct _MIB_UDPTABLE {
  DWORD       dwNumEntries;    // number of entries in table
  MIB_UDPROW  table[ANY_SIZE]; // table of MIB_UDPROW structs
} MIB_UDPTABLE, * PMIB_UDPTABLE;
```

Members

dwNumEntries
Specifies the number of entries in the table.

table[ANY_SIZE]
Pointer to an array of **MIB_UDPROW** structures.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

See Also

GetUdpTable, **MIB_UDPROW**

MIBICMPINFO

The **MIBICMPINFO** structure contains Internet Control Message Protocol (ICMP) statistics for a particular computer.

```
typedef  struct _MIBICMPINFO {
  MIBICMPSTATS   icmpInStats; // stats for incoming messages
  MIBICMPSTATS   icmpOutStats;// stats for outgoing messages
} MIBICMPINFO;
```

Members

icmpInStats
Specifies an **MIBICMPSTATS** structure that contains the statistics for incoming ICMP messages.

icmpOutStats

Specifies an **MIBICMPSTATS** structure that contains the statistics for outgoing ICMP messages.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

See Also

MIB_ICMP, **MIBICMPSTATS**

MIBICMPSTATS

The **MIBICMPSTATS** structure contains statistics for either incoming or outgoing Internet Control Message Protocol (ICMP) messages on a particular computer.

```
typedef struct _MIBICMPSTATS {
  DWORD   dwMsgs;           // number of messages
  DWORD   dwErrors;         // number of errors
  DWORD   dwDestUnreachs;   // destination unreachable
                            // messages
  DWORD   dwTimeExcds;      // time-to-live exceeded
                            // messages
  DWORD   dwParmProbs;      // parameter problem messages
  DWORD   dwSrcQuenchs;     // source quench messages
  DWORD   dwRedirects;      // redirection messages
  DWORD   dwEchos;          // echo requests
  DWORD   dwEchoReps;       // echo replies
  DWORD   dwTimestamps;     // time-stamp requests
  DWORD   dwTimestampReps;  // time-stamp replies
  DWORD   dwAddrMasks;      // address mask requests
  DWORD   dwAddrMaskReps;   // address mask replies
} MIBICMPSTATS;
```

Members

dwMsgs

Specifies the number of messages received or sent.

dwErrors

Specifies the number of errors received or sent.

dwDestUnreachs
Specifies the number of destination-unreachable messages received or sent. A destination-unreachable message is sent to the originating computer when a datagram fails to reach its intended destination.

dwTimeExcds
Specifies the number of Time-To-Live (TTL) exceeded messages received or sent. A time-to-live exceeded message is sent to the originating computer when a datagram is discarded because the number of routers it has passed through exceeds its time-to-live value.

dwParmProbs
Specifies the number of parameter problem messages received or sent. A parameter problem message is sent to the originating computer when a router or host detects an error in a datagram's IP header.

dwSrcQuenchs
Specifies the number of source quench messages received or sent. A source quench request is sent to a computer to request that it reduce its rate of packet transmission.

dwRedirects
Specifies the number of redirect messages received or sent. A redirect message is sent to the originating computer when a better route is discovered for a datagram sent by that computer.

dwEchos
Specifies the number of echo requests received or sent. An echo request causes the receiving computer to send an echo reply message back to the originating computer.

dwEchoReps
Specifies the number of echo replies received or sent. A computer sends an echo reply in response to receiving an echo request message.

dwTimestamps
Specifies the number of time-stamp requests received or sent. A time-stamp request causes the receiving computer to send a time-stamp reply back to the originating computer.

dwTimestampReps
Specifies the number of time-stamp replies received or sent. A computer sends a time-stamp reply in response to receiving a time-stamp request. Routers can use time-stamp requests and replies to measure the transmission speed of datagrams on a network.

dwAddrMasks
Specifies the number of address mask requests received or sent. A computer sends an address mask request to determine the number of bits in the subnet mask for its local subnet.

dwAddrMaskReps
Specifies the number of address mask responses received or sent. A computer sends an address mask response in response to an address mask request.

Remarks

Two **MIBICMPSTATS** structures are required to hold all the ICMP statistics for a given computer. One **MIBICMPSTATS** structure contains the statistics for incoming ICMP messages. The other contains the statistics for outgoing ICMP messages. For this reason, the **MIBICMPINFO** structure contains two **MIBICMPSTATS** structures.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in Iprtrmib.h.

+ See Also

MIB_ICMP, **MIBICMPINFO**

Transport and Protocol Constants

Use the following constants with router administration and configuration functions, and with the MIB API:

- Transport Identifiers
- Protocol Identifiers

Transport Identifiers

The following transport identifiers are also listed in Mprapi.h:

PID_IPX
PID_IP
PID_NBF

+ See Also

MIB Reference, Transport and Protocol Constants

Protocol Identifiers

The following protocol identifiers are also listed in Routprot.h.

IP Protocols

The routing protocols on the following page are associated with the IP transport.

Protocol	Description
PROTO_IP_OTHER	Protocol not listed here
PROTO_IP_LOCAL	Routes generated by the stack
PROTO_IP_NETMGMT	Routes added by "route add" or through SNMP
PROTO_IP_ICMP	Routes from ICMP redirects
PROTO_IP_EGP	Exterior Gateway Protocol
PROTO_IP_GGP	To be determined.
PROTO_IP_HELLO	HELLO routing protocol
PROTO_IP_RIP	Routing Informaton Protocol
PROTO_IP_IS_IS	To be determined.
PROTO_IP_ES_IS	To be determined.
PROTO_IP_CISCO	To be determined.
PROTO_IP_BBN	To be determined.
PROTO_IP_OSPF	Open Shortest Path First routing protocol
PROTO_IP_BGP	Border Gateway Protocol
PROTO_IP_BOOTP	Bootstrap Protocol
PROTO_IP_NT_AUTOSTATIC	Routes that were originally generated by a routing protocol, but which are now static
PROTO_IP_NT_STATIC	Routes that were added from the routing user interface, or by "routemon ip add"
PROTO_IP_NT_STATIC_NON_DOD	Identical to PROTO_IP_NET_STATIC, except these routes do not cause Dial On Demand (DOD)

Routes with a protocol identifier of PROTO_IP_LOCAL include:

- The loopback route
- The subnet route
- All nets broadcast route for subnetted interfaces
- All "1"s broadcast route
- Local multicast route
- Route to remote end of a PPP link

The identifier for the IP router manager is:

IPRTRMGR_PID

This identifier can be used instead of a routing protocol identifier for MIB calls with the IP router manager. This identifier is used for MIB-II, Forwarding MIB, and some enterprise specific information. This identifier is also listed in Iprtrmib.h.

IPX Protocols

The following routing protocols are associated with the IPX transport:

Protocol	Description
IPX_PROTOCOL_RIP	Routing Information Protocol for IPX
IPX_PROTOCOL_SAP	Service Advertisement Protocol
IPX_PROTOCOL_NLSP	Netware Link Services Protocol

The identifier for the IPX router manager is:

IPX_PROTOCOL_BASE

Use this identifier instead of a routing protocol identifier for MIB calls with the IPX router manager.

CHAPTER 8

Packet Filtering

Packet filtering enables the developer to create and manage input and output filters for IP packets. Each IP adapter interface can be associated with one or more filters. Filters can include source and destination addresses, address mask and port; and protocol identifiers.

With the exception of the **PfGetInterfaceStatistics** function, all of the functions described in this section require administrative permissions.

The following reference elements are found in the Fltdefs.h header file:

- Packet Filtering Functions
- Packet Filtering Structures
- Packet Filtering Enumerated Types

Packet Filtering Functions

Use the following functions to manage IP packet filters:

PfAddFiltersToInterface
PfAddGlobalFilterToInterface
PfBindInterfaceToIndex
PfBindInterfaceToIPAddress
PfCreateInterface
PfDeleteInterface
PfDeleteLog
PfGetInterfaceStatistics
PfMakeLog
PfRebindFilters
PfRemoveFilterHandles
PfRemoveFiltersFromInterface
PfRemoveGlobalFilterFromInterface
PfSetLogBuffer
PfTestPacket
PfUnBindInterface

PfAddFiltersToInterface

The **PfAddFiltersToInterface** function adds the specified filters to the specified interface.

```
PfAddFiltersToInterface(
  INTERFACE_HANDLE ih,
  DWORD cInFilters,
  PPF_FILTER_DESCRIPTOR pfiltIn,
  DWORD cOutFilters,
  PPF_FILTER_DESCRIPTOR pfiltOut,
  PFILTER_HANDLE pfHandle
);
```

Parameters

ih

Specifies a handle to the interface.

cInFilters

Specifies the number of input filter descriptions pointed to by the *pfiltIn* parameter.

pfiltIn

Pointer to an array of filter descriptions to use as input filters.

cOutFilters

Specifies the number of output filters descriptions pointed to by the *pfiltOut* parameter.

pfiltOut

Pointer to an array of filter descriptions to use as output filters.

pfHandle

Pointer to a buffer that, on successful return, contains an array of filter handles. If the caller doesn't not require the filter handles, the caller may set this parameter to NULL.

Remarks

A filter reverses the default processing rule for the interface, that is, the rule that was specified during the call to **PfCreateInterface**.

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
PFERROR_NO_FILTERS_GIVEN	No filter descriptions were supplied.
Other	Use **FormatMessage** to obtain the message string for the returned error.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

+ See Also

PfRemoveFiltersFromInterface, **PfRemoveFilterHandles**

PfAddGlobalFilterToInterface

The **PfAddGlobalFilterToInterface** function adds a global filter on the specified interface.

```
DWORD PfAddGlobalFilterToInterface(
  INTERFACE_HANDLE pInterface,
  GLOBAL_FILTER gfFilter
);
```

Parameters

pInterface
Handle to the interface.

gfFilter
Specifies the global filter to add to the interface.

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 global filter acts across all filters on the interface.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

+ See Also

GLOBAL_FILTER, **PfRemoveGlobalFilterFromInterface**

PfBindInterfaceToIndex

The **PfBindInterfaceToIndex** function associates an interface with the specified IP stack index.

```
DWORD PfBindInterfaceToIndex(
  INTERFACE_HANDLE pInterface,
  DWORD dwIndex,
  PFADDRESSTYPE pfatLinkType,
  PBYTE LinkIPAddress
);
```

Parameters

pInterface
Specifies a handle to the interface to associate with the IP stack index.

dwIndex
Specifies the IP stack index to which to associate the interface.

pfatLinkType
Specifies the address type for the interface. This parameter would be of type **PFADDRESSTYPE**.

LinkIPAddress
Pointer to an array of bytes that specifies the IP address for the interface.

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_IPV6_NOT_IMPLEMENTED	The IPV6 address type is not yet implemented.
Other	Use **FormatMessage** to obtain the message string for the returned error.

Remarks

Use the IP Helper functions to obtain a stack index.

An application should support the possibility of interface indices changing due to Plug and Play.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

See Also

PFADDRESSTYPE, **PfBindInterfaceToIPAddress**, **PfUnbindInterface**

PfBindInterfaceToIPAddress

The **PfBindInterfaceToIPAddress** function associates an interface with the IP stack index having the specified address.

```
DWORD PfBindInterfaceToIPAddress(
  INTERFACE_HANDLE pInterface,
  PFADDRESSTYPE pfatType,
  PBYTE IPAddress
);
```

Parameters

pInterface
  Specifies a handle to the interface to associate with the IP stack index.

pfatType
  Specifies the address type for the interface. This parameter would be of type **PFADDRESSTYPE**.

IPAddress
  Pointer to an array of bytes that specifies the IP address for the interface.

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_IPV6_NOT_IMPLEMENTED	The IPV6 address type is not yet implemented.
Other	Use **FormatMessage** to obtain the message string for the returned error.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

See Also

PFADDRESSTYPE, **PfBindInterfaceToIndex**

PfCreateInterface

The **PfCreateInterface** function creates a new filter interface. Use this interface to control the adding and deleting of filters to and from adapters.

```
PfCreateInterface(
  DWORD dwName,
```

(continued)

(continued)

```
  PFFORWARD_ACTION inAction,
  PFFORWARD_ACTION outAction,
  BOOL bUseLog,
  BOOL bMustBeUnique,
  INTERFACE_HANDLE *ppInterface
);
```

Parameters

dwName

Specifies the interface name. A zero value specifies a new, unique interface. Any other value is a potentially shared interface.

The *bMustBeUnique* parameter can turn a shared interface into a unique one. However, using *bMustBeUnique* in this way can cause the function to fail.

inAction

Default action for an input packet. This member can be one of the following values.

Value	Meaning
PF_ACTION_FORWARD	Forward the packet.
PF_ACTION_DROP	Discard the packet.

outAction

Default action for an output packet. This member can be one of the following values.

Value	Meaning
PF_ACTION_FORWARD	Forward the packet.
PF_ACTION_DROP	Discard the packet.

bUseLog

Specifies whether to bind the log to this interface. If this member is TRUE, the log will be bound to this interface.

bMustBeUnique

Specifies whether the interface is unique or shared. If this member is TRUE, this interface is unique, that is, it cannot be shared.

ppInterface

Pointer to a pointer that, on successful return, points to an interface handle to use with subsequent function calls.

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

An interface can either be unique to a process or shared. If an interface is shared, other processes may add or remove filters.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

See Also

PfAddFiltersToInterface, **PfRemoveFiltersFromInterface**, **PfDeleteInterface**

PfDeleteInterface

The **PfDeleteInteface** function deletes an interface previously created using **PfCreateInterface**.

```
PfDeleteInterface(
  INTERFACE_HANDLE pInterface
);
```

Parameters

pInterface

Specifies a interface handle obtained from a previous call to **PfCreateInterface**.

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.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

See Also

PfCreateInterface

PfDeleteLog

The **PfDeleteLog** function immediately disables the log on all interfaces with which it is associated. The log is deleted when all interfaces associated with the log are deleted.

```
DWORD PfDeleteLog(VOID);
```

Parameters

This function has no parameters.

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.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

+ See Also

PfMakeLog, **PfSetLogBuffer**

PfGetInterfaceStatistics

The **PfGetInterfaceStatistics** function retrieves statistics for the specified interface and, optionally, statistics for filters associated with the interface.

```
DWORD PfGetInterfaceStatistics(
  INTERFACE_HANDLE pInterface,
  PPF_INTERFACE_STATS ppfStats,
  PDWORD pdwBufferSize,
  BOOL fResetCounters
);
```

Parameters

pInterface
  Handle to the interface.

ppfStats

Pointer to a buffer that, on successful return contains the statistics for the interface.

If the caller requires only the statistics for the interface, this buffer should be of size equal to a **PF_INTERFACE_STATS** structure. If the caller supplies a buffer that is smaller than this size, PfGetInterfaceStatistics returns PFERROR_BUFFER_TOO_SMALL, and the pdwBufferSize parameter contains a size equal to a **PF_INTERFACE_STATS** structure.

If the caller requires the statistics for both the interface and the associated filters, the buffer should of a size greater than **PF_INTERFACE_STATS**. If the buffer is still not large enough **PfGetInterfaceStatistics** returns ERROR_INSUFFICIENT_BUFFER, and the *pdwBufferSize* parameter points to **DWORD** variable containing a buffer size that will contain both the interface and filter statistics.

pdwBufferSize

Pointer to a **DWORD** variable that contains the size of the buffer pointed to by the *ppfStats* parameter.

fResetCounters

Specifies whether the statistics counters for the interface should be reset. If this parameter is TRUE, the statistics counters are reset.

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_INSUFFICIENT_BUFFER	This error is specific to **PfGetInterfaceStatistics** and means that the supplied user buffer is too small for the filters. The correct size is returned as the interface statistics that contain the filter counts.
PFERROR_BUFFER_TOO_SMALL	This error is specific to **PfGetInterfaceStatistics** and means the user buffer is too small ever for the interface statistics. The returned size is the size of the interface statistics, but does not include space for filters. If the function is called using this size, the return value should be ERROR_INSUFFICIENT_BUFFER.
Other	Use **FormatMessage** to obtain the message string for the returned error.

Remarks

The caller may call **PfGetInterfaceStatistics** twice. Initially the call is made to obtain the correct buffer size; the call is made a second time to retrieve the statistics. If the caller calls **PfGetInterfaceStatistics** twice for a shared interface, the second call may fail with ERROR_INSUFFICIENT_BUFFER. This error can occur because the other sharers may add filters to the interface in the interval between the two calls. This type of error should not occur for a UNIQUE interface.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

See Also

PF_INTERFACE_STATS

PfMakeLog

The **PfMakeLog** function creates a log to use with an interface or set of interfaces.

```
WORD PfMakeLog(
  HANDLE hEvent
);
```

Parameters

hEvent

Handle to a Win32 event object. The caller can use this event object to obtain notification when a specified number of bytes have been used in the log's buffer, or when a certain number of entries have been created in the log. For more information, see **PfSetLogBuffer**.

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

Only one log exists. The log can be used with multiple interfaces.

The interface log must be created prior to the interface or interfaces with which it will be used. It is not possible to associate a log with an already existing interface.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

See Also

PfDeleteLog, **PfSetLogBuffer**

PfRebindFilters

The **PfRebindFilters** function rebinds the filters on the specified interface.

```
DWORD PfRebindFilters(
  INTERFACE_HANDLE pInterface,
  PPF_LATEBIND_INFO pLateBindInfo
);
```

Parameters

pInterface
Handle to the interface.

pLateBindInfo
Pointer to the late-binding information for the interface.

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 **fLateBound** member of **PF_FILTER_DESCRIPTOR** for each filter determines how the information pointed to by the *pLateBindInfo* parameter affects the filter.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

See Also

PF_FILTER_DESCRIPTOR

PfRemoveFilterHandles

The **PfRemoveFilterHandles** function removes the filter associated with the specified handles.

```
DWORD PfRemoveFilterHandles(
  INTERFACE_HANDLE pInterface,
  DWORD cFilters,
  PFILTER_HANDLE pvHandles
);
```

Parameters

pInterface
Specifies a handle to the interface.

cFilters
Specifies the number of filter handles pointed to by the *pvHandles* parameter. Obtain these handles from the **PfAddFiltersToInterface** function.

pvHandles
Pointer to an array of filter handles that specify the filters to remove.

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.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

See Also

PfAddFiltersToInterface, **PfRemoveFiltersFromInterface**

PfRemoveFiltersFromInterface

The **PfRemoveFiltersFromInterface** function removes the specified filters from the interface.

```
DWORD PfRemoveFiltersFromInterface(
  INTERFACE_HANDLE ih,
  DWORD cInFilters,
    PPF_FILTER_DESCRIPTOR pfiltIn,
    DWORD cOutFilters,
    PPF_FILTER_DESCRIPTOR pfiltOut
);
```

Parameters

ih

Specifies a handle to the interface.

cInFilters

Specifies the number of input filter descriptions pointed to by the *pfiltIn* parameter.

pfiltIn

Pointer to an array of filter descriptions to use as input filters.

cOutFilters

Specifies the number of output filters descriptions pointed to by the *pfiltOut* parameter.

pfiltOut

Pointer to an array of filter descriptions to use as output filters.

Return Values

If the function succeeds, the return value is NO_ERROR.

Value	Meaning
PFERROR_NO_FILTERS_GIVEN	No filter descriptions were supplied
Other	Use **FormatMessage** to obtain the message string for the returned error.

Remarks

The filter description passed in through the *pfiltIn* and *pfiltOut* parameters must be an exact match to a filter that was added previously.

No error is returned if a matching filter is not found.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

See Also

PfAddFiltersToInterface

PfRemoveGlobalFilterFromInterface

The **PfRemoveGlobalFilterFromInterface** function removes the specified global filter from the interface.

```
DWORD PfRemoveGlobalFilterFromInterface(
  INTERFACE_HANDLE pInterface,
  GLOBAL_FILTER gfFilter
);
```

Parameters

pInterface
Handle to the interface.

gfFilter
Specifies the global filter to remove from the interface.

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.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

See Also

GLOBAL_FILTER, **PfAddGlobalFilterToInterface**

PfSetLogBuffer

The **PfSetLogBuffer** function exchanges the log current buffer for a new one.

```
DWORD PfSetLogBuffer(
  PBYTE pbBuffer,
  DWORD dwSize,
  DWORD dwThreshold,
  DWORD dwEntries,
  PDWORD pdwLoggedEntries,
  PDWORD pdwLostEntries,
  PDWORD pdwSizeUsed
);
```

Parameters

pbBuffer
Pointer to the new log buffer. This buffer must be quad-word aligned.

dwSize
Specifies the size, in bytes, of the new buffer.

dwThreshold
Specifies the number of bytes used before signaling the event object associated with the log.

dwEntries
Specifies the number of entries in the log that will cause the event object to be signaled.

pdwLoggedEntries
Pointer to a **DWORD** variable that, on successful return, contains the number of entries in the old buffer.

pdwLostEntries
Pointer to a **DWORD** variable that, on successful return, contains the number of entries that could not be put into the old buffer.

pdwSizeUsed
Pointer to a **DWORD** variable that, on successful return, contains the number of bytes used in the old buffer.

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.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

+ See Also

PfMakeLog, **PfDeleteLog**

PfTestPacket

The **PfTestPacket** function tests the specified packet and returns the action that would be performed given the specified interface.

```
DWORD PfTestPacket(
  INTERFACE_HANDLE pInInterface  OPTIONAL,
  INTERFACE_HANDLE pOutInterface OPTIONAL,
  DWORD cBytes,
  PBYTE pbPacket,
  PPFFORWARD_ACTION ppAction
);
```

Parameters

pInInterface
Handle to an interface to use as an input interface for the packet. This parameter is optional and may be NULL.

pOutInterface
Handle to an interface to use an output interface for the packet. This parameter is optional and may be NULL.

cBytes
pbPacket
Pointer to a network packet to test with the specified interface (or interfaces).

ppAction
Pointer to a variable of type **PFFORWARD_ACTION**. On successful return, this variable contains the action that would have been taken given one or more specified interfaces and the packet.

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

Specifying only an input interface simulates a packet destined for the local computer. Specifying only an output interface simulates sending a packet from the local computer. Specifying both an input and an output interface simulates routing a network packet.

If the caller does not specify any interfaces, the **PfTestPacket** returns PF_ACTION_FORWARD in the *ppAction* parameter.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

+ See Also

PFFORWARD_ACTION

PfUnBindInterface

The **PfUnBindInterface** function unbinds the interface from the stack.

```
DWORD PfUnBindInterface(
  INTERFACE_HANDLE pInterface
);
```

Parameters

pInterface
Specifies the interface to unbind from the stack.

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

Unbinding the interface does not destroy the interface or any of its filters.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Fltdefs.h.
Library: Use Iphlpapi.lib.

See Also

PfBindInterfaceToIndex

Packet Filtering Structures

Use the following structures when managing IP packet filters:

PF_FILTER_DESCRIPTOR
PF_FILTER_STATS
PF_INTERFACE_STATS
PF_LATEBIND_INFO
PFLOGFRAME

PF_FILTER_DESCRIPTOR

The **PF_FILTER_DESCRIPTOR** structure contains the information that defines a packet filter.

```
typedef struct _PF_FILTER_DESCRIPTOR {
  DWORD         dwFilterFlags;    // see below
  DWORD         dwRule;           // copied into the log
                                  // when appropriate
  PFADDRESSTYPE pfatType;
  PBYTE         SrcAddr;
  PBYTE         SrcMask;
  PBYTE         DstAddr;
  PBYTE         DstMask;
  DWORD         dwProtocol;
  DWORD         fLateBound;
  WORD          wSrcPort;
  WORD          wDstPort;
  WORD          wSrcPortHighRange;
  WORD          wDstPortHighRange;
} PF_FILTER_DESCRIPTOR, *PPF_FILTER_DESCRIPTOR;
```

Members

dwFilterFlags
Currently only a single flag is supported for this member:

FD_FLAGS_NOSYN

dwRule
Specifies the rule for the filter.

pfatType
The address type for the filter. This member is of type **PFADDRESSTYPE**.

SrcAddr
The source address of the packets to filter.

SrcMask
The subnet mask for the source address.

DstAddr
The destination address of the packets to filter.

DstMask
The subnet mask for the destination address.

dwProtocol
Specifies the protocols to filter. This member can be one of the following values.

Value	Meaning
FILTER_PROTO_ANY	All protocols
FILTER_PROTO_ICMP	Internet Control Message Protocol
FILTER_PROTO_TCP	Transmission Control Protocol
FILTER_PROTO_UDP	User Datagram Protocol

fLateBound
Specifies the address information that should be updated when the filter is rebound. This member can be any combination of the following flags:

LB_SRC_ADDR_USE_SRCADDR_FLAG
LB_SRC_ADDR_USE_DSTADDR_FLAG
LB_DST_ADDR_USE_SRCADDR_FLAG
LB_DST_ADDR_USE_DSTADDR_FLAG

wSrcPort
Specifies the source port of the packets to filter.

wDstPort
Specifies the destination port of the packets to filter.

wSrcPortHighRange
Specifies the high range of the source port of packets to filter.

wDstPortHighRange
Specifies the high range of the destination port of packets to filter.

! Requirements

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

+ See Also

PF_LATEBIND_INFO, **PfAddFiltersToInterface**, **PfRebindFilters**

PF_FILTER_STATS

The **PF_FILTER_STATS** structure contains a description of a particular filter and the number of packets filtered by the filter.

```
typedef struct _PF_FILTER_STATS {
  DWORD       dwNumPacketsFiltered;
  PF_FILTER_DESCRIPTOR info;
} PF_FILTER_STATS, *PPF_FILTER_STATS;
```

Members

dwNumPacketsFiltered
Specifies the number of packets filtered by the filter specified by the info member.

info
A **PF_FILTER_DESCRIPTOR** that describes a particular filter.

Remarks

The **PF_INTERFACE_STATS** structure contains an array of **PF_FILTER_STATS** structures. Each element of the **PF_FILTER_STATS** array corresponds to a filter associated with the **PF_INTERFACE_STATS** interface.

Requirements

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

See Also

PF_FILTER_DESCRIPTOR, **PF_INTERFACE_STATS**, **PfGetInterfaceStatistics**

PF_INTERFACE_STATS

The **PF_INTERFACE_STATS** structure contains statistics for an interface.

```
typedef struct _PF_INTERFACE_STATS {
  PVOID             pvDriverContext;
  DWORD             dwFlags;
  DWORD             dwInDrops;
  DWORD             dwOutDrops;
  PFFORWARD_ACTION  eaInAction;
  PFFORWARD_ACTION  eaOutAction;
  DWORD             dwNumInFilters;
  DWORD             dwNumOutFilters;
  DWORD             dwFrag;
  DWORD             dwSpoof;
  DWORD             dwReserved1;
  DWORD             dwReserved2;
  LARGE_INTEGER     liSYN;
  LARGE_INTEGER     liTotalLogged;
  DWORD             dwLostLogEntries;
  PF_FILTER_STATS   FilterInfo[1];
} PF_INTERFACE_STATS, *PPF_INTERFACE_STATS;
```

Members

pvDriverContext
This member is not currently used.

dwFlags
No flags are currently defined for this member.

dwInDrops
Specifies the number of incoming packets that were dropped.

dwOutDrops
Specifies the number of outgoing packets that were dropped.

eaInAction
Specifies the default incoming action.

eaOutAction
Specifies the default outgoing action.

dwNumInFilters
Specifies the number of filters for incoming packets.

dwNumOutFilters
Specifies the number of filters for outgoing packets.

dwFrag
Specifies the state of global fragment checking. See ***GLOBAL_FILTER*** for more information.

dwSpoof
Specifies the state of global checking of destination addresses. See ***GLOBAL_FILTER*** for more information.

dwReserved1
This member is reserved and should be zero.

dwReserved2
This member is reserved and should be zero.

liSYN
Specifies the number of SYN packets discarded.

liTotalLogged
Specifies the number of packets logged.

dwLostLogEntries
Specifies the number of logged packets lost because of buffering problems.

FilterInfo[1]
Specifies an array of **PF_FILTER_STATS** structures. The array contains an element for each filter associated with the interface. Each element contains a description of the filter and the number of packets filtered by that filter.

! Requirements

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

See Also

GLOBAL_FILTER, **PfGetInterfaceStatistics**, **PF_FILTER_STATS**, **PFFORWARD_ACTION**

PF_LATEBIND_INFO

The **PF_LATEBIND_INFO** structure contains address information for late-binding interface.

```
typedef struct _PF_LATEBIND_INFO {
  PBYTE  SrcAddr;
  PBYTE  DstAddr;
  PBYTE  Mask;
} PF_LATEBIND_INFO, *PPF_LATEBIND_INFO;
```

Members

SrcAddr
Specifies a new source address.

DstAddr
Specifies a new destination address.

Mask
Subnet mask.

Remarks

Late-binding information is typically used with Wide Area Network (WAN) interfaces. The address information for such interfaces usually changes at the time they establish a connection.

Requirements

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

See Also

PF_FILTER_DESCRIPTOR, **PfRebindFilters**

PFLOGFRAME

The **PFLOGFRAME** structure stores the information for a log entry.

```
typedef struct _pfLogFrame {
  LARGE_INTEGER Timestamp;
  PFFRAMETYPE   pfeTypeOfFrame;
  DWORD         dwTotalSizeUsed;
  DWORD         dwFilterRule;
  WORD          wSizeOfAdditionalData;
  WORD          wSizeOfIpHeader;
  DWORD         dwInterfaceName;
  DWORD         dwIPIndex;
  BYTE          bPacketData[1];wsizeOfIpHeader
} PFLOGFRAME, *PPFLOGFRAME;
```

Members

Timestamp

pfeTypeOfFrame

Variable of type **PFFRAMETYPE** that specifies the reason the packet was filtered.

dwTotalSizeUsed

The total size, in bytes, of this entry. Use this value to find the next log entry in a sequence of entries.

dwFilterRule

Specifies the rule for the filter.

wSizeOfAdditionalData

Specifies additional data for the rule.

wSizeOfIpHeader

Size of the IP header for the packet.

dwInterfaceName

The name of the interface.

dwIPIndex

The index of the interface on which the packet was sent or received.

bPacketData[1]

! Requirements

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

+ See Also

PFFRAMETYPE, **PfMakeLog**, **PfSetLogBuffer**, **PfDeleteLog**

Packet Filtering Enumerated Types

Use the following enumerated types when managing IP packet filters:

GLOBAL_FILTER
PFADDRESSTYPE
PFFORWARD_ACTION
PFFRAMETYPE

GLOBAL_FILTER

The **GLOBAL_FILTER** type enumerates the kinds of global filters that can be applied to an interface.

```
typedef enum _GlobalFilter {
  GF_FRAGMENTS = 2,
  GF_STRONGHOST = 8,
  GF_FRAGCACHE = 9
} GLOBAL_FILTER, *PGLOBAL_FILTER;
```

Values

GF_FRAGMENTS
Causes a consistency check of packet fragments.

GF_STRONGHOST
Causes a check of the destination address of incoming packets.

GF_FRAGCACHE
Causes a check of the fragments from the cache.

Requirements

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

See Also

PF_INTERFACE_STATS, **PfAddGlobalFilterToInterface**, **PfRemoveGlobalFilterFromInterface**

PFADDRESSTYPE

The **PFADDRESSTYPE** type enumerates the address formats supported by filter interface.

```
typedef enum _PfAddresType {
  PF_IPV4,
  PF_IPV6
} PFADDRESSTYPE, *PPFADDRESSTYPE;
```

Values

PF_IPV4
The addresses format used with Internet Protocol v4.

PF_IPV6
The address format used with Internet Protocol v6.

! Requirements

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

+ See Also

PfBindInterfaceToIndex, **PfBindInterfaceToIPAddress**

PFFORWARD_ACTION

The **PFFORWARD_ACTION** type enumerates the possible ways in which a filter interface can process a network packet.

```
typedef enum _PfForwardAction {
  PF_ACTION_FORWARD = 0,
  PF_ACTION_DROP
} PFFORWARD_ACTION, *PPFFORWARD_ACTION;
```

Values

PF_ACTION_FORWARD
The interface forwards the network packet.

PF_ACTION_DROP
The interface discards the network packet.

! Requirements

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

+ See Also

PF_INTERFACE_STATS, **PfTestPacket**

PFFRAMETYPE

The **PFFRAMETYPE** type enumerates the reasons why a packet was filtered.

```
typedef enum _PfFrameType {
  PFFT_FILTER  = 1,
  PFFT_FRAG    = 2,
  PFFT_SPOOF   = 3
} PFFRAMETYPE, *PPFFRAMETYPE;
```

Values

PFFT_FILTER
The packet violated a filter rule.

PFFT_FRAG
A bad fragment was detected.

PFFT_SPOOF
A check of the destination address resulted in a "strong host" failure.

Requirements

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

See Also

PFLOGFRAME, **GLOBAL_FILTER**

CHAPTER 9

Routing Protocol Interface

Routing Protocol Interface Overview

The following sections describe the integration of third-party routing protocols into the Routing and Remote Access Service (RRAS). RRAS is a feature of Microsoft® Windows® 2000 that acts as a multiprotocol router. RRAS defines the interface between the router manager and the Dynamic-Link Library (DLL) for routing protocols.

Use this interface to implement routing protocols, for example, IGRP, NLSP, and BGP, as user-mode DLLs that work with RRAS.

Adapters

An adapter represents the physical point of attachment to a network segment. A bound LAN card is one example of an adapter. Similarly, a machine with two modems, each capable of connecting to a remote network, will have two adapters, one to represent each modem.

Interfaces

An interface represents a network that can be reached over a LAN or WAN adapter. Each interface has a unique identifier on the router. Interfaces that are active have an adapter that is providing connectivity to the network they represent. Interfaces that are inactive do not have an adapter providing connectivity.

Routing a packet to a network represented by an interface will cause the router to allocate an adapter for that interface, and establish a WAN connection to the remote network. Allocating an adapter to an interface is referred to as "binding."

Interfaces are manageable objects. Each interface appears as a row in the Interface Table of the appropriate SNMP MIB.

Static and Autostatic Routes

Typically, routes to remote networks are obtained dynamically through routing protocols. However, the administrator can also "seed" the routing table by providing routes manually. These routes are referred to as *static*. A static route is associated with an interface that represents the remote network. Unlike dynamic routes, static routes are retained even if the router is restarted or the interface is disabled.

An *autostatic* route is obtained through a routing protocol, but once obtained behaves like a static route. The process for obtaining autostatic routes is as follows: The IP or IPX router manager issues a request that a routing protocol update the routing information for a specific interface. The results of the update are then converted into static routes. Note that only certain routing protocols support requests for autostatic route updates.

Routing Protocol Interface Reference

This section describes the functions and structures that are used to implement a routing protocol as a user-mode DLL.

Routing Protocol Interface Functions

Implement the following functions for a routing protocol DLL:

AddInterface
ConnectClient
DeleteInterface
DisconnectClient
DoUpdateRoutes
GetEventMessage
GetGlobalInfo
GetInterfaceInfo
GetMfeStatus
GetNeighbors
InterfaceStatus
MibCreate
MibDelete
MibGet
MibGetFirst
MibGetNext
MibSet
MibSetTrapInfo
QueryPower
RegisterProtocol
SetGlobalInfo
SetInterfaceInfo
SetPower
StartComplete
StartProtocol
StopProtocol
UnbindInterface

If the routing protocol supports service handling, implement the following function in addition to those listed preceding:

DoUpdateServices

AddInterface

The **AddInterface** function adds an interface to be managed by the routing protocol. The protocol should consider the interface to be in a disabled state. The router manager enables the interface by calling **InterfaceStatus** with the RIS_INTERFACE_ENABLED flag.

```
DWORD AddInterface(
  LPWSTR InterfaceName,  // name of the interface
  ULONG InterfaceIndex,  // index for the interface
  NET_INTERFACE_TYPE InterfaceType,
                          // type of the interface
  DWORD MediaType,
  WORD AccessType,
  WORD ConnectionType,
  PVOID InterfaceInfo,    // interface information block
  ULONG StructureVersion,
  ULONG StructureSize,
  ULONG StructureCount
);
```

Parameters

InterfaceName

[in] Pointer to a Unicode string. The string contains a name that uniquely identifies the interface in the set of interfaces configured on the router.

InterfaceIndex

[in] Identifies the interface in the set of interfaces configured on the router.

InterfaceType

[in] The type of the interface.

Value	Description
PERMANENT	Permanent connectivity (e.g., LAN, Frame Relay).
DEMAND_DIAL	Demand dial connectivity (analog, ISDN, PPTP, switched FR).
LOCAL_WORKSTATION_DIAL	Local workstation connectivity only.
REMOTE_WORKSTATION_DIAL	Remote workstation connectivity only.

MediaType

[in] Specifies the media type.

AccessType

[in] Specifies the type of network access.

ConnectionType

[in] Specifies the type of network connection.

InterfaceInfo

[in] Pointer to a buffer that contains protocol-defined configuration information associated with the interface. This information is private to the routing protocol.

StructureVersion

[in] Specifies the version of the information structures pointed to by the *InterfaceInfo* parameter. In some cases, this is equal to the version of the routing protocol.

StructureSize
[in] Specifies the size of each of the information structures pointed to by the *InterfaceInfo* parameter. Since some information structures contain variable length members, the routing protocol isn't necessarily able to determine the size of the information from the version.

StructureCount
[in] Specifies a count of the number of information structures pointed to by the *InterfaceInfo* parameter. This parameter is always one.

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	Description
ERROR_CAN_NOT_COMPLETE	The attempt to add the interface failed.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid (for example, an interface with that index already exists), or one of the parameters pointed to by *InterfaceInfo* is invalid.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **DeleteInterface**

ConnectClient

The router manager calls the **ConnectClient** function when a client connects to an interface on which the routing protocol is running.

```
typedef DWORD (WINAPI *PCONNECT_CLIENT) (
  ULONG InterfaceIndex,
  PVOID ClientAddress
);
```

Parameters

InterfaceIndex
[in] Specifies the index of the interface on which the client is connecting.

ClientAddress
[in] Pointer to the address (e.g. the IP address) of the connecting client.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value should be one of the following error codes.

Value	Description
ERROR_INVALID_PARAMETER	At least one of the following is true: The *InterfaceIndex* parameter is invalid, for example, no interface exists with that index.

Requirements

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

See Also

DisconnectClient

DeleteInterface

The **DeleteInterface** function removes an interface from the set managed by the routing protocol.

```
DWORD DeleteInterface(
  ULONG InterfaceIndex  // index of the interface
);
```

Parameters

InterfaceIndex
Identifies the interface in the set of interfaces configured on the router.

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	Description
ERROR_CAN_NOT_COMPLETE	The attempt to delete the interface failed.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid (for example, no interface exists with that index).

! Requirements

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

+ See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **AddInterface**

DisconnectClient

The router manager calls the **DisconnectClient** function when a client disconnects from an interface on which the routing protocol is running.

```
typedef DWORD (WINAPI *PDISCONNECT_CLIENT) (
  ULONG InterfaceIndex,
  PVOID ClientAddress
);
```

Parameters

InterfaceIndex
[in] Specifies the index of the interface on which the client is connecting.

ClientAddress
[in] Pointer to the address (e.g. the IP address) of the connecting client.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value should be one of the following error codes.

Value	Description
ERROR_INVALID_PARAMETER	At least one of the following is true: The *InterfaceIndex* parameter is invalid, for example, no interface exists with that index.

! Requirements

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

+ See Also

ConnectClient

DoUpdateRoutes

The **DoUpdateRoutes** function requests the routing protocol to perform a routing information update over the specified interface to obtain static route information. (This process is called an autostatic route update.)

```
DWORD DoUpdateRoutes(
  ULONG InterfaceIndex    // index of the interface
);
```

Parameters

InterfaceIndex
Identifies the interface in the set of interfaces configured on the router.

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	Description
ERROR_CAN_NOT_COMPLETE	The update operation could not be performed.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid (for example, no interface exists with that index).

Remarks

If the function returns NO_ERROR, the update operation started successfully on the interface. Check the routing protocol event queue for a completion event (see ***GetEventMessage***).

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **DoUpdateServices**, **GetEventMessage**

DoUpdateServices

The **DoUpdateServices** function requests the routing protocol to perform a service information update over the interface to obtain static service information. This process is called an autostatic service update.

```
DWORD DoUpdateServices(
  ULONG InterfaceIndex    // index of the interface
);
```

Parameters

InterfaceIndex
  Identifies the interface in the set of interfaces configured on the router.

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	Description
ERROR_CAN_NOT_COMPLETE	The update operation could not be performed.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid (for example, no interface exists with that index).

Remarks

If the function returns NO_ERROR, the update operation started successfully on the interface. Check the routing protocol event queue for a completion event (see ***GetEventMessage***).

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **DoUpdateRoutes**, **GetEventMessage**

GetEventMessage

The **GetEventMessage** function gets an entry from the routing protocol's message queue. The routing protocol uses the queue to inform the router manager of asynchronous events.

```
DWORD GetEventMessage(
  ROUTING_PROTOCOL_EVENTS *Event,
                      // address of message queue entry
  MESSAGE *Result     // event dependent message
);
```

Parameters

Event

Pointer to an event. Information about this event is reported in the associated message. Note that this is *not* a Win32 event object.

(The **ROUTING_PROTOCOL_EVENTS** type is declared in Routprot.h.)

Event values

Value	**Description**
ROUTER_STOPPED	The router protocol shut down successfully. The message is empty for this event. (See ***StopProtocol***)
SAVE_GLOBAL_CONFIG_INFO	The routing protocol reports that its global configuration information has been changed by an external agent, that is, through means other than **SetGlobalInfo**. The routing protocol requests that the router manager retrieve and permanently store this information. The message is empty for this event.
SAVE_INTERFACE_CONFIG_INFO	The routing protocol reports that the configuration information associated with one of its interfaces has been changed by an external agent, that is, through means other than **SetInterfaceInfo**. The routing protocol requests that the router manager retrieve and permanently store this information. The message contains the ID of the interface.
UPDATE_COMPLETE	The routing protocol has completed an autostatic update request from the router manager. The router manager can proceed with converting received routing information to static. The message contains the index of the interface on which the update was performed, the type of the information received (routes or services), and the result field, which indicates whether the update succeeded. See ***DoUpdateRoutes*** and ***DoUpdateServices***.

Result

Pointer to a **MESSAGE** union. The contents of the message are specific to the reported event.

This parameter is optional; the caller may specify NULL for this parameter.

Return Values

If the entry is retrieved successfully, the return value is NO_ERROR.

If the routing protocol's message queue does not contain any entries, the return value is ERROR_NO_MORE_ITEMS.

! Requirements

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

+ See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **DoUpdateRoutes**, **DoUpdateServices**, **MESSAGE**, **SetGlobalInfo**, **SetInterfaceInfo**, **StopProtocol**

GetGlobalInfo

The **GetGlobalInfo** function retrieves global (as opposed to interface-specific) configuration information kept by the routing protocol.

```
DWORD GetGlobalInfo(
  PVOID GlobalInfo    // address of configuration information
  PULONG GlobalInfoSize  // size of configuration information
);
```

Parameters

GlobalInfo
Pointer to a buffer to receive the protocol-defined global configuration information. The format of this information is specific to the routing protocol.

GlobalInfoSize
Pointer to a **DWORD** variable.

On input this variable contains the size, in bytes, of the buffer pointed to by the *GlobalInfo* parameter.

On output this variable contains the size, in bytes, of the data placed in the output buffer. If the initial size was not large enough, the variable contains the size required to hold all of the output data.

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	Description
ERROR_CAN_NOT_COMPLETE	The routing protocol could not retrieve the global information.
ERROR_INSUFFICIENT_BUFFER	The size of the output buffer provided is not large enough to hold the requested information. The required size is returned in the **DWORD** variable pointed to by *OutputDataSize*.
ERROR_INVALID_PARAMETER	The *GlobalInfoSize* parameter is NULL.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **SetInterfaceInfo**, **SetGlobalInfo**

GetInterfaceInfo

The **GetInterfaceInfo** function gets the configuration information kept by the routing protocol for a specific interface.

```
DWORD GetInterfaceInfo(
  ULONG InterfaceIndex, // the index of the interface
  PVOID InterfaceInfo,  // buffer for interface information
  PULONG BufferSize,    // size of the buffer for interface
                        // information
  ULONG StructureVersion,
  ULONG StructureSize,
  ULONG StructureCount
);
```

Parameters

InterfaceIndex
[in] Identifies the interface in the set of interfaces configured on the router.

InterfaceInfo
[in] Pointer to a buffer that receives the protocol-defined configuration information associated with the interface. This information is private to the routing protocol.

BufferSize

[in, out] Pointer to a **DWORD** variable.

On input: This variable contains the size, in bytes, of the buffer provided to receive the configuration information.

On output: This variable contains the size, in bytes, of the data placed in the buffer. If the initial size was not large enough, this variable contains the size required to hold all of the data.

StructureVersion

[in] Specifies the version of the information structures pointed to by the *InterfaceInfo* parameter. In some cases, this is equal to the version of the routing protocol.

StructureSize

[in] Specifies the size of each of the information structures pointed to by the *InterfaceInfo* parameter. Since some information structures contain variable length members, the routing protocol isn't necessarily able to determine the size of the information from the version.

StructureCount

[in] Specifies a count of the number of information structures pointed to by the *InterfaceInfo* parameter. This parameter is always one.

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	Description
ERROR_CAN_NOT_COMPLETE	The attempt to retrieve the information failed.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid (for example, no interface exists with that index), or the *InterfaceInfoSize* parameter is NULL.
ERROR_INSUFFICIENT_BUFFER	The size of the output buffer provided is not large enough to hold the requested information. The required size is returned in the **DWORD** variable pointed to by *InterfaceInfoSize*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Iphlpapi.h.
Library: Use Iphlpapi.lib.

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **SetInterfaceInfo**

GetMfeStatus

The router manager calls the **GetMfeStatus** function to obtain the status of the multicast forwarding entry (MFE) for the specified interface, group address, and source address.

```
typedef DWORD (WINAPI *PGET_MFE_STATUS) (
  DWORD InterfaceIndex,
  DWORD GroupAddress,
  DWORD SourceAddress,
  PBYTE StatusCode
);
```

Parameters

InterfaceIndex
[in] Specifies the index of the interface for this MFE.

GroupAddress
[in] Specifies the multicast group address for this MFE.

SourceAddress
[in] Specifies the multicast source address for this MFE.

StatusCode
[out] Pointer to a **BYTE** variable. The routing protocol should fill in this variable with one of the following values. The routing protocol should select the highest-valued code that applies.

Value	Meaning
MFE_NO_ERROR	None of the following values apply.
MFE_REACHED_CORE	The local computer is this router is an rendezvous point (RP)/core router for the multicast group.
MFE_OIF_PRUNED	This value should be set only by the owner of the outgoing interface. The value indicates that no downstream receivers exist on the outgoing interface.
MFE_PRUNED_UPSTREAM	This value should be set only by the owner of the incoming interface. The value indicates that a prune message was sent upstream.
MFE_OLD_ROUTER	This value should be set only by the owner of the incoming interface. The value indicates that the upstream neighbor doesn't support mtrace.

Return Values

If the function succeeds, the return value should be NO_ERROR.

If the function fails, the return value shoudl be one of the following error codes.

Value	Description
ERROR_CAN_NOT_COMPLETE	The routing protocol could not complete the request.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid (for example, no interface exists with that index), or the group or source address is invalid.

Remarks

Only multicast routing protocols need implement this function. Non-multicast routing protocols should pass NULL as the pointer value for this function in **MPR_ROUTING_CHARACTERISTICS**

Requirements

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

See Also

GetNeighbors

GetNeighbors

The router manager calls the **GetNeighbors** function to obtain the querier for the network attached through the specified interface.

```
typedef DWORD (WINAPI *PGET_NEIGHBORS) (
  DWORD InterfaceIndex,
  PDWORD NeighborList,
  PDWORD NeighborListSize,
  PBYTE InterfaceFlags
);
```

Parameters

InterfaceIndex
[in] Specifies the index of the interface on which the routing protocol should provide the querier.

NeighborList

[in] Pointer to an array **DWORD** variables. The routing protocol should fill in this array with the address of the querier.

If the local computer is the querier for the network attached through the specified interface, the routing protocol need not fill in this variable. Instead, the routing protocol should set the value pointed to by *NeighborListSize* to zero. Also, the routing protocol should add MRINFO_QUERIER_FLAG to the flags returned in the *InterfaceFlags* parameter.

NeighborListSize

[in, out] Pointer to a **DWORD** variable. The routing protocol should fill in this variable with the length (in bytes) of the address returned in the *NeighborList* parameter.

InterfaceFlags

[out] Specifies one or more of the following flags. The flags describe the relationship of the local computer to other computers on the network attached through the specified interface.

MRINFO_TUNNEL_FLAG
MRINFO_PIM_FLAG
MRINFO_DOWN_FLAG
MRINFO_DISABLED_FLAG
MRINFO_QUERIER_FLAG
MRINFO_LEAF_FLAG

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	Description
ERROR_CAN_NOT_COMPLETE	The routing protocol could not complete the request.
ERROR_INSUFFICIENT_BUFFER	The size of the buffer pointed to by *NeighborList* is not large enough to hold the address. The required size is returned in the **DWORD** variable pointed to by the *NeigbhorListSize* parameter.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid (for example, no interface exists with that index).

Remarks

Only multicast routing protocols need implement this function. Non-multicast routing protocols should pass NULL as the pointer value for this function in **MPR_ROUTING_CHARACTERISTICS**

Requirements

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

See Also

GetMfeStatus

InterfaceStatus

Router manager calls the **InterfaceStatus** function to change the status of an interface.

```
typedef DWORD (WINAPI * PINTERFACE_STATUS) (
  ULONG InterfaceIndex,
  BOOL InterfaceActive,
  DWORD StatusType,
  PVOID StatusInfo
);
```

Parameters

InterfaceIndex
[in] Specifies the index of the interface to change.

InterfaceActive
[in] Specifies whether the interface is active.

StatusType
[in] Specifies the new interface status. This parameter is one of the following values:

RIS_INTERFACE_ADDRESS_CHANGE
RIS_INTERFACE_ENABLED
RIS_INTERFACE_DISABLED
RIS_INTERFACE_MEDIA_PRESENT
RIS_INTERFACE_MEDIA_ABSENT

StatusInfo
[in] Pointer to a structure that contains information appropriate to the type of interface status type. For example, if the *StatusType* parameter specifies an address change, the StatusInfo parameter will point to a structure that contains the new address information, e.g. **IP_ADAPTER_BINDING_INFO** or **IPX_ADAPTER_BINDING_INFO**. This parameter may be NULL.

Return Values

If the function succeeds, the return value should be NO_ERROR.

If the function fails, the return value should be one of the following error codes.

Value	Description
ERROR_CAN_NOT_COMPLETE	Unspecified failure.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid (for example, no interface exists with that index).

Requirements

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

See Also

AddInterface, **DeleteInterface**

MibCreate

The **MibCreate** function passes an SNMP MIB-style Create Request to the routing protocol.

```
DWORD MibCreate(
  ULONG InputDataSize,  // size of the data for the request
  PVOID InputData      // address of the data for the request
);
```

Parameters

InputDataSize
Specifies the size of the data for the Create Request.

InputData
Pointer to a buffer that contains the data for the Create Request.

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	Description
ERROR_CAN_NOT_COMPLETE	The routing protocol could not complete the request.
ERROR_INVALID_PARAMETER	Specifies the size or content of the data is inappropriate for the request.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **MibDelete**

MibDelete

The **MibDelete** function passes an SNMP MIB-style Delete Request to the routing protocol.

```
DWORD MibDelete(
  ULONG InputDataSize,  // size of the data for the request
  PVOID InputData       // address of the data for the request
);
```

Parameters

InputDataSize
Specifies the size of the data for the Delete Request.

InputData
Pointer to a buffer that contains the data for the Delete Request.

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	Description
ERROR_CAN_NOT_COMPLETE	The routing protocol could not complete the request.
ERROR_INVALID_PARAMETER	The size or content of the data is inappropriate for the request.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **MibCreate**

MibGet

The **MibGet** function passes an SNMP MIB-style Get Request to the routing protocol DLL.

```
DWORD MibGet(
  ULONG InputDataSize,   // size of the data for the request
  PVOID InputData,    // address of the data for the request
  ULONG *OutputDataSize, // size of the data returned
  PVOID OutputData       // address of data returned
);
```

Parameters

InputDataSize
Specifies the size of the data for the Get Request.

InputData
Pointer to a buffer that contains the data for the Get Request.

OutputDataSize
Pointer to a **ULONG** variable:

On input: This variable contains the size of the output buffer.

On output: This variable contains the size of the data placed in the output buffer. If the initial size was not large enough, the variable contains the buffer size required to hold all of the output data.

OutputData
Pointer to a buffer to receive the data from the MIB entry.

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	Description
ERROR_CAN_NOT_COMPLETE	The routing protocol could not complete the request.
ERROR_INVALID_PARAMETER	The size or content of the data is inappropriate for the request.
ERROR_INSUFFICIENT_BUFFER	The size of the output buffer provided is not large enough to hold the requested information. The required size is returned in the **ULONG** variable pointed to by the *OutputDataSize* parameter.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **MibGetFirst**, **MibGetNext**, **MibSet**

MibGetFirst

The **MibGetFirst** function passes a SNMP MIB-style Get First Request to the routing protocol.

```
DWORD MibGetFirst(
  ULONG InputDataSize,    // size of the data for the request
  PVOID InputData,        // address of the data for the request
  ULONG *OutputDataSize,  // size of the data returned
  PVOID OutputData        // address of the data returned
);
```

Parameters

InputDataSize
Specifies the size of the data for the Get First Request.

InputData
Pointer to the data to be passed with the Get First Request.

OutputDataSize
Pointer to a **ULONG** variable:

On input: This variable contains the size of the output buffer.

On output: This variable contains the size of the data placed in the output buffer. If the initial size was not large enough, the variable contains the buffer size required to hold all of the output data.

OutputData
Pointer to a buffer to receive the data from the MIB entry.

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	Description
ERROR_CAN_NOT_COMPLETE	The routing protocol could not complete the request.
ERROR_INVALID_PARAMETER	The size or content of the data is inappropriate for the request.
ERROR_INSUFFICIENT_BUFFER	The size of the output buffer provided is not large enough to hold the requested information. The required size is returned in the **ULONG** variable pointed to by the *OutputDataSize* parameter.

! Requirements

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

+ See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **MibGet**, **MibGetNext**, **MibSet**

MibGetNext

The **MibGetNext** function passes a SNMP MIB-style Get Next Request to the routing protocol.

```
DWORD MibGetNext(
  ULONG InputDataSize,   // size of the data for the request
  PVOID InputData,      // address of the data for the request
  ULONG *OutputDataSize, // size of the returned data
  PVOID OutputData       // address of the returned data
);
```

Parameters

InputDataSize
Specifies the size of the data for the Get Next Request.

InputData
Pointer to the data for the Get Next Request.

OutputDataSize
Pointer to a **ULONG** variable:

On input: This variable that contains the size of the output buffer.

On output: This variable contains the size of data placed in the output buffer. If the initial size was not large enough, the variable contains the buffer size required to hold all of the output data.

OutputData

Pointer to a buffer to receive the data from the MIB entry.

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	Description
ERROR_CAN_NOT_COMPLETE	The routing protocol could not complete the request.
ERROR_INVALID_PARAMETER	The size or content of the data is inappropriate for the request.
ERROR_INSUFFICIENT_BUFFER	The size of the output buffer provided is not large enough to hold the requested information. The required size is returned in the **ULONG** variable pointed to by the *OutputDataSize* parameter.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **MibGet**, **MibGetFirst**, **MibSet**

MibGetTrapInfo

The **MibEntryGetTrapInfo** function queries the module that set a trap event for more information about the trap.

```
typedef DWORD (WINAPI * PMIB_GET_TRAP_INFO) (
  ULONG InputDataSize,
  PVOID InputData,
  PULONG OutputDataSize,
  PVOID OutputData
);
```

Parameters

InputDataSize
[in] Specifies a **ULONG** variable that contains the size in bytes of the data pointed to by *InputData*.

InputData
[in] Pointer to the input data.

OutputDataSize
[out] Pointer to a **ULONG** variable that contains the size in bytes of the data pointed to by **OutputData*.

OutputData
[out] Specifies on successful return, the address of a pointer to the output data.

Return Values

If the functions succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes.

Value	Description
ERROR_ACCESS_DENIED	The caller does not have sufficient privileges.
ERROR_NOT_ENOUGH_MEMORY	There are insufficient resources to complete the operation.

Requirements

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

See Also

MibSetTrapInfo

MibSet

The **MibSet** function passes a SNMP MIB-style Set Request to the routing protocol.

```
DWORD MibSet(
  ULONG InputDataSize,  // size of the data for the request
  PVOID InputData       // address of the data for the request
);
```

Parameters

InputDataSize
Specifies the size of the data for the Set Request.

InputData
Pointer to a buffer that contains the data for the Set Request.

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	Description
ERROR_CAN_NOT_COMPLETE	The routing protocol could not complete the request.
ERROR_INVALID_PARAMETER	The size or content of the data is inappropriate for the request.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **MibGet**, **MibGetFirst**, **MibGetNext**

MibSetTrapInfo

The **MibSetTrapInfo** function passes in a handle to an event which is signaled whenever a TRAP needs to be issued.

```
typedef DWORD (WINAPI * PMIB_SET_TRAP_INFO) (
  HANDLE Event,
  ULONG InputDataSize,
  PVOID InputData,
  PULONG OutputDataSize,
  PVOID OutputData
);
```

Parameters

Event
[in] Handle to an event that is signaled when a TRAP needs to be issued.

InputDataSize
[in] Specifies a **ULONG** variable that contains the size in bytes of the data pointed to by *InputData*.

InputData
[in] Pointer to the input data.

OutputDataSize
[out] Pointer to a **ULONG** variable that contains the size in bytes of the data pointed to by **OutputData*.

OutputData
[out] Specifies on successful return, the address of a pointer to the output data.

Return Values

If the functions succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes.

Value	Description
ERROR_ACCESS_DENIED	The caller does not have sufficient privileges.
ERROR_NOT_ENOUGH_MEMORY	There are insufficient resources to complete the operation.

! Requirements

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

+ See Also

MibGetTrapInfo

QueryPower

The **QueryPower** function is reserved for future use. It is not currently called by the router manager. Routing protocols should pass NULL as the pointer value for this function in **MPR_ROUTING_CHARACTERISTICS**.

```
typedef DWORD (WINAPI * PQUERY_POWER) (
  DWORD PowerType
);
```

! Requirements

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

See Also

SetPower

RegisterProtocol

The **RegisterProtocol** function registers the routing protocol with the router manager. It also informs the router manager of the functionality that the routing protocol supports.

```
DWORD RegisterProtocol (
  PMPR_ROUTING_CHARACTERISTICS pRoutingChar,
  PMPR_SERVICE_CHARACTERISTICS pServiceChar
);
```

Parameters

pRoutingChar
Pointer to an **MPR_ROUTING_CHARACTERISTICS** structure. See the reference page for this structure for more information on how to use it with the **RegisterProtocol** function.

pServiceChar
Pointer to an **MPR_SERVICE_CHARACTERISTICS** structure. See the reference page for this structure for more information on how to use it with the **RegisterProtocol** function.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is ERROR_NOT_SUPPORTED.

Remarks

All routing protocol DLLs must fill in values for the **MPR_ROUTING_CHARACTERISTICS** structure.

Routing protocol DLLs that provide services must fill in values for the **MPR_SERVICE_CHARACTERISTICS** structure. If a routing protocol DLL does not provide services, it should fill in zero for the **fSupportedFunctionality** member of this structure, but need not fill in values for the other members.

Routing protocols are implemented in user-mode DLLs. A single DLL may implement multiple routing protocols. Therefore, router manager may call **RegisterProtocol** multiple times, once for each routing protocol implemented in the DLL.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **MPR_ROUTING_CHARACTERISTICS**, **MPR_SERVICE_CHARACTERISTICS**

SetGlobalInfo

The **SetGlobalInfo** function sets the global (as opposed to interface-specific) configuration information kept by the routing protocol. The format of this information is specific to the routing protocol.

```
DWORD SetGlobalInfo(
  PVOID GlobalInfo  // address of GlobalInfo block
);
```

Parameters

GlobalInfo

Pointer to a buffer containing the protocol-defined global configuration information.

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	Description
ERROR_CAN_NOT_COMPLETE	The routing protocol could not set the configuration information.
ERROR_INVALID_PARAMETER	The *GlobalInfo* parameter is NULL, or one of the parameters in the configuration information is invalid.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **GetInterfaceInfo**, **GetGlobalInfo**

SetInterfaceInfo

The **SetInterfaceInfo** function sets the configuration of a specific interface managed by the routing protocol.

```
DWORD SetInterfaceInfo(
  ULONG InterfaceIndex,  // index of interface
  PVOID InterfaceInfo,   // interface information block
  ULONG StructureVersion,
  ULONG StructureSize,
  ULONG StructureCount
);
```

Parameters

InterfaceIndex
[in] Identifies the interface in the set of interfaces configured on the router.

InterfaceInfo
[in] Pointer to a buffer that holds the protocol-defined configuration information associated with the interface. This information is private to the routing protocol.

StructureVersion
[in] Specifies the version of the information structures pointed to by the *InterfaceInfo* parameter. In some cases, this is equal to the version of the routing protocol.

StructureSize
[in] Specifies the size of each of the information structures pointed to by the *InterfaceInfo* parameter. Since some information structures contain variable length members, the routing protocol isn't necessarily able to determine the size of the information from the version.

StructureCount
[in] Specifies a count of the number of information structures pointed to by the *InterfaceInfo* parameter. This parameter is always one.

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	Description
ERROR_CAN_NOT_COMPLETE	The attempt to set the interface configuration failed.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid (for example, no interface exists with that index), the *InterfaceInfo* parameter is NULL, or one of the parameters in the configuration information is invalid.

! Requirements

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

+ See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **GetInterfaceInfo**

SetPower

The **SetPower** function is reserved for future use. It is not currently called by the router manager. Routing protocols should pass NULL as the pointer value for this function in **MPR_ROUTING_CHARACTERISTICS**.

```
typedef DWORD (WINAPI * PSET_POWER) (
  DWORD PowerType
);
```

! Requirements

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

+ See Also

QueryPower

StartComplete

Router Manager calls the **StartComplete** function to inform the routing protocol that initialization is complete and all interfaces have been added. The routing protocol should wait for this call before starting any protocol-specific behavior.

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

Parameters

This function takes no parameters.

Return Values

This function should return NO_ERROR.

Requirements

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

See Also

StartProtocol

StartProtocol

The **StartProtocol** function initializes the routing protocol's functionality. The router manager uses this function to pass the routing protocol global configuration parameters and a set of API entry points. The protocol uses these entry points to call into the router manager.

```
DWORD StartProtocol(
  HANDLE NotificationEvent,
                  // handle to an event object
  PSUPPORT_FUNCTIONS SupportFunctions,
                  // address of SUPPORT_FUNCTIONS structure
  PVOID GlobalInfo // address of configuration information
  ULONG StructureVersion,
  ULONG StructureSize,
  ULONG StructureCount
);
```

Parameters

NotificationEvent
: [in] Handle to an event object. The routing protocol will signal this event when it wants the router manager to retrieve an asynchronous message from the queue maintained by the protocol.

SupportFunctions
: [in] Pointer to a **SUPPORT_FUNCTIONS** structure. The fields of this structure are pointers to functions in the router manager. These functions allow the protocol to access information that spans routing protocols.

GlobalInfo
: [in] Pointer to protocol-defined global (as opposed to interface-specific) configuration information. This information is private to the routing protocol.

StructureVersion
: [in] Specifies the version of the information structures pointed to by the *GlobalInfo* parameter. In some cases, this is equal to the version of the routing protocol.

StructureSize
[in] Specifies the size of each of the information structures pointed to by the *GlobalInfo* parameter. Since some information structures contain variable length members, the routing protocol isn't necessarily able to determine the size of the information from the version.

StructureCount
[in] Specifies a count of the number of information structures pointed to by the *GlobalInfo* parameter. This parameter is always one.

Return Values

If the function succeeds, and the protocol is ready to receive interface information, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes.

Value	Description
ERROR_CAN_NOT_COMPLETE	The attempt to initialize the routing protocol failed.
ERROR_INVALID_PARAMETER	One of the parameters pointed to by the *GlobalInfo* parameter is invalid.

! Requirements

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

+ See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **GetEventMessage**, **StopProtocol**, **SUPPORT_FUNCTIONS**

StopProtocol

The **StopProtocol** function causes the routing protocol to perform an orderly shutdown.

```
DWORD StopProtocol (void);
```

Return Values

If the routing protocol shutdown successfully (synchronous completion), the return value is NO_ERROR.

If routing protocol is shutting down asynchronously, the return value is ERROR_PROTOCOL_STOP_PENDING. In this case, the protocol will report the results of the shutdown through the event message queue.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions, **GetEventMessage**, **StartProtocol**

UnbindInterface

The **UnbindInterface** function tells the routing protocol that an adapter has been deallocated from the specified interface. The function directs the routing protocol to stop protocol-defined activities over the adapter.

```
DWORD UnbindInterface(
  ULONG InterfaceIndex   // index of the interface
);
```

Parameters

InterfaceIndex
Identifies the interface in the set of interfaces configured on the router.

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	Description
ERROR_CAN_NOT_COMPLETE	The attempt to unbind the interface failed.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid (for example, no interface exists with that index, or the interface exists, but is already unbound).

Remarks

The routing protocol should no longer consider routes dynamically obtained through the interface to be valid. It should remove these routes from the routing table.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Functions

Routing Protocol Interface Structures

The Routing Protocol Interface Functions use the following structures:

MESSAGE
MPR_ROUTING_CHARACTERISTICS
MPR_SERVICE_CHARACTERISTICS
UPDATE_COMPLETE_MESSAGE

MESSAGE

The **MESSAGE** union contains information about an event reported to the router manager through the routing protocol's message queue.

```
typedef union _MESSAGE {
  UPDATE_COMPLETE_MESSAGE UpdateCompleteMessage;
  DWORD                   InterfaceIndex;
}  MESSAGE, *PMESSAGE;
```

Members

UpdateCompleteMessage
Provides information associated with an UPDATE_COMPLETE event.

InterfaceIndex
Identifies the interface associated with a SAVE_INTERFACE_CONFIG_INFO event.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Structures, **DoUpdateRoutes**, **DoUpdateServices**, **GetEventMessage**, **UPDATE_COMPLETE_MESSAGE**

MPR_ROUTING_CHARACTERISTICS

The **MPR_ROUTING_CHARACTERISTICS** structure contains information used to register routing protocols with the router manager.

```
typedef struct _MPR50_ROUTING_CHARACTERISTICS {
  DWORD                dwVersion;
  DWORD                dwProtocolId;
  DWORD                fSupportedFunctionality;

  PSTART_PROTOCOL      pfnStartProtocol;
  PSTART_COMPLETE      pfnStartComplete;
  PSTOP_PROTOCOL       pfnStopProtocol;
  PGET_GLOBAL_INFO     pfnGetGlobalInfo;
  PSET_GLOBAL_INFO     pfnSetGlobalInfo;
  PQUERY_POWER         pfnQueryPower;
  PSET_POWER           pfnSetPower;

  PADD_INTERFACE       pfnAddInterface;
  PDELETE_INTERFACE    pfnDeleteInterface;
  PINTERFACE_STATUS    pfnInterfaceStatus;
  PGET_INTERFACE_INFO pfnGetInterfaceInfo;
  PSET_INTERFACE_INFO pfnSetInterfaceInfo;

  PGET_EVENT_MESSAGE   pfnGetEventMessage;

  PDO_UPDATE_ROUTES    pfnUpdateRoutes;

  PCONNECT_CLIENT      pfnConnectClient;
  PDISCONNECT_CLIENT   pfnDisconnectClient;

  PGET_NEIGHBORS       pfnGetNeighbors;
  PGET_MFE_STATUS      pfnGetMfeStatus;

  PMIB_CREATE          pfnMibCreateEntry;
  PMIB_DELETE          pfnMibDeleteEntry;
  PMIB_GET             pfnMibGetEntry;
  PMIB_SET             pfnMibSetEntry;
  PMIB_GET_FIRST       pfnMibGetFirstEntry;
  PMIB_GET_NEXT        pfnMibGetNextEntry;
  PMIB_SET_TRAP_INFO   pfnMibSetTrapInfo;
  PMIB_GET_TRAP_INFO   pfnMibGetTrapInfo;
} MPR50_ROUTING_CHARACTERISTICS

typedef MPR50_ROUTING_CHARACTERISTICS MPR_ROUTING_CHARACTERISTICS;

typedef MPR_ROUTING_CHARACTERISTICS *PMPR_ROUTING_CHARACTERISTICS;
```

Members

dwVersion

On input: specifies the version of RRAS currently running.

On output: the routing protocol should specify the version of RRAS that it requires.

The symbol **MS_ROUTER_VERSION** in the header file Routprot.h is defined to be the RRAS version for a given implementation.

dwProtocolId

Specifies the routing protocol that the router manager requests the DLL to register. (A common name space is used for all protocol families.)

fSupportedFunctionality

On input: specifies the functionality that the router manager supports.

On output: the routing protocol should reset these flags to indicate the subset of functionality that it supports.

Supported Functionality Values

Value	Description
ROUTING	The protocol participates in Multiprotocol routing by importing routing table manager APIs.
SERVICES	The protocol assumes responsibility for managing services (such as IPX SAP), and provides Service Table Management APIs.
DEMAND_UPDATE_ROUTES	The protocol is able to perform autostatic updates of routes when requested by the router manager.
DEMAND_UPDATE_SERVICES	The protocol is able to perform autostatic updates of services when requested by the router manager.

pfnStartProtocol

Pointer to an implementation of the **StartProtocol** function for this routing protocol.

pfnStartComplete

Pointer to an implementation of the **StartComplete** function for this routing protocol.

pfnStopProtocol

Pointer to an implementation of the **StopProtocol** function for this routing protocol.

pfnGetGlobalInfo

Pointer to an implementation of the **GetGlobalInfo** function for this routing protocol.

pfnSetGlobalInfo

Pointer to an implementation of the **SetGlobalInfo** function for this routing protocol.

pfnQueryPower

Pointer to an implementation of the **QueryPower** function for this routing protocol.

pfnSetPower

Pointer to an implementation of the **SetPower** function for this routing protocol.

pfnAddInterface
Pointer to an implementation of the **AddInterface** function for this routing protocol.

pfnDeleteInterface
Pointer to an implementation of the **DeleteInterface** function for this routing protocol.

pfnInterfaceStatus
Pointer to an implementation of the **InterfaceStatus** function for this routing protocol.

pfnGetInterfaceInfo
Pointer to an implementation of the **GetInterfaceInfo** function for this routing protocol.

pfnSetInterfaceInfo
Pointer to an implementation of the **SetInterfaceInfo** function for this routing protocol.

pfnGetEventMessage
Pointer to an implementation of the **GetEventMessage** function for this routing protocol.

pfnUpdateRoutes
Pointer to an implementation of the **DoUpdateRoutes** function for this routing protocol.

pfnConnectClient
Pointer to an implementation of the **ConnectClient** function for this routing protocol.

pfnDisconnectClient
Pointer to an implementation of the **DisconnectClient** function for this routing protocol.

pfnGetNeighbors
Pointer to an implementation of the **GetNeighbors** function for this routing protocol.

pfnGetMfeStatus
Pointer to an implementation of the **GetMfeStatus** function for this routing protocol.

pfnMibCreateEntry
Pointer to an implementation of the **MibCreate** function for this routing protocol.

pfnMibDeleteEntry
Pointer to an implementation of the **MibDelete** function for this routing protocol.

pfnMibGetEntry
Pointer to an implementation of the **MibGet** function for this routing protocol.

pfnMibSetEntry
Pointer to an implementation of the **MibSet** function for this routing protocol.

pfnMibGetFirstEntry
Pointer to an implementation of the **MibGetFirst** function for this routing protocol.

pfnMibGetNextEntry
Pointer to an implementation of the **MibGetNext** function for this routing protocol.

pfnMibSetTrapInfo
Pointer to an implementation of the **MibSetTrapInfo** function for this routing protocol.

pfnMibGetTrapInfo
Pointer to an implementation of the **MibGetTrapInfo** function for this routing protocol.

Remarks

Most of the members of this structure are pointers to functions implemented in the routing protocol DLL. The routing protocol fills in the address values for these pointers during a call to the **RegisterProtocol** function.

For a complete description of a particular function pointed to by one of the structure members, see the reference page for that function.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Structures, **RegisterProtocol**, Protocol Identifiers

MPR_SERVICE_CHARACTERISTICS

The **MPR_SERVICE_CHARACTERISTICS** structure contains information used to register a routing protocol with the router manager.

```
typedef struct _MPR_SERVICE_CHARACTERISTICS {
  DWORD                                dwVersion;
  DWORD                                dwProtocolId;
  DWORD                                fSupportedFunctionality;
  PIS_SERVICE                          pfnIsService;
  PCREATE_SERVICE_ENUMERATION_HANDLE  pfnCreateServiceEnumerationHandle;
  PENUMERATE_GET_NEXT_SERVICE          pfnEnumerateGetNextService;
  PCLOSE_SERVICE_ENUMERATION_HANDLE    pfnCloseServiceEnumerationHandle;
  PGET_SERVICE_COUNT                   pfnGetServiceCount;
  PCREATE_STATIC_SERVICE               pfnCreateStaticService;
  PDELETE_STATIC_SERVICE               pfnDeleteStaticService;
  PBLOCK_CONVERT_SERVICES_TO_STATIC    pfnBlockConvertServicesToStatic;
  PBLOCK_DELETE_STATIC_SERVICES        pfnBlockDeleteStaticServices;
  PGET_FIRST_ORDERED_SERVICE           pfnGetFirstOrderedService;
  PGET_NEXT_ORDERED_SERVICE            pfnGetNextOrderedService;
}MPR_SERVICE_CHARACTERISTICS, *PMPR_SERVICE_CHARACTERISTICS;
```

Members

dwVersion

On input: specifies the version of RRAS currently running.

On output: the routing protocol should specify the version of RRAS that it requires.

The symbol **MS_ROUTER_VERSION** in the header file Routprot.h is defined to be the RRAS version for a given implementation.

dwProtocolId
Specifies the routing protocol that the router manager requests the DLL to register. (A common name space is used for all protocol families.)

fSupportedFunctionality
On input: specifies the functionality that the router manager supports.

On output: the routing protocol should reset these flags to indicate the subset of functionality that it supports. If this routing protocol does not provide services, **fSupportedFunctionality** should be zero.

Supported Functionality Values

Value	Description
ROUTING	The protocol participates in Multiprotocol routing by importing routing table manager APIs.
SERVICES	The protocol assumes responsibility for managing services (such as IPX SAP), and provides Service Table Management APIs.
DEMAND_UPDATE_ROUTES	The protocol is able to perform autostatic updates of routes when requested by the router manager.
DEMAND_UPDATE_SERVICES	The protocol is able to perform autostatic updates of services when requested by the router manager.

pfnIsService
Pointer to an implementation of the **IsService** function for this routing protocol.

pfnCreateServiceEnumerationHandle
Pointer to an implementation of the **CreateServiceEnumerationHandle** function for this routing protocol.

pfnEnumerateGetNextService
Pointer to an implementation of the **EnumerateGetNextService** function for this routing protocol.

pfnCloseServiceEnumerationHandle
Pointer to an implementation of the **CloseServiceEnumerationHandle** function for this routing protocol.

pfnGetServiceCount
Pointer to an implementation of the **GetServiceCount** function for this routing protocol.

pfnCreateStaticService
Pointer to an implementation of the **CreateStaticService** function for this routing protocol.

pfnDeleteStaticService
Pointer to an implementation of the **DeleteStaticService** function for this routing protocol.

pfnBlockConvertServicesToStatic
Pointer to an implementation of the **BlockConvertServicesToStatic** function for this routing protocol.

pfnBlockDeleteStaticServices
Pointer to an implementation of the **BlockDeleteStaticServices** function for this routing protocol.

pfnGetFirstOrderedService
Pointer to an implementation of the **GetFirstOrderedService** function for this routing protocol.

pfnGetNextOrderedService
Pointer to an implementation of the **GetNextOrderedService** function for this routing protocol.

Remarks

The members of this structure are pointers to Service Table Management functions implemented in the routing protocol DLL. The routing protocol fills in the address values for these pointers during a call to the **RegisterProtocol** function.

Only routing protocol DLLs that support services need to fill in the **MPR_SERVICE_CHARACTERISTICS** structure.

For a complete description of a particular function pointed to by one of the structure members, see the reference page for that function.

To use this structure, the user should add -DMPR50=1 to the compiler flags.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Structures, **MPR_ROUTING_CHARACTERISTICS**, **RegisterProtocol**, Protocol Identifiers

UPDATE_COMPLETE_MESSAGE

The **UPDATE_COMPLETE_MESSAGE** structure contains information describing the completion status of an update operation.

```
typedef struct _UPDATE_COMPLETE_MESSAGE {
  ULONG    InterfaceIndex;
  ULONG    UpdateType;
  ULONG    UpdateStatus;
}  UPDATE_COMPLETE_MESSAGE, *PUPDATE_COMPLETE_MESSAGE;
```

Members

InterfaceIndex
Identifies the interface over which the update was performed.

UpdateType
Indicates the type of information that was received in this update.

DEMAND_UPDATE_ROUTES
Routing information was reported to the routing table manager.

DEMAND_UPDATE_SERVICES
Services information that is accessible through the Services Table Management functions provided by the routing protocol.

UpdateStatus
Indicates the result of the update operation.

NO_ERROR
The update was completed successfully.

ERROR_CAN_NOT_COMPLETE
The update was unsuccessful.

Requirements

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

See Also

Routing Protocol Interface Reference, Routing Protocol Interface Structures, **MESSAGE**

Support Functions Reference

The following functions are provided to routing protocols by the router manager. When the router manager calls the **StartProtocol** function (implemented by the routing protocol), the router manager passes the routing protocol a **SUPPORT_FUNCTIONS** structure containing pointers to these functions.

DemandDialRequest
MIBEntryCreate
MIBEntryDelete
MIBEntryGet
MIBEntryGetFirst
MIBEntryGetNext
MIBEntrySet
SetInterfaceReceiveType
ValidateRoute

SUPPORT_FUNCTIONS

The **SUPPORT_FUNCTIONS** structure is used by the router manager to pass the routing protocol a set of pointers to functions provided by the router manager.

```
typedef struct _SUPPORT_FUNCTIONS {
  union
  {
    ULONGLONG   _Align8;

    struct
    {
      DWORD   dwVersion;
      DWORD   dwReserved;
    };
  };
  DWORD (WINAPI *DemandDialRequest)( DWORD, DWORD );
  DWORD (WINAPI *SetInterfaceReceiveType)( DWORD, DWORD, DWORD, BOOL );
  DWORD (WINAPI *ValidateRoute)( DWORD, PVOID, PVOID );
  DWORD (WINAPI *MIBEntryCreate)( DWORD, DWORD, LPVOID );
  DWORD (WINAPI *MIBEntryDelete)( DWORD, DWORD, LPVOID );
  DWORD (WINAPI *MIBEntrySet)( DWORD, DWORD, LPVOID );
  DWORD (WINAPI *MIBEntryGet)( DWORD, DWORD, LPVOID, LPDWORD LPVOID );
  DWORD (WINAPI *MIBEntryGetFirst)( DWORD, DWORD, LPVOID, LPDWORD, LPVOID );
  DWORD (WINAPI *MIBEntryGetNext)( DWORD, DWORD, LPVOID, LPVOID, LPVOID);
} SUPPORT_FUNCTIONS, *PSUPPORT_FUNCTIONS;
```

Members

DemandDialRequest
Pointer to the **DemandDialRequest** function provided by the router manager for the routing protocol.

SetInterfaceReceiveType
Pointer to the **SetInterfaceReceiveType** function provided by the router manager for the routing protocol.

ValidateRoute
Pointer to the **ValidateRoute** function provided by the router manager for the routing protocol.

MIBEntryCreate
Pointer to the **MIBEntryCreate** function provided by the router manager for the routing protocol.

MIBEntryDelete
Pointer to the **MIBEntryDelete** function provided by the router manager for the routing protocol.

MIBEntrySet
Pointer to the **MIBEntrySet** function provided by the router manager for the routing protocol.

MIBEntryGet
Pointer to the **MIBEntryGet** function provided by the router manager for the routing protocol.

MIBEntryGetFirst
Pointer to the **MIBEntryGetFirst** function provided by the router manager for the routing protocol.

MIBEntryGetNext
Pointer to the **MIBEntryGetNext** function provided by the router manager for the routing protocol.

Requirements

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

See Also

StartProtocol

DemandDialRequest

The routing protocol should call **DemandDialRequest** to initiate a demand dial connection.

```
DWORD ( * DemandDialRequest) (
  DWORD InitiatingProtocolId,    // ID of DLL to process request
  DWORD InterfaceIndex           // ID of the interface
);
```

Parameters

InitiatingProtocolId

Specifies the identifier of the routing protocol on behalf of which the connection should be established. (Normally, this parameter is the identifier of the calling routing protocol.)

InterfaceIndex

Specifies the identifier of the interface for which the connection should be established.

Return Value

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_CAN_NOT_COMPLETE	The attempt to establish the connection failed.
ERROR_INVALID_PARAMETER	The *InitiatingProtocolId* parameter and/or the *InterfaceIndex* parameter were/was invalid.

Requirements

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

See Also

Protocol Identifiers

MibEntryCreate

The routing protocol should call **MibEntryCreate** to execute an SNMP MIB-style Create request of the router manager or a peer protocol DLL.

```
DWORD ( * MibEntryCreate ) (
  DWORD TargetProtocolId,    // ID of DLL to process the
                             // request
  DWORD InputDataSize,       // size of the data for the
                             // request
  PVOID InputData            // address of the data for the
                             // request
);
```

Parameters

TargetProtocolId
Specifies the identifier of the DLL that should process this request. This parameter may be the identifier of the router manager or the identifier of a routing protocol.

InputDataSize
Specifies the size, in bytes, of the data to pass with the Create request.

InputData
Pointer to the data to pass with the Create request.

Return Value

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_CAN_NOT_COMPLETE	The attempt to create the MIB entry failed.
ERROR_INVALID_PARAMETER	The size or content of the input data are incompatible with the request.

Requirements

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

See Also

Protocol Identifiers, **MibEntryDelete**

MibEntryDelete

The routing protocol should call **MibEntryDelete** to execute an SNMP MIB-style Delete request of the router manager or a peer protocol DLL.

```
DWORD ( * MibEntryDelete ) (
  DWORD TargetProtocolId,    // ID of DLL to process request
  DWORD InputDataSize,       // size of the data for the
                             // request
  PVOID InputData            // address of the data for the
                             // request
);
```

Parameters

TargetProtocolId
: Specifies the identifier of the DLL that should process this request. This parameter may be the identifier of the router manager or the identifier of a routing protocol.

InputDataSize
: Specifies the size, in bytes, of the data to pass with the Delete request.

InputData
: Pointer to the data to pass with the Delete request.

Return Value

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_CAN_NOT_COMPLETE	The attempt to delete the MIB entry failed.
ERROR_INVALID_PARAMETER	The size or content of the input data are incompatible with the request.

Requirements

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

See Also

Protocol Identifiers, **MibEntryCreate**

MibEntryGet

The routing protocol should call **MibEntryGet** to execute an SNMP MIB-style Get request of the router manager or a peer protocol DLL.

```
DWORD ( * MibEntryGet ) (
  DWORD TargetProtocolId,    // ID of DLL to process request
  DWORD InputDataSize,       // size of the data for the
                             // request
  PVOID InputData,           // address of the data for the
                             // request
  DWORD * OutputDataSize,    // size of the data returned
  PVOID OutputData           // address of the data returned
);
```

Parameters

TargetProtocolId
Specifies the identifier of the DLL that should process this request. This parameter may be the identifier of the router manager or the identifier of a routing protocol.

InputDataSize
Specifies the size, in bytes, of the data to pass with the Get request.

InputData
Pointer to the data to pass with the Get request.

OutputDataSize
A pointer to a **DWORD** variable:

On input: This variable contains the size, in bytes, of the output buffer.

On output: This variable contains the size, in bytes, of data placed in the output buffer. If the initial size was not large enough, this variable contains the buffer size required to hold all of the output data.

OutputData
Pointer to a buffer to hold the data from the MIB entry.

Return Value

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_CAN_NOT_COMPLETE	The operation failed.
ERROR_INVALID_PARAMETER	The size or content of the input data are incompatible with the request.
ERROR_INSUFFICIENT_BUFFER	The size of the output buffer provided is not large enough to hold the requested information. The required size is returned in the **DWORD** variable pointed to by the *OutputDataSize* parameter.

Requirements

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

See Also

Protocol Identifiers, **MibEntrySet**, **MibEntryGetFirst**, **MibEntryGetNext**

MibEntryGetFirst

The routing protocol should call **MibEntryGetFirst** to execute an SNMP MIB-style Get First request of the router manager or a peer protocol DLL.

```
DWORD ( * MibEntryGetFirst ) (
  DWORD TargetProtocolId,     // ID of DLL to process request
  DWORD InputDataSize,        // size of the data for the
                              // request
  PVOID InputData,            // address of the data for the
                              // request
  DWORD * OutputDataSize,     // size of the data returned
  PVOID OutputData            // address of the data returned
);
```

Parameters

TargetProtocolId
Specifies the identifier of the DLL that should process this request. This parameter may be the identifier of the router manager or the identifier of a routing protocol.

InputDataSize
Specifies the size, in bytes, of the data to pass with the Get First request.

InputData
Pointer to the data to pass with the Get First request.

OutputDataSize
A pointer to a **DWORD** variable:

On input: This variable contains the size, in bytes, of the output buffer.

On output: This variable contains the size, in bytes, of data placed in the output buffer. If the initial size is not large enough, this variable contains the buffer size required to hold all of the output data.

OutputData
Pointer to a buffer to hold the data from the MIB entry.

Return Value

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_CAN_NOT_COMPLETE	The operation failed.
ERROR_INVALID_PARAMETER	The size or content of the input data is incompatible with the request.
ERROR_INSUFFICIENT_BUFFER	The size of the output buffer provided is not large enough to hold the requested information. On return, the required size is pointed to by the *OutputDataSize* parameter.

Requirements

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

See Also

Protocol Identifiers, **MibEntryGet**, **MibEntryGetNext**

MibEntryGetNext

The routing protocol should call **MibEntryGetNext** to execute an SNMP MIB-style Get Next request of the router manager or a peer protocol DLL.

```
DWORD ( * MibEntryGetNext ) (
  DWORD TargetProtocolId,     // ID of DLL to process request
  DWORD InputDataSize,        // size of the data for the
                              // request
  PVOID InputData,            // address of the data for the
                              // request
  DWORD * OutputDataSize,     // size of the data returned
  PVOID OutputData            // address of the data returned
);
```

Parameters

TargetProtocolId
Specifies the identifier of the DLL that should process this request. This parameter may be the identifier of the router manager or the identifier of a routing protocol.

InputDataSize
Specifies the size, in bytes, of the data to pass with the Get Next request.

InputData
Pointer to the data to pass with the Get Next request.

OutputDataSize
A pointer to a **DWORD** variable:

On input: This variable contains the size, in bytes, of the output buffer.

On output: This variable contains the size, in bytes, of data placed in the output buffer. If the initial size is not large enough, this variable contains the buffer size required to hold all of the output data.

OutputData
Pointer to a buffer to hold the data from the MIB entry.

Return Value

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_CAN_NOT_COMPLETE	The operation failed.
ERROR_INVALID_PARAMETER	The size or content of the input data is incompatible with the request.
ERROR_INSUFFICIENT_BUFFER	The size of the output buffer provided is not large enough to hold the requested information. On return, *OutputDataSize* points to the required size.

Requirements

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

See Also

Protocol Identifiers, **MibEntryGet**, **MibEntryGetFirst**

MibEntrySet

The routing protocol should call **MibEntrySet** to execute an SNMP MIB-style Set request of the router manager or a peer protocol DLL.

```
DWORD ( * MibEntrySet ) (
  DWORD TargetProtocolId,    // ID of DLL to process request
  DWORD InputDataSize,       // size of the data for the
                             // request
  PVOID InputData            // address of the data for the
                             // request
);
```

Parameters

TargetProtocolId
Specifies the identifier of the DLL that should process this request. This parameter may be the identifier of the router manager or the identifier of a routing protocol.

InputDataSize
Specifies the size, in bytes, of the data to pass with the Set request.

InputData
Pointer to the data to pass with the Set request.

Return Value

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following values.

Value	Description
ERROR_CAN_NOT_COMPLETE	The operation failed.
ERROR_INVALID_PARAMETER	The size or content of the input data are incompatible with the request.

Requirements

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

See Also

Protocol Identifiers, **MibEntryGet**

SetInterfaceReceiveType

The routing protocol can call the **SetInterfaceReceiveType** function to set the receive capability of the specified interface.

```
DWORD (WINAPI *SetInterfaceReceiveType)(
  DWORD ProtocolId,
  DWORD InterfaceIndex,
  DWORD InterfaceReceiveType,
  BOOL bActivate
);
```

Parameters

ProtocolId
[in] Specifies the ID of the routing protocol.

InterfaceIndex
[in] Specifies the index of the interface on which to set the receive type.

InterfaceReceiveType
[in] Specifies the receive type. This parameter should be one of the following values.

IR_PROMISCUOUS
IR_PROMISCUOUS_MULTICAST

bActivate
[in] Specifies whether to activate the interface.

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 value specified by the *dwInterfaceReceiveType* parameter is not valid.
Other	The call failed. Use **FormatMessage** to retrieve the error message corresponding to the returned error code.

! Requirements

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

+ See Also

Protocol Identifiers

ValidateRoute

The routing protocol must call the **ValidateRoute** function to set the route preference and perform other route validation.

```
DWORD (WINAPI *ValidateRoute)(
  DWORD ProtocolId,
  PVOID RouteInfo,
  PVOID DestAddress       // OPTIONAL
);
```

Parameters

ProtocolId
[in] Specifies the ID of the routing protocol.

RouteInfo
[in] Pointer to information describing the route to validate.

DestAddress
[in] Pointer to information describing the destination address. This parameter is optional and may be NULL.

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 some or all of the information specified by the *RouteInfo* or *DestAddress* parameters is invalid.
Other	The call failed. Use **FormatMessage** to retrieve the error message corresponding to the returned error code.

Requirements

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

See Also

Protocol Identifiers

IPX Service Table Management

An IPX routing protocol that registers for service handling should maintain a service bindery table as defined by Novell's IPX Service Advertising Protocol (SAP) specifications. The routing protocol should also provide the router manager access to this table through the following functions. (See the ***RegisterProtocol*** function for more information on how a routing protocol registers for service handling.)

Service Table Management Functions

Implement the following functions for routing protocols that register for service handling:

BlockConvertServicesToStatic
BlockDeleteStaticServices
CloseServiceEnumerationHandle
CreateServiceEnumerationHandle
CreateStaticService
DeleteStaticService
EnumerateGetNextService
GetFirstOrderedService
GetNextOrderedService
GetServiceCount
IsService

BlockConvertServicesToStatic

The **BlockConvertServicesToStatic** function converts all services received on a specified interface to static.

```
DWORD BlockConvertServicesToStatic (
  IN ULONG InterfaceIndex
  );
```

Parameters

InterfaceIndex
A unique number that identifies the interface associated with the services intended for conversion.

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	Description
ERROR_CAN_NOT_COMPLETE	The SAP Agent is down.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.

See Also

IPX Service Table Management, Service Table Management Functions, **BlockDeleteStaticServices**

BlockDeleteStaticServices

The **BlockDeleteStaticServices** function deletes all static services associated with a specified interface.

```
DWORD BlockDeleteStaticServices (
  IN ULONG InterfaceIndex
  );
```

Parameters

InterfaceIndex
A unique number that identifies the interface associated with the services to be deleted.

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	Description
ERROR_CAN_NOT_COMPLETE	The SAP Agent is down.
ERROR_INVALID_PARAMETER	The *InterfaceIndex* parameter is invalid.

! Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.

+ See Also

IPX Service Table Management, Service Table Management Functions, **BlockConvertServicesToStatic**

CloseServiceEnumerationHandle

The **CloseServiceEnumerationHandle** function terminates the enumeration and frees associated resources.

```
DWORD CloseServiceEnumerationHandle (
  IN HANDLE EnumerationHandle
  );
```

Parameters

EnumerationHandle
Handle that identifies the enumeration to terminate, obtained from a previous call to **CreateServiceEnumerationHandle**.

Return Values

If the functions succeeds, the return value is NO_ERROR.

If the function fails, the return value is ERROR_CAN_NOT_COMPLETE.

! Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.

+ See Also

IPX Service Table Management, Service Table Management Functions, **CreateServiceEnumerationHandle**

CreateServiceEnumerationHandle

The **CreateServiceEnumerationHandle** function returns a handle that allows the use of fast and change-tolerant enumeration functions. Such functions can scan through all services or a specified subset.

```
HANDLE CreateServiceEnumerationHandle (
  IN DWORD ExclusionFlags,
  IN PIPX_SERVICE CriteriaService
  );
```

Parameters

ExclusionFlags

Limits the set of services that **CreateServiceEnumerationHandle** returns to a subset defined by a combination of *ExclusionFlags* and values in the corresponding members of *CriteriaService*. This parameter must be one of the following values.

Value	Defintion
STM_ONLY_THIS_INTERFACE	Enumerate only those services that were obtained through the interface specified in the **InterfaceIndex** member of *CriteriaService.*
STM_ONLY_THIS_PROTOCOL	Enumerate only those services that were obtained through the protocol specified in the **Protocol** member of *CriteriaService* (for example, **IPX_PROTOCOL_SAP** for services obtained by the DLL protocol or **IPX_PROTOCOL_STATIC** for services maintained by the router manager).
STM_ONLY_THIS_TYPE	Enumerate only those services that have the same type as those in the **Service** member of *CriteriaService.*

CriteriaService

Pointer to an **IPX_SERVICE** structure with member values that correspond to those specified in *ExclusionFlags.*

Return Values

If the function succeeds, the return value is a handle for use with the service enumeration function.

A NULL handle indicates no services exists with the specified criteria, or that the operation failed. For more information, call **GetLastError** and check the error code against the table below.

Value	Description
ERROR_NO_SERVICES	No services exist with the specified criteria.
ERROR_INVALID_PARAMETER	One or more of the input parameters is invalid. For example, invalid enumeration flags or invalid members in *CriteriaService*.

! Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.

+ See Also

IPX Service Table Management, Service Table Management Functions, **CloseServiceEnumerationHandle**, **EnumerateGetNextService**, **GetLastError**, **IPX_SERVICE**

CreateStaticService

The **CreateStaticService** function adds a static service to the table.

```
DWORD CreateStaticService (
  IN ULONG InterfaceIndex,
  IN PIPX_STATIC_SERVICE_INFO ServiceEntry
  );
```

Parameters

InterfaceIndex

A unique number that identifies the interface associated with the new service.

ServiceEntry

Pointer to an **IPX_STATIC_SERVICE_INFO** structure containing parameters of the static service to be added.

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	Description
ERROR_CAN_NOT_COMPLETE	The SAP Agent is down.
ERROR_INVALID_PARAMETER	One or more of the input parameters is invalid, for example, invalid interface index, or invalid fields in *ServiceEntry*.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.

See Also

IPX Service Table Management, Service Table Management Functions, **DeleteStaticService**, **IPX_STATIC_SERVICE_INFO**

DeleteStaticService

The **DeleteStaticService** function deletes a static service from the table.

```
DWORD DeleteStaticService (
  IN ULONG InterfaceIndex,
  IN PIPX_STATIC_SERVICE_INFO ServiceEntry
  );
```

Parameters

InterfaceIndex
A unique number that identifies the interface associated with the service intended for deletion.

ServiceEntry
Pointer to an **IPX_STATIC_SERVICE_INFO** structure containing the parameters of the static service intended for deletion.

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	Description
ERROR_CAN_NOT_COMPLETE	The SAP Agent is down.
ERROR_INVALID_PARAMETER	One or more of the input parameters is invalid; for example,invalid interface index, or invalid fields in *ServiceEntry*.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.

See Also

IPX Service Table Management, Service Table Management Functions, **CreateStaticService**, **IPX_STATIC_SERVICE_INFO**

EnumerateGetNextService

The **EnumerateGetNextService** function returns the next service entry in an enumeration started by **CreateServiceEnumerationHandle**.

```
DWORD EnumerateGetNextService (
  IN HANDLE EnumerationHandle,
  OUT PIPX_SERVICE Service
  );
```

Parameters

EnumerationHandle

Handle that identifies the enumeration and specifies the subset of services on which the enumeration will operate. The handle is obtained from a call to **CreateServiceEnumerationHandle**.

Service

Pointer to an **IPX_SERVICE** structure that will contain the next service in the enumeration. Although services are returned in no particular order, each service in the subset is returned only once.

Return Values

If the function succeeds, the buffer pointed to by the *Service* parameter receives the next service in the enumeration. In this case the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes.

Value	Description
ERROR_NO_MORE_ITEMS	No more services exist with the specified criteria.
ERROR_CAN_NOT_COMPLETE	The operation failed.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.

See Also

IPX Service Table Management, Service Table Management Functions, **CreateServiceEnumerationHandle**, **IPX_SERVICE**

GetFirstOrderedService

The **GetFirstOrderedService** function returns the first service in the specified order from the designated subset of services in the table.

```
DWORD GetFirstOrderedService (
  IN DWORD OrderingMethod,
  IN DWORD ExclusionFlags,
  IN OUT PIPX_SERVICE Service
  );
```

Parameters

OrderingMethod
Indicates the order in which the services are searched. This parameter must be one of the following values.

STM_ORDER_BY_TYPE_AND_NAME
Search the services in type.name order.

STM_ORDER_BY_INTERFACE_TYPE_NAME
Search the services in interface index.type.name order.

ExclusionFlags
Limits the set of examined services to a subset defined by *ExclusionFlags* and the values in the members of the structure pointed to by the *Service* parameter. See ***CreateServiceEnumerationHandle*** for a description of the possible flags.

Service
Pointer to an **IPX_SERVICE** structure.

Value of *Service* at Input:
Values in the members correspond to flags specified in *ExclusionFlags*.

Value of *Service* upon Output:
The first service that matches specified criteria.

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	Description
ERROR_NO_MORE_ITEMS	Services that match the specified criteria do not exist.
ERROR_INVALID_PARAMETER	One or more input parameters are invalid, that is, invalid ordering method, enumeration flags, or field values in *Service*.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.

See Also

IPX Service Table Management, Service Table Management Functions, **CreateServiceEnumerationHandle**, **IPX_SERVICE**

GetNextOrderedService

The **GetNextOrderedService** function returns the next service from a subset of services in the table. The service returned is the next service after a given input service using the ordering method specified.

```
DWORD GetNextOrderedService (
  IN DWORD OrderingMethod,
  IN DWORD ExclusionFlags,
  IN OUT PIPX_SERVICE Service
  );
```

Parameters

OrderingMethod
Indicates the order in which the services are searched. See ***GetFirstOrderedService*** for a description of the various ordering methods.

ExclusionFlags
Limits the set of examined services to a subset defined by *ExclusionFlags* and the values in the corresponding members of the structure pointed to by the *Service* parameter. See ***CreateServiceEnumerationHandle*** for a description of the possible flags.

Service
Pointer to an **IPX_SERVICE** structure.

Value of *Service* at Input:
Contains the service from which to continue searching; also contains member values that correspond to the specified *ExclusionFlags*.

Value of *Service* upon Output:
The structure contains the first service that follows the input service and matches the specified criteria.

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	Description
ERROR_NO_MORE_SERVICES	There are no more services matching the specified criteria.
ERROR_INVALID_PARAMETER	One or more of the input parameters is invalid; for example, invalid ordering method, enumeration flags, or member values in *Service*.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.

See Also

IPX Service Table Management, Service Table Management Functions, **CreateServiceEnumerationHandle**, **IPX_SERVICE**

GetServiceCount

The **GetServiceCount** function returns the number of services in the table.

```
ULONG GetServiceCount (void);
```

Return Values

If the function succeeds, the return value is the number of services in the table.

If the function fails, the return value is one of the following error codes.

Value	Description
NO_ERROR	Operation succeeded but no services are available.
0 (Zero)	No services are available in the table or the operation failed. Call **GetLastError** to obtain more information.

Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.

See Also

IPX Service Table Management, Service Table Management Functions, **GetLastError**

IsService

The **IsService** function checks whether a service of specified type and name exists in the service table, and optionally returns the service's parameters.

```
BOOL IsService (
  IN USHORT Type,
  IN PUCHAR Name,
  OUT PIPX_SERVICE Service    //OPTIONAL
  );
```

Parameters

Type
Specifies the type of the service being checked.

Name
Specifies the name of the service being checked.

Service
Receives a pointer to a structure in which to place the information about the matching service (if any).

Return Values

The **IsService** function returns one of the following values.

Value	Description
TRUE	The service exists in the table.
FALSE	No such service exists, or the operation failed. Call **GetLastError** for more information about the failure.
NO_ERROR	The operation succeeded, but no such service exists.
ERROR_INVALID_PARAMETER	The service type or name is invalid.

! Requirements

Windows NT/2000: Requires Windows 2000. Available as a redistributable for Windows NT 4.0.

+ See Also

IPX Service Table Management, Service Table Management Functions, **GetLastError**, **IPX_SERVICE**

Service Table Management Structures

Use the following structures to implement routing protocols that register for service handling:

IPX_SERVER_ENTRY
IPX_SERVICE

IPX_SERVER_ENTRY

The **IPX_SERVER_ENTRY** structure describes a particular IPX service.

```
typedef struct _IPX_SERVER_ENTRY {
  USHORT    Type;
  UCHAR     Name[48];
  UCHAR     Network[4];
  UCHAR     Node[6];
  UCHAR     Socket[2];
  USHORT    HopCount;
  } IPX_SERVER_ENTRY, * PIPX_SERVER_ENTRY;
```

Members

Type
Contains the service type as defined by the SAP specification.

Name[48]
Contains the service name as defined by SAP specifications.

Network[4]
Contains the network number portion of the service address.

Node[6]
Contains the node number portion of the service address.

Socket[2]
Contains the socket number portion of service address.

HopCount
Contains the service hop count.

Requirements

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

See Also

IPX Service Table Management, Service Table Management Structures, **IPX_SERVICE**

IPX_SERVICE

The **IPX_SERVICE** structure contains information about an IPX service, and identifies the interface and protocol through which this information was obtained.

```
typedef struct _IPX_SERVICE {
  ULONG                 InterfaceIndex; // index of interface
  ULONG                 Protocol;
  IPX_SERVER_ENTRY      Service;
} IPX_SERVICE, *PIPX_SERVICE;
```

Parameters

InterfaceIndex
Contains the index of the interface through which the service information was obtained.

Protocol
Contains the identifier of the protocol that obtained the service information. (Static services are viewed as services obtained through the **IPX_PROTOCOL_STATIC** protocol.)

Service
Specifies an **IPX_SERVER_ENTRY** structure.

! Requirements

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

+ See Also

IPX Service Table Management, Service Table Management Structures, **IPX_SERVER_ENTRY**

CHAPTER 10

Routing Table Manager Version 1

Routing Table Manager Version 1 Overview

The routing table manager is a central repository of routing information for all routing protocols that operate under Routing and Remote Access Service (RRAS). The routing table manager provides routing information to all interested components, such as routing protocols, management agents, and monitoring agents. The routing table manager also determines the best route to each destination network known to the routing protocols. It determines this route based on routing protocol priorities and on metrics associated with the routes. Note that the administrator is able to configure routing protocol priorities. The routing table manager then passes the best-route information on to the forwarders and back to the routing protocols.

Each routing protocol calls **RtmRegisterClient** to register with the routing table manager. **RtmRegisterClient** returns a handle that is used by the routing protocol to add or delete route entries. **RtmRegisterClient** also allows the routing protocol to register an event object with the routing table manager. The routing table manager signals this event object to notify the routing protocol of changes in best-route information. All other components can obtain information stored in the routing table manager through route enumeration.

Route Tables and Route Table Entries

The routing table manager maintains distinct route tables for each protocol family. Currently explicit support is provided for the Internet Protocol (IP) and Internet Packet Exchange (IPX) routing protocol families. Regardless of the protocol family, each route entry contains the following information:

- Destination network.
- Identifier of the protocol that added the route.
- Index of interface through which the route was obtained.
- Address of the next hop router. RRAS uses this router to forward packets to the destination network if the network is not directly connected.
- The time the route was created or last updated.
- The amount of time this route should be kept in the routing table. If this amount of time elapses, and the route has not been updated, the routing table manager removes the route from the table (in this case, the route is said to have “aged out”).

- Data specific to the protocol family. This data is transparent to RTMv1. However, if this data changes for a route that is designated as a "best route," the routing table manager sends out route-change notification.
- Data specific to the routing protocol. This data is completely transparent to the routing table manager in that changes to this data do not cause route change notification.

The following values taken together uniquely identify a route in the routing table:

- Destination network
- Protocol identifier
- Interface index
- Address of next-hop router

In general, the routing table manager creates separate entries for routes that differ in any of these parameter values. However, an exception is made for routing protocols that do not keep more that one entry for each destination network. For these protocols, the routing table manager ignores differences in interface index or next-hop address. An example of such a protocol would be the RRAS implementation of Open Shortest Path First (OSPF).

Changes to the Best Route to a Network

A change in any of the following values for the best route to a given destination network, causes the routing table manager to generate a notification message that is sent to each registered client and to the forwarders:

- Protocol identifier
- Interface index
- Address of next-hop router
- Protocol-family specific data that includes route metrics

A change in protocol identifier, interface index, or next-hop router address can occur when a new, better-route entry is added, or when the current best-route entry is deleted or aged out, leaving another route as the new best route.

Routing Table Manager Version 1 Reference

The following functions, structures, and constants provide an interface that routing protocols can use to access the routing tables maintained by the routing table manager.

Routing Table Manager Version 1 Functions

Use the following functions to access the routing tables maintained by the routing table manager.

RtmRegisterClient
RtmDeregisterClient
RtmDequeueRouteChangeMessage
RtmAddRoute
RtmDeleteRoute
RtmIsRoute
RtmGetNetworkCount
RtmGetRouteAge
RtmCreateEnumerationHandle
RtmEnumerateGetNextRoute
RtmCloseEnumerationHandle
RtmBlockDeleteRoutes
RtmGetFirstRoute
RtmGetNextRoute

RtmRegisterClient

The **RtmRegisterClient** function registers a client as a handler of the specified protocol. It establishes a route change notification mechanism for the client, and sets protocol options.

```
HANDLE RtmRegisterClient(
  DWORD ProtocolFamily,    // identifier of protocol family
  DWORD RoutingProtocol,   // identifier of routing protocol
  HANDLE ChangeEvent,      // event to signal when best
                           // routes change
  DWORD Flags              // flags to indicate special
                           // handling of routing protocol
);
```

Parameters

ProtocolFamily
Specifies the protocol family of the routing protocol to register.

RoutingProtocol
Specifies the routing protocol identifier, the same as that used when registering with the router manager (see ***RegisterProtocol***).

ChangeEvent
Specifies that a best route to a network in the table has changed. The routing table manager signals this event after a change to the best route to any network in the table. See ***RtmDequeueRouteChangeMessage*** for more information about route-change notification.

This parameter is optional. If the caller specifies NULL for this parameter, the routing table manager does not notify the client of changes in best route status.

Flags
Miscellaneous options for special handling of the routing protocol. The following value is currently supported.

Flags	Values
RTM_PROTOCOL_SINGLE_ROUTE	The routing table manager keeps only one route per destination network for the routing protocol. In other words, the routing table manager replaces route entries that have the same destination network numbers instead of adding new ones.

Return Values

On successful return, a **HANDLE** value that identifies the client in subsequent calls to the routing table manager.

A NULL handle indicates that the routing table manager was unable to register the client. Call **GetLastError** to obtain the reason for the failure.

Value	Description
ERROR_CLIENT_ALREADY_EXISTS	Another client has already registered to handle the specified protocol.
ERROR_INVALID_PARAMETER	The specified protocol family is not supported, or the *Flags* parameter is invalid.
ERROR_NO_SYSTEM_RESOURCES	Insufficient resources to carry out the operation.
ERROR_NOT_ENOUGH_MEMORY	Insufficient memory to allocate data structures for the client.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **GetLastError**, **RegisterProtocol**, RTMv1 Protocol Family Identifiers, **RtmDequeueRouteChangeMessage**, **RtmDeregisterClient**

RtmDeregisterClient

The **RtmDeregisterClient** function deregisters the client, and frees resources associated with the client.

```
DWORD RtmDeregisterClient(
  HANDLE ClientHandle   // handle that identifies the client
);
```

Parameters

ClientHandle

A handle that identifies the client to deregister. Obtain this handle by calling **RtmRegisterClient**.

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	Description
ERROR_INVALID_HANDLE	The *ClientHandle* parameter is not a valid handle.
ERROR_NO_SYSTEM_RESOURCES	Insufficient resources to carry out the operation.

Remarks

This function removes all routes that were added by the client.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **RtmRegisterClient**

RtmDequeueRouteChangeMessage

The **RtmDequeueRouteChangeMessage** function returns the next route-change message in the queue associated with the specified client.

```
DWORD RtmDequeueRouteChangeMessage(
  HANDLE ClientHandle,  // handle that identifies the client
  DWORD Flags,          // type of change message
  PVOID CurBestRoute,   // the new best route
  PVOID PrevBestRoute   // the previous best route
);
```

Parameters

ClientHandle

Handle that identifies the client for which the operation is performed. Obtain this handle by calling **RtmRegisterClient**.

Flags

Pointer to a **DWORD** variable. The value of this variable is set by the routing table manager. The value indicates the type of the change message, and what information was returned in the provided buffers. This parameter is one of the following.

Flags	Values
RTM_ROUTE_ADDED	The first route was added for a particular destination network. The *CurBestRoute* parameter points to the information for the added route.
RTM_ROUTE_DELETED	The only route available for a particular destination network was deleted. The *PrevBestRoute* parameter points to the information for the deleted route.
RTM_ROUTE_CHANGED	At least one of the significant parameters was changed for a best route to a particular destination network. The significant parameters are: Protocol identifier Interface index Next-hop address Protocol-family-specific data (including route metrics)

The *PrevBestRoute* parameter points to the route information as it was before the change. The *CurBestRoute* parameter points to current (that is, after-change) route information.

CurBestRoute

Pointer to a structure to receive the current best-route information (if any). The type of the structure is specific to the protocol family (for example, IP or IPX).

This parameter is optional. If the caller specifies NULL for this parameter, the current best-route information is not returned.

PrevBestRoute

Pointer to a structure to receive the previous best-route information, if any. The type of the structure is specific to the protocol family (for example, IP or IPX).

This parameter is optional. If the caller specifies NULL for this parameter, the previous best-route information is not returned.

Return Values

The return value is one of the following codes.

Value	Description
NO_ERROR	This message was the last message in the client's queue. The event object is reset.
ERROR_INVALID_HANDLE	The *ClientHandle* parameter is not a valid handle, or at registration the client did not provide an event object for change message notification (see ***RtmRegisterClient***).
ERROR_MORE_MESSAGES	The client's queue contains additional messages. The client should call **RtmDequeueRouteChangeMessage** again as soon as possible to allow the routing table manager to free the resources associated with the pending messages.
ERROR_NO_MESSAGES	The client's queue contains no messages; the call was unsolicited. The event is reset.
ERROR_NO_SYSTEM_RESOURCES	There are insufficient resources to carry out the operation.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **RtmRegisterClient**

RtmAddRoute

The **RtmAddRoute** function adds a route entry, or updates an existing route entry.

```
DWORD RtmAddRoute(
  HANDLE ClientHandle,  // handle that identifies the client
  PVOID Route,          // pointer to new route structure
  DWORD TimeToLive,     // how long to keep the route
  DWORD Flags,          // best route change status
  PVOID CurBestRoute,   // pointer to new best route
  PVOID PrevBestRoute   // pointer to previous best route
);
```

Parameters

ClientHandle

Handle that identifies the client, and therefore the routing protocol, that added or updated the route. Obtain this handle by calling **RtmRegisterClient**.

Route

Pointer to a protocol-family-specific structure that contains the new or updated route. The following fields are used by the routing table manager to update the routing table:

RR_Network

Specifies the destination network number.

RR_InterfaceID

Specifies the index of the interface through which the route was received.

RR_NextHopAddress

Specifies the address of the next-hop router.

RR_FamilySpecificData

Specifies data that is specific to the protocol family. Although the data is transparent to the routing table manager, it is considered when comparing routes to determine if route information has changed. The data is also used to set metric values that are independent of the routing protocol. Consequently, this data is used to determine the best route for the destination network.

RR_ProtocolSpecificData

Specifies data which is specific to the routing protocol that supplied the route.

RR_TimeStamp

Specifies the current system time. This field is set by the routing table manager.

TimeToLive

Specifies the number of seconds the specified route should be kept in the routing table. If this parameter is set to INFINITE, the route is kept until it is explicitly deleted. The current limit for *TimeToLive* is 2147483 sec (24+ days).

Flags

Pointer to a **DWORD** variable. The value of this variable is set by the routing table manager. The value indicates the type of the change, and what information was returned in the provided buffers. This parameter is one of the following.

Flags	**Values**
RTM_NO_CHANGE	The addition or update either did not change any of the significant route parameters, or the route entry affected is not the best route among the entries for the destination network.
RTM_ROUTE_ADDED	The route was added for the destination network. The *CurBestRoute* parameter points to the information for the added route.

Flags	Values
RTM_ROUTE_CHANGED	At least one of the significant parameters was changed for the best route to the destination network. The significant parameters are: Protocol identifier Interface index Next-hop address Protocol-family-specific data (including route metrics)

The *PrevBestRoute* parameter points to the route information as it was before the change. The *CurBestRoute* parameter points to the current (that is, after-change) route information.

CurBestRoute

Pointer to a structure to receive the current best-route information, if any. The type of the structure is specific to the protocol family (for example, IP or IPX).

This parameter is optional. If the caller specifies NULL for this parameter, the current best-route information is not returned.

PrevBestRoute

Pointer to a structure to receive the previous best-route information, if any. The type of the structure is specific to the protocol family (for example, IP or IPX).

This parameter is optional. If the caller specifies NULL for this parameter the previous best-route information is not returned.

Return Value

The return value is one of the following codes.

Value	Description
NO_ERROR	The route was added or updated successfully.
ERROR_INVALID_HANDLE	The client handle parameter is not a valid handle.
ERROR_INVALID_PARAMETER	The route structure contains an invalid parameter.
ERROR_NO_SYSTEM_RESOURCES	There are insufficient resources to carry out the operation.
ERROR_NOT_ENOUGH_MEMORY	There is insufficient memory to allocate the route entry.

Remarks

The function generates a route-change message if the best route to a destination network has changed as the result of this operation. However, the route-change message is not sent to the client that makes this call. Instead, relevant information is returned by this function directly to that client through the *Flags*, *CurBestRoute*, and *PrevBestRoute* parameters.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **RtmDeleteRoute**, **RtmDequeueRouteChangeMessage**

RtmDeleteRoute

The **RtmDeleteRoute** function deletes a route entry.

```
DWORD RtmDeleteRoute(
  HANDLE ClientHandle,     // handle that identifies client
  PVOID Route,             // pointer to route structure
  DWORD Flags,             // best-route change status
  PVOID CurBestRoute       // pointer to new best route
);
```

Parameters

ClientHandle
Handle that identifies the client and therefore the routing protocol of the added or updated route. Obtain this handle by calling **RtmRegisterClient**.

Route
Pointer to a protocol-family-specific structure containing the new or updated route. The following fields will be used by the routing table manager to update the routing table:

RR_Network
Specifies the destination network number.

RR_InterfaceID
Specifies the index of the interface through which the route was received.

RR_NextHopAddress
Specifies the network address of the next-hop router.

Flags

A pointer to a set of flags that indicate the type of the change message, and what information was placed in the provided buffers. This parameter is one of the following values.

Flags	Values
RTM_NO_CHANGE	Deleting the route did not affect the best route to any destination network. In other words, another entry represents a route to the same destination network and has a lower metric.
RTM_ROUTE_DELETED	The route deleted was the only route available for a particular destination network.
RTM_ROUTE_CHANGED	After this route was deleted, another route became the best route to a particular destination network. *CurBestRoute* points to the information for the new best route.

CurBestRoute

Pointer to a structure to receive the current best-route information, if any. The type of the structure is specific to the protocol family (for example, IP or IPX).

This parameter is optional. If the caller specifies NULL for this parameter, the current best-route information is not returned.

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	Description
ERROR_INVALID_HANDLE	The client handle parameter is not a valid handle.
ERROR_INVALID_PARAMETER	The route structure pointed to by the *Route* parameter contains a member value.
ERROR_NO_SUCH_ROUTE	There are no entries in the routing table that match the parameters of the specified route.
ERROR_NO_SYSTEM_RESOURCES	There are insufficient resources to perform the operation.

Remarks

The function generates a route-change message if the best route to a destination network has changed as the result of the deletion. However, the route-change message is not sent to the client that makes this call. Instead, relevant information is returned by this function directly to that client.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **RtmAddRoute**, **RtmDequeueRouteChangeMessage**

RtmIsRoute

The **RtmIsRoute** function determines if one or more routes to a specified destination network exist. If so, the function returns information for the best route to that network.

```
BOOL RtmIsRoute(
  DWORD ProtocolFamily,    // specifies type of network
  PVOID Network,           // specifies the network
  PVOID BestRoute          // receives best route for the
                           // network
);
```

Parameters

ProtocolFamily
Specifies the type of data structure pointed to by the *Network* parameter (for example, **IP_NETWORK**, **IPX_NETWORK**).

Network
Pointer to a structure that contains protocol-family-specific network number data. This data identifies the network for which the caller seeks route information.

BestRoute
Pointer to a protocol-family-specific structure to receive the current best route information, if any.

Return Values

The return value is one of the following codes.

Value	Description
TRUE	At least one route to the specified network exists. The best route is returned in the structure pointed to by the *BestRoute* parameter.

Value	Description
FALSE	There is no route to the specified network, or the operation failed. Call **GetLastError** to obtain more information:
NO_ERROR	The operation succeeded, but there is no route to the specified network.
ERROR_INVALID_PARAMETER	The value of the *ProtocolFamily* parameter does not correspond to any installed protocol family.
ERROR_NO_SYSTEM_RESOURCES	There are insufficient resources to carry out the operation.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **GetLastError**, **IP_NETWORK**, **IPX_NETWORK**, RTMv1 Protocol Family Identifiers

RtmGetNetworkCount

The **RtmGetNetworkCount** function retrieves the number of networks to which the routing table manager has routes.

```
ULONG RtmGetNetworkCount(
  DWORD ProtocolFamily    // type of network (IP or IPX)
);
```

Parameters

ProtocolFamily
Specifies for which type of network (for example, IP or IPX) to obtain route information.

Return Values

If the function succeeds, the return value is the network count, the number of networks known to the routing protocols of the specified protocol family.

If the return value is zero, either no routes are available, or the operation failed. Call **GetLastError** to obtain more information.

Value	Description
NO_ERROR	The operation succeeded, but no routes are available.
ERROR_INVALID_PARAMETER	The value of the *ProtocolFamily* parameter does not correspond to any installed protocol family.

! Requirements

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

+ See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **GetLastError**, RTMv1 Protocol Family Identifiers

RtmGetRouteAge

The **RtmGetRouteAge** function returns the age of a route. The age is the time, in seconds, since it was created or last updated.

```
ULONG RtmGetRouteAge(
  PVOID Route    // pointer to route structure
);
```

Parameters

Route
Pointer to a protocol-family-specific structure that contains route data recently obtained from the routing table manager.

Return Values

The return value is one of the following values.

Value	Description
RouteAge	The time in seconds since a route was created or last updated.
INFINITE	The content of the route structure is invalid. In this case, a call to **GetLastError** returns **ERROR_INVALID_PARAMETER**.

Remarks

The route age is computed from the **RR_TimeStamp** member of the structure that is pointed to by the *Route* parameter. The routing table manager sets the value of this member when a route is added or updated.

Requirements

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

See Also

Routing Table Manager Version_1 Reference, Routing Table Manager Version 1 Functions, **GetLastError**, **RTM_IP_ROUTE**, **RTM_IPX_ROUTE**

RtmCreateEnumerationHandle

The **RtmCreateEnumerationHandle** function returns a handle to use with **RtmEnumerateGetNextRoute** to scan through all routes, or a subset of routes, known to the routing table manager.

```
HANDLE RtmCreateEnumerationHandle(
  DWORD ProtocolFamily,      // type of network (IP or IPX)
  DWORD EnumerationFlags,    // flags that specify type of
                             // criteria
  PVOID CriteriaRoute        // structure that holds
                             // criteria values
);
```

Parameters

ProtocolFamily
Specifies the protocol family of the routes to enumerate.

EnumerationFlags
Specifies which routes should be enumerated. This parameter limits the set of routes returned by the enumeration API to a subset defined by the following flags and the values in the corresponding members of the structure pointed to by the *CriteriaRoute* parameter. This parameter can be one of the following values.

EnumerationFlags	Values
RTM_ONLY_THIS_NETWORK	Enumerate only those routes that have the same network number as the RR_Network member of the structure pointed to by CriteriaRoute.

(continued)

(continued)

EnumerationFlags	Values
RTM_ONLY_THIS_INTERFACE	Enumerate only those routes that were obtained through the interface specified by the RR_InterfaceID member of the structure pointed to by CriteriaRoute.
RTM_ONLY_THIS_PROTOCOL	Enumerate only those routes that were added by the protocol handler specified by the RR_RoutingProtocol field of the structure pointed to by CriteriaRoute.
RTM_ONLY_BEST_ROUTES	Enumerate only the best routes to each of the networks in the set.

CriteriaRoute

Pointer to a protocol-family-specific route structure (**RTM_IP_ROUTE** or **RTM_IPX_ROUTE**). The member values in this structure correspond to the flags specified by the *EnumerationFlags* parameter.

Return Values

If the function succeeds, the return value is a HANDLE to use with subsequent enumeration calls.

If the function fails, or no routes exist with the specified criteria, the return value is NULL. Call **GetLastError** to obtain more information.

Value	Description
ERROR_NO_ROUTES	There are no routes that have the specified criteria.
ERROR_INVALID_PARAMETER	One or more of the input parameters is invalid (for example, unknown protocol family, invalid enumeration flags).
ERROR_NO_SYSTEM_RESOURCES	There are insufficient resources to carry out the operation.
ERROR_NOT_ENOUGH_MEMORY	There is insufficient memory to allocate the handle.

! Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **GetLastError**, **RTM_IP_ROUTE**, **RTM_IPX_ROUTE**, **RtmCloseEnumerationHandle**, **RtmEnumerateGetNextRoute**

RtmEnumerateGetNextRoute

The **RtmEnumerateGetNextRoute** function returns the next-route entry in the enumeration started by a call to **RtmCreateEnumerationHandle**.

```
DWORD RtmEnumerateGetNextRoute(
  HANDLE EnumerationHandle,    // handle that identifies
                               // enumeration
  PVOID Route                  // structure to receive next
                               // route
);
```

Parameters

EnumerationHandle

Handle that identifies the enumeration and specifies its scope. Obtain this handle by calling **RtmCreateEnumerationHandle**.

Route

Pointer to a protocol-family-specific route structure (**RTM_IP_ROUTE** or **RTM_IPX_ROUTE**). This structure will receive the next route in the enumeration.

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	Description
ERROR_INVALID_HANDLE	The *EnumerationHandle* parameter is not valid.
ERROR_NO_MORE_ROUTES	There are no more routes in the enumeration.
ERROR_NO_SYSTEM_RESOURCES	There are insufficient resources to carry out the operation.

Remarks

Although routes are not returned in any particular order, each route in the enumeration is returned only once.

! Requirements

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

+ See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **RTM_IP_ROUTE**, **RTM_IPX_ROUTE**, **RtmCloseEnumerationHandle**, **RtmCreateEnumerationHandle**

RtmCloseEnumerationHandle

The **RtmCloseEnumerationHandle** terminates a specified enumeration and frees the associated resources.

```
DWORD RtmCloseEnumerationHandle(
  HANDLE EnumerationHandle    // handle that identifies
                              // enumeration
);
```

Parameters

EnumerationHandle
Handle to the enumeration to terminate. Obtain this handle by calling **RtmCreateEnumerationHandle**.

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	Description
ERROR_INVALID_HANDLE	The *EnumerationHandle* parameter is not valid.
ERROR_NO_SYSTEM_RESOURCES	There are insufficient resources to carry out the operation.

! Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **RtmCreateEnumerationHandle**, **RtmEnumerateGetNextRoute**

RtmBlockDeleteRoutes

The **RtmBlockDeleteRoutes** function deletes all routes in the specified subset of routes in the table.

```
HANDLE RtmBlockDeleteRoutes(
  HANDLE ClientHandle,       // handle that identifies client
  DWORD EnumerationFlags,    // flags that specify type of
                             // criteria
  PVOID CriteriaRoute        // structure that hold criteria
                             // values
);
```

Parameters

ClientHandle
Handle that identifies the client, and therefore the routing protocol, of the routes to be deleted.

EnumerationFlags
Specifies which routes should be enumerated. This parameter limits the set of deleted routes to a subset defined by the following flags and the values in the corresponding members of the structure pointed to by the *CriteriaRoute* parameter. The flags are the same as those used in **RtmCreateEnumerationHandle** except that **RTM_ONLY_BEST_ROUTES** is redundant for **RtmBlockDeleteRoutes**. The best-route designation is adjusted as routes are deleted, so the function eventually deletes all the routes in the subset.

CriteriaRoute
Pointer to a protocol-family-specific route structure (**RTM_IP_ROUTE** or **RTM_IPX_ROUTE**). The member values in this structure correspond to the flags specified by the *EnumerationFlags* parameter.

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	Description
ERROR_NO_ROUTES	There are no routes that have the specified criteria.

(continued)

(continued)

Value	Description
ERROR_INVALID_HANDLE	The *ClientHandle* parameter is not valid.
ERROR_INVALID_PARAMETER	One or more of the input parameters is invalid (for example, the enumeration flags are invalid).
ERROR_NO_SYSTEM_RESOURCES	There are insufficient resources to carry out the operation.
ERROR_NOT_ENOUGH_MEMORY	There is insufficient memory to carry out the operation.

Remarks

The function generates appropriate notification messages to all registered clients including the caller.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **RtmCreateEnumerationHandle**, **RtmRegisterClient**

RtmGetFirstRoute

The **RtmGetFirstRoute** function returns the first route from the specified subset of routes in the table.

```
DWORD RtmGetFirstRoute(
  DWORD ProtocolFamily,      // type of network (IP or IPX)
  DWORD EnumerationFlags,    // flags that specify type of
                             // criteria
  PVOID Route                // structure for criteria
                             // values and returned route
);
```

Parameters

ProtocolFamily
Identifies the protocol family (for example, IP or IPX) of routes to retrieve.

EnumerationFlags

Limits the set of deleted routes to a subset defined by these flags and the values in the corresponding members of the structure pointed to by the *CriteriaRoute* parameter. The flags are the same as those used in **RtmCreateEnumerationHandle.**

Route

Pointer to a protocol-family-specific structure (**RTM_IP_ROUTE** or **RTM_IPX_ROUTE**).

The calling function provides member values for this structure. These values, in conjunction with the *EnumerationFlags* parameter, specify the set from which to return routes.

On successful return, *Route* points to the first route that matched the specified criteria.

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	Description
ERROR_INVALID_PARAMETER	One or more of the input parameters is invalid (for example, the protocol family is unknown, or the enumeration flags are invalid).
ERROR_NO_ROUTES	There are no routes that match the specified criteria.
ERROR_NO_SYSTEM_RESOURCES	There are insufficient resources to carry out the operation.

Remarks

The routes are returned in the following order:

1. Network number
2. Routing protocol
3. Interface identifier
4. Next-hop address

This function is less efficient than the corresponding enumeration handle function (**RtmEnumerateGetNextRoute**).

! Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **RtmCloseEnumerationHandle**, **RtmCreateEnumerationHandle**, **RtmEnumerateGetNextRoute**, **RtmGetNextRoute**

RtmGetNextRoute

The **RtmGetNextRoute** function returns the next route from the specified subset of routes in the table.

```
DWORD RtmGetNextRoute(
  DWORD ProtocolFamily,      // type of network (IP or IPX)
  DWORD EnumerationFlags,    // flags that specify type of
                             // criteria
  PVOID Route                // structure for criteria
                             // values and returned route
);
```

Parameters

ProtocolFamily

Specifies the protocol family (for example, IP or IPX) of routes to retrieve.

EnumerationFlags

Specifies which routes should be enumerated. This parameter limits the set of deleted routes to a subset defined by the following flags and the values in the corresponding members of the structure pointed to by the *CriteriaRoute* parameter. The flags are the same as those used in **RtmCreateEnumerationHandle**.

Route

Pointer to a protocol-family-specific structure (**RTM_IP_ROUTE** or **RTM_IPX_ROUTE**).

The calling function provides member values for this structure. These values, in conjunction with the *EnumerationFlags* parameter, specify the set from which to return routes.

On successful return, this structure receives the first route that matched the specified criteria.

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	Description
ERROR_INVALID_PARAMETER	One or more of the input parameters is invalid (for example, the protocol family is unknown, or the enumeration flags are invalid).
ERROR_NO_ROUTES	There are no routes that match the specified criteria.
ERROR_NO_SYSTEM_RESOURCES	There are insufficient resources to carry out the operation.

Remarks

The routes are returned in the following order:

1. Network number
2. Routing protocol
3. Interface identifier
4. Next-hop address

This function is less efficient than the corresponding enumeration handle functions.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Functions, **RtmCloseEnumerationHandle**, **RtmCreateEnumerationHandle**, **RtmEnumerateGetNextRoute**, **RtmGetFirstRoute**

Routing Table Manager Version 1 Structures

The Routing Table Manager Version 1 Functions use the following structures:

IP_NETWORK
IP_NEXT_HOP_ADDRESS
IP_SPECIFIC_DATA
IPX_NETWORK
IPX_NEXT_HOP_ADDRESS
IPX_SPECIFIC_DATA
PROTOCOL_SPECIFIC_DATA
RTM_IP_ROUTE
RTM_IPX_ROUTE

IP_NETWORK

The **IP_NETWORK** structure describes an IP network address.

```
typedef struct _IP_NETWORK {
  DWORD    N_NetNumber;
  DWORD    N_NetMask;
} IP_NETWORK, *PIP_NETWORK;
```

Members

N_NetNumber
Specifies the IP network number expressed as an IP address in machine-byte order.

N_NetMask
Specifies the network mask. Apply this mask to the IP address in order to extract the network address. The network mask is in machine-byte order.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Structures, **RTM_IP_ROUTE**

IP_NEXT_HOP_ADDRESS

The **IP_NEXT_HOP_ADDRESS** structure contains the address for the next-hop router for an IP route.

```
typedef struct _IP_NETWORK {
  DWORD    N_NetNumber;
  DWORD    N_NetMask;
} IP_NETWORK, *PIP_NETWORK;

typedef IP_NETWORK IP_NEXT_HOP_ADDRESS, *PIP_NEXT_HOP_ADDRESS;
```

Members

N_NetNumber
Specifies the IP network address expressed as an IP address in machine-byte order.

N_NetMask
Specifies the network mask. Apply this mask to the IP address in order to extract the network address. The network mask is in machine-byte order.

Remarks

The **IP_NEXT_HOP_ADDRESS** structure is a **typedef** of the **IP_NETWORK** structure. The **typedef** is in Rtm.h.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Structures, **IP_NETWORK**, **RTM_IP_ROUTE**

IP_SPECIFIC_DATA

```
typedef struct _IP_SPECIFIC_DATA {
  DWORD    FSD_Type;
  DWORD    FSD_Policy;
  DWORD    FSD_NextHopAS;
  DWORD    FSD_Priority;
  DWORD    FSD_Metric;
  DWORD    FSD_Metric1;
  DWORD    FSD_Metric2;
  DWORD    FSD_Metric3;
  DWORD    FSD_Metric4;
  DWORD    FSD_Metric5;
  DWORD    FSD_Flags;
} IP_SPECIFIC_DATA, *PIP_SPECIFIC_DATA;
```

Members

FSD_Type

Specifies the route type as defined in RFC 1354. The following table shows the possible values for this member.

Member	Value
1	The route type is not specified. The type is different from those listed here.
2	The route is invalid. Normally, this value is used to invalidate a route. However, since invalidation is not supported by the routing table manager, the route is still considered in best-route computations. Therefore, routing protocols should not use this value.
3	The route is a local route, that is, the next hop is the final destination.
4	The route is a remote route, that is, the next hop is not the final destination.

FSD_Policy
Specifies the set of conditions that would cause the selection of a multi-path route. This member is typically in IP TOS format. For more information, see RFC 1354.

FSD_NextHopAS
Specifies the autonomous system number of the next hop.

FSD_Priority
Specifies a metric value. The routing table manager uses this value to compare this route entry to route entries obtained from other routing protocols. The value of this member is set by the routing table manager.

FSD_Metric
Specifies a metric value. The routing table manager uses this value to compare this route entry to other route entries obtained from the same routing protocol. The value of this member is set by the routing protocol.

FSD_Metric1
Specifies a routing-protocol-specific metric value. This metric value is documented in RFC 1354.

FSD_Metric2
Specifies a routing-protocol-specific metric value. This metric value is documented in RFC 1354.

FSD_Metric3
Specifies a routing-protocol-specific metric value. This metric value is documented in RFC 1354.

FSD_Metric4
Specifies a routing-protocol-specific metric value. This metric value is documented in RFC 1354.

FSD_Metric5
Specifies a routing-protocol-specific metric value. This metric value is documented in RFC 1354.

FSD_Flags
Specifies whether the route is valid. The routing protocol should first clear these flags, then set the route as either valid or invalid. The routing protocol should use the macros **ClearRouteFlags()**, **SetRouteValid()**, and **ClearRouteValid()** to perform these operations. These macros are defined in Rtm.h.

Remarks

The routing table manager uses the **FSD_Priority** and **FSD_Metric** members to compute the best route to a particular destination network.

The **FSD_Metric[1-5]** members are for MIB II conformance. MIB II agents display only these metric values. They do not display the **FSD_Metric** metric value. To have the **FSD_Metric** displayed, the routing protocol should also store the value in one of the **FSD_Metric[1-5]** members.

The routing table manager does not use the metric values in the **FSD_Metric[1-5]** members when computing the best route to a destination network. Therefore, the routing protocol should ensure that the **FSD_Metric** member has an appropriate metric value.

A routing protocol could use the **FSD_Flags** to mark a route as invalid, if the route should not be used by other routing protocols.

! Requirements

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

+ See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Structures, **RTM_IP_ROUTE**

IPX_NETWORK

The **IPX_NETWORK** structure describes an IPX network address.

```
typedef struct _IPX_NETWORK {
  DWORD    N_NetNumber;
} IPX_NETWORK, *PIPX_NETWORK;
```

Members

N_NetNumber;
Contains the IPX network number in machine-byte order.

! Requirements

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

+ See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Structures, **RTM_IPX_ROUTE**

IPX_NEXT_HOP_ADDRESS

The **IPX_NEXT_HOP_ADDRESS** structure contains the address for the next-hop router for an IPX route.

```
typedef struct _IPX_NEXT_HOP_ADDRESS {
  BYTE    NHA_Mac[6];
} IPX_NEXT_HOP_ADDRESS, *PIPX_NEXT_HOP_ADDRESS;
```

Members

NHA_Mac[6]
Specifies the MAC address of next-hop router.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Structures, **RTM_IPX_ROUTE**

IPX_SPECIFIC_DATA

```
typedef struct _IPX_SPECIFIC_DATA {
  DWORD     FSD_Flags;
  USHORT    FSD_TickCount;
  USHORT    FSD_HopCount;
} IPX_SPECIFIC_DATA, *PIPX_SPECIFIC_DATA;
```

Members

FSD_Flags
Specifies flags that describe the route. Currently, this member is either zero or the following flag value:

IPX_GLOBAL_CLIENT_WAN_ROUTE
Specifies that this route is for the global network allocated for all WAN clients.

FSD_TickCount
Specifies the number of ticks it takes to reach the destination network. Routing protocols other than RIP should convert their metrics into ticks.

FSD_HopCount
Specifies the hop count associated with the route.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Structures, **RTM_IPX_ROUTE**

PROTOCOL_SPECIFIC_DATA

The **PROTOCOL_SPECIFIC_DATA** structure contains memory reserved for data specific to a particular routing protocol.

```
typedef struct _PROTOCOL_SPECIFIC_DATA {
  DWORD    PSD_Data[4];
} PROTOCOL_SPECIFIC_DATA, *PPROTOCOL_SPECIFIC_DATA;
```

Members

PSD_Data[4]

Specifies an array of **DWORD** variables. This memory is reserved for data that is specific to routing protocols.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Structures, **RTM_IP_ROUTE**, **RTM_IPX_ROUTE**

RTM_IP_ROUTE

The **RTM_IP_ROUTE** structure contains information that describes a route owned by the IP protocol family:

```
typedef struct _RTM_IP_ROUTE {
  FILETIME                  RR_TimeStamp;
  DWORD                     RR_RoutingProtocol;
  DWORD                     RR_InterfaceID;
  PROTOCOL_SPECIFIC_DATA    RR_ProtocolSpecificData;
  IP_NETWORK                RR_Network;
  IP_NEXT_HOP_ADDRESS       RR_NextHopAddress;
  IP_SPECIFIC_DATA          RR_FamilySpecificData;
} RTM_IP_ROUTE, * PRTM_IP_ROUTE;
```

Members

RR_TimeStamp
Specifies the time that the route entry was created or last updated. This member is set by the routing table manager. The time is expressed as a **FILETIME** structure.

RR_RoutingProtocol
Specifies the routing protocol that added the route.

RR_InterfaceID
Specifies the interface through which the route was obtained.

RR_ProtocolSpecificData
Specifies a **PROTOCOL_SPECIFIC_DATA** structure that contains memory reserved for routing-protocol-specific data.

RR_Network
Specifies an **IP_NETWORK** structure that contains an IP network address.

RR_NextHopAddress
Specifies an **IP_NEXT_HOP_ADDRESS** structure that contains the address of the next-hop router.

RR_FamilySpecificData
Specifies an **IP_SPECIFIC_DATA** structure that contains IP protocol-family-specific data.

Remarks

The members of the **RTM_IP_ROUTE** structure are all **DWORD** aligned.

! Requirements

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

+ See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version 1 Structures, **IP_NETWORK**, **IP_NEXT_HOP_ADDRESS**, **IP_SPECIFIC_DATA**

RTM_IPX_ROUTE

The **RTM_IPX_ROUTE** structure contains information that describes a route for the IPX protocol family.

```
typedef struct _RTM_IPX_ROUTE {
  FILETIME                    RR_TimeStamp;
  DWORD                       RR_RoutingProtocol;
  DWORD                       RR_InterfaceID;
```

```
    PROTOCOL_SPECIFIC_DATA    RR_ProtocolSpecificData;
    IPX_NETWORK               RR_Network;
    IPX_NEXT_HOP_ADDRESS      RR_NextHopAddress;
    IPX_SPECIFIC_DATA         RR_FamilySpecificData;
} RTM_IPX_ROUTE, * PRTM_IPX_ROUTE;
```

Members

RR_TimeStamp
Specifies the time that the route entry was created or last updated. This member is set by the routing table manager. The time is expressed as a FILETIME structure.

RR_RoutingProtocol
Specifies the routing protocol that added the route.

RR_InterfaceID
Specifies the interface through which the route was obtained.

RR_ProtocolSpecificData
Specifies a **PROTOCOL_SPECIFIC_DATA** structure containing memory reserved for data specific to routing protocols.

RR_Network
Specifies an **IPX_NETWORK** structure that contains an IP network address.

RR_NextHopAddress
Specifies an **IPX_NEXT_HOP_ADDRESS** structure that contains the address of the next-hop router.

RR_FamilySpecificData
Specifies an **IPX_SPECIFIC_DATA** structure that contains data specific to the IPX protocol family.

Remarks

The members of the **RTM_IPX_ROUTE** structure are all **DWORD** aligned.

Requirements

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

See Also

Routing Table Manager Version 1 Reference, Routing Table Manager Version_1_Structures, **IPX_NETWORK**, **IPX_NEXT_HOP_ADDRESS**, **IPX_SPECIFIC_DATA**

Routing Table Manager Version 1 Protocol Family Identifiers

The following transport identifiers are listed in Rtm.h. Use these identifiers with the RTMv1 API.

Constant	Description
RTM_PROTOCOL_FAMILY_IPX	Identifies the IPX address family.
RTM_PROTOCOL_FAMILY_IP	Identifies the IP address family.

See Also

Routing Table Manager Version 1 Reference

CHAPTER 11

Routing Table Manager Version 2

Routing Table Manager Version 2 Overview

This chapter describes the Routing Table Manager Version 2 (RTMv2) technology. The RTMv2 API is a feature of Microsoft® Windows® 2000 that you can use to write routing protocols that interact with the routing table manager.

The routing table manager is the central repository of routing information for all routing protocols that operate under the Routing and Remote Access Service (RRAS).

RTMv2 is not available for Microsoft® Windows NT® version 4.0. Additionally, RTMv2 cannot be used for IPX routing protocols that run on Windows NT 4.0 or Windows 2000. If you are using IPX or writing routing protocols for Windows NT 4.0, you must use the Routing Table Manager Version 1 (RTMv1) API.

Components of the Routing Table Manager Architecture

This section describes the major components of the Routing Table Manager Version 2 (RTMv2) technology:

- Router
- Client
- Router Manager
- Routing Protocol
- Forwarder
- Routing Table Manager
- Routing Table Manager Instance
- Address Family
- Routing Table
- View
- Routing Table Entries
  - Destinations
  - Routes and the Best Route
  - Next Hops

Router

For the purposes of this documentation, a router is a Windows NT/Windows 2000 server that is running the Routing and Remote Access Service (RRAS) and the routing table manager.

Client

A client of RTMv2 is either an instance of a routing protocol, or a management component that registers and interacts with the routing table manager using the RTMv2 API. Unless otherwise specified, any reference to a routing table manager client refers to a client of RTMv2.

A client must run in the same process as the router manager.

Router Manager

The router manager is the component of a Windows NT/Windows 2000 router that starts and manages the different routing components. The components include routing protocols, the routing table manager, and the multicast group manager.

Routing Protocol

A routing protocol is a type of client that registers with the routing table manager. Routers use routing protocols to exchange information regarding routes to a destination.

Routing protocols are either unicast or multicast. Routing protocols advertise routes to a destination. A unicast route to a destination is used by a unicast routing protocol to forward unicast data to that destination. A multicast route to a destination is used by some multicast routing protocols to create the information that is used to forward multicast data from hosts on the destination network of the route (known as reverse path forwarding).

Examples of unicast routing protocols include: Routing Information Protocol (RIP), Open Shortest Path First (OSPF), and Border Gateway Protocol (BGP). Examples of multicast routing protocols include: Multicast Open Shortest Path First (MOSPF), Distance Vector Multicast Routing Protocol (DVMRP), and Protocol Independent Multicast (PIM).

The routing table manager supports multiple instances of the same protocol (such as Microsoft OSPF and a third-party OSPF) running on the router. This allows routers to use the different capabilities of each version. These protocols have different protocol identifiers.

Protocol identifiers are comprised of a vendor identifier and a protocol-specific identifier. The protocol-specific identifier is the same for different implementations of the protocol, such as Microsoft OSPF and a third-party implementation of OSPF. Only when the vendor and protocol-specific identifiers are combined is there a unique identifier for a routing protocol.

A protocol with the same protocol identifier (that is, the same vendor identifier and protocol-specific identifier) can register with the routing table manager multiple times. Each time, the protocol registers using a different protocol instance identifier. For example, an implementation of OSPF from a particular vendor can register as Vendor-OSPF-1 and Vendor-OSPF-2. This enables a specific protocol implementation to partition the information that it keeps in the routing table.

Forwarder

The forwarder is the kernel-mode component of the router that is responsible for forwarding data from one router interface to the others. The forwarder also decides whether a packet is destined for local delivery, whether it is destined to be forwarded out of another interface, or both.

There are two kernel-mode forwarders: unicast and multicast.

The router manager obtains the best routes to all destinations from the routing table manager. These routes are passed to the unicast forwarder. The unicast forwarder uses these routes to perform the actual forwarding of unicast data. In this manner, the unicast forwarder maintains a cache of the best routes in the unicast view of the routing table.

The multicast group manager uses information from the multicast view of the routing table to add multicast forwarding entries to the multicast forwarder.

Routing Table Manager

The routing table manager is the central repository of routing information for all routing protocols that operate under the Routing and Remote Access Service (RRAS). It notifies clients when changes have occurred, and allows clients to exchange private information.

The routing table manager provides routing information to all interested clients, such as routing protocols, management programs, and monitoring programs. The routing table manager also determines the best route to each destination network that is known to the routing protocols. The routing table manager determines this route based on routing protocol priorities and on the metrics associated with the routes. The person administering the router can configure routing protocol priorities using the RRAS management console.

The routing table manager passes the best-route information to the forwarders (one per address family, or one per interface) and to all interested clients.

Each client registers with the routing table manager, and in return receives a handle that the client uses to add or delete routes. Clients can retrieve information stored in the routing table. Additionally, clients can register with the routing table manager to receive notification of changes to the best route to a destination.

Routing Table Manager Instance

An instance is a separate table that the routing table manager uses to maintain routing information about a subset of interfaces. Instances are typically used to enable a physical router to act as a set of virtual routers—one virtual router per logical network.

Currently, the routing table manager supports only one instance (identified as zero, the default). The client can register with other instances, but any virtual router except the default one is recognized or used by the router manager.

Address Family

Address families include Internet Packet Exchange (IPX), Internet Protocol Version 4 (IPv4), and Internet Protocol Version 6 (IPv6). Address families can also be referred to as protocol families. These address families are defined in RFC 1700.

Routing Table

The routing table manager maintains a distinct routing table for each address family (such as IPX and IPv4). The IPX address family is supported by the RTMv1 API; IPv4 is supported by the RTMv2 API.

The routing table consists of destinations, routes, and next hops. These entries define a route to a destination network.

View

A view is a subset of the routing table and contains a group of related routes (for example, multicast routes). Currently, only unicast and multicast views are supported. Views are sometimes called Routing Information Bases (RIBs).

Routing Table Entries

The routing table consists of three types of entries: destinations, routes, and next hops.

Destinations

A destination in the routing table is a network entry represented by a network address and a network mask.

A destination entry in the routing table includes:

- The address, expressed as a network address and network mask.
- A list of routes to the destination.
- A list of opaque pointer slots.
- The views in which this destination is valid. The destination contains a structure for each view that contains the following information:
  - An identifier for the view.
  - A pointer to the best route to the destination in this view.
  - The owner of the best route in this view.
  - Flags associated with the best route in this view.
  - Handle to any routes that are in a hold-down state in this view.

Routes and the Best Route

A route is a "network path" to a destination that has a certain cost associated with it. The cost is represented by its administrative preference and its protocol-specific metric. Routes with lower costs are preferred over all other routes.

A route entry in the routing table includes:

- A handle to the destination
- The owner of this route
- The neighbor (peer) that provided the route information
- Flags associated with the state of the route

- Flags associated with the route
- The preference and metric for the route
- The list of views to which the route belongs
- Information that is private to the owner of the route
- A list of next hops used to reach the destination

The following values, taken together, uniquely identify a route in the routing table:

- The destination network
- The owner of the route
- The neighbor who supplied the route

Metrics and Preference

Each route has an administrative preference (specified by the routing policy), and a client-dependent metric. The routing table manager uses this information to determine which route is the better route to a destination. Routes with lower preference are better routes (one being lowest, and therefore best). If two routes have the same preference, the route with the lower metric is the better route.

Normally, the preference of a route is determined by the preference of the client that added the route. However, for any routes added using the Netsh.exe management tool, a preference value can be specified on a per-route basis.

Preference is normally used to indicate priority between clients. For example, an administrator can assign OSPF a lower (better) preference than RIP. In this case, OSPF routes are preferable to RIP routes.

Next Hops

Routes have one or more next hops associated with them. If the destination is not on a directly connected network, the next hop is the address of the next router (or network) on the outgoing network that can most effectively route data to the destination. Each next hop can be used to forward data on the path to the destination. All routes owned by a client share a common set of next-hop entries that were added by the client.

Each next hop is uniquely identified by the address of the next hop and the interface index used to reach the next hop.

If the next hop itself is not directly connected, it is marked as a "remote" next hop. In this case, the forwarder must perform another lookup using the next hop's network address. This lookup is necessary to find the "local" next hop used to reach the remote next hop and the destination.

A next-hop entry in the routing table includes:

- The network address of the next hop
- The owner of the next hop
- The identifier of the outgoing interface

- The state of the next hop
- Flags associated with the next hop
- Information that is private to the owner of the next hop
- A handle to the destination corresponding to the remote next hop

How the Routing Table Manager Architecture Fits Together

Figure 11-1 shows the relationship between the different components of a router.

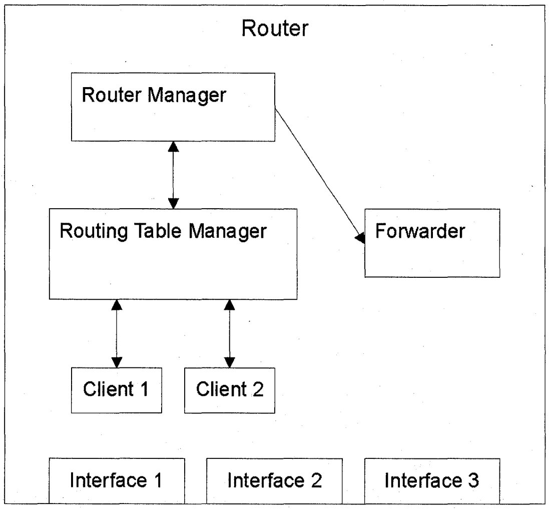


Figure 11-1: Router Components.

When the router is bootstrapped, the router manager service is started, as well as one or more routing protocols. Routing protocols are associated with the various interfaces on the router. The router manager also starts the routing table manager.

Figure 11-2 shows the relationship between clients and the different components of the routing table manager.

The router manager starts one or more instances of the routing table manager. When multiple instances of the routing table manager are started, the router has been configured to act as one or more virtual routers. Each instance of the routing table manager owns one or more interfaces; each interface can only be owned by one instance of the routing table manager.

Each instance of the routing table manager is independent from the others; no information is exchanged between the instances.

Each instance of the routing table manager contains one or more routing tables. Each routing table is associated with an address family.

Each routing table contains one or more views. In this example, the routing table is shown with a unicast and multicast view. Each view is a subset of the routing table.

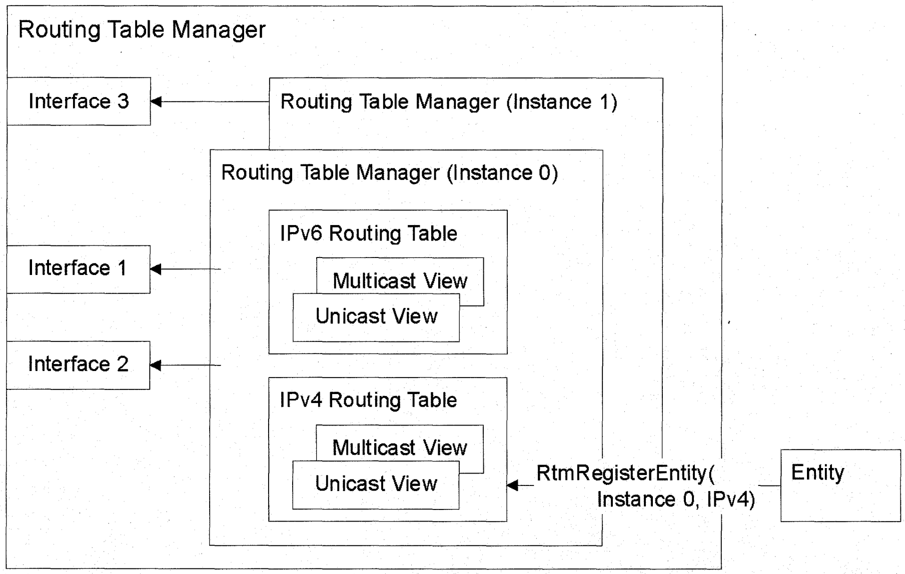


Figure 11-2: Relationship Between Clients and Components of Routing Table Manager.

Figure 11-3 shows the relationship between clients and multiple instances of the routing table manager, routing tables, and views.

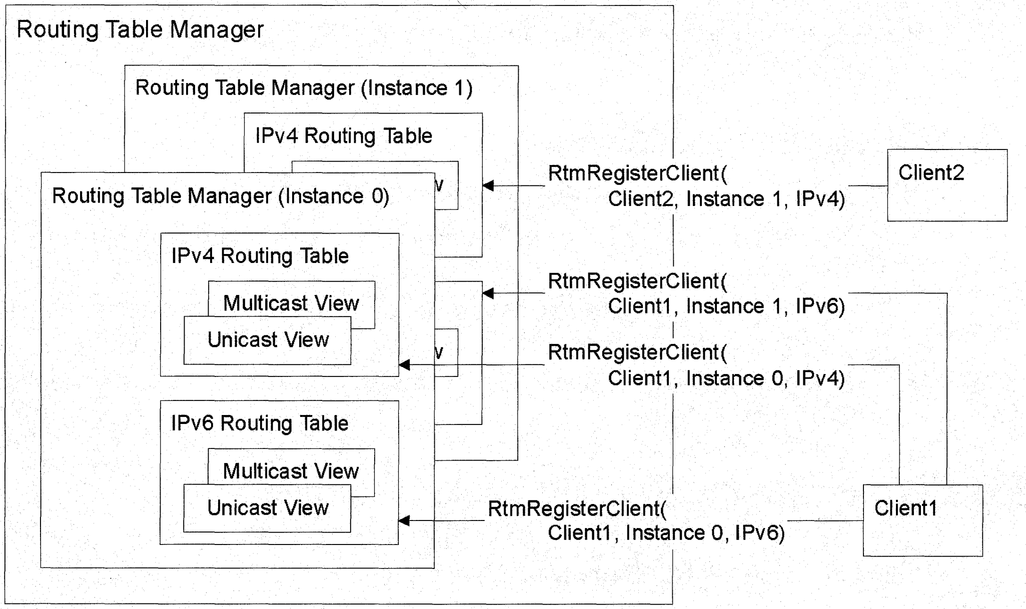

Routing Table Manager
Routing Table Manager (Instance 1)
IPv4 Routing Table
Routing Table Manager (Instance 0)
IPv4 Routing Table
Multicast View
Unicast View
IPv6 Routing Table
Multicast View
Unicast View
RtmRegisterClient(Client2, Instance 1, IPv4)
Client2
RtmRegisterClient(Client1, Instance 1, IPv6)
RtmRegisterClient(Client1, Instance 0, IPv4)
RtmRegisterClient(Client1, Instance 0, IPv6)
Client1

Figure 11-3: Relationship Between Clients and Multiple Instances of the Routing Table Manager, Routing Tables, and Views.

An instance of the client can register multiple times with an instance of the routing table manager—once per address family. A client can register with each instance of the routing table manager.

Figure 11-4 shows how the routing table entries are related. For more information on routing table entries, see *Routing Table Entries*.

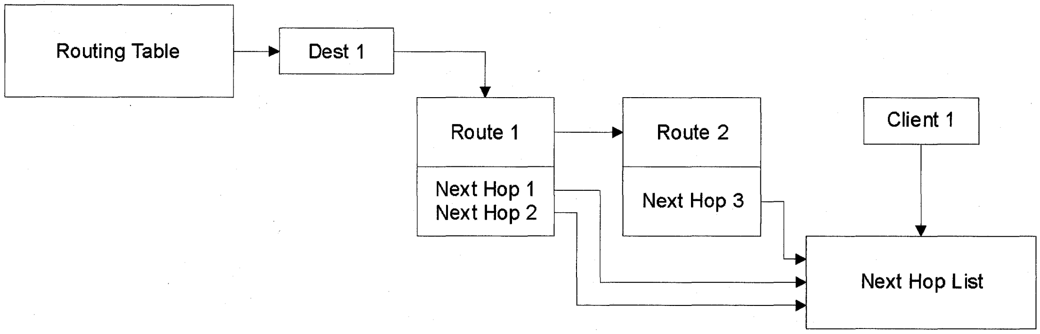


Figure 11-4: Relationship Between Routing Tables.

The routing table contains destinations. Each destination is related to one or more routes. Each route has zero, one, or more pointers to next hops that are associated with the route. Each pointer refers to the actual next hop in the list of next hops. Each client that registers with the routing table manager creates a list of next hops that the client owns.

Routes can only contain pointers to the next-hop list associated with the client that owns the route.

RTMv2 Programming Issues

RTMv2 functions are written with the following assumptions:

- RTMv2 functions do not allocate memory for the client. The client must always allocate memory.
- When a client is unregistering, it must perform "clean-up" operations itself, such as releasing all memory allocated.
- Clients must release handles correctly; memory leaks can occur if a client does not observe this practice.

Registering with the Routing Table Manager

Before a client can access the routing table, it first must register with the routing table manager using the **RtmRegisterEntity** function.

When a client registers, it passes to the routing table manager an **RTM_ENTITY_INFO** structure. This structure contains the information that uniquely identifies a client, the address family, and the instance of the routing table manager with which the client is registering. A client can also establish the **RTM_EVENT_CALLBACK** callback. The routing table manager will use this callback to notify the client of events such as change notifications and client registrations.

The routing table manager completes its registration processing and returns a handle to the client. The client must use this handle for all subsequent calls to RTMv2 functions.

The **RtmRegisterEntity** function that is used in RTMv2 is analogous to the **RtmRegisterClient** function that is used in RTMv1. The **RtmRegisterClient** function is obsolete, except for clients using IPX.

Once a client has finished interacting with the routing table manager, it must call **RtmDeregisterEntity**. The routing table manager destroys the handle associated with the client. To avoid memory leaks, the client must ensure that it releases all handles and deletes all the routes and next hops that it owns before calling **RtmDeregisterEntity** .

For sample code that shows how to use these functions, see *Register with the Routing Table Manager* and *Use the Event Notification Callback*.

Enumerating Registered Entities

Once a client has registered, the client can obtain a list all the other clients that have registered with the routing table manager. Some clients must perform certain operations if the presence of a particular type of client is detected.

The client can call the **RtmGetRegisteredEntities** function. A buffer of **RTM_ENTITY_INFO** structures is returned. Once the client has processed this information, the client should call **RtmReleaseEntities** to release the handles in the **RTM_ENTITY_INFO** structures.

If the routing table manager client supplied a callback function in the call to **RtmRegisterEntity**, the client is notified when any other clients register or unregister.

For sample code that shows how to use these functions, see *Enumerate the Registered Entities* and *Use the Event Notification Callback*.

Using Methods

When a client registers with the routing table manager, it can export a set of methods. These methods are used by other clients to obtain client-specific information. Methods enable private communication between clients of the routing table manager.

A client can obtain the list of methods exported by another client. The client calls the **RtmGetEntityMethods** function, supplying the target client's handle.

Each method exported by a client is uniquely identified by its method identifier. Each client can export up to 32 methods. Each method corresponds to a bit set in the **MethodType** member of the **RTM_ENTITY_EXPORT_METHOD** structure. To invoke multiple methods, the client should perform a logical OR of the identifiers for those methods. The syntax and semantics of input and output structures for each protocol must be specified when the protocol is implemented.

Note Method identifier values that correspond to a bit set in the lower half of the **MethodType** member (the lower 16 bits) are reserved by Microsoft.

To invoke a second client's method, a client calls the **RtmInvokeMethod** function. The routing table manager arbitrates all requests to invoke a client's methods. The routing table manager performs two functions when it arbitrates between clients:

- Preventing the second client from invoking a method for a client that is unregistering.
- Holding an "invoke" request when methods are blocked, and allowing the request to continue when the methods are unblocked.

If a client must prevent other clients from executing its methods, the client can call **RtmBlockMethods**. The routing table manager will not allow a call to **RtmInvokeMethod** to be processed until the client unblocks its methods again.

Clients typically block methods when making changes to the private data that is exchanged between clients. Blocking methods is an optional action. A client can also use internal locks to prevent other clients from invoking methods.

For more sample code that shows how to use these functions, see *Obtain and Call the Exported Methods for a Client*.

Using Opaque Pointers

Clients often must store additional, client-specific information about destinations. The routing table manager enables clients to store this information in destination structures in the routing table. The information is stored and retrieved using "opaque pointers". The information stored is private, and accessible only to the client that owns the opaque pointer.

For example, the multicast group manager keeps a list of multicast forwarding entries that are dependent on a particular destination. The multicast group manager uses an opaque pointer in that destination. In another example, a routing protocol that advertises a particular destination can keep information related to its own route advertisement of the destination using an opaque pointer, even though it does not own the best route.

The number of opaque pointers is limited; these pointers are allocated to clients on a first-come, first-served basis. The router administrator must allocate the correct number of pointers during the router configuration; therefore, routing protocols and other clients must document their use of opaque pointers.

Accessing Opaque Pointers

Clients are able to access the information stored in destinations by using opaque pointers. To use the storage, the client must first call **RtmGetOpaqueInformationPointer** to obtain the pointer. Whenever a change to the information is necessary, the client must first lock the destination by calling **RtmLockDestination** (with the *LockDest* parameter set to TRUE). Once the destination is locked, the client can make the necessary change. The destination can be unlocked using another call to **RtmLockDestination** (with the *LockDest* parameter set to FALSE).

The **RtmLockDestination** function also allows a client to use either a read lock or a write lock, using the *Exclusive* parameter. A client should use the write lock only when it is making changes to the information kept using the opaque pointer. Clients can use the use the read lock to view the opaque pointer information stored in a destination.

For sample code that shows how to use these functions, see *Access the Opaque Pointers in a Destination.*

Marking Routes for the Hold-Down State

Some clients, such as distance vector protocols like RIP and DVMRP, require that destinations be advertised as unreachable for a certain time after the last route to the destination is deleted. The last route that is deleted must be advertised as unreachable even if newer routes arrive in the meantime. The last route deleted is marked as being in a "hold-down state". The hold-down process prevents the formation of routing loops. Routing loops are caused when a routing protocol advertises obsolete routing information. When the hold-down expires, these protocols resume their advertisement with the new best route.

A protocol that implements hold-down states indicates that a destination is in a hold-down state by using the **RtmHoldDestination** function. The client calls this function when it advertises the best route to this destination. If all routes to this destination are later deleted, the last route that is deleted is kept in a hold-down state for the period of time specified in the earlier call to **RtmHoldDestination**.

When a protocol advertises a destination, the route information that is used depends on whether the protocol uses hold-down states and if a route in the hold-down state exists for the destination.

Protocols that do not use hold-down states can ignore route information that relates to hold-down states for a destination, and always advertise the best route.

For sample code that shows how to use these functions, see *Use the Route Hold-Down State.*

Adding Routes

Once a client has discovered a route, the client can add that route to the routing table.

▶ To add a route, the client should take the following steps

1. If the client has already cached the next-hop handle, go to step 4.
2. Create an **RTM_NEXTHOP_INFO** structure and fill it with the appropriate information.
3. Add the next hop to the routing table by calling **RtmAddNextHop**. The routing table manager returns a handle to the next hop. If the next hop already exists, the routing table does not add the next hop; instead it returns the handle to the next hop.
4. Create an **RTM_ROUTE_INFO** structure and fill it with the appropriate information, including the next-hop handle returned by the routing table manager.
5. Add the route to the routing table by calling **RtmAddRouteToDest**. The routing table manager compares the new route to the routes that are already in the routing table. Two routes are equal if all of the following conditions are true:
   - The route is being added to the same destination.
   - The route is being added by the same client (as specified by the **Owner** member of the **RTM_ROUTE_INFO** structure).
   - The route is advertised by the same neighbor (as specified by the **Neighbor** member of the **RTM_ROUTE_INFO** structure).

   If the route exists, the routing table manager returns the handle to the existing route. Otherwise, the routing table manager adds the route and returns the handle to the new route.

   The client can set the *Change_Flags* parameter to RTM_ROUTE_CHANGE_NEW to instruct the routing table manager to add a new route on the destination, even if another route with the same owner and neighbor fields exists.

   The client can set the *Change_Flags* parameter to RTM_ROUTE_CHANGE_FIRST to cause the routing table manager to update the first route on the destination owned by the caller. This update can be performed if such a route exists, even if the neighbor field does not match. This flag is used by clients that maintain a single route per destination.

A client can remove routes from the routing table by calling the **RtmDeleteRouteToDest** function.

For sample code that shows how to use these functions, see *Add and Update Routes Using RtmAddRouteToDest*.

Retrieving Route Information

There are three methods used to obtain route information from the routing table manager:

1. Enumerating routes (described in Enumerating Routing Table Entries)
2. Searching for specific routes (described in Finding Specific Information in the Routing Table)
3. Retrieving changed destinations (described in Receiving Notification of Changes)

Updating Routes

A client can use the either of the following two methods to update or remove routes that it owns:

- Updating Routes Using **RtmAddRouteToDest**
- Updating Routes In Place Using **RtmUpdateAndUnlockRoute**

Updating Routes Using RtmAddRouteToDest

If the client does not require efficiency when adding a route, it should use the following method of updating routes. This method is less efficient since it requires obtaining a handle to the route, requires passing an **RTM_ROUTE_INFO** structure to and from the routing table manager, and takes more time. Since **RtmAddRouteToDest** does not manipulate the routing table directly, using this method trades efficiency for simplicity.

▶ To update a route, the client should

1. Call **RtmGetRouteInfo** with the handle to the route. The handle is either one previously cached by the client, or returned by the routing table manager from a call that returns a route handle (such as **RtmGetRouteInfo**).
2. Make the changes to the **RTM_ROUTE_INFO** structure that is returned by the routing table manager.
3. Call **RtmAddRouteToDest** with the handle to the route and the changed **RTM_ROUTE_INFO** structure.

For sample code that shows how to use these functions, see *Add and Update Routes Using RtmAddRouteToDest.*

Updating Routes In Place Using RtmUpdateAndUnlockRoute

In-place updating is generally more efficient than updating the routing table with an indirect method such as that used by the **RtmAddRouteToDest** function. This method is more efficient because the client is not required to obtain a handle, not required to pass an **RTM_ROUTE_INFO** structure to and from the routing table manager, and takes less time. However, directly updating the routing table can be risky, since the routing table manager is not functioning as an intermediary.

▶ To update a route, the client should take the following steps

1. Lock the route by calling **RtmLockRoute**. (Currently, this function actually locks the route's destination). The routing table manager returns a pointer to the route.
2. Use the pointer to the routing table manager's **RTM_ROUTE_INFO** structure (obtained in step 1) to make the necessary changes to the route. Only certain members can be modified when updating in place. These members are: **Neighbour**, **PrefInfo**, **EntitySpecificInfo**, **BelongsToViews**, and **NextHopsList**.

Note If the client adds information to either the **Neighbour** or **NextHopList** members, the client must call **RtmReferenceHandles** to explicitly increment the reference count that the routing table manager keeps on the next-hop object. Similarly, if the client removes information from the *NextHopList* member, the client must call **RtmReleaseNextHops** to decrement the reference count.

3. Call **RtmUpdateAndUnlockRoute** to notify the routing table manager that a change has taken place. The routing table manager commits the changes, updates the destination to reflect the new information, and then unlocks the route.

For sample code that shows how to use these functions, see *Update a Route In Place Using RtmUpdateAndUnlockRoute*.

Receiving Notification of Changes

Many clients can simultaneously update the routing table, and clients must be notified when changes to routing information occur. For example, a client that is not notified of another client's changes to the routing table could advertise outdated route information. This can be prevented by programming clients to register with the routing table manager to be notified of changes in the routing table. The routing table manager sends notifications of changes to all clients that register to receive them.

Change notification applies only to destinations. There is no way to query the routing table manager for changes to a particular route.

When a change is made to one of the routes to a destination, the routing table manager sends out a notification that a change has occurred. This notification goes only to those clients that have registered with the routing table manager for the type of change that has occurred. All changes to routing information in the routing table manager occur in one or more views, and change notification messages can be requested in any subset of supported views.

There are currently three types of change notifications for which a client can register:

- Notification of any change to the routes for the destination. This request is made using the RTM_CHANGE_TYPE_ALL flag.
- Notification if the best route to the destination changes, or any of the following information for the current best route changes:
  - preference
  - next hops
  - route flags

  This request is made using the RTM_CHANGE_TYPE_BEST flag.

- Notification of all changes of the type RTM_CHANGE_TYPE_BEST, except changes in non-forwarding flags in the best route. For example, the router manager waits for changes of this type in the unicast view, and updates information in the unicast forwarder. This request is made using the RTM_CHANGE_TYPE_FORWARDING flag.

Requests for notifications of changes can also be restricted to a subset of destinations by registering for notifications of changes only to "marked" destinations. The client can mark a destination for change notification by calling **RtmMarkDestForChangeNotification**.

When a change occurs, the routing table manager checks to see if there are any clients that must be notified of this change. A client must be notified of a change if all of the following conditions are met:

- the type of change that occurred is a type for which the client has registered for notification
- changes to a destination that the client has marked have occurred (or any destination, if the client has requested changes for all destinations)
- the client requested change notification for the view in which this change occurred

If the change meets all of the above criteria, the change is cached and the client is notified.

The notification does not specify what the actual changes are, only that they have occurred. The client must retrieve the changes by calling **RtmGetChangedDests** using the notification handle that was obtained from a previous call to **RtmRegisterForChangeNotification**.

Registering for Change Notification

A client can register to receive notification of changes to routing information that is stored in the routing table manager. This request can only be made after a client has registered with the routing table manager.

To register for change notification, a client must call **RtmRegisterForChangeNotification**, specifying the types of changes for which the client must receive notification. If the client must be notified of change to specific destinations, the client calls **RtmMarkDestForChangeNotification** once for each destination.

The client can stop receiving change notifications by calling **RtmDeregisterFromChangeNotification**.

For sample code that shows how to use these functions, see *Register For Change Notification*.

Retrieving Changes

Once a client has registered for notification of certain changes and one or more of these changes occurs, the client receives a notification from the routing table manager. The routing table manager uses the **RTM_EVENT_CALLBACK** callback that was supplied in a previous call to **RtmRegisterEntity**. The routing table manager indicates that a RTM_CHANGE_NOTIFICATION event has occurred.

▶ **After the client receives notice of the change, the client can retrieve the changes by taking the following steps**

1. Call **RtmGetChangedDests** to retrieve a set of changes.
2. Process the changes.
3. Release the destinations using **RtmReleaseChangedDests**.
4. Repeat steps 1, 2, and 3 until **RtmGetChangedDests** returns ERROR_NO_MORE_ITEMS.

For sample code that shows how to use these functions, see *Use the Event Notification Callback*.

Retrieving Change Status and Ignoring Changes

The client can query the routing table manager to find out if a notification of a change to a destination is pending by calling **RtmGetChangeStatus**. This function returns TRUE until the caller retrieves this change by calling **RtmGetChangedDests**.

A client can use this query to avoid performing an action that would have to be undone after the change notification is received and processed. Using this feature allows the client to work efficiently by deferring certain operations to a later time.

The client can also ignore a pending change notification for a destination by calling **RtmIgnoreChangedDests**. A later call to **RtmGetChangedDests** will not return this destination unless another change occurs after the call to **RtmIgnoreChangedDests**.

Working with Next Hops

The RTMv2 API allows clients to work with the list of next hops that are associated with routes and destinations. Clients can add or update a next hop by calling **RtmAddNextHop**. Clients can delete a next hop using **RtmDeleteNextHop**. Clients can search for next hops by calling **RtmFindNextHop**.

▶ **Clients can also can update the next hop directly**

1. Call **RtmLockNextHop** with the *Lock* parameter set to TRUE to lock the next hop and obtain a direct pointer to the routing table manager's **RTM_NEXTHOP_INFO** structure.
2. Use the pointer returned by the routing table manager to make the necessary changes to the next hop.

Note The next-hop address and interface index fields in the next hop uniquely identify the next hop and should not be modified.

3. Call **RtmLockNextHop** with the *Lock* parameter set to FALSE to unlock the next hop.

Enumerating Routing Table Entries

The enumeration functions allow a client to retrieve information about a specific type of routing table object (routes, destinations, and next hops). Both routing protocols and administration programs can use these functions to locate specific data.

▶ **The basic process for each enumeration is as follows**

1. Start the enumeration by obtaining a handle from the routing table manager. Call **RtmCreate*Data_Type*Enum** (where ***Data_Type*** is either **Dest**, **Route**, or **NextHop**), and supply the criteria that specifies the kind of information being enumerated. This criteria includes (but is not limited to) a range of destinations, a particular interface, and the views in which the information resides.
2. Call **RtmGetEnum*Data_Type*** one or more times to retrieve data until the routing table manager returns ERROR_NO_MORE_ITEMS. The route, destination, and next-hop data is returned in order of the address information (and the preference and metric values, if routes are being enumerated).
3. Call **RtmRelease*Data_Type*** when the handles or information structures associated with the enumeration are no longer needed.
4. Call **RtmDeleteEnumHandle** to release the enumeration handle (returned when the enumeration was created). This function is used to release the handle for all types of enumerations.

Note Routes that are in the hold-down state are only enumerated when a client requests data from all views using RTM_VIEW_MASK_ANY.

For sample code that shows how to use these functions, see *Enumerate All Destinations and Enumerate All Routes.*

Finding Specific Information in the Routing Table

Clients must be able to locate specific information in the routing table, rather than being required to enumerate all the data. The RTMv2 API provides the ability to search for specific destinations, routes, and next hops based on certain criteria.

Retrieving Information

RTMv2 allows a client to retrieve the information referred to by a given handle using the **RtmGet*Data_Type*Info** functions, where ***Data_Type*** is either **Entity**, **Dest**, **Route**, or **NextHop**.

Each of these function calls has a corresponding **RtmRelease*Data_Type*** function to release the handles associated with the information structure returned by the routing table manager.

Note Information for clients is available only in the current instance and address family.

For sample code that shows how to use these functions, see *Search For the Best Route*.

Using RtmGetExactMatchRoute and RtmGetExactMatchDestination

The **RtmGetExactMatchRoute** and **RtmGetExactMatchDestination** functions are used by clients to find a specific route or destination. These functions save time by doing the comparison work for the client.

For example, when RIP updates a route, RIP must retain the old metric information. RIP searches for the route and its information. Then RIP can copy the information and update the route.

Using RtmGetMostSpecificDestination

The **RtmGetMostSpecificDestination** function is used to locate the destination that best matches the specified network prefix.

For example, the multicast group manager uses this function to perform a Reverse-Path-Forwarding (RPF) check on a single address. The function can also be used to find the local next hop for a given remote next hop.

For sample code that shows how to use these functions, see *Search for Routes Using RtmGetMostSpecificDestination and RtmGetLessSpecificDestination* and *Search For the Best Route*.

Using RtmGetLessSpecificDestination

The **RtmGetLessSpecificDestination** function is used to locate the destination that is the next-best match for the specified network prefix. This function can be called repeatedly to return the next successive less-specific match, until no more destinations match.

This function is called after a call to **RtmGetMostSpecificDestination**.

For sample code that illustrates how to use these functions, see *Search for Routes Using RtmGetMostSpecificDestination and RtmGetLessSpecificDestination*.

Using RtmIsBestRoute

The **RtmIsBestRoute** function enables a client to quickly find out whether or not a particular route is the best route to a destination. For example, a client may need to store a particular route only if it is the best route. Therefore, the client can call this function, instead of enumerating all routes and making the comparison itself.

Maintaining Client-Specific Lists

RTMv2 provides functions that enable clients to create a private list of routes stored in the routing table. Using this list can be more efficient than enumerating routes from the routing table.

▶ To use this feature, a client should take the following steps

1. Call **RtmCreateRouteList** to obtain a handle from the routing table manager.
2. Call **RtmInsertInRouteList** whenever the client must add a route to this list.
3. When the client no longer requires the list, it should call **RtmDeleteRouteList** to remove the list.

▶ If the client must enumerate the routes on the list, the client should take the following steps

1. Call **RtmCreateRouteListEnum** to obtain an enumeration handle from the routing table manager.
2. Call **RtmGetListEnumRoutes** to obtain the handles to the routes in the list.
3. Call **RtmReleaseRoutes** to release the handles when no longer required.

For sample code that shows how to use these functions, see *Use a Client-Specific Route List*.

Managing Handles

The routing table manager maintains a reference count for all the information that it maintains. This prevents the routing table manager from returning to a client handles to memory that has been freed. Each time a handle is returned to the caller, either as an explicit handle or as part of an information structure (such as **RTM_DEST_INFO**), the reference count for the object that corresponds to the handle is incremented. When the handle or the information structure is released, the appropriate reference count is decremented. When the reference count becomes zero, the object is freed.

Each **RtmGet*Data_Type*Info** function (where ***Data_Type*** is either **Dest**, **Route**, **NextHop**, or **Entity**) returns an information structure that has a corresponding **RtmRelease*Data_Type*Info** function. The release functions free the handles returned by the routing table manager. Similarly there are functions of type **RtmRelease*Data_Types*** that release handles that were returned by other functions.

Note The **RtmReleaseChangedDests** function should be used to release handles that have been returned by a call to **RtmGetChangedDests**. Do not use **RtmReleaseDests** for changed destination structures.

If a client must keep a specific handle in an information structure while releasing the rest, the client can call **RtmReferenceHandles** with that handle before releasing the information structure. The handle can then later be released by a call to **RtmRelease*Data_Type***.

Using Routing Table Manager Version 2

This section contains sample code that can be used when developing clients such as routing protocols.

Register with the Routing Table Manager

The following sample code shows how to register with the routing table manager.

```
// In the following example, VendorId stands for the 14 bit ID that identifies
// the vendor. For more information, see the Routprot.h file in the SDK.

RTM_ENTITY_INFO EntityInfo;
DWORD       Status;

EntityInfo.RtmInstanceId = 0;   // Must be set to default for now
EntityInfo.AddressFamily = AF_INET;
EntityInfo.EntityId.EntityProtocolId = PROTO_IP_RIP;
EntityInfo.EntityId.EntityInstanceId =
    PROTOCOL_ID(PROTO_TYPE_UCAST, VendorId, PROTO_IP_RIP);

Status = RtmRegisterEntity(EntityInfo,
                           (PRTM_ENTITY_EXPORT_METHODS) ExportMethods,
                           EntityEventCallback,
                           TRUE,
                           &RegnProfile,
                           &RtmRegHandle);

if (Status != NO_ERROR)
{
    // Registration failed - Log an Error and Quit
    ...
    return Status;
}
```

Enumerate the Registered Entities

The following sample code shows how to create an enumeration of registered clients and obtain the client information from the routing table manager.

```
NumEntities = 0;

Status = RtmGetRegisteredEntities(RtmRegHandle,
                                  &NumEntities,
                                  NULL,
                                  NULL);

if (Status == ERROR_INSUFFICIENT_BUFFER)
{
    // Plan for growth
    NumEntities += 5;

    // Allocate space (on stack for convenience) to get info
    EntityInfos = _alloca(NumEntities * sizeof(RTM_ENTITY_INFO));
    EntityHandles = _alloca(NumEntities * sizeof(HANDLE));

    Status = RtmGetRegisteredEntities(RtmRegHandle,
                                      &NumEntities,
                                      EntityHandles,
                                      EntityInfos);

    for (i = 0; i < NumEntities; i++)
    {
        // Process entity info
        if (EntityInfo[i].EntityProtocolId == PROTO_IP_OSPF)
        {
            // This is an OSPF registration

            // Do you want to get methods?
            ...
        }
    }

    RtmReleaseEntities(RtmRegHandle, NumEntities, EntityHandles);

    // At this point, we do not have to release entities since
    // the RTM_ENTITY_INFO structures do not contain handles
}
```

Obtain and Call the Exported Methods for a Client

The following sample code shows how to obtain a list of methods a client has exported, and how to invoke a method for that client.

```
//_____________________________________

//Obtaining an entity's export methods
//_____________________________________

// Specify 0 to get actual number of methods

NumMethods = 0;

Status = RtmGetEntityMethods(RtmRegHandle,
                             EntityHandle,
                             &NumMethods,
                             NULL);

if (Status == ERROR_INSUFFICIENT_BUFFER)
{
        // Get actual number of methods now

    ExportMethods = malloc(NumMethods * sizeof(PVOID));

    Status = RtmGetEntityMethods(RtmRegHandle,
                                 EntityHandle,
                                 &NumMethods,
                                 ExportMethods);

    if (Status == NO_ERROR)
    {
        // You have obtained methods for the
        // entity corresponding to EntityHandle
    }
}

//__________________________________

//Calling an entity's export methods
//__________________________________

// You cannot call a method directly
// You must use the RtmInvokeMethod API

Input.MethodType = METHOD_TYPE_ALL_METHODS; // Or 'OR' of methods to call

// Set the method input and its size
// (assume input is a route handle)
```

```
Input.InputSize = sizeof(HANDLE);
Input.InputData[0] = RouteHandle;

// Assume that there is no output data for
// any function other than an error code

OutputHdrSize = FIELD_OFFSET(RTM_ENTITY_METHOD_OUTPUT, OutputData);
OutputSize = OutputHdrSize * MAX_METHODS;
Output = _alloca(OutputSize);

Status = RtmInvokeMethod(RtmRegHandle,
                         EntityHandle,
                         &Input,
                         &OutputSize,
                         Output);

if (Status == NO_ERROR)
{
    // Parse the output from each method
    ;
}
```

Register for Change Notification

The following sample code shows how to register for changes to best routes to all destinations in the unicast view of the routing table.

```
// In this call, you are registering for changes on
// all destinations (not just those destinations marked before)

Status = RtmRegisterForChangeNotification(RtmRegHandle,
                                          RTM_VIEW_MASK_UCAST,
                              // Unicast route changes only
                                          RTM_CHANGE_TYPE_BEST,
                              // Changes in best routes only
                                          NotifyContext,
                              // Context was returned RTM_EVENT_CALLBACK
                                          &NotifyHandle);

if (Status != NO_ERROR)
{
    // Registration for change notifications failed
    ...
    return Status;
}

// For processing changes upon notification, please
// refer to the implementation of RTM_EVENT_CALLBACK.
```

Add and Update Routes Using RtmAddRouteToDest

The following sample code shows how to add a route to a destination using the routing table manager as an intermediary.

```
// Add a route to a destination given by (addr, masklen)
// using a next hop reachable with an interface

RTM_NEXTHOP_INFO NextHopInfo;

// First, create and add a next hop to the caller's
// next-hop tree (if it does not already exist)

ZeroMemory(&NextHopInfo, sizeof(RTM_NEXTHOP_INFO);

RTM_IPV4_MAKE_NET_ADDRESS(&NextHopInfo.NextHopAddress,
                          nexthop, // Address of the next hop
                          32);

NextHopInfo.InterfaceIndex = interface;

NextHopHandle = NULL;

Status = RtmAddNextHop(RtmRegHandle,
                       &NextHopInfo,
                       &NextHopHandle,
                       &ChangeFlags);

if (Status == NO_ERROR)
{
    // Created a new next hop or found an old one

        // Fill in the route information for the route

    ZeroMemory(&RouteInfo, sizeof(RTM_ROUTE_INFO);

    // Fill in the destination network's address and mask values
    RTM_IPV4_MAKE_NET_ADDRESS(&NetAddress, addr, masklen);

    // Assume 'neighbour learnt from' is the first next hop
    RouteInfo.Neighbour = NextHopHandle;

    // Set metric for route; Preference set internally
    RouteInfo.PrefInfo.Metric = metric;

    // Adding a route to both the unicast and multicast views
    RouteInfo.BelongsToViews = RTM_VIEW_MASK_UCAST|RTM_VIEW_MASK_MCAST;

    RouteInfo.NextHopsList.NumNextHops = 1;
    RouteInfo.NextHopsList.NextHops[0] = NextHopHandle;

    // If you want to add a new route, regardless of
    // whether a similar route already exists, use the following
    //     ChangeFlags = RTM_ROUTE_CHANGE_NEW;

    ChangeFlags = 0;

    Status = RtmAddRouteToDest(RtmRegHandle,
                               &RouteHandle,     // Can be NULL if you
                                                 // do not need handle
                               &NetAddress,
                               &RouteInfo,
                               1000,             // Time out route after
                                                 // 1000 ms
                               RouteListHandle1, // Also add the route
                                                 // to this list
                               0,
                               NULL,
                               &ChangeFlags);

    if (Status == NO_ERROR)
    {
        if (ChangeFlags & RTM_ROUTE_CHANGE_NEW)
        {
        ; // A new route has been created
        }
        else
        {
        ; // An existing route is updated
        }

        if (ChangeFlags & RTM_ROUTE_CHANGE_BEST)
        {
        ; // Best route information has changed
        }

        // Release the route handle if you do not need it
        RtmReleaseRoutes(RtmRegHandle, 1, &RouteHandle);
    }

    // Also release the next hop since it is no longer needed
    RtmReleaseNextHops(RtmRegHandle, 1, &NextHopHandle);
}
```

Update a Route In Place Using RtmUpdateAndUnlockRoute

The following sample code shows how to update a route directly, using a pointer to the actual route information in the routing table.

```
Status = RtmLockRoute(RtmRegHandle,
                      RouteHandle,
                      TRUE,
                      TRUE,
                      &RoutePointer);

if (Status == NO_ERROR)
{
        // Update route parameters in place (i.e., directly on
// the routing table manager's copy)

    // Update the metric and views of the route
    RoutePointer->PrefInfo.Metric = 16;

    // Change the views so that the route belongs to only
    // the multicast view
    RoutePointer->BelongsToViews = RTM_VIEW_MASK_MCAST;

    // Set the entity-specific information to X
    RoutePointer->EntitySpecificInfo = X;

    // Note that the following manipulation of
    // next-hop references is not needed when
    // using RtmAddRouteToDest, as it is done
    // by the routing table manager automatically

    // Change next hop from NextHop1 to NextHop2
    NextHop1 = RoutePointer->NextHopsList.NextHop[0];

    // Explicitly dereference the old next hop
    RtmReleaseNextHops(RtmRegHandle, 1, &NextHop1);

    RoutePointer->NextHopsList.NextHop[0] = NextHop2;

    // Explicitly reference next hop being added
    RtmReferenceHandles(RtmRegHandle, 1, &NextHop2);

    // Call the routing table manager to indicate that route information
    // has changed, and that its position might
    // have to be rearranged and the corresponding destination
    // needs to be updated to reflect this change.

    Status = RtmUpdateAndUnlockRoute(RtmRegHandle,
                                     RouteHandle,
                                     INFINITE, // Keep forever
                                     NULL,
                                     0,
                                     NULL,
                                     &ChangeFlags);
    ASSERT(Status == NO_ERROR);
```

Use the Route Hold-Down State

The following sample code shows how to mark a destination for the hold-down state, and how to create a destination enumeration that includes routes that are in the hold-down state.

```
// Macro to allocate a RTM_DEST_INFO on the stack

#define ALLOC_RTM_DEST_INFO(NumViews, NumInfos)
        (PRTM_DEST_INFO) _alloca(RTM_SIZE_OF_DEST_INFO(NumViews) * NumInfos)


// Mark the destination for hold down in Unicast view

Status = RtmHoldDestination(RtmRegHandle,
                            DestHandle,
                            RTM_VIEW_MASK_UCAST,
                            30*1000); // 30 seconds
// Check Status

// When the last route on this destination is deleted,
// it is moved from the ViewInfo's Route to HoldRoute.

// To enumerate all destinations - even the ones with
// just hold down routes, use the following API calls

// Code to enumerate destinations in the table

MaxHandles = RegnProfile.MaxHandlesInEnum;

DestInfos = ALLOC_RTM_DEST_INFO(NumViews, MaxHandles);

DestInfoSize = RTM_SIZE_OF_DEST_INFO(NumViews);

// Enumerate all destinations in the subtree (0 / 0)
```

(continued)

(continued)

```
// (basically the whole tree; you can
// also achieve this by using RTM_ENUM_START)

RTM_IPV4_MAKE_NET_ADDRESS(&NetAddress,
                          0x00000000,
                          0);

// Create a destination enumeration in views = RTM_VIEW_MASK_ANY.

Status = RtmCreateDestEnum(RtmRegHandle,
                           RTM_VIEW_MASK_ANY, // MUST BE EXACTLY
                                              // THE SAME AS THIS
                           RTM_ENUM_RANGE,
                           &NetAddress,
                           RTM_BEST_PROTOCOL, // Get best route on
                                              // each destination
                           &EnumHandle);

if (Status == NO_ERROR)
{
    do
    {
        NumInfos = MaxHandles;

        Status = RtmGetEnumDests(RtmRegHandle
                                 EnumHandle,
                                 &NumInfos,
                                 DestInfos);

        for (i = 0; i < NumInfos; i++)
        {
            DestInfo = (PRTM_DEST_INFO) ((PUCHAR)DestInfos+(i*DestInfoSize));

            // Process the current destination information
            // As RTM_VIEW_MASK_ANY = 0, we have
            // no view information returned in the call

            // Get actual DestInfo in the specified views
            // (Assume that only unicast is specified)

            Status = RtmGetDestInfo(RtmRegHandle,
                                    DestInfo.DestHandle,
                                    RTM_BEST_PROTOCOL,
                                    RTM_VIEW_MASK_UCAST,
                                    &DestInfoActual);
            if (Status == NO_ERROR)
            {
                HoldRoute = DestInfoActual.ViewInfo[0].HoldRoute;
                BestRoute = DestInfoActual.ViewInfo[0].Route;

                // Advertise best unicast route (or the hold down route)

                if (HoldRoute)
                {
                // HoldRoute exists; Advertise it if hold down protocol
                    :
                }

                RtmReleaseDestInfo(RtmRegHandle, &DestInfoActual);
            }

            ...
        }

        RtmReleaseDests(RtmRegHandle, NumInfos, DestInfos);
    }
    while (Status == NO_ERROR)

    // Close the enumeration and release its resources
    RtmDeleteEnumHandle(RtmRegHandle, EnumHandle);
}
```

Enumerate All Destinations

The following sample code shows how to enumerate all destinations in the routing table.

```
// Macro to allocate a RTM_DEST_INFO on the stack

#define ALLOC_RTM_DEST_INFO(NumViews, NumInfos)
        (PRTM_DEST_INFO) _alloca(RTM_SIZE_OF_DEST_INFO(NumViews) * NumInfos)

// Code to enumerate destinations in the table

MaxHandles = RegnProfile.MaxHandlesInEnum;

DestInfos = ALLOC_RTM_DEST_INFO(NumViews, MaxHandles);

DestInfoSize = RTM_SIZE_OF_DEST_INFO(NumViews);

// Enumerate all destinations in the subtree (0 / 0)
```

(continued)

(continued)

```
// (basically the whole tree; you can
// also achieve this by using RTM_ENUM_START)

RTM_IPV4_MAKE_NET_ADDRESS(&NetAddress,
                          0x00000000,
                          0);

Status = RtmCreateDestEnum(RtmRegHandle,
                           RTM_VIEW_MASK_UCAST | RTM_VIEW_MASK_MCAST,
                           RTM_ENUM_RANGE,
                           &NetAddress,
                           RTM_BEST_PROTOCOL, // Get best route for
                                              // each destination
                           &EnumHandle1);
if (Status == NO_ERROR)
{
    do
    {
        NumInfos = MaxHandles;

        Status = RtmGetEnumDests(RtmRegHandle
                                 EnumHandle1,
                                 &NumInfos,
                                 DestInfos);

        for (i = 0; i < NumInfos; i++)
        {
            DestInfo = (PRTM_DEST_INFO) ((PUCHAR)DestInfos+(i*DestInfoSize));

                        // Process the current destination information

            ASSERT(DestInfo->ViewInfo[0].ViewId == RTM_VIEW_ID_UCAST);
            BestUnicastRoute = DestInfo->ViewInfo[0].Route;

            // Advertise the best unicast route for the destination
            ...

            // You can enumerate all routes for a destination by
            // creating a route enumeration using
            // RtmCreateRouteEnum ( .. DestInfo->DestHandle .. );
        }

        RtmReleaseDests(RtmRegHandle, NumInfos, DestInfos);
    }
    while (Status == NO_ERROR)
    // Close the enumeration and release its resources
    RtmDeleteEnumHandle(RtmRegHandle, EnumHandle1);
}
```

Enumerate All Routes

The following sample code shows how to enumerate all routes in the routing table.

```
MaxHandles = RegnProfile.MaxHandlesInEnum;

RouteHandles = _alloca(MaxHandles * sizeof(HANDLE));

// Do a "route enumeration" over the whole table
// by passing a NULL DestHandle in this function.

DestHandle = NULL; // Give a valid handle to enumerate
                   // over a particular destination

Status = RtmCreateRouteEnum(RtmRegHandle,
                            DestHandle,
                            RTM_VIEW_MASK_UCAST|RTM_VIEW_MASK_MCAST,
                            RTM_ENUM_OWN_ROUTES, // Get only your
                                                 // own routes
                            NULL,
                            0,
                            NULL,
                            0,
                            &EnumHandle2);
if (Status == NO_ERROR)
{
    do
    {
        NumHandles = MaxHandles;

        Status = RtmGetEnumRoutes(RtmRegHandle
                                  EnumHandle2,
                                  &NumHandles,
                                  RouteHandles);

        for (k = 0; k < NumHandles; k++)
        {
            Print("Route %d: %p\n", l++, RouteHandles[k]);

            // Get route information using the route's handle
```

(continued)

(continued)

```
            Status1 = RtmGetRouteInfo(...RouteHandles[k]...);

            if (Status1 == NO_ERROR)
            {
                // Do whatever you want with the route info
                ...

                // Release the route information once you are done
                RtmReleaseRouteInfo(...);
            }
        }

        RtmReleaseRoutes(RtmRegHandle, NumHandles, RouteHandles);
    }
    while (Status == NO_ERROR)

    // Close the enumeration and release its resources
    RtmDeleteEnumHandle(RtmRegHandle, EnumHandle2);
}
```

Search for the Best Route

The following sample code shows how to search the routing table for the best route.

```
// Search for a best matching route to a network
// given (address,mask) for this destination network

// First get the best matching destination

RTM_IPV4_SET_ADDR_AND_MASK(NetAddress, Addr, Mask);

Status = RtmGetMostSpecificDestination(RtmRegHandle,
                                       &NetAddress,
                                       RTM_BEST_PROTOCOL,
                  // Determines which route information is returned.
                                       RTM_VIEW_MASK_UCAST,
                  // Give the information for the best unicast route
                                       DestInfo);
if (Status == NO_ERROR)
{
    ASSERT(DestInfo->ViewInfo[0].ViewId == RTM_VIEW_ID_UCAST);

    // Get the best unicast route for the destination
    NewBestRoute = DestInfo.ViewInfo[0].Route;
    // Call RtmGetRouteInfo using the handle for information
    ...

    // Release information from RtmGetMostSpecificDestination
    RtmReleaseDestInfo(RtmRegHandle, DestInfo);
}

// To search for an exact match, use RtmGetExactMatchDestionation
```

Search for Routes Using RtmGetMostSpecificDestination and RtmGetLessSpecificDestination

The following sample code shows how to use **RtmGetMostSpecificDestination** and **RtmGetLessSpecificDestination** to walk up the prefix tree in the routing table.

```
// Used to walk up the prefix tree, given the destination and mask

// Search for a best matching route to a network
// given (address,mask) for this destination network

// First get the best matching destination

RTM_IPV4_SET_ADDR_AND_MASK(NetAddress, Addr, Mask);

Status = RtmGetMostSpecificDestination(RtmRegHandle,
                                       &NetAddress,
                                       RTM_BEST_PROTOCOL,
                      // Determines which route information is returned.
                                       RTM_VIEW_MASK_UCAST,
                      // Give the information for best unicast route
                                       DestInfo1);
// Use RtmGetLessSpecificDestination to go up the
// tree until it returns ERROR_NOT_FOUND. Use two
// RTM_DEST_INFO buffers - DestInfo1 & DestInfo2
// alternately to avoid all unnecessary copying.

while (Status == NO_ERROR)
{
    if (DestInfo1->DestAddress.NumBits == NetAddress.NumBits)
    {
        Print("Exact Match of Destination\n");
    }

    Status = RtmGetLessSpecificDestination(RtmRegHandle,
```

(continued)

(continued)

```
                                           DestInfo1->DestHandle,
                                           RTM_BEST_PROTOCOL,
                                           RTM_VIEW_MASK_UCAST,
                                           DestInfo2);

    // NOTE that the buffer is DestInfo1 and not DestInfo2
    RtmReleaseDestInfo(RtmRegHandle, DestInfo1);

    if (Status != NO_ERROR)
    {
        break;
    }

    // Use any information you want in DestInfo2
    Print("Mask Len: %d\n", DestInfo2->DestAddress.NumBits);

    Status = RtmGetLessSpecificDestination(RtmRegHandle,
                                           DestInfo2->DestHandle,
                                           RTM_BEST_PROTOCOL,
                                           RTM_VIEW_MASK_UCAST,
                                           DestInfo1);

    // NOTE that the buffer is DestInfo2 and not DestInfo1
    RtmReleaseDestInfo(RtmRegHandle, DestInfo2);

    if (Status != NO_ERROR)
    {
        break;
    }

    // Use any information you want in DestInfo1
    Print("Mask Len: %d\n", DestInfo1->DestAddress.NumBits);
}
```

Access the Opaque Pointers in a Destination

The following sample code shows how to access the opaque pointer in a destination.

```
// Opaque Info Blob Defn

typedef struct _OPAQUE_INFO
{
    ULONG  Info1;
    ULONG  Info2;
}
```

```
OPAQUE_INFO, *POPAQUE_INFO;

PVOID       OpaqueInfoSlotPointer;  // Pointer to the opaque
                                    // pointer slot
PVOID       OpaqueInfoSlotInfo;     // Information in the opaque
                                    // pointer slot
POPAQUE_INFO OpaqueInfoPtr;         // Pointer to opaque information

// Lock the destination in exclusive mode to sync opaque pointer access
// If you know that you will only be reading the opaque pointer
// and not modifying it, then you can use a shared lock

Status = RtmLockDestination(RtmRegHandle, DestHandle, TRUE, TRUE);

if (Status != NO_ERROR)
{
   return Status;
}

// You can get a pointer to your opaque pointer slot,
// assuming that you have reserved one during registration.

Status = RtmGetOpaqueInformationPointer(RtmRegHandle,
                                        DestHandle,
                                        &OpaqueInfoSlotPointer);
if (Status == NO_ERROR)
{
    OpaqueInfoSlotInfo = * (PVOID *) OpaqueInfoSlotPointer;

    if (OpaqueInfoSlotInfo == NULL)
    {
        // No information set yet - create private
        // information BLOB (if required)
        OpaqueInfoPtr = (POPAQUE_INFO) malloc(OpaqueInfoSize);

        if (OpaqueInfoPtr)
        {
            // Set certain information in the opaque information BLOB
            OpaqueInfoPtr->Info1 = 1;
            OpaqueInfoPtr->Info2 = 2;

            * (PVOID *) OpaqueInfoSlotPointer = OpaqueInfoPtr;
        }
        else
        {
```

(continued)

(continued)

```
            // Already exists; do something with the opaque information
            OpaqueInfoPtr = (POPAQUE_INFO) OpaqueInfoSlotInfo;

            // Set certain information in the opaque information BLOB
            OpaqueInfoPtr->Info1 = 3;
            OpaqueInfoPtr->Info2 = 4;
        }
    }
}

// Unlock destination from exclusive mode that we locked earlier
Status = RtmLockDestination(RtmRegHandle, DestHandle, TRUE, FALSE);

ASSERT(Status == NO_ERROR);
```

Use a Client-Specific Route List

The following sample code shows how to create and use a client-specific route list.

```
HANDLE RouteListHandle1;
HANDLE RouteListHandle2;

// Create two entity-specific lists to add routes to

Status = RtmCreateRouteList(RtmRegHandle,
                            &RouteListHandle1);
// Check Status
...

Status = RtmCreateRouteList(RtmRegHandle,
                            &RouteListHandle2);
// Check Status
...

// Assume you have added a bunch of routes
// by calling RtmAddRouteToDest many times
// with 'RouteListHandle1' specified similar to
// Status = RtmAddRouteToDest(RtmRegHandle,
//                              ...
//                              RouteListHandle1,
//                              ...
//                              &ChangeFlags);

// Enumerate routes in RouteListHandle1 list

// Create an enumeration on the route list
```

```
Status = RtmCreateRouteListEnum(RtmRegHandle,
                                RouteListHandle1,
                                &EnumHandle);

if (Status == NO_ERROR)
{
        // Allocate space on the top of the stack

    MaxHandles = RegnProfile.MaxHandlesInEnum;

    RouteHandles = _alloca(MaxHandles * sizeof(HANDLE));

    // Note how the termination condition is different
    // from other enumerations - the call to the function
    // RtmGetListEnumRoutes always returns NO_ERROR
    // Quit when you get fewer handles than requested

    do
    {
                // Get next set of routes from the list enumeration

        NumHandles = MaxHandles;

        Status = RtmGetListEnumRoutes(RtmRegHandle,
                                      EnumHandle,
                                      &NumHandles,
                                      RouteHandles);

        for (i = 0; i < NumHandles; i++)
        {
            Print("Route Handle %5lu: %p\n", i, RouteHandles[i]);
        }

        // Move all these routes to another route list
        // They are automatically removed from the old one

        Status = RtmInsertInRouteList(RtmRegHandle,
                                      RouteListHandle2,
                                      NumHandles,
                                      RouteHandles);
        // Check Status...

                // Release the routes that have been enumerated
```

(continued)

(continued)

```
        RtmReleaseRoutes(RtmRegHandle, NumHandles, RouteHandles);
    }
    while (NumHandles == MaxHandles);

    RtmDeleteEnumHandle(RtmRegHandle, EnumHandle);
}

// Destroy all the entity-specific route lists
// after removing all routes from these lists;
// the routes themselves are not deleted

Status = RtmDeleteRouteList(RtmRegHandle, RouteListHandle1);
ASSERT(Status == NO_ERROR);


Status = RtmDeleteRouteList(RtmRegHandle, RouteListHandle2);
ASSERT(Status == NO_ERROR);
```

Use the Event Notification Callback

The following sample code shows how to process an **RTM_EVENT_CALLBACK** callback received from the routing table manager.

```
// Macro to allocate an RTM_DEST_INFO on the stack

#define ALLOC_RTM_DEST_INFO(NumViews, NumInfos)
        (PRTM_DEST_INFO) _alloca(RTM_SIZE_OF_DEST_INFO(NumViews) * NumInfos)


// Routing table manager entity event callback

DWORD
EntityEventCallback (
    IN      RTM_ENTITY_HANDLE               RtmRegHandle,
    IN      RTM_EVENT_TYPE                  EventType,
    IN      PVOID                 .         Context1,
    IN      PVOID                           Context2
    )
{
    RTM_ENTITY_HANDLE EntityHandle;
    PENTITY_CHARS     EntityChars;
    PRTM_ENTITY_INFO  EntityInfo;
    RTM_NOTIFY_HANDLE NotifyHandle;
    PRTM_DEST_INFO    DestInfos;
    ULONG             DestInfoSize;
    UINT              NumDests;
UINT              NumViews, i;
UINT              MaxHandles;
RTM_ROUTE_HANDLE  RouteHandle;
PRTM_ROUTE_INFO   RoutePointer;
DWORD             ChangeFlags;
DWORD             Status;

Print("\nEvent callback called for %p :", RtmRegHandle);

Print("\n\tEntity Event = ");

Status = ERROR_NOT_SUPPORTED;

switch (EventType)
{
case RTM_ENTITY_REGISTERED:

            // Get the handle and information of the
            // entity that registered

    EntityHandle = (RTM_ENTITY_HANDLE) Context1;
    EntityInfo   = (PRTM_ENTITY_INFO)  Context2;

    Print("Registration\n\tEntity Handle = %p\n\tEntity IdInst = %p\n\n",
          EntityHandle,
          EntityInfo->EntityId);

            // Do something if you are interested in the entity
            ;

    Status = NO_ERROR;
    break;

case RTM_ENTITY_DEREGISTERED:

            // Get the handle and information of
            // the entity that deregistered

    EntityHandle = (RTM_ENTITY_HANDLE) Context1;
    EntityInfo   = (PRTM_ENTITY_INFO)  Context2;

    Print("Deregistration\n\tEntity Handle = %p\n\tEntity IdInst = %p\n\n",
           EntityHandle,
          EntityInfo->EntityId);
```

(continued)

(continued)

```
                // Do something if you are interested in the entity
                ;

            Status = NO_ERROR;
            break;

        case RTM_CHANGE_NOTIFICATION:

            // Get the notification registration on which changes exist and
            // context supplied in RtmRegisterForChangeNotification

            NotifyHandle = (RTM_NOTIFY_HANDLE) Context1;

            NotifyContext = (PVOID) Context2; // Unused

            Print("Changes Available\n\tNotify Handle = %p\n\tNotify Context =
%p\n\n",
                   NotifyHandle,
                   NotifyContext);

            MaxHandles = RegnProfile.MaxHandlesInEnum;

            NumViews = RegnProfile.NumberOfViews;

            DestInfoSize = RTM_SIZE_OF_DEST_INFO(NumViews);

                // Get all changes to destinations

            DestInfos = ALLOC_RTM_DEST_INFO(NumViews, MaxHandles);

            do
            {
                // Try to get as many as possible in one
                // routing table managercall
                NumDests = MaxHandles;

                Status = RtmGetChangedDests(RtmRegHandle,
                                            NotifyHandle,
                                            &NumDests,
                                            DestInfos);

                ASSERT((Status == NO_ERROR) || (Status == ERROR_NO_MORE_ITEMS));

                DestInfo = (PRTM_DEST_INFO) DestInfos;
        for (i = 0; i < NumDests; i++)
        {
            // Process the current destination information

            // Assuming you asked for unicast view information,
            ASSERT(DestInfo->ViewInfo[0].ViewId == RTM_VIEW_ID_UCAST);

            // Get the best unicast route for the destination.
            NewBestRoute = DestInfo.ViewInfo[0].Route;

            // Do whatever you want with above route
            ...

            // Move to the next changed one
            DestInfo = (PRTM_DEST_INFO) ((PUCHAR)DestInfo + DestInfoSize);
        }

        RtmReleaseChangedDests(RtmRegHandle,
                               NotifyHandle,
                               NumDests,
                               DestInfos);
    }
    while (NumDests > 0);

    Status = NO_ERROR;
    break;

case RTM_ROUTE_EXPIRED:

            // Get handle and a direct pointer to
            // the route whose timer expired

    RouteHandle = (RTM_ROUTE_HANDLE) Context1;

    RoutePointer = (PRTM_ROUTE_INFO) Context2;

    Print("Route Aged Out\n\tRoute Handle = %p\n\tRoute Pointer = %p\n\n",
           RouteHandle,
          RoutePointer);

    // To just let the routing table manager delete
    // the route, return ERROR_NOT_SUPPORTED
    // If you return any other value, the routing table
    // manager assumes that you have handled the time-out condition
    // in callback for route timer expiration
```

(continued)

(continued)

```
        // Suppose we just want to update the metric and leave the route

        Status = RtmLockRoute(RtmRegHandle,
                              RouteHandle,
                              TRUE,
                              TRUE,
                              NULL);

        // Check(Status, 46);

        if (Status == NO_ERROR)
        {
            // Set the metric to indicate that it is unreachable
            RoutePointer->PrefInfo.Metric = METRIC_UNREACHABLE;

            Status = RtmUpdateAndUnlockRoute(RtmRegHandle,
                                             RouteHandle,
                                             INFINITE,        // To stay forever
                                             NULL,
                                             0,
                                             NULL,
                                             &ChangeFlags);
            ASSERT(Status == NO_ERROR);
        }

        // If ERROR_NOT_SUPPORTED not returned, release the handle
        RtmReleaseRoutes(RtmRegHandle, 1, &RouteHandle);

        Status = NO_ERROR;
        break;
    }

    return Status;
}
```

Routing Table Manager Version 2 Reference

The following documentation describes the functions, callbacks, structures, and enumeration types to use when interacting with the routing table manager.

Routing Table Manager Version 2 Functions

The following functions are used to interact with the routing table manager.

Registration Functions

RtmGetRegisteredEntities
RtmReleaseEntities
RtmRegisterEntity
RtmDeregisterEntity

Opaque Pointer Functions

RtmLockDestination
RtmGetOpaqueInformationPointer

Export Method Functions

RtmGetEntityMethods
RtmInvokeMethod
RtmBlockMethods

Handle to Information Structure Functions

RtmGetEntityInfo
RtmGetDestInfo
RtmGetRouteInfo
RtmGetNextHopInfo
RtmReleaseEntityInfo
RtmReleaseDestInfo
RtmReleaseRouteInfo
RtmReleaseNextHopInfo

Routing Table Insertion and Deletion Functions

RtmAddRouteToDest
RtmDeleteRouteToDest
RtmHoldDestination
RtmGetRoutePointer
RtmLockRoute
RtmUpdateAndUnlockRoute

Routing Table Query Functions

RtmGetExactMatchDestination
RtmGetMostSpecificDestination
RtmGetLessSpecificDestination
RtmGetExactMatchRoute
RtmIsBestRoute

Next-Hop Insertion and Deletion Functions

RtmAddNextHop
RtmFindNextHop
RtmDeleteNextHop
RtmGetNextHopPointer
RtmLockNextHop

Routing Table Enumeration Functions

RtmCreateDestEnum
RtmGetEnumDests
RtmReleaseDests
RtmCreateRouteEnum
RtmGetEnumRoutes
RtmReleaseRoutes
RtmCreateNextHopEnum
RtmGetEnumNextHops
RtmReleaseNextHops
RtmDeleteEnumHandle

Change Notification Functions

RtmRegisterForChangeNotification
RtmGetChangedDests
RtmReleaseChangedDests
RtmIgnoreChangedDests
RtmGetChangeStatus
RtmMarkDestForChangeNotification
RtmIsMarkedForChangeNotification
RtmDeregisterFromChangeNotification

Route List Function

RtmCreateRouteList
RtmInsertInRouteList
RtmCreateRouteListEnum
RtmGetListEnumRoutes
RtmDeleteRouteList

Handle Management Functions

RtmReferenceHandles

RtmAddNextHop

The **RtmAddNextHop** function adds a new next-hop entry or updates an existing next-hop entry to a client's next-hop list. If a next hop already exists, the routing table manager returns a handle to the next hop. This handle can then be used to specify a next hop to a destination when adding or updating a route.

```
DWORD
RtmAddNextHop (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PRTM_NEXTHOP_INFO NextHopInfo,
  PRTM_NEXTHOP_HANDLE NextHopHandle,
  PRTM_NEXTHOP_CHANGE_FLAGS ChangeFlags
);
```

Parameters

RtmRegHandle

[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NextHopInfo

[in] Pointer to a structure that contains information identifying the next hop to add or update. The **NextHopOwner** and **State** members are ignored; these members are set by the routing table manager. The **Flags** member can be one of the following values.

Flag	Description
RTM_NEXTHOP_FLAGS_REMOTE	This next hop points to a remote destination that is not directly reachable. To obtain the complete path, the client must perform a recursive lookup.
RTM_NEXTHOP_FLAGS_DOWN	This flag is reserved for future use.

NextHopHandle

[in, out] If the client has a handle (client is updating a next hop): On input, *NextHopHandle* is a pointer to the next-hop handle. On output, *NextHopHandle* is unchanged.

If the client does not have a handle and a handle must be returned (client is adding or updating a next hop): On input, *NextHopHandle* is a pointer to NULL. On output, *NextHopHandle* receives a pointer to the next-hop handle. The values in *NextHopInfo* are used to identify the next hop to update.

If a handle does not need to be returned (client is adding or updating a next hop): On input, *NextHopHandle* is NULL. The values in *NextHopInfo* are used to identify the next hop to update.

ChangeFlags

[out] On input, *ChangeFlags* is a pointer to an **RTM_NEXTHOP_CHANGE_FLAGS** structure. On output, *ChangeFlags* receives a flag indicating whether a next hop was added or updated. If *ChangeFlags* is zero, a next hop was updated; if *ChangeFlags is* RTM_NEXTHOP_CHANGE_NEW, a next hop was added.

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_ACCESS_DENIED	The calling client does not own this next hop.
ERROR_NOT_ENOUGH_MEMORY	There is not enough memory to complete this operation.

Remarks

If *NextHopHandle* points to a non-NULL handle, the next hop specified by the handle is updated. Otherwise, a search is made for the address specified by *NextHopInfo*. If a next hop is found, it is updated. If no match is found, a new next hop is added.

If a handle was returned, release the handle when it is no longer required by calling **RtmReleaseNextHops**.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

Next Hop Flags, **RTM_NEXTHOP_INFO**, **RtmDeleteNextHop**, **RtmFindNextHop**, **RtmGetNextHopPointer**, **RtmLockNextHop**, **RtmReleaseNextHops**

RtmAddRouteToDest

The **RtmAddRouteToDest** function adds a new route to the routing table or updates an existing route in the routing table. If the best route changes, a change notification is generated.

```
DWORD
RtmAddRouteToDest (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PRTM_ROUTE_HANDLE RouteHandle,
  PRTM_NET_ADDRESS DestAddress,
  PRTM_ROUTE_INFO RouteInfo,
  ULONG TimeToLive,
  RTM_ROUTE_LIST_HANDLE RouteListHandle,
  RTM_NOTIFY_FLAGS NotifyType,
  RTM_NOTIFY_HANDLE NotifyHandle,
  PRTM_ROUTE_CHANGE_FLAGS ChangeFlags
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteHandle
[in, out] If the client has a handle (updating a route): On input, *RouteHandle* is a pointer to the route handle. On output, *RouteHandle* is unchanged.

If the client does not have a handle and a handle must be returned (client is adding or updating a route): On input, *RouteHandle* is a pointer to NULL. On output, *RouteHandle* receives a pointer to the route handle. The values in *RouteInfo* are used to identify the route to update.

If a handle does not need to be returned (client is adding or updating a route): On input, *RouteHandle* is NULL. The values in *RouteInfo* are used to identify the route to update.

DestAddress
[in] Pointer to the destination network address to which the route is being added or updated.

RouteInfo
[in] Pointer to the route information to add or update.

TimeToLive
[in] Specifies the time (in milliseconds) after which the route is expired. Specify INFINITE to prevent routes from expiring.

RouteListHandle
[in] Handle to a route list to which to move the route. This parameter is optional and can be set to NULL.

NotifyType
[in] Set this parameter to NULL. *NotifyType* is reserved for future use.

NotifyHandle
[in] Set this parameter to NULL. *NotifyHandle* is reserved for future use.

ChangeFlags
[in, out] On input, *ChangeFlags* is a pointer to an **RTM_ROUTE_CHANGE_FLAGS** structure indicating whether the routing table manager should add a new route or update an existing one. On output, *ChangeFlags* is a pointer to an **RTM_ROUTE_CHANGE_FLAGS** structure that receives the flag indicating the type of change that was actually performed, and if the best route was changed. The following flags are used.

Constant	Description
RTM_ROUTE_CHANGE_FIRST	Indicates that the routing table manager should not check the **Neighbour** member of the *RouteInfo* parameter when determining if two routes are equal.
RTM_ROUTE_CHANGE_NEW	Returned by the routing table manager to indicate a new route was created.
RTM_ROUTE_CHANGE_BEST	Returned by the routing table manager to indicate that the route that was added or updated was the best route, or that because of the change, a new route became the best route.

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_ACCESS_DENIED	The calling client does not own this route.
ERROR_INVALID_HANDLE	The handle is invalid.
ERROR_INVALID_PARAMETER	A parameter contains incorrect information.
ERROR_NOT_ENOUGH_MEMORY	There is not enough memory to complete this operation.

Remarks

Two routes are considered equal if the following values are equal:

- The destination network
- The owner of the route
- The neighbor that supplied the route

When a client is updating a route, it is more efficient to pass a handle to the route to update in the *RouteHandle* parameter, because the routing table manager does not have to perform a search for the route in the routing table.

If a handle was returned, release the handle when it is no longer required by calling **RtmReleaseRoutes**.

For sample code using this function, see *Add and Update Routes Using RtmAddRouteToDest*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

+ See Also

RTM_NET_ADDRESS, **RTM_ROUTE_INFO**, **RtmDeleteRouteToDest**, **RtmGetRoutePointer**, **RtmHoldDestination**, **RtmLockRoute**, **RtmReleaseRoutes**, **RtmUpdateAndUnlockRoute**

RtmBlockMethods

The **RtmBlockMethods** function blocks or unblocks the execution of methods for a specified destination, route, or next hop, or for all destinations, routes, and next hops.

```
DWORD
RtmBlockMethods (
  RTM_ENTITY_HANDLE RtmRegHandle,
  HANDLE TargetHandle,
  UCHAR TargetType,
  DWORD BlockingFlag
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

TargetHandle
[in] Handle to a destination, route, or next hop for which to block methods. This parameter is optional and can be set to NULL to block methods for all targets.

TargetType
[in] Specifies the type of the handle in *TargetHandle*. This parameter is optional and can be set to NULL to block methods for all targets. The following flags are used.

Type	Description
DEST_TYPE	*TargetHandle* is a destination.
NEXTHOP_TYPE	*TargetHandle* is a next hop.
ROUTE_TYPE	*TargetHandle* is a route.

BlockingFlag
[in] Specifies whether to block or unblock methods. The following flags are used.

Constant	Description
RTM_BLOCK_METHODS	Block methods for the specified target.
RTM_RESUME_METHODS	Unblock methods for the specified target.

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_HANDLE	The handle is invalid.

Remarks

Currently, this function does not support blocking methods for a specific destination, route, or next hop.

Methods are typically blocked when client-specific data in the route is being changed; a client blocks methods, rearranges data, and then unblocks methods.

Clients should only block methods for a short period of time. If a second client calls **RtmInvokeMethod** and the first client's methods are blocked, **RtmInvokeMethod** will not return until methods are unblocked and the function call is completed.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmGetEntityMethods, **RtmInvokeMethod**

RtmCreateDestEnum

The **RtmCreateDestEnum** function starts an enumeration of the destinations in the routing table. A client can enumerate destinations for one or more views, or for all views.

```
DWORD
RtmCreateDestEnum (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_VIEW_SET TargetViews,
  RTM_ENUM_FLAGS EnumFlags,
  PRTM_NET_ADDRESS NetAddress,
  ULONG ProtocolId,
  PRTM_ENUM_HANDLE RtmEnumHandle
);
```

Parameters

RtmRegHandle

[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

TargetViews

[in] Specifies the set of views to use when creating the enumeration. The following flags are used.

Constant	Description
RTM_VIEW_MASK_ANY	Return destinations from all views. This is the default value.
RTM_VIEW_MASK_UCAST	Return destinations from the unicast view.
RTM_VIEW_MASK_MCAST	Return destinations from the multicast view.

EnumFlags

[in] Specifies which destinations to include in the enumeration. Two sets of flags are used; use one flag from each set (for example, use RTM_ENUM_ALL_DESTS and RTM_ENUM_START).

Constant	Description
RTM_ENUM_ALL_DESTS	Return all destinations.
RTM_ENUM_OWN_DESTS	Return destinations for which the client owns the best route to a destination in any of the specified views.

Constant	Description
RTM_ENUM_NEXT	Enumerate destinations starting at the specified address/mask length (such as 10/8). The enumeration continues to the end of the routing table.
RTM_ENUM_RANGE	Enumerate destinations in the range specified by the address/mask length (such as 10/8).
RTM_ENUM_START	Enumerate destinations starting at 0/0. Specify NULL for *NetAddress*.

NetAddress

[in] Pointer to an **RTM_NET_ADDRESS** structure that contains the starting address of the enumeration. Specify NULL if *EnumFlags* contains RTM_ENUM_START.

ProtocolId
[in] Specifies the protocol identifier used to determine the best route information returned by the **RtmGetEnumDests** function. The *ProtocolID* is not part of the search criteria. The routing table manager uses this identifier to determine which route information to return (for example, if a client specifies the RIP protocol identifier, the best RIP route is returned, even if a non-RIP route is the best route to the destination).

Specify RTM_BEST_PROTOCOL to return a route regardless of which protocol owns it. Specify RTM_THIS_PROTOCOL to return the best route for the calling protocol.

RtmEnumHandle
[out] On input, *RtmEnumHandle* is a pointer to NULL. On output, *RtmEnumHandle* receives a pointer to a handle to the enumeration. Use this handle in all subsequent calls to **RtmGetEnumDests**, **RtmReleaseDests**, and **RtmDeleteEnumHandle**.

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	A parameter contains incorrect information.
ERROR_NOT_ENOUGH_MEMORY	There is not enough memory to complete this operation.
ERROR_NOT_SUPPORTED	One or more of the specified views is not supported.

Remarks

If *EnumFlags* contains RTM_ENUM_RANGE, use *NetAddress* to specify the range of the routing table to enumerate. For example, if a client sets *NetAddress* to 10/8, destinations in the range 10.0.0.0/8 to 10.255.255.255/32 are returned.

When the enumeration handle is no longer required, release it by calling **RtmDeleteEnumHandle**.

For sample code using this function, see *Enumerate All Destinations*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_NET_ADDRESS , **RtmDeleteEnumHandle**, **RtmGetEnumDests**, **RtmReleaseDests**

RtmCreateNextHopEnum

The **RtmCreateNextHopEnum** enumerates the next hops in the next-hop list.

```
DWORD
RtmCreateNextHopEnum (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ENUM_FLAGS EnumFlags,
  PRTM_NET_ADDRESS NetAddress,
  PRTM_ENUM_HANDLE RtmEnumHandle
);
```

Parameters

RtmRegHandle

[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

EnumFlags

[in] Specifies which next hops to include in the enumeration. The following flags are used.

Constant	Description
RTM_ENUM_NEXT	Enumerate next hops starting at the specified address/mask length (such as 10/8). The enumeration continues to the end of the next hop list.
RTM_ENUM_RANGE	Enumerate next hops in the specified range specified by the address/mask length (such as 10/8).
RTM_ENUM_START	Enumerate next hops starting at 0/0. Specify NULL for *NetAddress*.

NetAddress

[in] Pointer to an **RTM_NET_ADDRESS** structure that contains the starting address of the enumeration. Specify NULL if *EnumFlags* contains RTM_ENUM_START.

RtmEnumHandle

[out] On input, *RtmEnumHandle* is a pointer to NULL. On output, *RtmEnumHandle* receives a pointer to a handle to the enumeration. Use this handle in all subsequent calls to **RtmGetEnumNextHops**, **RtmReleaseNextHops**, and **RtmDeleteEnumHandle**.

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	A parameter contains incorrect information.
ERROR_NOT_ENOUGH_MEMORY	There is not enough memory to complete this operation.

Remarks

If *EnumFlags* contains RTM_ENUM_RANGE, use *NetAddress* to specify the range of the routing table to enumerate. For example, if a client sets *NetAddress* to 10/8, next hops in the range 10.0.0.0/8 to 10.255.255.255/32 are returned.

When the enumeration handle is no longer required, release it by calling **RtmDeleteEnumHandle**.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_NET_ADDRESS, **RtmDeleteEnumHandle**, **RtmGetEnumNextHops**, **RtmReleaseNextHops**

RtmCreateRouteEnum

The **RtmCreateRouteEnum** function creates an enumeration of the routes for a particular destination or range of destinations in the routing table. A client can enumerate routes for one or more views, or for all views.

```
DWORD
RtmCreateRouteEnum (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_DEST_HANDLE DestHandle,
  RTM_VIEW_SET TargetViews,
  RTM_ENUM_FLAGS EnumFlags,
  PRTM_NET_ADDRESS StartDest,
  RTM_MATCH_FLAGS MatchingFlags,
  PRTM_ROUTE_INFO CriteriaRoute,
  ULONG CriteriaInterface,
  PRTM_ENUM_HANDLE RtmEnumHandle
);
```

Parameters

RtmRegHandle

[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

DestHandle

[in] Handle to the destination for which to enumerate routes. This parameter is optional, and can be set to NULL; specifying NULL enumerates all routes for all destinations. Specify NULL if *EnumFlags* contains RTM_ENUM_START.

TargetViews

[in] Specifies the set of views to use when creating the enumeration. The following flags are used.

Constant	Description
RTM_VIEW_MASK_ANY	Return destinations from all views. This is the default value.
RTM_VIEW_MASK_UCAST	Return destinations from the unicast view.
RTM_VIEW_MASK_MCAST	Return destinations from the multicast view.

EnumFlags

[in] Specifies which routes to include in the enumeration. Two sets of flags are used; use one flag from each set (such as RTM_ENUM_ALL_ROUTES and RTM_ENUM_START).

Constant	Description
RTM_ENUM_ALL_ROUTES	Return all routes.
RTM_ENUM_OWN_ROUTES	Return only those routes that the client owns.

Constant	Description
RTM_ENUM_NEXT	Enumerate routes starting at the specified address/mask length (such as 10/8). The enumeration continues to the end of the routing table.
RTM_ENUM_RANGE	Enumerate routes in the specified range specified by the address/mask length (such as 10/8).
RTM_ENUM_START	Enumerate routes starting at 0/0. Specify NULL for *NetAddress*.

StartDest

[in] Pointer to an **RTM_NET_ADDRESS** structure that contains the starting address of the enumeration. This parameter is ignored if *EnumFlags* contains RTM_ENUM_START.

MatchingFlags

[in] Specifies the elements of the route to match. Only routes that match the criteria specified in *CriteriaRoute* and *CriteriaInterface* are returned, unless otherwise noted. The following flags are used.

Constant	Description
RTM_MATCH_FULL	Match routes with all criteria.
RTM_MATCH_INTERFACE	Match routes that are on the same interface. The client can specify NULL for *CriteriaRoute.*
RTM_MATCH_NEIGHBOUR	Match routes with the same neighbor.
RTM_MATCH_NEXTHOP	Match routes that have the same next hop.
RTM_MATCH_NONE	Match none of the criteria; all routes for the destination are returned. The *CriteriaRoute* parameter is ignored if this flag is set.
RTM_MATCH_OWNER	Match routes with same owner.
RTM_MATCH_PREF	Match routes that have the same preference.

CriteriaRoute

[in] Specifies which routes to enumerate. This parameter is optional and can be set to NULL if *MatchingFlags* contains RTM_MATCH_INTERFACE or RTM_MATCH_NONE.

CriteriaInterface

[in] Pointer to a ULONG that specifies on which interfaces routes should be located. This parameter is ignored unless *MatchingFlags* contains RTM_MATCH_INTERFACE.

RtmEnumHandle

[out] On input, *RtmEnumHandle* is a pointer to NULL. On output, *RtmEnumHandle* receives a pointer to a handle to the enumeration. Use this handle in all subsequent calls to **RtmGetEnumRoutes**, **RtmReleaseRoutes**, and **RtmDeleteEnumHandle**.

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	A parameter contains incorrect information.
ERROR_NOT_ENOUGH_MEMORY	There is not enough memory to complete this operation.
ERROR_NOT_SUPPORTED	One or more of the specified views is not supported.

Remarks

If *EnumFlags* contains RTM_ENUM_RANGE, use *NetAddress* to specify the range of the routing table to enumerate. For example, if a client sets *NetAddress* to 10/8, destinations in the range 10.0.0.0/8 to 10.255.255.255/32 are returned.

When the enumeration handle is no longer required, release it by calling **RtmDeleteEnumHandle**.

For sample code using this function, see *Enumerate All Routes*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

+ See Also

RTM_NET_ADDRESS, **RTM_ROUTE_INFO**, **RtmDeleteEnumHandle**, **RtmGetEnumRoutes**, **RtmReleaseRoutes**

RtmCreateRouteList

The **RtmCreateRouteList** function creates a list in which the caller can keep a copy of the routes it owns.

```
DWORD
RtmCreateRouteList (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PRTM_ROUTE_LIST_HANDLE RouteListHandle
);
```

Parameters

RtmRegHandle
: [in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteListHandle
: [out] On input, *RouteListHandle* is a pointer to NULL. On output, *RouteListHandle* receives a pointer to a handle to the new route list.

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_NOT_ENOUGH_MEMORY	There is not enough memory to complete this operation.

Remarks

For sample code using this function, see *Use a Client-Specific Route List*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmDeleteRouteList, **RtmInsertInRouteList**

RtmCreateRouteListEnum

The **RtmCreateRouteListEnum** function creates an enumeration of routes on the specified route list.

```
DWORD
RtmCreateRouteListEnum (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ROUTE_LIST_HANDLE RouteListHandle,
  PRTM_ENUM_HANDLE RtmEnumHandle
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteListHandle
[in] Handle to the route list to enumerate obtained from a previous call to **RtmCreateRouteList**.

RtmEnumHandle
[out] On input, *RtmEnumHandle* is a pointer to NULL. On output, *RtmEnumHandle* receives a pointer to a handle to the enumeration. Use this handle in all subsequent calls to functions that enumerate the list of routes.

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_NOT_ENOUGH_MEMORY	There is not enough memory to complete this operation.

Remarks

When the enumeration handle is no longer required, release it by calling **RtmDeleteEnumHandle**.

For sample code using this function, see *Use a Client-Specific Route List*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmDeleteEnumHandle, **RtmGetListEnumRoutes**

RtmDeleteEnumHandle

The **RtmDeleteEnumHandle** function deletes the specified enumeration handle and frees all resources allocated for the enumeration.

```
DWORD
RtmDeleteEnumHandle (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ENUM_HANDLE EnumHandle
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

EnumHandle
[in] Handle to be released. Any resources associated with the handle are also freed.

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_HANDLE	The handle is invalid.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

+ See Also

RtmCreateDestEnum, **RtmCreateNextHopEnum**, **RtmCreateRouteEnum**, **RtmCreateRouteListEnum**, **RtmGetEnumDests**, **RtmGetEnumNextHops**, **RtmGetEnumRoutes**, **RtmReleaseDests**, **RtmReleaseNextHops**, **RtmReleaseRoutes**

RtmDeleteNextHop

The **RtmDeleteNextHop** function deletes a next hop from the next-hop list.

```
DWORD
RtmDeleteNextHop (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_NEXTHOP_HANDLE NextHopHandle,
  PRTM_NEXTHOP_INFO NextHopInfo
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NextHopHandle
[in] Handle to the next hop to delete. This parameter is optional and can be set to NULL; if it is NULL, the values in *NextHopInfo* are used to identify the next hop to delete.

NextHopInfo
[in] Pointer to a structure that contains information identifying the next hop to delete. This parameter is optional and can be set to NULL; if it is NULL, the handle in *NextHopHandle* is used to identify the next hop to delete.

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_ACCESS_DENIED	The calling client does not own this next hop.
ERROR_NOT_ENOUGH_MEMORY	There is not enough memory to complete this operation.
ERROR_NOT_FOUND	The specified next hop was not found.

Remarks

If a client specifies a *NextHopHandle*, the client should not subsequently release the handle using **RtmReleaseNextHops**.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_NEXTHOP_INFO, **RtmAddNextHop**, **RtmFindNextHop**, **RtmGetNextHopPointer**

RtmDeleteRouteList

The **RtmDeleteRouteList** function removes all routes from a client-specific route list, then frees any resources allocated to the list.

```
DWORD
RtmDeleteRouteList (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ROUTE_LIST_HANDLE RouteListHandle
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteListHandle
[in] Handle to the route list to delete.

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_HANDLE	The handle is invalid.

Remarks

This function also releases the handle to the route list.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmCreateRouteList, **RtmInsertInRouteList**

RtmDeleteRouteToDest

The **RtmDeleteRouteToDest** function deletes a route from the routing table and updates the best-route information for the corresponding destination, if the best route changed. If the best route changes, a change notification is generated.

```
DWORD
RtmDeleteRouteToDest (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ROUTE_HANDLE RouteHandle,
  PRTM_ROUTE_CHANGE_FLAGS ChangeFlags
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteHandle
[in] Handle to the route to delete.

ChangeFlags
[out] On input, *ChangeFlags* is a pointer to RTM_ROUTE_CHANGE_FLAGS. On output, *ChangeFlags* receives RTM_ROUTE_CHANGE_BEST if the best route was changed.

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_ACCESS_DENIED	The calling client does not own this route.
ERROR_INVALID_HANDLE	The handle is invalid.
ERROR_NOT_FOUND	The specified route was not found.

Remarks

The *RouteHandle* should not subsequently be released by a client if the client has already called **RtmDeleteRouteToDest** using that handle. The **RtmDeleteRouteToDest** function deletes the route and releases the handle.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmAddRouteToDest, **RtmGetRoutePointer**, **RtmHoldDestination**, **RtmLockRoute**, **RtmUpdateAndUnlockRoute**

RtmDeregisterEntity

The **RtmDeregisterEntity** function unregisters a client from a routing table manager instance and address family.

```
DWORD
RtmDeregisterEntity (
  RTM_ENTITY_HANDLE RtmRegHandle
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

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_HANDLE	The handle is invalid.

Remarks

Before calling this function, the client must ensure that all locks, handles, and information structures are released with the appropriate functions.

When the client calls **RtmDeregisterEntity**, the handle that was returned by a previous call to **RtmRegisterEntity** is released. The client must not call any RTMv2 functions after releasing this handle.

If the client does call any functions that access the routing table manager after the client has unregistered, the client's process may be terminated.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmRegisterEntity

RtmDeregisterFromChangeNotification

The **RtmDeregisterFromChangeNotification** function unregisters a client from change notification and frees all resources allocated to the notification.

```
DWORD
RtmDeregisterFromChangeNotification (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_NOTIFY_HANDLE NotifyHandle
);
```

Parameters

RtmRegHandle
: [in] Handle to the client obtained from a previous call to **RtmRegisterForChangeNotification**.

NotifyHandle
: [in] Handle to the change notification to unregister, obtained from a previous call to **RtmRegisterForChangeNotification**.

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_HANDLE	The handle is invalid.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

+ See Also

RtmMarkDestForChangeNotification, **RtmRegisterForChangeNotification**, **RtmReleaseChangedDests**

RtmFindNextHop

The **RtmFindNextHop** function finds a specific next hop in a client's next-hop list.

```
DWORD
RtmFindNextHop (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PRTM_NEXTHOP_INFO NextHopInfo,
  PRTM_NEXTHOP_HANDLE NextHopHandle,
  PRTM_NEXTHOP_INFO *NextHopPointer
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NextHopInfo
[in] Pointer to an **RTM_NEXTHOP_INFO** structure that contains information identifying the next hop to find. Use the **NextHopAddress** and **InterfaceIndex** members to identify the next hop to find.

NextHopHandle
[out] If a handle must be returned: On input, *NextHopPointer* is a pointer to NULL. On output, if the client owns the next hop, *NextHopPointer* receives a pointer to the next-hop handle; otherwise, *NextHopPointer* remains unchanged.

If a handle does not need to be returned: On input, *NextHopPointer* is NULL.

NextHopPointer
[out] If a pointer must be returned: On input, *NextHopPointer* is a pointer to NULL. On output, if the client owns the next hop, *NextHopPointer* receives a pointer to the next-hop; otherwise, *NextHopPointer* remains unchanged.

If a pointer does not need to be returned: On input, *NextHopPointer* is NULL.

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_ACCESS_DENIED	The calling client does not own this next hop.
ERROR_NOT_FOUND	The specified next hop was not found.

Remarks

The *NextHopPointer* is valid as long as the client has not released *NextHopHandle*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_NEXTHOP_INFO, **RtmAddNextHop**, **RtmDeleteNextHop**, **RtmGetNextHopPointer**, **RtmLockNextHop**

RtmGetChangedDests

The **RtmGetChangedDests** function returns a set of destinations with changed information.

```
DWORD
RtmGetChangedDests (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_NOTIFY_HANDLE NotifyHandle,
  PUINT NumDests,
  PRTM_DEST_INFO ChangedDests
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NotifyHandle
[in] Handle to a change notification obtained from a previous call to **RtmRegisterForChangeNotification**.

NumDests
[in, out] On input, *NumDests* is a pointer to a **UINT** value specifying the maximum number of destinations that can be received by *ChangedDests*. On output, *NumDests* receives the actual number of destinations received by *ChangedDests*.

ChangedDests
[out] On input, *ChangedDests* is a pointer to an array of **RTM_DEST_INFO** structures. On output, *ChangedDests* is filled with the changed destination information.

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	A parameter contains incorrect information.
ERROR_NO_MORE_ITEMS	No more changed destinations to retrieve.

Remarks

A client is notified of changes by an **RTM_EVENT_CALLBACK**. The **RTM_EVENT_CALLBACK** is only used to notify the client, not deliver the changes. After a change notification is received, the client must call **RtmGetChangedDests** repeatedly to retrieve all the changes.

If two or more changes to the same destination have occurred since the notification, only the latest change is returned.

When a client no longer needs the handles in *ChangedDests*, the client must use **RtmReleaseChangedDests** to release the handles.

For sample code using this function, see *Use the Event Notification Callback*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_EVENT_CALLBACK, **RTM_DEST_INFO**, **RtmGetChangeStatus**, **RtmIgnoreChangedDests**, **RtmIsMarkedForChangeNotification**, **RtmMarkDestForChangeNotification**, **RtmReleaseChangedDests**

RtmGetChangeStatus

The **RtmGetChangeStatus** function checks whether there are pending changes that have not been retrieved with **RtmGetChangedDests**.

```
DWORD
RtmGetChangeStatus (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_NOTIFY_HANDLE NotifyHandle,
  RTM_DEST_HANDLE DestHandle,
  PBOOL ChangeStatus
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NotifyHandle
[in] Handle to a change notification.

DestHandle
[in] Handle to the destination for which to return change status.

ChangeStatus
[out] On input, *ChangeStatus* is a pointer to a **BOOL** value. On output, *ChangeStatus* receives either TRUE or FALSE to indicate if the destination specified by *DestHandle* has a change notification pending.

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_HANDLE	The handle is invalid.

Remarks

This function can be used to make portions of the client code more efficient. For example, a client may postpone some operation if there are changes that the client has not yet processed.

This function can also be used to monitor change notification in another thread.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_EVENT_CALLBACK, **RtmGetChangedDests**, **RtmIgnoreChangedDests**, **RtmIsMarkedForChangeNotification**, **RtmMarkDestForChangeNotification**, **RtmReleaseChangedDests**

RtmGetDestInfo

The **RtmGetDestInfo** function returns information about a destination.

```
DWORD
RtmGetDestInfo (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_DEST_HANDLE DestHandle,
  ULONG ProtocolId,
  RTM_VIEW_SET TargetViews,
  PRTM_DEST_INFO DestInfo
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

DestHandle
[in] Handle to the destination for which to return information.

ProtocolId
[in] Specifies the protocol identifier. The *ProtocolID* is not part of the search criteria. The routing table manager uses this identifier to determine which route information to return (for example, if a client specifies the RIP protocol identifier, the best RIP route is returned, even if a non-RIP route is the best route to the destination).

Specify RTM_BEST_PROTOCOL to return a route regardless of which protocol owns it. Specify RTM_THIS_PROTOCOL to return the best route for the calling protocol.

TargetViews
[in] Specifies the views from which to return information. If the client specifies RTM_VIEW_MASK_ANY, destination information is returned from all views; however, no view-specific information is returned.

DestInfo
[out] On input, *DestInfo* is a pointer to an **RTM_DEST_INFO** structure. On output, *DestInfo* is filled with the requested destination information.

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_HANDLE	The handle is invalid.

Remarks

The *DestInfo* structure is a variable-sized structure. If the client specifies more than one view with *TargetViews*, the size of *DestInfo* increases for each view. Use the **RTM_SIZE_OF_DEST_INFO** macro to determine how large a *DestInfo* structure to allocate before calling this function. Use the value specified for *TargetViews* as a parameter to **RTM_SIZE_OF_DEST_INFO**.

Use **RtmReleaseDestInfo** to release the *DestInfo* buffer.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_DEST_INFO, **RtmReleaseDestInfo**

RtmGetEntityInfo

The **RtmGetEntityInfo** function returns information about a previously registered client.

```
DWORD
RtmGetEntityInfo (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ENTITY_HANDLE EntityHandle,
  PRTM_ENTITY_INFO EntityInfo
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

EntityHandle
[in] Handle to the client for which to return information.

EntityInfo
[out] On input, *EntityInfo* is a pointer to an **RTM_ENTITY_INFO** structure. On output, *EntityInfo* is filled with the requested information.

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_HANDLE	The handle is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_ENTITY_INFO, **RtmReleaseEntityInfo**

RtmGetEntityMethods

The **RtmGetEntityMethods** function queries the specified client to determine which methods are available for another client to invoke.

```
DWORD
RtmGetEntityMethods (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ENTITY_HANDLE EntityHandle,
  PUINT NumMethods,
  PRTM_ENTITY_EXPORT_METHOD ExptMethods
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

EntityHandle
[in] Handle to the client for which to obtain methods.

NumMethods
[in, out] On input, *NumMethods* specifies a valid pointer to a **UINT** value. Specify zero to return the number of methods available to be exported. On output, *NumMethods* receives the number of methods exported by the client.

ExptMethods
[out] Receives a pointer to an **RTM_ENTITY_EXPORT_METHOD** structure containing the set of method identifiers requested by the calling client.

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_INSUFFICIENT_BUFFER	The buffer supplied is not large enough to hold all the requested information.

Remarks

Do not call the another client's method directly, always use **RtmInvokeMethod**. The routing table manager performs error checking when using **RtmInvokeMethod** to ensure that the client is not unregistering or already unregistered.

If ERROR_INSUFFICIENT_BUFFER is returned, there may be some data in *ExptMethods*; *NumMethods* specifies how many methods actually fit in the buffer.

When the entity handle is no longer required, release it by calling **RtmReleaseEntities**.

For sample code using this function, see *Obtain and Call the Exported Methods for a Client*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmBlockMethods, **RtmInvokeMethod**

RtmGetEnumDests

The **RtmGetEnumDests** function retrieves the next set of destinations in the specified enumeration.

```
DWORD
RtmGetEnumDests (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ENUM_HANDLE EnumHandle,
  PUINT NumDests,
  PRTM_DEST_INFO DestInfos
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

EnumHandle
[in] Handle to the destination enumeration.

NumDests
[in, out] On input, *NumDests* is a pointer to a **UINT** value specifying the maximum number of destinations that can be received by *DestInfos*. On output, *NumDests* receives the actual number of destinations received by *DestInfos*.

DestInfos
[out] On input, *DestInfo* is a pointer to an **RTM_DEST_INFO** structure. On output, *DestInfo* receives an array of handles to destinations.

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 value pointed to by *NumRoutes* is larger than the maximum number of routes a client is allowed to retrieve with one call. Check **RTM_REGN_PROFILE** for the maximum number of destinations that the client is allowed to retrieve with one call.
ERROR_NO_MORE_ITEMS	There are no more destinations to enumerate.

Remarks

The *DestInfo* structure is a variable-sized structure. If the client specifies more than one view with *TargetViews*, the size of *DestInfo* increases for each view. Use the **RTM_SIZE_OF_DEST_INFO** macro to determine how large a *DestInfo* structure to allocate before calling this function. Use the value specified for *TargetViews* as a parameter to **RTM_SIZE_OF_DEST_INFO**.

When the destinations are no longer required, release them by calling **RtmReleaseDests**.

For sample code using this function, see *Enumerate All Destinations.*

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

+ See Also

RTM_DEST_INFO, **RtmCreateDestEnum**, **RtmDeleteEnumHandle**, **RtmReleaseDests**

RtmGetEnumNextHops

The **RtmGetEnumNextHops** function retrieves the next set of next hops in the specified enumeration.

```
DWORD
RtmGetEnumNextHops (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ENUM_HANDLE EnumHandle,
  PUINT NumNextHops,
  PRTM_NEXTHOP_HANDLE NextHopHandles
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

EnumHandle
[in] Handle to the next-hop enumeration.

NumNextHops
[in, out] On input, *NumNextHops* is a pointer to a **UINT** value specifying the maximum number of next hops that can be received by *NextHopHandles*. On output, *NumNextHops* receives the actual number of next hops received by *NextHopHandles*.

NextHopHandles
[out] On input, *NextHopHandles* pointers to an **RTM_NEXTHOP_INFO** structure. On output, *NextHopHandles* receives an array of handles to next hops.

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 value pointed to by *NumRoutes* is larger than the maximum number of routes a client is allowed to retrieve with one call. Check **RTM_REGN_PROFILE** for the maximum number of next hops that the client is allowed to retrieve with one call.
ERROR_NO_MORE_ITEMS	There are no more next hops to enumerate.

Remarks

When the next hops are no longer required, release them by calling **RtmReleaseNextHops**.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmCreateNextHopEnum, **RtmDeleteEnumHandle**, **RtmReleaseNextHops**

RtmGetEnumRoutes

The **RtmGetEnumRoutes** function retrieves the next set of routes in the specified enumeration.

```
DWORD
RtmGetEnumRoutes (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ENUM_HANDLE EnumHandle,
  PUINT NumRoutes,
  PRTM_ROUTE_HANDLE RouteHandles
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

EnumHandle
[in] Handle to the route enumeration.

NumRoutes
[in, out] On input, *NumRoutes* is a pointer to a **UINT** value specifying the maximum number of routes that can be received by *RouteHandles*. On output, *NumRoutes* receives the actual number of routes received by *RouteHandles*.

RouteHandles
[out] On input, *RouteHandles* is a pointer to an **RTM_ROUTE_INFO** structure. On output, *RouteHandles* receives an array of handles to routes.

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 value pointed to by *NumRoutes* is larger than the maximum number of routes a client is allowed to retrieve with one call. Check **RTM_REGN_PROFILE** for the maximum number of routes that the client is allowed to retrieve with one call.
ERROR_NO_MORE_ITEMS	There are no more routes to enumerate.
ERROR_NOT_ENOUGH_MEMORY	There is not enough memory to complete this operation.

Remarks

When the routes are no longer required, release them by calling **RtmReleaseRoutes**.

For sample code using this function, see *Enumerate All Routes*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmCreateRouteEnum, **RtmDeleteEnumHandle**, **RtmReleaseRoutes**

RtmGetExactMatchDestination

The **RtmGetExactMatchDestination** function searches the routing table for a destination that exactly matches the specified network address and subnet mask. If an exact match is found, the information for that destination is returned.

```
DWORD
RtmGetExactMatchDestination (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PRTM_NET_ADDRESS DestAddress,
  ULONG ProtocolId,
  RTM_VIEW_SET TargetViews,
  PRTM_DEST_INFO DestInfo
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

DestAddress
[in] Pointer to the destination network address.

ProtocolId
[in] Specifies the protocol identifier. The *ProtocolID* is not part of the search criteria. The routing table manager uses this identifier to determine which destination and route information to return (for example, if a client specifies the RIP protocol identifier, the best RIP route is returned, even if a non-RIP route is the best route to the destination).

Specify RTM_BEST_PROTOCOL to return a route regardless of which protocol owns it. Specify RTM_THIS_PROTOCOL to return the best route for the calling protocol.

TargetViews
[in] Specifies the views to return information from. If the client specifies RTM_VIEW_MASK_ANY, destination information is returned from all views; however, no view-specific information is returned.

DestInfo
[out] On input, *DestInfo* is a pointer to an **RTM_DEST_INFO** structure. On output, *DestInfo* is filled with the requested destination information.

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_NOT_FOUND	The specified destination was not found.

Remarks

The *DestInfo* structure is a variable-sized structure. If the client specifies more than one view with *TargetViews*, the size of *DestInfo* increases for each view. Use the **RTM_SIZE_OF_DEST_INFO** macro to determine how large a *DestInfo* structure to allocate before calling this function. Use the value specified for *TargetViews* as a parameter to **RTM_SIZE_OF_DEST_INFO**.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_DEST_INFO, **RTM_NET_ADDRESS**, **RtmGetExactMatchRoute**, **RtmGetLessSpecificDestination**, **RtmGetMostSpecificDestination**, **RtmIsBestRoute**

RtmGetExactMatchRoute

The **RtmGetExactMatchRoute** function searches the routing table for a route that exactly matches the specified route (indicated by a network address, subnet mask, and other route-matching criteria). If an exact match is found, the route information is returned.

```
DWORD
RtmGetExactMatchRoute (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PRTM_NET_ADDRESS DestAddress,
  RTM_MATCH_FLAGS MatchingFlags,
  PRTM_ROUTE_INFO RouteInfo,
  ULONG InterfaceIndex,
  RTM_VIEW_SET TargetViews,
  PRTM_ROUTE_HANDLE RouteHandle
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

DestAddress
[in] Pointer to the destination network address.

MatchingFlags
[in] Specifies the criteria to use when searching for the route. The following flags are used.

Constant	Description
RTM_MATCH_FULL	Match routes with all criteria.
RTM_MATCH_INTERFACE	Match routes that are on the same interface.
RTM_MATCH_NEIGHBOUR	Match routes with the same neighbor.
RTM_MATCH_NEXTHOP	Match routes that have the same next hop.
RTM_MATCH_NONE	Match none of the criteria; all routes for the destination are returned.
RTM_MATCH_OWNER	Match routes with the same owner.
RTM_MATCH_PREF	Match routes that have the same preference.

RouteInfo
[in, out] On input, *RouteInfo* is a pointer an **RTM_ROUTE_INFO** structure containing the criteria that specifies the route to find. On output, *RouteInfo* receives the route information for the route that matched the criteria.

InterfaceIndex

[in] If RTM_MATCH_INTERFACE is specified in *MatchingFlags, InterfaceIndex* specifies the interface on which the route should be present (that is, the route has a next hop on that interface).

TargetViews

[in] Specifies the views from which to return information. If the client specifies RTM_VIEW_MASK_ANY, destination information is returned from all views; however, no view-specific information is returned.

RouteHandle

[out] If a handle must be returned: On input, *RouteHandle* is a pointer to NULL. On output, *RouteHandle* receives a pointer to the route handle; otherwise, *RouteHandle* remains unchanged.

If a handle does not need to be returned: On input, *RouteHandle* is NULL.

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_NOT_FOUND	The specified route was not found.

Remarks

Consider using **RtmGetExactMatchDestination** if you have no route-matching criteria specified in the *MatchingFlags* parameter.

The following members of the **RTM_ROUTE_INFO** structure that is passed in the *RouteInfo* parameter are used to match a route:

- **Neighbour**
- **NextHopsList**
- **PrefInfo**
- **RouteOwner**

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_NET_ADDRESS, **RTM_ROUTE_INFO**, **RtmGetExactMatchDestination**, **RtmGetLessSpecificDestination**, **RtmGetMostSpecificDestination**, **RtmIsBestRoute**

RtmGetLessSpecificDestination

The **RtmGetLessSpecificDestination** function searches the routing table for a destination with the next-best-match (longest) prefix, given a destination prefix. The requested destination information is returned.

```
DWORD
RtmGetLessSpecificDestination (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_DEST_HANDLE DestHandle,
  ULONG ProtocolId,
  RTM_VIEW_SET TargetViews,
  PRTM_DEST_INFO DestInfo
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

DestHandle
[in] Handle to the destination.

ProtocolId
[in] Specifies the protocol identifier. The *ProtocolID* is not part of the search criteria. The routing table manager uses this identifier to determine which route information to return (for example, if a client specifies the RIP protocol identifier, the best RIP route is returned, even if a non-RIP route is the best route to the destination).

Specify RTM_BEST_PROTOCOL to return a route regardless of which protocol owns it. Specify RTM_THIS_PROTOCOL to return the best route for the calling protocol.

TargetViews
[in] Specifies the views from which to return information. If the client specifies RTM_VIEW_MASK_ANY, destination information is returned from all views; however, no view-specific information is returned.

DestInfo
[out] On input, *DestInfo* is a pointer to an **RTM_DEST_INFO** structure. On output, *DestInfo* is filled with the requested destination information.

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	A parameter contains incorrect information.
ERROR_NOT_FOUND	The next best destination cannot be found.

Remarks

The *DestInfo* parameter is a variable-sized **RTM_DEST_INFO** structure. If the client specifies more than one view using *TargetViews*, the size of *DestInfo* increases for each view. Use the **RTM_SIZE_OF_DEST_INFO** macro to determine how much memory to allocate for the *DestInfo* structure before calling this function. Use the value specified for *TargetViews* as a parameter to **RTM_SIZE_OF_DEST_INFO**.

The **RtmGetLessSpecificDestination** function is used after a call to **RtmGetMostSpecificDestination** to return the next-best match for a destination. This call is also used after a prior call to **RtmGetLessSpecificDestination** to return the next successive less-specific match. Clients can use this function to "walk up" the prefix tree for a destination.

The **RtmGetLessSpecificDestination** function returns matches until it reaches the default route, if it exists. Once the default route is found, **RtmGetLessSpecificDestination** returns ERROR_NOT_FOUND.

One common use for the **RtmGetLessSpecificDestination** and **RtmGetMostSpecificDestination** functions, is to retrieve each of the matching destinations.

For sample code using this function, see *Search for Routes Using RtmGetMostSpecificDestination and RtmGetLessSpecificDestination*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_DEST_INFO, **RtmGetExactMatchDestination**, **RtmGetExactMatchRoute**, **RtmGetMostSpecificDestination**, **RtmIsBestRoute**

RtmGetListEnumRoutes

The **RtmGetListEnumRoutes** function enumerates a set of routes in a specified route list.

```
DWORD
RtmGetListEnumRoutes (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ENUM_HANDLE EnumHandle,
  PUINT NumRoutes,
  PRTM_ROUTE_HANDLE RouteHandles
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

EnumHandle
[in] Handle to the route list to enumerate.

NumRoutes
[in, out] On input, *NumRoutes* is a pointer to a **UINT** value that specifies the maximum number of routes that can be received by *RouteHandles*. On output, *NumRoutes* receives the actual number of routes received by *RouteHandles*.

RouteHandles
[out] On input, *DestInfo* is a pointer to an array of **RTM_DEST_INFO** structures. On output, *DestInfo* is filled with the requested destination information.

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 value pointed to by *NumRoutes* is larger than the maximum number of routes a client is allowed to retrieve with one call. Check **RTM_REGN_PROFILE** for the maximum number of routes that the client is allowed to retrieve with one call.

Remarks

Call this function repeatedly to retrieve all routes.

There are no more routes to enumerate when the routing table manager returns zero in *NumRoutes*.

For sample code using this function, see *Use a Client-Specific Route List*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

+ See Also

RtmCreateRouteListEnum, **RtmDeleteEnumHandle**

RtmGetMostSpecificDestination

The **RtmGetMostSpecificDestination** function searches the routing table for a destination with the exact match for a specified network address and subnet mask; if the exact match is not found, the best prefix is matched. The destination information is returned.

```
DWORD
RtmGetMostSpecificDestination (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PRTM_NET_ADDRESS DestAddress,
  ULONG ProtocolId,
  RTM_VIEW_SET TargetViews,
  PRTM_DEST_INFO DestInfo
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

DestAddress
[in] Pointer to the destination network address.

ProtocolId
[in] Specifies the protocol identifier. The *ProtocolID* is not part of the search criteria. The routing table manager uses this identifier to determine which route information to return (for example, if a client specifies the RIP protocol identifier, the best RIP route is returned, even if a non-RIP route is the best route to the destination).

Specify RTM_BEST_PROTOCOL to return a route regardless of which protocol owns it. Specify RTM_THIS_PROTOCOL to return the best route for the calling protocol.

TargetViews
[in] Specifies the views from which to return information. If the client specifies RTM_VIEW_MASK_ANY, destination information is returned from all views; however, no view-specific information is returned.

DestInfo
[out] On input, *DestInfo* is a pointer to an **RTM_DEST_INFO** structure. On output, *DestInfo* is filled with the requested destination information.

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_HANDLE	The handle was invalid.
ERROR_NOT_FOUND	The specified destination was not found.

Remarks

The *DestInfo* structure is a variable-sized structure. If the client specifies more than one view with *TargetViews*, the size of *DestInfo* increases for each view. Use the **RTM_SIZE_OF_DEST_INFO** macro to determine how much memory to allocate for the *DestInfo* structure before calling this function. Use the value specified for *TargetViews* as a parameter to **RTM_SIZE_OF_DEST_INFO**.

For sample code using this function, see *Search for Routes Using RtmGetMostSpecificDestination and RtmGetLessSpecificDestination*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_DEST_INFO, **RTM_NET_ADDRESS**, **RtmGetExactMatchDestination**, **RtmGetExactMatchRoute**, **RtmGetLessSpecificDestination**, **RtmIsBestRoute**

RtmGetNextHopInfo

The **RtmGetNextHopInfo** function returns information about the specified next hop.

```
DWORD
RtmGetNextHopInfo (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_NEXTHOP_HANDLE NextHopHandle,
  PRTM_NEXTHOP_INFO NextHopInfo
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NextHopHandle
[in] Handle to the next hop.

NextHopInfo
[out] On input, *NextHopInfo* a pointer to an **RTM_NEXTHOP_INFO** structure. On output, *NextHopInfo* is filled with the requested next-hop information.

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_HANDLE	The handle is invalid.

Remarks

When the next hop handle is no longer required, release it by calling **RtmDeleteNextHop**.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_NEXTHOP_INFO, **RtmReleaseNextHopInfo**

RtmGetNextHopPointer

The **RtmGetNextHopPointer** function obtains a direct pointer to the specified next hop. The pointer allows the next-hop owner direct read access to the routing table manager's **RTM_NEXTHOP_INFO** structure.

```
DWORD
RtmGetNextHopPointer (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_NEXTHOP_HANDLE NextHopHandle,
  PRTM_NEXTHOP_INFO *NextHopPointer
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NextHopHandle
[in] Handle to the next hop.

NextHopPointer
[out] If the client is the owner of the next hop, *NextHopPointer* receives a pointer to the next hop.

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_ACCESS_DENIED	The calling client does not own this next hop.
ERROR_INVALID_HANDLE	The handle is invalid.

Remarks

Clients should only use this pointer for read-only access.

When the next hop handle is no longer required, release it by calling **RtmReleaseNextHopInfo**.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_NEXTHOP_INFO, **RtmAddNextHop**, **RtmDeleteNextHop**, **RtmFindNextHop**, **RtmLockNextHop**

RtmGetOpaqueInformationPointer

The **RtmGetOpaqueInformationPointer** function returns a pointer to the opaque information field in a destination that is reserved for this client. The pointer enables the client to store client-specific information with the destination in the routing table.

```
DWORD
RtmGetOpaqueInformationPointer (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_DEST_HANDLE DestHandle,
  PVOID *OpaqueInfoPointer
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

DestHandle
[in] Handle to the destination.

OpaqueInfoPointer
[out] On input, *OpaqueInfoPointer* is a pointer to NULL. On output, *OpaqueInfoPointer* receives a pointer to the opaque information pointer. If a client has not reserved an opaque pointer during registration, this parameter remains unchanged.

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_HANDLE	The handle is invalid.
ERROR_NOT_FOUND	No opaque pointer was reserved by the client.

Remarks

For sample code using this function, see *Access the Opaque Pointers in a Destination*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmLockDestination

RtmGetRegisteredEntities

The **RtmGetRegisteredEntities** function returns information about all clients that have registered with the specified instance of the routing table manager and specified address family.

```
DWORD
RtmGetRegisteredEntities (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PUINT NumEntities,
  PRTM_ENTITY_HANDLE EntityHandles,
  PRTM_ENTITY_INFO EntityInfos
);
```

Parameters

RtmRegHandle
: [in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NumEntities
: [in, out] On input, *NumEntities* is a pointer to a **UINT** value, which specifies the maximum number of clients that can be received by *EntityInfos*. On output, *NumEntities* receives the actual number of clients received by *EntityInfos*.

EntityHandles

[out] If handles must be returned: On input, *EntityHandles* is a pointer to NULL. On output, *EntityHandles* receives a pointer to an array of entity handle; otherwise, *EntityHandles* remains unchanged.

If handles do not need to be returned: On input, *EntityHandles* is NULL.

EntityInfos

[out] If a pointer must be returned: On input, *EntityInfos* is a pointer to NULL. On output, *EntityInfos* receives a pointer to an array of **RTM_ENTITY_INFO** structures; otherwise, *EntityInfos* remains unchanged.

If a pointer does not need to be returned: On input, *EntityInfos* is NULL.

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_INSUFFICIENT_BUFFER	The buffer supplied is not large enough to hold all the requested information.

Remarks

If ERROR_INSUFFICIENT_BUFFER is returned, there may be some data in *EntityHandles*. The *NumEntities* parameter specifies how many entities were actually returned.

The **RtmGetRegisteredEntities** function can be used by routing protocols to verify which other protocols are running for that address family and routing table manager instance. Based on the information returned, a client can then perform protocol-specific processing.

The RTMv2 API supports only one instance of the routing table manager.

When the entities are no longer required, release them by calling **RtmReleaseEntities**.

For sample code using this function, see *Enumerate the Registered Entities*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

+ See Also

RTM_ENTITY_INFO, **RtmReleaseEntities**

RtmGetRouteInfo

The **RtmGetRouteInfo** function returns information for the specified route.

```
DWORD
RtmGetRouteInfo (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ROUTE_HANDLE RouteHandle,
  PRTM_ROUTE_INFO RouteInfo,
  PRTM_NET_ADDRESS DestAddress
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteHandle
[in] Handle to the route to find.

RouteInfo
[out] If a pointer must be returned: On input, *RouteInfo* is a pointer to NULL. On output, *RouteInfo* receives a pointer to the route; otherwise, *RouteInfo* remains unchanged.

If a pointer does not need to be returned: On input, *RouteInfo* is NULL.

DestAddress
[out] If a pointer must be returned: On input, *DestAddress* is a pointer to NULL. On output, *DestAddress* receives a pointer to the destination's **RTM_NET_ADDRESS** structure; otherwise, *DestAddress* remains unchanged.

If a pointer does not need to be returned: On input, *DestAddress* is NULL.

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_HANDLE	The handle is invalid.

When the routes are no longer required, release them by calling **RtmReleaseRouteInfo**.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_NET_ADDRESS, **RTM_ROUTE_INFO**, **RtmReleaseRouteInfo**

RtmGetRoutePointer

The **RtmGetRoutePointer** function obtains a direct pointer to a route that allows the owner of the route read access.

```
DWORD
RtmGetRoutePointer (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ROUTE_HANDLE RouteHandle,
  PRTM_ROUTE_INFO *RoutePointer
);
```

Parameters

RtmRegHandle
  [in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteHandle
  [in] Handle to the route.

RoutePointer
  [in] On input, *RoutePointer* is a pointer to NULL. On output, *RoutePointer* receives a pointer to the route.

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_ACCESS_DENIED	The calling client does not own this route.
ERROR_INVALID_HANDLE	The handle is invalid.

Remarks

The pointer that was returned points to the public part of the route.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_ROUTE_INFO, **RtmAddRouteToDest**, **RtmDeleteRouteToDest**, **RtmHoldDestination**, **RtmLockRoute**, **RtmUpdateAndUnlockRoute**

RtmHoldDestination

The **RtmHoldDestination** function marks a destination to be put in the hold-down state for a certain amount of time. A hold down only happens if the last route for the destination in any view is deleted.

Routing protocols that use hold-down states continue to advertise the last route until the hold-down expires, even if newer routes arrive in the meantime. The route is advertised as a deleted route. The newer routes are, however, used by the routing protocols for forwarding purposes. New routes are advertised when the hold down expires.

```
DWORD
RtmHoldDestination (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_DEST_HANDLE DestHandle,
  RTM_VIEW_SET TargetViews,
  ULONG HoldTime
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

DestHandle
[in] Handle to the destination to mark for holding.

TargetViews
[in] Specifies the views in which to hold the destination.

HoldTime
[in] Specifies how long, in milliseconds, to hold the destination.

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 hold time specified was zero.
ERROR_INVALID_HANDLE	The handle is invalid.

Remarks

All routes in a hold-down state are held for all views for a single, maximum hold-down time, regardless of the *HoldTime* specified.

For sample code using this function, see *Use the Route Hold-Down State*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmAddRouteToDest, **RtmDeleteRouteToDest**, **RtmLockRoute**, **RtmUpdateAndUnlockRoute**

RtmIgnoreChangedDests

The **RtmIgnoreChangedDests** function skips the next change for each destination if it has already occurred. This function can be used after **RtmGetChangeStatus** to prevent the routing table manager returning this change in response to a call to **RtmGetChangedDests**.

```
DWORD
RtmIgnoreChangedDests (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_NOTIFY_HANDLE NotifyHandle,
  UINT NumDests,
  PRTM_DEST_HANDLE ChangedDests
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NotifyHandle
[in] Handle to a change notification.

NumDests
[in] Specifies the number of destinations in *ChangedDests*.

ChangedDests
[in] Pointer to an array of **RTM_DEST_HANDLE** handles indicating the destinations for which to ignore any pending changes.

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_HANDLE	The handle is invalid.

When the destinations are no longer required, release them by calling **RtmReleaseChangedDests**.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmGetChangedDests, **RtmGetChangeStatus**, **RtmIsMarkedForChangeNotification**, **RtmMarkDestForChangeNotification**, **RtmReleaseChangedDests**

RtmInsertInRouteList

The **RtmInsertInRouteList** function inserts the specified set of routes into the client's route list. If a route is already in another list, the route is removed from the old list and inserted into the new one.

```
DWORD
RtmInsertInRouteList (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ROUTE_LIST_HANDLE RouteListHandle,
  UINT NumRoutes,
  PRTM_ROUTE_HANDLE RouteHandles
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteListHandle
[in] Handle to the route list to which to add routes. Specify NULL to remove the specified routes from their old lists.

NumRoutes
[in] Specifies the number of routes in *RouteHandles*.

RouteHandles
[in] Pointer to an array of route handles to move from the old list to the new list.

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_HANDLE	The handle is invalid.

When the routes are no longer required, release them by calling **RtmReleaseRoutes**.

For sample code using this function, see *Use a Client-Specific Route List*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmCreateRouteList, **RtmDeleteRouteList**

RtmInvokeMethod

The **RtmInvokeMethod** function invokes a method exported by another client.

```
DWORD
RtmInvokeMethod (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ENTITY_HANDLE EntityHandle,
  PRTM_ENTITY_METHOD_INPUT Input,
  PUINT OutputSize,
  PRTM_ENTITY_METHOD_OUTPUT Output
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

EntityHandle
[in] Handle to the client whose methods are being invoked.

Input
[in] Pointer to an **RTM_ENTITY_METHOD_INPUT** structure containing the method to be invoked and a common input buffer.

OutputSize
[in, out] On input, *OutputSize* is a pointer to a **UNIT** value specifying the size, in bytes, of *Output*. On output, *OutputSize* receives a pointer to a **UINT** value specifying the actual size, in bytes, of *Output*.

Output
[out] Receives a pointer to an array of **RTM_ENTITY_METHOD_OUTPUT** structures. Each structure consists of a (method identifier, correct output) tuple.

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_HANDLE	The handle is invalid.

Remarks

For sample code using this function, see *Obtain and Call the Exported Methods for a Client*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_ENTITY_METHOD_INPUT, **RTM_ENTITY_METHOD_OUTPUT**, **RtmBlockMethods**, **RtmGetEntityMethods**

RtmIsBestRoute

The **RtmIsBestRoute** function returns the set of views in which the specified route is the best route to a destination.

```
DWORD
RtmIsBestRoute (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ROUTE_HANDLE RouteHandle,
  PRTM_VIEW_SET BestInViews
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteHandle
[in] Handle to the route to check.

BestInViews
[out] Receives a pointer to the set of views for which the specified route is the best route.

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_HANDLE	The handle is invalid.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

+ See Also

RtmGetExactMatchDestination, **RtmGetExactMatchRoute**, **RtmGetLessSpecificDestination**, **RtmGetMostSpecificDestination**

RtmIsMarkedForChangeNotification

The **RtmIsMarkedForChangeNotification** function queries the routing table manager to determine if a destination has previously been marked by a call to **RtmMarkDestForChangeNotification**.

```
DWORD
RtmIsMarkedForChangeNotification (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_NOTIFY_HANDLE NotifyHandle,
  RTM_DEST_HANDLE DestHandle,
  PBOOL DestMarked
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NotifyHandle
[in] Handle to a change notification, obtained from a previous call to **RtmRegisterForChangeNotification**.

DestHandle
[in] Handle to the destination to check.

DestMarked
[out] Pointer to a **BOOL** variable that is TRUE if the destination is marked, FALSE if it is not.

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_HANDLE	The handle is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmGetChangedDests, **RtmGetChangeStatus**, **RtmIgnoreChangedDests**, **RtmMarkDestForChangeNotification**, **RtmReleaseChangedDests**

RtmLockDestination

The **RtmLockDestination** function locks or unlocks a destination in the routing table. Use this function to protect a destination while changing opaque pointers.

```
DWORD
RtmLockDestination (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_DEST_HANDLE DestHandle,
  BOOL Exclusive,
  BOOL LockDest
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

DestHandle
[in] Handle to the destination to lock.

Exclusive
[in] Specifies whether to lock or unlock the destination in an exclusive (TRUE) or shared (FALSE) mode.

LockDest
[in] Specifies whether to lock or unlock the destination. Specify TRUE to lock the destination; specify FALSE to unlock it.

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_ACCESS_DENIED	The calling client does not own this destination.
ERROR_INVALID_HANDLE	The handle is invalid.

Remarks

This function also locks the associated routes. Avoid locking destinations for long periods of time, because no other client can access the destination and associated routes until the lock is released.

A client can use also this function when reading information for a destination, while preventing changes during the client's read operation. In this case, consider using **RtmGetDestInfo** instead.

For sample code using this function, see *Update a Route In Place Using RtmUpdateAndUnlockRoute*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmGetOpaqueInformationPointer

RtmLockNextHop

The **RtmLockNextHop** function locks or unlocks a next hop. This function should be called by the next hop's owner to lock the next hop before making changes to the next hop. A pointer to the next hop is returned.

```
DWORD
RtmLockNextHop (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_NEXTHOP_HANDLE NextHopHandle,
  BOOL Exclusive,
  BOOL LockNextHop,
  PRTM_NEXTHOP_INFO *NextHopPointer
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NextHopHandle
[in] Handle to the next hop to lock or unlock.

Exclusive
[in] Specifies whether to lock or unlock the next hop in an exclusive (TRUE) or shared (FALSE) mode.

LockNextHop
[in] Specifies whether to lock or unlock the next hop. Specify TRUE to lock the next hop; specify FALSE to unlock it.

NextHopPointer
[out] On input, *NextHopPointer* is a pointer to NULL. On output, if the client owns the next hop, *NextHopPointer* receives a pointer to the next-hop; otherwise, *NextHopPointer* remains unchanged.

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_ACCESS_DENIED	The calling client does not own this next hop.
ERROR_NOT_FOUND	The specified next hop was not found.

Remarks

Clients cannot change the **NextHopAddress** and **InterfaceIndex** members; these values are used to uniquely identify a next hop.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_NEXTHOP_INFO, **RtmAddNextHop**, **RtmDeleteNextHop**, **RtmFindNextHop**, **RtmGetNextHopPointer**

RtmLockRoute

The **RtmLockRoute** function locks or unlocks a route in the routing table. This protects the route while a client makes the necessary changes to the client's opaque route pointer.

```
DWORD
RtmLockRoute (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ROUTE_HANDLE RouteHandle,
  BOOL Exclusive,
  BOOL LockRoute,
  PRTM_ROUTE_INFO *RoutePointer
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteHandle
[in] Handle to the route to lock.

Exclusive
[in] Specifies whether to lock or unlock the route in an exclusive (TRUE) or shared (FALSE) mode.

LockRoute
[in] Specifies whether to lock or unlock the route. Specify TRUE to lock the route; specify FALSE to unlock it.

RoutePointer
[out] If a pointer must be returned: On input, *RoutePointer* is a pointer to NULL. On output, if the client owns the route, *RoutePointer* receives a pointer to the next-hop; otherwise, *RoutePointer* remains unchanged.

If a handle does not need to be returned: On input, *RoutePointer* is NULL.

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_ACCESS_DENIED	The calling client does not own this route.
ERROR_INVALID_HANDLE	The handle is invalid.

Remarks

Do not call any other RTMv2 functions until the route is unlocked by a call to **RtmLockRoute** (and the *LockRoute* parameter is set to FALSE) or a call to **RtmUpdateAndUnlockRoute**.

Currently, this function locks the entire destination, not just the route.

Clients can only change the **Neighbour**, **PrefInfo**, **BelongsToViews**, **EntitySpecificInfo**, and **NextHopsList** members of the **RTM_ROUTE_INFO** structure.

If any of these values are changed, the client must call **RtmUpdateAndUnlockRoute** to notify the routing table manager of the changes.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_ROUTE_INFO, **RtmAddRouteToDest**, **RtmDeleteRouteToDest**, **RtmGetRoutePointer**, **RtmHoldDestination**, **RtmUpdateAndUnlockRoute**

RtmMarkDestForChangeNotification

The **RtmMarkDestForChangeNotification** function marks a destination for a client, requesting that the routing table manager send the client change notifications messages for marked destination. The client receives change notification messages when a destination changes. The change notifications inform the client of changes to best-route information for the specified destination. This function should be used when **RtmRegisterForChangeNotification** is called to request changes for specific destinations (RTM_NOTIFY_ONLY_MARKED_DESTS).

```
DWORD
RtmMarkDestForChangeNotification (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_NOTIFY_HANDLE NotifyHandle,
  RTM_DEST_HANDLE DestHandle,
  BOOL MarkDest
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NotifyHandle
[in] Handle to a change notification obtained via a previous call to **RtmRegisterForChangeNotification**.

DestHandle
[in] Handle to the destination that the client is marking for notification of changes.

MarkDest
[in] Specifies whether to mark a destination and receive change notifications. Specify TRUE to mark a destination and start receive change notifications for the specified destination. Specify FALSE to stop receiving change notifications a previously marked destination.

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_HANDLE	The handle is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmGetChangedDests, **RtmGetChangeStatus**, **RtmIgnoreChangedDests**, **RtmIsMarkedForChangeNotification**, **RtmRegisterForChangeNotification**, **RtmReleaseChangedDests**

RtmReferenceHandles

The **RtmReferenceHandles** function increases the reference count for objects pointed to by one or more handles that the routing manager used to access those objects. A client should use this function when the client must keep a handle but release the rest of the information structure associated with the handle.

```
DWORD
RtmReferenceHandles (
  RTM_ENTITY_HANDLE RtmRegHandle,
  UINT NumHandles,
  HANDLE *RtmHandles
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NumHandles
[in] Specifies the number of handles in *RtmHandles*.

RtmHandles
[in] Array of handles to increase the reference count for.

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_HANDLE	The handle is invalid.

Remarks

Always call this function when caching a handle returned by the routing table manager. This notifies the routing table manager that it should not destroy the object the handle refers to until the handle is released by the client.

When a client must release the handle, the client must call the appropriate "release" function, based on the type of handle (for example, for a route, call **RtmReleaseRoutes**).

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmDeleteEnumHandle, **RtmReleaseChangedDests**, **RtmReleaseDestInfo**, **RtmReleaseEntityInfo**, **RtmReleaseNextHopInfo**, **RtmReleaseRouteInfo**

RtmRegisterEntity

The **RtmRegisterEntity** function registers a client with an instance of the routing table manager for a specific address family. The routing table manager returns a registration handle and a profile of the instance. The profile contains a list of values that are used when calling other functions (values include the maximum number of destinations returned in a buffer by a single call).

Registration is the first action a client should take.

```
DWORD
RtmRegisterEntity (
  PRTM_ENTITY_INFO RtmEntityInfo,
  PRTM_ENTITY_EXPORT_METHODS ExportMethods,
  RTM_EVENT_CALLBACK EventCallback,
  BOOL ReserveOpaquePointer,
  PRTM_REGN_PROFILE RtmRegProfile,
  PRTM_ENTITY_HANDLE RtmRegHandle
);
```

Parameters

RtmEntityInfo
[in] Pointer to an **RTM_ENTITY_INFO** structure. This structure contains information identifying the client to the routing table manager (such as the routing table manager instance and address family to register with).

ExportMethods
[in] Pointer to a list of methods exported by the client. This parameter is optional and can be set to NULL if the calling client has no methods to export.

EventCallback
[in] Specifies the callback the routing table manager will use to notify the client of events. The events are when a client registers and unregisters, when routes expire, and when changes to the best route to a destination have occurred (only those changes specified when the client called **RtmRegisterForChangeNotification**).

ReserveOpaquePointer
[in] Specifies whether to reserve an opaque pointer in each destination for this instance of the protocol. Specify TRUE to reserve an opaque pointer in each destination. Specify FALSE to skip this action. These opaque pointers can be used to point to a private, protocol-specific context for each destination.

RtmRegProfile
[out] On input, *RtmRegProfile* is a pointer to an **RTM_REGN_PROFILE** structure. On output, *RtmRegProfile* is filled with the requested registration profile structure. The client must use the information returned in other function calls (information returned includes the number of equal-cost next hops and the maximum number of items returned by an enumeration function call).

RtmRegHandle
[out] Receives a registration handle for the client. This handle must be used in all subsequent calls to the routing table manager.

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_ALREADY_EXISTS	The specified client has already registered with the routing table manager.
ERROR_BAD_CONFIGURATION	Registry information for the routing table manager is corrupt.
ERROR_FILE_NOT_FOUND	Registry information for the routing table manager was not found.
ERROR_INVALID_DATA	A parameter contains incorrect information.
ERROR_INVALID_PARAMETER	A parameter contains incorrect information.
ERROR_NO_SYSTEM_RESOURCES	There are not enough available system resources to complete this operation.
ERROR_NOT_ENOUGH_MEMORY	There is not enough memory to complete this operation.

Remarks

For sample code using this function, see *Register With the Routing Table Manager.*

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_EVENT_CALLBACK, **RTM_ENTITY_EXPORT_METHODS**, **RTM_ENTITY_INFO**, **RTM_REGN_PROFILE**, **RtmDeregisterEntity**, **RtmGetRegisteredEntities**, **RtmReleaseEntities**

RtmRegisterForChangeNotification

The **RtmRegisterForChangeNotification** function informs the routing table manager that the client should receive change notifications for the specified types of changes. The routing table manager returns a change notification handle, which the client must use when requesting change information after receiving a change notification message.

```
DWORD
RtmRegisterForChangeNotification (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_VIEW_SET TargetViews,
  RTM_NOTIFY_FLAGS NotifyFlags,
  PVOID NotifyContext,
  PRTM_NOTIFY_HANDLE NotifyHandle
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

TargetViews
[in] Specifies the views to register for change notification in.

NotifyFlags
[in] Specifies the flags that indicate the type of changes for which the client requests notification. The following flags are used. (The flags are to be joined using a logical OR.)

Constant	Description
RTM_CHANGE_TYPE_ALL	Notify the client of any change to a destination.
RTM_CHANGE_TYPE_BEST	Notify the client of changes to the current best route, or when the best route changes.
RTM_CHANGE_TYPE_FORWARDING	Notify the client of any best route changes that affect forwarding (such as next hop changes).
RTM_NOTIFY_ONLY_MARKED_DESTS	Notify the client of changes to destinations that the client has marked. If this flag is not specified, change notification messages for all destinations are sent.

NotifyContext
[in] Pointer to a **VOID** that specifies the context that the **RTM_EVENT_CALLBACK** uses to indicate new changes.

NotifyHandle
[out] Receives a handle to a change notification. The handle must be used when calling **RtmGetChangedDests**.

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	A parameter contains incorrect information.
ERROR_NO_SYSTEM_RESOURCES	There are not enough available system resources to complete this operation. The routing table manager has exceeded the maximum number of change notifications that can be cached.
ERROR_NOT_ENOUGH_MEMORY	There is not enough memory to complete this operation.
ERROR_NOT_SUPPORTED	One or more of the specified views is not supported.

Remarks

A client calls **RtmMarkDestForChangeNotification** when it is registering for changes to a specific destination.

The routing table manager uses the **RTM_EVENT_CALLBACK** callback (specified when the client called **RtmRegisterEntity**) to notify the client when changes have occurred; the client must call **RtmGetChangedDests** to receive the actual change information.

For sample code using this function, see *Register For Change Notification*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmDeregisterFromChangeNotification, **RtmGetChangedDests**, **RtmMarkDestForChangeNotification**

RtmReleaseChangedDests

The **RtmReleaseChangedDests** function releases the changed destination handles.

```
DWORD
RtmReleaseChangedDests (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_NOTIFY_HANDLE NotifyHandle,
  UINT NumDests,
  PRTM_DEST_INFO ChangedDests
);
```

Parameters

RtmRegHandle
: [in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NotifyHandle
: [in] Handle to a change notification, obtained from a previous call to **RtmRegisterForChangeNotification**.

NumDests
: [in] Specifies the number of destinations in *ChangedDests*.

ChangedDests
: [in] Pointer to an array of **RTM_DEST_INFO** structures to release. The changed destinations were obtained from a prior call to **RtmGetChangedDests**.

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_HANDLE	The handle is invalid.

Remarks

Always use this function to release changed **RTM_DEST_INFO** structures obtained from a call to **RtmGetChangedDests**.

The **RTM_DEST_INFO** structure is a variable-sized structure. If a destination contains information for more than one view, the size of **RTM_DEST_INFO** increases for each view. Use the **RTM_SIZE_OF_DEST_INFO** macro to determine how large a *ChangedDests* buffer to allocate before calling this function.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_DEST_INFO, **RtmGetChangedDests**, **RtmGetChangeStatus**, **RtmIgnoreChangedDests**, **RtmIsMarkedForChangeNotification**, **RtmMarkDestForChangeNotification**

RtmReleaseDestInfo

The **RtmReleaseDestInfo** function releases a destination structure.

```
DWORD
RtmReleaseDestInfo (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PRTM_DEST_INFO DestInfo
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

DestInfo
[in] Pointer to the destination to release. The destination was obtained from a previous call to **RtmGetDestInfo**.

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_HANDLE	The handle is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_DEST_INFO, **RtmGetDestInfo**

RtmReleaseDests

The **RtmReleaseDests** function releases the destination handles.

```
DWORD
RtmReleaseDests (
  RTM_ENTITY_HANDLE RtmRegHandle,
  UINT NumDests,
  PRTM_DEST_INFO DestInfos
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NumDests
[in] Specifies the number of destinations in *DestInfos*.

DestInfos
[in] Pointer to an array of **RTM_DEST_INFO** structures to release. The destinations were obtained from a previous call to **RtmGetEnumDests**.

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_HANDLE	The handle is invalid.

Remarks

Do not use this function to release **RTM_DEST_INFO** structures obtained from a call to **RtmGetChangedDests**. Use **RtmReleaseChangedDests** instead.

The **RTM_DEST_INFO** structure is a variable-sized structure. If a destination contains information for more than one view, the size of **RTM_DEST_INFO** increases for each view. Use the **RTM_SIZE_OF_DEST_INFO** macro to determine how large a *DestInfos* buffer to allocate before calling this function.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_DEST_INFO, **RtmCreateDestEnum**, **RtmDeleteEnumHandle**, **RtmGetEnumDests**

RtmReleaseEntities

The **RtmReleaseEntities** function releases the client handles returned by **RtmGetRegisteredEntities**.

```
DWORD
RtmReleaseEntities (
  RTM_ENTITY_HANDLE RtmRegHandle,
  UINT NumEntities,
  PRTM_ENTITY_HANDLE EntityHandles
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NumEntities
[in] Specifies the number of clients in *EntityHandles*.

EntityHandles
[in] Pointer to an array of client handles to release. The handles were obtained from a previous call to **RtmGetRegisteredEntities**.

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_HANDLE	The handle is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmGetRegisteredEntities

RtmReleaseEntityInfo

The **RtmReleaseEntityInfo** function releases a client structure.

```
DWORD
RtmReleaseEntityInfo (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PRTM_ENTITY_INFO EntityInfo
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

EntityInfo
[in] Pointer to the handle to release. The handle was obtained with a previous call to **RtmGetEntityInfo**.

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_HANDLE	The handle is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_ENTITY_INFO, **RtmGetEntityInfo**

RtmReleaseNextHopInfo

The **RtmReleaseNextHopInfo** function releases a next-hop structure.

```
DWORD
RtmReleaseNextHopInfo (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PRTM_NEXTHOP_INFO NextHopInfo
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NextHopInfo
[in] Pointer to the **RTM_NEXTHOP_INFO** structure to release. The next hop was obtained with a previous call to **RtmGetNextHopInfo**.

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_HANDLE	The handle is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_NEXTHOP_INFO, **RtmFindNextHop**, **RtmGetNextHopInfo**

RtmReleaseNextHops

The **RtmReleaseNextHops** function releases the next-hop handles.

```
DWORD
RtmReleaseNextHops (
  RTM_ENTITY_HANDLE RtmRegHandle,
  UINT NumNextHops,
  PRTM_NEXTHOP_HANDLE NextHopHandles
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NumNextHops
[in] Specifies the number of next hops in *NextHopHandles*.

NextHopHandles
[in] Pointer to an array of next-hop handles to release. The handles were obtained with a previous call to **RtmGetEnumNextHops**.

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_HANDLE	The handle is invalid.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

+ See Also

RtmCreateNextHopEnum, **RtmGetEnumNextHops**

RtmReleaseRouteInfo

The **RtmReleaseRouteInfo** function releases a route structure.

```
DWORD
RtmReleaseRouteInfo (
  RTM_ENTITY_HANDLE RtmRegHandle,
  PRTM_ROUTE_INFO RouteInfo
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteInfo
[in] Pointer to the **RTM_ROUTE_INFO** structure to release. The route was obtained with a previous call to **RtmGetRouteInfo**.

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_HANDLE	The handle is invalid.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RTM_ROUTE_INFO, **RtmGetRouteInfo**

RtmReleaseRoutes

The **RtmReleaseRoutes** function releases the route handles.

```
DWORD
RtmReleaseRoutes (
  RTM_ENTITY_HANDLE RtmRegHandle,
  UINT NumRoutes,
  PRTM_ROUTE_HANDLE RouteHandles
);
```

Parameters

RtmRegHandle
: [in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

NumRoutes
: [in] Specifies the number of routes in *RouteHandles*.

RouteHandles
: [in] Pointer to an array of route handles to release. The handles were obtained with a previous call to **RtmGetEnumRoutes**.

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_HANDLE	The handle is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmCreateRouteEnum, **RtmDeleteEnumHandle**, **RtmGetEnumRoutes**

RtmUpdateAndUnlockRoute

The **RtmUpdateAndUnlockRoute** function updates the position of the route in the set of routes for a destination, and adjusts the best route information for the destination.

This function is used after a client has locked a route and updated it directly (also known as "in-place updating").

```
DWORD
RtmUpdateAndUnlockRoute (
  RTM_ENTITY_HANDLE RtmRegHandle,
  RTM_ROUTE_HANDLE RouteHandle,
  ULONG TimeToLive,
  RTM_ROUTE_LIST_HANDLE RouteListHandle,
  RTM_NOTIFY_FLAGS NotifyType,
  RTM_NOTIFY_HANDLE NotifyHandle,
  PRTM_ROUTE_CHANGE_FLAGS ChangeFlags
);
```

Parameters

RtmRegHandle
[in] Handle to the client obtained from a previous call to **RtmRegisterEntity**.

RouteHandle
[in] Handle to the route to change.

TimeToLive
[in] Specifies the time (in milliseconds) after which the route is expired. Specify INFINITE to prevent routes from expiring.

RouteListHandle
[in] Handle to an optional route list to which to move the route. This parameter is optional and can be set to NULL.

NotifyType
[in] Set this parameter to NULL. *NotifyType* is reserved for future use.

NotifyHandle
[in] Set this parameter to NULL. *NotifyHandle* is reserved for future use.

ChangeFlags
[out] Receives RTM_ROUTE_CHANGE_BEST if the best route was changed.

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_ACCESS_DENIED	The calling client does not own this route.

Remarks

Before calling this function, the client should lock the route using **RtmLockRoute**, which returns a pointer to the route. Then, the client can update the route information using the pointer. Finally, the client should call **RtmUpdateAndUnlockRoute**. If the function executes successfully, the route is unlocked. If the call failed, the client must unlock the route by calling **RtmLockRoute** with the *LockRoute* parameter set to FALSE.

For sample code using this function, see *Update a Route In Place Using RtmUpdateAndUnlockRoute*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.
Library: Use Rtm.lib.

See Also

RtmAddRouteToDest, **RtmDeleteRouteToDest**, **RtmGetRoutePointer**, **RtmHoldDestination**, **RtmLockRoute**

Routing Table Manager Version 2 Callbacks

The following callbacks are used to inform clients of registration events.

RTM_ENTITY_EXPORT_METHOD
RTM_EVENT_CALLBACK

RTM_ENTITY_EXPORT_METHOD

The **RTM_ENTITY_EXPORT_METHOD** callback is the prototype for any method exported by a client.

```
typedef VOID (WINAPI * _ENTITY_METHOD) (
  RTM_ENTITY_HANDLE CallerHandle,
  RTM_ENTITY_HANDLE CalleeHandle,
  RTM_ENTITY_METHOD_INPUT *Input,
  RTM_ENTITY_METHOD_OUTPUT *Output
);
```

Parameters

CallerHandle
Handle to the calling client.

CalleeHandle
Handle to the client being called.

Input
Handle to the method to be invoked. Contains an input buffer.

Output
An array of **RTM_ENTITY_METHOD_OUTPUT** structures. Each structure consists of a (method identifier, correct output) tuple.

Remarks

Methods can be exported when a client registers. Other clients, such as routing protocols, can invoke these methods to obtain client-specific information. For example, BGP can use a method to obtain OSFP information.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RTM_ENTITY_METHOD_INPUT, **RTM_ENTITY_METHOD_OUTPUT**

RTM_EVENT_CALLBACK

The **RTM_EVENT_CALLBACK** callback is used by the routing table manager to inform a client that the specified event occurred.

```
typedef DWORD (WINAPI * _EVENT_CALLBACK) (
  IRTM_ENTITY_HANDLE RtmRegHandle,
  RTM_EVENT_TYPE EventType,
  PVOID Context1,
  PVOID Context2
);
```

Parameters

RtmRegHandle
Handle to the client the that routing table manager is sending the notification to.

EventType
Specifies the event the routing table manager is notifying the client about. The following values are used.

Value	Description
RTM_ENTITY_REGISTERED	A client has just registered with the routing table manager.
RTM_ENTITY_DEREGISTERED	A client has just unregistered.

Value	Description
RTM_ROUTE_EXPIRED	A route has timed out.
RTM_CHANGE_NOTIFICATION	A change notification has been made.

Context1

For RTM_ENTITY_REGISTERED calls: Contains the handle to the entity that registered.

For RTM_ENTITY_DEREGISTERED calls: Contains the handle to the entity that unregistered.

For RTM_ROUTE_EXPIRED calls: Contains the handle to the route that timed out.

For RTM_CHANGE_NOTIFICATION calls: Contains the handle to the change notification.

Context2

For RTM_ENTITY_REGISTERED calls: Contains a pointer to the **RTM_ENTITY_INFO** structure referred to by the handle in **Context1**. If the client must retain this information, the client must copy it to a structure it has allocated.

For RTM_ENTITY_DEREGISTERED calls: Contains a pointer to the **RTM_ENTITY_INFO** structure referred to by the handle in **Context1**. If the client must retain this information, the client must copy it to a structure it has allocated.

For RTM_ROUTE_EXPIRED calls: Contains a pointer to the **RTM_ROUTE_INFO** structure referred to by the handle in **Context1**. If the client must retain this information, the client must copy it to a structure it has allocated.

For RTM_CHANGE_NOTIFICATION calls: Contains the notification context that was given to the client by a previous call to **RtmRegisterForChangeNotification**.

Return Values

If the routing table manager issues an RTM_ROUTE_EXPIRED callback, and the client returns to the routing table manager the value ERROR_NOT_SUPPORTED, the routing table manager will delete the route from the routing table.

All other errors returned by the client are ignored.

Remarks

After a client has registered for change notification, the routing table manager uses this callback to keep the client informed about events.

If a client receives an **RTM_EVENT_CALLBACK** for the RTM_ENTITY_REGISTERED or RTM_ENTITY_DEREGISTERED events, the client must not make calls to **RtmRegisterEntity**, **RtmDeregisterEntity**, or **RtmGetRegisteredEntities** in the context of this callback.

If a client receives an **RTM_EVENT_CALLBACK** for the RTM_CHANGE_NOTIFICATION event, the client must not call **RtmRegisterForChangeNotification** in the context of this callback.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

+ See Also

RTM_EVENT_TYPE, **RtmRegisterEntity**

Routing Table Manager Version 2 Structures

The RTMv2 functions use the following structures:

RTM_DEST_INFO
RTM_ENTITY_EXPORT_METHODS
RTM_ENTITY_ID
RTM_ENTITY_INFO
RTM_ENTITY_METHOD_INPUT
RTM_ENTITY_METHOD_OUTPUT
RTM_NET_ADDRESS
RTM_NEXTHOP_INFO
RTM_NEXTHOP_LIST
RTM_PREF_INFO
RTM_REGN_PROFILE
RTM_ROUTE_INFO

RTM_DEST_INFO

The **RTM_DEST_INFO** structure is used to exchange destination information with clients registered with the routing table manager.

```
typedef struct _RTM_DEST_INFO {
  RTM_DEST_HANDLE       DestHandle;
  RTM_NET_ADDRESS       DestAddress;
  FILETIME              LastChanged;
  RTM_VIEW_SET          BelongsToViews;
  UINT                  NumberOfViews;
  struct
  {
    RTM_VIEW_ID         ViewId;
    UINT                NumRoutes;
    RTM_ROUTE_HANDLE    Route;
    RTM_ENTITY_HANDLE   Owner;
    DWORD               DestFlags;
    RTM_ROUTE_HANDLE    HoldRoute;
  }                 ViewInfo[1];
} RTM_DEST_INFO, *PRTM_DEST_INFO;
```

Members

DestHandle
Handle to the destination.

DestAddress
Specifies the destination network address as an address and a mask.

LastChanged
Specifies the last time this destination was updated.

BelongsToViews
Specifies the views to which this destination belongs.

NumberOfViews
Indicates the number of ViewInfo structures present in this destination.

ViewInfo
Structure of the following components.

ViewId
Specifies the view this information applies to.

NumRoutes
Specifies the number of routes in each of the supported views.

Route
Handle to the best route (with matching criteria) in each of the supported views.

Owner
Handle to the owner of the best route in each of the supported views.

DestFlags
Specifies the flags for the best route in each of the supported views.

HoldRoute
Handle to the route that is in a hold-down state in each of the supported views.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

+ See Also

RTM_NET_ADDRESS, **RtmGetChangedDests**, **RtmGetDestInfo**, **RtmGetEnumDests**, **RtmGetExactMatchDestination**, **RtmGetLessSpecificDestination**, **RtmGetMostSpecificDestination**, **RtmReleaseChangedDests**, **RtmReleaseDestInfo**, **RtmReleaseDests**

RTM_ENTITY_EXPORT_METHODS

The **RTM_ENTITY_EXPORT_METHODS** structure contains the set of methods exported by a client.

```
typedef struct _RTM_ENTITY_EXPORT_METHODS {
  UINT                     NumMethods;
  RTM_ENTITY_EXPORT_METHOD Methods[1];
} RTM_ENTITY_EXPORT_METHODS, *PRTM_ENTITY_EXPORT_METHODS;
```

Members

NumMethods
Specifies the number of methods exported by the client in the **Methods** member.

Methods
Specifies which methods the client is exporting.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RtmRegisterEntity

RTM_ENTITY_ID

The **RTM_ENTITY_ID** structure is used to uniquely identify a client to the routing table manager. The protocol identifier and the instance identifier are the values that are used to uniquely identify a client.

```
typedef struct _RTM_ENTITY_ID {
  union
  {
    struct {
      ULONG    EntityProtocolId;
      ULONG    EntityInstanceId;
    };
    ULONGLONG  EntityId;
  };
} RTM_ENTITY_ID, *PRTM_ENTITY_ID;
```

Members

EntityProtocolId
Specifies a client's protocol identifier (such as RIP or OSPF).

EntityInstanceId
Specifies a client's protocol instance (such as RIPv1 or RIPv2).

EntityId
Specifies a client's identifier, which is a combination of the **EntityProtocolId** and the **EntityInstanceId**.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

+ See Also

RTM_ENTITY_INFO

RTM_ENTITY_INFO

The **RTM_ENTITY_INFO** structure is used to exchange client information with the routing table manager.

```
typedef struct _RTM_ENTITY_INFO {
  USHORT         RtmInstanceId;
  USHORT         AddressFamily;
  RTM_ENTITY_ID  EntityId;
} RTM_ENTITY_INFO, *PRTM_ENTITY_INFO;
```

Members

RtmInstanceId
Specifies the instance of the routing table manager with which the client registered.

AddressFamily
Specifies the address family to which the client belongs.

EntityId
Specifies the identifier that uniquely identifies a client.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

+ See Also

RTM_ENTITY_ID, **RtmGetEntityInfo**, **RtmGetRegisteredEntities**, **RtmRegisterEntity**, **RtmReleaseEntityInfo**

RTM_ENTITY_METHOD_INPUT

The **RTM_ENTITY_METHOD_INPUT** structure is used to pass information to a client when invoking its method.

```
typedef struct _RTM_ENTITY_METHOD_INPUT {
  RTM_ENTITY_METHOD_TYPE  MethodType;
  UINT                    InputSize;
  UCHAR                   InputData[1];
} RTM_ENTITY_METHOD_INPUT, *PRTM_ENTITY_METHOD_INPUT;
```

Members

MethodType
Specifies the method.

InputSize
Specifies the size, in bytes, of **InputData**.

InputData
Buffer for input data for the method.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

+ See Also

RtmInvokeMethod

RTM_ENTITY_METHOD_OUTPUT

The **RTM_ENTITY_METHOD_INPUT** structure is used to pass information to the calling client when the routing table manager invokes a method.

```
typedef struct _RTM_ENTITY_METHOD_OUTPUT {
  RTM_ENTITY_METHOD_TYPE  MethodType;
  DWORD                   MethodStatus;
  UINT                    OutputSize;
  UCHAR                   OutputData[1];
} RTM_ENTITY_METHOD_OUTPUT, *PRTM_ENTITY_METHOD_OUTPUT;
```

Members

MethodType
Specifies the method identifier

MethodStatus
Receives the status of the method after execution.

OutputSize
Specifies the size, in bytes, of **OutputData**.

OutputData
Buffer for data returned by the method.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RtmInvokeMethod

RTM_NET_ADDRESS

The **RTM_NET_ADDRESS** structure is used to communicate address information to the routing table manager for any address family. The address family must use only with contiguous address masks that are less than 8 bytes.

```
typedef struct _RTM_NET_ADDRESS {
  USHORT   AddressFamily;
  USHORT   NumBits;
  UCHAR    AddrBits[RTM_MAX_ADDRESS_SIZE];
} RTM_NET_ADDRESS, *PRTM_NET_ADDRESS;
```

Members

AddressFamily
Specifies the type of network address for this address (such as IPv4).

NumBits
Specifies the number of bits in the network part of the **AddrBits** bit array (for example, 255.0.0.0 has 8 bits).

AddrBits
Specifies an array of bits that form the address prefix.

Remarks

If the client specifies an address and a mask length that do not correspond to each other, inconsistent results will be returned by the routing table manager. For example, if a client specifies an address as 10.10.10.10 and a length as 24 when calling **RTM_IPV4_SET_ADDR_AND_LEN**, the routing table manager may return an incorrect *NetAddress*.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RTM_DEST_INFO, **RTM_NEXTHOP_INFO**, **RtmAddRouteToDest**, **RtmCreateDestEnum**, **RtmCreateNextHopEnum**, **RtmCreateRouteEnum**, **RtmGetExactMatchDestination**, **RtmGetExactMatchRoute**, **RtmGetMostSpecificDestination**, **RtmGetRouteInfo**

RTM_NEXTHOP_INFO

The **RTM_NEXTHOP_INFO** structure is used to exchange next-hop information with the routing table manager.

```
typedef struct _RTM_NEXTHOP_INFO {
  //
  // Information that the owner can directly access
  // for read only
  //
  RTM_NET_ADDRESS     NextHopAddress;
  RTM_ENTITY_HANDLE   NextHopOwner;
  ULONG               InterfaceIndex;
  USHORT              State;
  //
  // Information that the owner can directly access
  // for read/write
  //
  USHORT              Flags;
  PVOID               EntitySpecificInfo;
  RTM_DEST_HANDLE     RemoteNextHop;
} RTM_NEXTHOP_INFO, *PRTM_NEXTHOP_INFO;
```

Members

NextHopAddress
Specifies the network address for this next hop.

NextHopOwner
Handle to the client that owns this next hop.

InterfaceIndex
Specifies the outgoing interface index.

State
Flags that indicates the state of this next hop. The following flags are used.

Constant	Description
RTM_NEXTHOP_STATE_CREATED	The next hop has been created.
RTM_NEXTHOP_STATE_DELETED	The next hop has been deleted.

Flags

Flags that convey status information for the next hop. The following flags are used.

Constant	Description
RTM_NEXTHOP_FLAGS_REMOTE	This next hop points to a remote destination that is not directly reachable. To obtain the complete path, the client must perform a recursive lookup.
RTM_NEXTHOP_FLAGS_DOWN	This flag is reserved for future use.

EntitySpecificInfo

Contains information specific to the client that owns this next hop.

RemoteNextHop

Handle to the destination with the indirect next-hop address. This member is only valid when the **Flags** member is set to RTM_NEXTHOP_FLAGS_REMOTE. This cached handle can prevent multiple lookups for this indirect next hop.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RTM_NET_ADDRESS, **RtmAddNextHop**, **RtmDeleteNextHop**, **RtmFindNextHop**, **RtmGetNextHopInfo**, **RtmGetNextHopPointer**, **RtmLockNextHop**, **RtmReleaseNextHopInfo**

RTM_NEXTHOP_LIST

The **RTM_NEXTHOP_LIST** structure contains a list of next hops used to determine equal-cost paths in a route.

```
typedef struct _RTM_NEXTHOP_LIST {
  USHORT             NumNextHops;
  RTM_NEXTHOP_HANDLE  NextHops[1];
} RTM_NEXTHOP_LIST, *PRTM_NEXTHOP_LIST;
```

Members

NumNextHops

Specifies the number of equal cost next hops in **NextHops**.

NextHops

Array of next-hop handles.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RTM_ROUTE_INFO

RTM_PREF_INFO

The **RTM_PREF_INFO** structure contains the information used when comparing any two routes. The value of the **Preference** member is given more weight than the value of the **Metric** member.

```
typedef struct _RTM_PREF_INFO {
  ULONG                Metric;
  ULONG                Preference;
} RTM_PREF_INFO, *PRTM_PREF_INFO;
```

Members

Metric
Specifies a metric. The metric is specific to a particular routing protocol.

Preference
Specifies a preference. The preference is determined by the router policy.

Remarks

Preference is more important than metric. The metric will only be checked if the preferences are equal.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RTM_ROUTE_INFO

RTM_REGN_PROFILE

The **RTM_REGN_PROFILE** structure contains information returned during the registration process. The information is used for later function calls (such as the maximum number of routes that can be returned by a call to **RtmGetEnumRoutes**).

```
typedef struct _RTM_REGN_PROFILE {
  UINT          MaxNextHopsInRoute;
  UINT          MaxHandlesInEnum;
  RTM_VIEW_SET  ViewsSupported;
  UINT          NumberOfViews;
} RTM_REGN_PROFILE, *PRTM_REGN_PROFILE;
```

Members

MaxNextHopsInRoute
Specifies the maximum number of equal-cost next hops in a route.

MaxHandlesInEnum
Specifies the maximum number of handles that can be returned in one call to **RtmGetEnumDests**, **RtmGetChangedDests**, **RtmGetEnumRoutes**, or **RtmGetListEnumRoutes**. The number of handles that can be returned is limited (and configurable) to improve efficiency and performance of the routing table manager.

ViewsSupported
Views supported by this address family.

NumberOfViews
Number of views.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RtmRegisterEntity

RTM_ROUTE_INFO

The **RTM_ROUTE_INFO** structure is used to exchange route information with the routing table manager. Do not change the read-only information.

```
typedef struct _RTM_ROUTE_INFO {
  //
  // Information that the owner can directly access
  // for read only
  //
  RTM_DEST_HANDLE     DestHandle;
  RTM_ENTITY_HANDLE   RouteOwner;
  RTM_NEXTHOP_HANDLE  Neighbour;
  UCHAR               State;
  //
```

(continued)

(continued)

```
    // Information that the owner can directly access
    // for read/write
    //
    UCHAR               Flags1;
    USHORT              Flags;
    RTM_PREF_INFO       PrefInfo;
    RTM_VIEW_SET        BelongsToViews;
    PVOID               EntitySpecificInfo;
    RTM_NEXTHOP_LIST    NextHopsList;
} RTM_ROUTE_INFO, *PRTM_ROUTE_INFO;
```

Members

DestHandle
Handle to the destination that owns the route.

RouteOwner
Handle to the client that owns this route.

Neighbour
Handle to the neighbor that informed the routing table manager of this route. This member is NULL for a link-state protocol.

State
Flags the specify the state of this route. The following flags are used.

Constant	Description
RTM_ROUTE_STATE_CREATED	Route has been created.
RTM_ROUTE_STATE_DELETING	Route is being deleted.
RTM_ROUTE_STATE_DELETED	Route has been deleted.

Flags1
Flags used for compatibility with RTMv1.

Flags
Flags used to specify information about the route. The following flags are used.

Constant	Description
RTM_ROUTE_FLAGS_ANY_ BCAST	The route is one of the following broadcast types: RTM_ROUTE_FLAGS_LIMITED_BC, RTM_ROUTE_FLAGS_ONES_NETBC, RTM_ROUTE_FLAGS_ONES_SUBNET_BC, RTM_ROUTE_FLAGS_ZEROS_NETBC, RTM_ROUTE_FLAGS_ZEROS_SUBNETBC
RTM_ROUTE_FLAGS_ANY_ MCAST	The route is one of the following multicast types: RTM_ROUTE_FLAGS_MCAST, RTM_ROUTE_FLAGS_LOCAL_MCAST

Constant	Description
RTM_ROUTE_FLAGS_ANY_UNICAST	The route is one of the following unicast types: RTM_ROUTE_FLAGS_LOCAL, RTM_ROUTE_FLAGS_REMOTE, RTM_ROUTE_FLAGS_MYSELF
RTM_ROUTE_FLAGS_LIMITED_BC	Indicates that this route is a limited broadcast address. Packets to this destination should not be forwarded.
RTM_ROUTE_FLAGS_LOCAL	Indicates a destination is on a directly reachable network.
RTM_ROUTE_FLAGS_LOCAL_MCAST	Indicates that this route is a route to a local multicast address.
RTM_ROUTE_FLAGS_MCAST	Indicates that this route is a route to a multicast address.
RTM_ROUTE_FLAGS_MYSELF	Indicates the destination is one of the router's addresses.
RTM_ROUTE_FLAGS_NET_BCAST	Flag grouping that contains: RTM_ROUTE_FLAGS_ONES_NETBC, RTM_ROUTE_FLAGS_ZEROS_NETBC
RTM_ROUTE_FLAGS_ONES_NETBC	Indicates that the destination matches an interface's "all-ones" broadcast address. If broadcast forwarding is enabled, packets should be received and resent out all appropriate interfaces.
RTM_ROUTE_FLAGS_ONES_SUBNETBC	Indicates that the destination matches an interface's "all-ones" subnet broadcast address. If subnet broadcast forwarding is enabled, packets should be received and resent out all appropriate interfaces.
RTM_ROUTE_FLAGS_REMOTE	Indicates that the destination is not on a directly reachable network.
RTM_ROUTE_FLAGS_ZEROS_SUBNETBC	Indicates that the destination matches an interface's "all-zeros" subnet broadcast address. If subnet broadcast forwarding is enabled, packets should be received and resent out all appropriate interfaces.
RTM_ROUTE_FLAGS_ZEROS_NETBC	Indicates that the destination matches an interface's "all-zeros" broadcast address. If broadcast forwarding is enabled, packets should be received and resent out all appropriate interfaces.

PrefInfo
Specifies the preference and metric information for this route.

BelongsToViews
Specifies the views that this route is included in.

EntitySpecificInfo
Contains the client-specific information for the client that owns this route.

NextHopsList
Specifies a list of equal-cost next hops.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

+ See Also

RTM_PREF_INFO, **RTM_NEXTHOP_LIST**, **RtmAddRouteToDest**, **RtmCreateRouteEnum**, **RtmGetExactMatchRoute**, **RtmGetRouteInfo**, **RtmGetRoutePointer**, **RtmLockRoute**, **RtmReleaseRouteInfo**

Routing Table Manager Version 2 Macros

RTMv2 uses the following macros:

Size of Structure Macros

RTM_SIZE_OF_DEST_INFO
RTM_SIZE_OF_ROUTE_INFO

IPv4 Address Macros

RTM_IPV4_GET_ADDR_AND_LEN
RTM_IPV4_GET_ADDR_AND_MASK
RTM_IPV4_LEN_FROM_MASK
RTM_IPV4_MAKE_NET_ADDRESS
RTM_IPV4_MASK_FROM_LEN
RTM_IPV4_SET_ADDR_AND_LEN
RTM_IPV4_SET_ADDR_AND_MASK

RTM_IPV4_GET_ADDR_AND_LEN

The **RTM_IPV4_GET_ADDR_AND_LEN** macro converts a generic net address and length to an IPv4 address and a length.

```
RTM_IPV4_GET_ADDR_AND_LEN(
  Addr,
  Len,
  NetAddress
);
```

Parameters

Addr
Receives the converted IPv4 address.

Len
Receives the converted length.

NetAddress
Specifies the network address to convert.

Remarks

For example, if a client supplies the *NetAddress* 10.10.10/24, the *Addr* 10.10.10.0 and the *Len* 24 are returned.

The macro is defined as follows:

```
#define RTM_IPV4_GET_ADDR_AND_LEN(Addr, Len, NetAddress)    \
        (Len) = (NetAddress)->NumBits;                      \
        (Addr) = (* (ULONG *) ((NetAddress)->AddrBits));    \
```

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RTM_IPV4_GET_ADDR_AND_MASK, **RTM_IPV4_LEN_FROM_MASK**, **RTM_IPV4_MAKE_NET_ADDRESS**, **RTM_IPV4_MASK_FROM_LEN**, **RTM_IPV4_SET_ADDR_AND_LEN**, **RTM_IPV4_SET_ADDR_AND_MASK**, **RTM_NET_ADDRESS**

RTM_IPV4_GET_ADDR_AND_MASK

The **RTM_IPV4_GET_ADDR_AND_MASK** macro converts a generic net address and length to an IPv4 address and mask.

```
RTM_IPV4_GET_ADDR_AND_MASK(
  Addr,
  Mask,
  NetAddress
);
```

Parameters

Addr
Receives the converted IPv4 address.

Mask
Receives the converted IPv4 mask.

NetAddress
Specifies the network address to convert.

Remarks

For example, if a client supplies the *NetAddress* 10.10.10/24, the *Addr* 10.10.10.0 and the *Mask* 255.255.255.255 are returned.

The macro is defined as follows:

```
#define RTM_IPV4_GET_ADDR_AND_MASK(Addr, Mask, NetAddress)  \
        (Addr) = (* (ULONG *) ((NetAddress)->AddrBits));     \
        (Mask) = RTM_IPV4_MASK_FROM_LEN((NetAddress)-       \
             >NumBits);
```

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RTM_IPV4_GET_ADDR_AND_LEN, **RTM_IPV4_LEN_FROM_MASK**, **RTM_IPV4_MAKE_NET_ADDRESS**, **RTM_IPV4_MASK_FROM_LEN**, **RTM_IPV4_SET_ADDR_AND_LEN**, **RTM_IPV4_SET_ADDR_AND_MASK**, **RTM_NET_ADDRESS**

RTM_IPV4_LEN_FROM_MASK

The **RTM_IPV4_LEN_FROM_MASK** macro converts an IPv4 mask to a generic route length.

```
RTM_IPV4_LEN_FROM_MASK(
  Len,
  Mask
);
```

Parameters

Len
  Receives the converted length

Mask
  Specifies the mask to convert.

Remarks

For example, if a client supplies the *Mask* 255.255.255.255, the *Len* 24 is returned, the mask is returned.

The macro is defined as follows.

```
#define RTM_IPV4_LEN_FROM_MASK(Len, Mask)                  \
        {                                                  \
            ULONG _Temp_ = ntohl(Mask);                    \
            (Len) = 0;                                     \
            RTM_CHECK_NTH_BIT(_Temp_, 16, (Len));          \
            RTM_CHECK_NTH_BIT(_Temp_,  8, (Len));          \
            RTM_CHECK_NTH_BIT(_Temp_,  4, (Len));          \
            while (_Temp_)                                 \
            {                                              \
                (Len) +=  1; _Temp_ <<=  1;                \
            }                                              \
        }                                                  \
```

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RTM_IPV4_GET_ADDR_AND_LEN, **RTM_IPV4_GET_ADDR_AND_MASK**, **RTM_IPV4_MAKE_NET_ADDRESS**, **RTM_IPV4_MASK_FROM_LEN**, **RTM_IPV4_SET_ADDR_AND_LEN**, **RTM_IPV4_SET_ADDR_AND_MASK**, **RTM_NET_ADDRESS**

RTM_IPV4_MAKE_NET_ADDRESS

The **RTM_IPV4_MAKE_NET_ADDRESS** macro converts an IPv4 address and a length to a generic **RTM_NET_ADDRESS** structure.

```
RTM_IPV4_MAKE_NET_ADDRESS(
  NetAddress,
  Addr,
  Len
);
```

Parameters

NetAddress
Receives the converted address structure.

Addr
Specifies the IPv4 address to convert.

Len
Specifies the length to convert.

Remarks

For example, if a client supplies the *Addr* 10.10.10.0 and the *Len* 24, the *NetAddress* 10.10.10/24 is returned.

The macro is defined as follows:

```
#define RTM_IPV4_MAKE_NET_ADDRESS(NetAddress, Addr, Len)     \
        RTM_IPV4_SET_ADDR_AND_LEN(NetAddress, Addr, Len)
```

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RTM_IPV4_GET_ADDR_AND_LEN, RTM_IPV4_GET_ADDR_AND_MASK, RTM_IPV4_LEN_FROM_MASK, RTM_IPV4_MASK_FROM_LEN, RTM_IPV4_SET_ADDR_AND_LEN, RTM_IPV4_SET_ADDR_AND_MASK, RTM_NET_ADDRESS

RTM_IPV4_MASK_FROM_LEN

The **RTM_IPV4_MASK_FROM_LEN** macro converts a generic route length to an IPv4 mask.

```
RTM_IPV4_MASK_FROM_LEN(
  Len
);
```

Parameters

Len
Specifies the generic length to convert.

Return Values

The return value is the size of the subnet mask.

Remarks

For example, if a client supplies the *Len* 24, the mask 255.255.255.255 is returned.

The macro is defined as follows:

```
#define RTM_IPV4_MASK_FROM_LEN(Len)                          \
        ((Len) ? htonl(~0 << (32 - (Len))): 0);              \
```

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

+ See Also

RTM_IPV4_GET_ADDR_AND_LEN, **RTM_IPV4_GET_ADDR_AND_MASK**, **RTM_IPV4_LEN_FROM_MASK**, **RTM_IPV4_MAKE_NET_ADDRESS**, **RTM_IPV4_SET_ADDR_AND_LEN**, **RTM_IPV4_SET_ADDR_AND_MASK**, **RTM_NET_ADDRESS**

RTM_IPV4_SET_ADDR_AND_LEN

The **RTM_IPV4_SET_ADDR_AND_LEN** macro converts an IPv4 address and a length to a generic **RTM_NET_ADDRESS** structure.

```
RTM_IPV4_SET_ADDR_AND_LEN(
  NetAddress,
  Addr,
  Len
);
```

Parameters

NetAddress
  Receives the converted address structure.

Addr
  Specifies the IPv4 address to convert.

Len
  Specifies the length to convert.

Remarks

For example, if a client supplies the *Addr* 10.10.10.0 and the *Len* 24, the *NetAddress* 10.10.10/24 is returned.

The macro is defined as follows.

```
#define RTM_IPV4_SET_ADDR_AND_LEN(NetAddress, Addr, Len)  \
        (NetAddress)->AddressFamily = AF_INET;            \
        (NetAddress)->NumBits  = (USHORT) (Len);          \
        (* (ULONG *) ((NetAddress)->AddrBits)) = (Addr);  \
```

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RTM_IPV4_GET_ADDR_AND_LEN, **RTM_IPV4_GET_ADDR_AND_MASK**, **RTM_IPV4_LEN_FROM_MASK**, **RTM_IPV4_MAKE_NET_ADDRESS**, **RTM_IPV4_MASK_FROM_LEN**, **RTM_IPV4_SET_ADDR_AND_MASK**, **RTM_NET_ADDRESS**

RTM_IPV4_SET_ADDR_AND_MASK

The **RTM_IPV4_SET_ADDR_AND_MASK** macro converts an IPv4 address and mask to a generic **RTM_NET_ADDRESS** structure.

```
RTM_IPV4_SET_ADDR_AND_MASK(
  NetAddress,
  Addr,
  Mask
);
```

Parameters

NetAddress
Receives the converted address structure.

Addr
Specifies the IPv4 address to convert.

Mask
Specifies the IPv4 mask to convert.

Remarks

For example, if a client supplies the *Addr* 10.10.10.0 and the *Mask* 255.255.255.255, the *NetAddress* 10.10.10/24 is returned.

The macro is defined as follows:

```
#define RTM_IPV4_SET_ADDR_AND_MASK(NetAddress, Addr, Mask)  \
        (NetAddress)->AddressFamily = AF_INET;              \
        (* (ULONG *) ((NetAddress)->AddrBits)) = (Addr);    \
        RTM_IPV4_LEN_FROM_MASK((NetAddress)->NumBits, Mask)
```

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

See Also

RTM_IPV4_GET_ADDR_AND_LEN, **RTM_IPV4_GET_ADDR_AND_MASK**, **RTM_IPV4_LEN_FROM_MASK**, **RTM_IPV4_MAKE_NET_ADDRESS**, **RTM_IPV4_MASK_FROM_LEN**, **RTM_IPV4_SET_ADDR_AND_LEN**, **RTM_NET_ADDRESS**

RTM_SIZE_OF_DEST_INFO

The **RTM_SIZE_OF_DEST_INFO** macro returns the size the destination information structure (**RTM_DEST_INFO**). The size of this structure is variable, and is based on the number of views for which it contains information. Use this macro when allocating memory for destination information.

```
ULONG RTM_SIZE_OF_DEST_INFO(
  NumViews
);
```

Parameters

NumViews
Specifies the number of views in the destination structure.

Return Values

The return value is the size of the destination information structure with the specified number of views.

Remarks

If the client will only use one view per destination, the client can allocate an **RTM_DEST_INFO** structure statically.

The macro is defined as follows:

```
#define RTM_BASIC_DEST_INFO_SIZE                                  \
    FIELD_OFFSET(RTM_DEST_INFO, ViewInfo)
#define RTM_DEST_VIEW_INFO_SIZE                                   \
    (sizeof(RTM_DEST_INFO) - RTM_BASIC_DEST_INFO_SIZE)
#define RTM_SIZE_OF_DEST_INFO(NumViews)                           \
    (RTM_BASIC_DEST_INFO_SIZE + (NumViews) *                      \
    RTM_DEST_VIEW_INFO_SIZE)
```

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

RTM_SIZE_OF_ROUTE_INFO

The **RTM_SIZE_OF_ROUTE_INFO** macro returns the size of the route information structure, **RTM_ROUTE_INFO**. The size of this structure is variable, and is based on the number of next hops associated with the route. Use this macro when allocating memory for route structures.

```
ULONG RTM_SIZE_OF_ROUTE_INFO(
  NumHops
);
```

Parameters

NumHops

Specifies the number of next hops in the route structure.

Return Values

The return value is the size of the route information structure with the specified number of next hops.

Remarks

If the client will only allocate one next hop per route, the client can allocate an **RTM_ROUTE_INFO** structure statically.

The macro is defined as follows:

```
#define RTM_BASIC_ROUTE_INFO_SIZE                              \
    FIELD_OFFSET(RTM_ROUTE_INFO, NextHopsList.NumNextHops)
#define RTM_SIZE_OF_ROUTE_INFO(NumHops)                        \
    (RTM_BASIC_ROUTE_INFO_SIZE + (NumHops) *                   \
     sizeof(RTM_NEXTHOP_HANDLE))
```

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

Routing Table Manager Version 2 Constants

The RTMv2 functions use the following constants:

View Flags	**Routing Table Query Flags**
Route Flags	**Enumeration Flags**
Next Hop Flags	**Change Notification Flags**

View Flags

Constant	Value	Description
RTM_MAX_ ADDRESS_SIZE	16	Max address size for an address family.
RTM_MAX_VIEWS	32	Maximum number of views that can be active in the routing table.
RTM_VIEW_ID_UCAST	0	Specifies a unicast view.
RTM_VIEW_ID_MCAST	1	Specifies a multicast view.
RTM_VIEW_MASK_SIZE	0x20	Specifies the maximum number of views that can be configured.
RTM_VIEW_MASK_NONE	0x00000000	Return information regardless of the view.
RTM_VIEW_MASK_ANY	0x00000000	Return destinations from all views. This is the default value.
RTM_VIEW_MASK_UCAST	0x00000001	Return destinations from the unicast view.
RTM_VIEW_MASK_MCAST	0x00000002	Return destinations from the multicast view.
RTM_VIEW_MASK_ALL	0xFFFFFFFF	Return information from all views.

Route Flags

State of the Route Constants

Constant	Value	Description
RTM_ROUTE_STATE_CREATED	0	Route has been created.
RTM_ROUTE_STATE_DELETING	1	Route is being deleted.
RTM_ROUTE_STATE_DELETED	q	Route has been deleted.

Route Update Flags

Constant	Value	Description
RTM_ROUTE_CHANGE_FIRST	0x01	Indicates that the routing table manager should not check the **Neighbour** member of the **RTM_ROUTE_INFO** structure when determining when two routes are equal.
RTM_ROUTE_CHANGE_NEW	0x02	Returned by the routing table manager to indicate a new route was created.

(continued)

(continued)

Constant	Value	Description
RTM_ROUTE_CHANGE_BEST	0x00010000	Returned by the routing table manager to indicate that the route that was added or updated was the best route, or that because of the change, a new route became the best route.

Unicast Flags

Constant	Value	Description
RTM_ROUTE_FLAGS_LOCAL	0x0010	Indicates a destination is on a directly reachable network.
RTM_ROUTE_FLAGS_REMOTE	0x0020	Indicates that the destination is not on a directly reachable network.
RTM_ROUTE_FLAGS_MYSELF	0x0040	Indicates the destination is one of the router's addresses.

Broadcast and Multicast Flags

Constant	Value	Description
RTM_ROUTE_FLAGS_MCAST	0x0100	Indicates that this route is a route to a multicast address.
RTM_ROUTE_FLAGS_LOCAL_ MCAST	0x0200	Indicates that this route is a route to a local multicast address.
RTM_ROUTE_FLAGS_LIMITED_ BC	0x0400	Indicates that this route is a limited broadcast address. Packets to this destination should not be forwarded.
RTM_ROUTE_FLAGS_ZEROS_ NETBC	0x1000	Indicates that the destination matches an interface's "all-zeros" broadcast address. If broadcast forwarding is enabled, packets should be received and resent out all appropriate interfaces.
RTM_ROUTE_FLAGS_ZEROS_ SUBNETBC	0x2000	Indicates that the destination matches an interface's "all-zeros" subnet broadcast address. If subnet broadcast forwarding is enabled, packets should be received and resent out all appropriate interfaces.
RTM_ROUTE_FLAGS_ONES_ NETBC	0x4000	Indicates that the destination matches an interface's "all-ones" broadcast address. If broadcast forwarding is enabled, packets should be received and resent out all appropriate interfaces.

Constant	Value	Description
RTM_ROUTE_FLAGS_ONES_ SUBNETBC	0x8000	Indicates that the destination matches an interface's "all-ones" subnet broadcast address. If subnet broadcast forwarding is enabled, packets should be received and resent out all appropriate interfaces.

Grouping of Flags

Group	Members	Description
RTM_ROUTE_FLAGS_ FORWARDING	RTM_ROUTE_FLAGS_MARTIAN, RTM_ROUTE_FLAGS_BLACKHOLE, RTM_ROUTE_FLAGS_DISCARD, RTM_ROUTE_FLAGS_INACTIVE	Specifies any forwarding flags.
RTM_ROUTE_FLAGS_ ANY_UNICAST	RTM_ROUTE_FLAGS_LOCAL, RTM_ROUTE_FLAGS_REMOTE, RTM_ROUTE_FLAGS_MYSELF	Specifies any unicast flags.
RTM_ROUTE_FLAGS_ ANY_MCAST	RTM_ROUTE_FLAGS_MCAST, RTM_ROUTE_FLAGS_LOCAL_MCAST	Specifies any unicast flags.
RTM_ROUTE_FLAGS_ SUBNET_BCAST	RTM_ROUTE_FLAGS_ONES_SUBNET_ BC, RTM_ROUTE_FLAGS_ZEROS_ SUBNETBC	Specifies any subnet broadcast flags.
RTM_ROUTE_FLAGS_ NET_BCAST	RTM_ROUTE_FLAGS_ONES_NETBC, RTM_ROUTE_FLAGS_ZEROS_NETBC	Specifies any net-wide broadcast flags.
RTM_ROUTE_FLAGS_ ANY_BCAST	RTM_ROUTE_FLAGS_LIMITED_BC, RTM_ROUTE_FLAGS_ONES_NETBC, RTM_ROUTE_FLAGS_ONES_SUBNET_ BC, RTM_ROUTE_FLAGS_ZEROS_NETBC, RTM_ROUTE_FLAGS_ZEROS_ SUBNETBC	Specifies any of the subnet or net-wide broadcast flags.

Next Hop Flags

Next Hop State Flags

Constant	Value	Description
RTM_NEXTHOP_STATE_ CREATED	0	Indicates that the next hop was created.
RTM_NEXTHOP_STATE_ DELETED	1	Indicates that the next hop was deleted.

Next Hop Flags

Constant	Value	Description
RTM_NEXTHOP_FLAGS_ REMOTE	0x0001	This next hop points to a remote destination that is not directly reachable. To obtain the complete path, the client must perform a recursive lookup.
RTM_NEXTHOP_FLAGS_DOWN	0x0002	This flag is reserved for future use.

Next Hop Added

Constant	Value	Description
RTM_NEXTHOP_CHANGE_NEW	0x01	A new next hop was created.

Routing Table Query Flags

Constant	Value	Description
RTM_MATCH_NONE	0x00000000	Match none of the criteria; all routes for the destination are returned.
RTM_MATCH_OWNER	0x00000001	Match routes with same owner.
RTM_MATCH_ NEIGHBOUR	0x00000002	Match routes with the same neighbor.
RTM_MATCH_ PREF	0x00000004	Match routes that have the same preference.
RTM_MATCH_ NEXTHOP	0x00000008	Match routes that have the same next hop.
RTM_MATCH_ NTERFACE	0x00000010	Match routes that are on the same interface.
RTM_MATCH_FULL	0x0000FFF F	Match routes with all criteria.
RTM_BEST_ PROTOCOL	0	Return a route regardless of which protocol owns it.
RTM_THIS_ PROTOCOL	~0	Returns the best route for the calling protocol.

Enumeration Flags

Constant	Value	Description
RTM_ENUM_START	0x00000000	Enumerate routes or destinations starting at 0/0.
RTM_ENUM_NEXT	0x00000001	Enumerate routes or destinations starting at the specified address/mask length (such as 10/8). The enumeration continues to the end of the routing table.
RTM_ENUM_RANGE	0x00000002	Enumerate routes or destinations in the specified subtree specified by the address/mask length (such as 10/8).
RTM_ENUM_ALL_DESTS	0x00000000	Return all destinations.
RTM_ENUM_OWN_DESTS	0x01000000	Return only those destinations that the client owns.
RTM_ENUM_ALL_ROUTES	0x00000000	Return all routes.
RTM_ENUM_OWN_ROUTES	0x00010000	Return only those routes that the client owns.

Change Notification Flags

Constant	Value	Description
RTM_NUM_CHANGE_TYPES	3	Specifies the number of change types that are currently used by the routing table manager.
RTM_CHANGE_TYPE_ALL	0x0001	Notify the client of any change to a destination.
RTM_CHANGE_TYPE_BEST	0x0002	Notify the client of changes to the best route, or when the best route changes.
RTM_CHANGE_TYPE_FORWARDING	0x0004	Notify the client of any best route changes that affect forwarding (such as next hop changes).
RTM_NOTIFY_ONLY_MARKED_DESTS	0x00010000	Notify the client of changes to destinations that the client has marked. If this flag is not specified, change notification messages for all destinations are sent.

Routing Table Manager Version 2 Enumerations

The RTMv2 functions use the following enumerations:

RTM_EVENT_TYPE

RTM_EVENT_TYPE

Enumerates the events that the routing table manager can notify the client about using the **RTM_EVENT_CALLBACK** callback.

```
typedef enum _RTM_EVENT_TYPE {
  RTM_ENTITY_REGISTERED,
  RTM_ENTITY_DEREGISTERED,
  RTM_ROUTE_EXPIRED,
  RTM_CHANGE_NOTIFICATION
} RTM_EVENT_TYPE, *PRTM_EVENT_TYPE;
```

Values

RTM_ENTITY_REGISTERED
A client has just registered with the routing table manager.

RTM_ENTITY_DEREGISTERED
A client has just unregistered.

RTM_ROUTE_EXPIRED
A route has timed out.

RTM_CHANGE_NOTIFICATION
A change notification has been made.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Rtmv2.h.

+ See Also

RTM_EVENT_CALLBACK

Routing Table Manager Version 2 Simple Data Types

The RTMv2 API defines several simple data types. The following table lists these data types.

Data Type	Description	Typedef
RTM_VIEW_ID, *PRTM_VIEW_ID	Identifies a particular view	INT
DWORD RTM_VIEW_SET, *PRTM_VIEW_SET	Identifies a set of views; expressed as a mask	DWORD
RTM_ENTITY_HANDLE, *PRTM_ENTITY_HANDLE, RTM_DEST_HANDLE, *PRTM_DEST_HANDLE, RTM_ROUTE_HANDLE, *PRTM_ROUTE_HANDLE, RTM_NEXTHOP_HANDLE, *PRTM_NEXTHOP_HANDLE, RTM_ENUM_HANDLE, *PRTM_ENUM_HANDLE, RTM_ROUTE_LIST_HANDLE, *PRTM_ROUTE_LIST_HANDLE, RTM_NOTIFY_HANDLE, *PRTM_NOTIFY_HANDLE	Handles pointing to RTMv2 data	HANDLE
RTM_ENTITY_METHOD_TYPE, *PRTM_ENTITY_METHOD_TYPE	Identifies methods exported by a registered client	DWORD
RTM_ENTITY_EXPORT_METHOD, *PRTM_ENTITY_EXPORT_METHOD	Specifies the common prototype for client methods	_ENTITY_METHOD
RTM_ROUTE_CHANGE_FLAGS, PRTM_ROUTE_CHANGE_FLAGS	Input and output flags used to specify the state when a route is added or updated	DWORD
RTM_NEXTHOP_CHANGE_FLAGS, *PRTM_NEXTHOP_CHANGE_FLAGS	Output flags used to specify the state when a next hop is added	DWORD
RTM_MATCH_FLAGS, *PRTM_MATCH_FLAGS	Input flags used to specify criteria when matching routes in the routing table	DWORD
RTM_ENUM_FLAGS, *PRTM_ENUM_FLAGS	Identifies enumerations	DWORD
RTM_NOTIFY_FLAGS, *PRTM_NOTIFY_FLAGS	Output flags used to specify which type of notification is being issued; composed as follows: (Change Types I Dests) a client is interested in	DWORD
RTM_EVENT_CALLBACK, *PRTM_EVENT_CALLBACK;	Identifies the callback used to notify clients that a change has occurred in route state or clients registered	RTM_EVENT_ CALLBACK

CHAPTER 12

Multicast Group Manager

Multicast Group Manager Overview

This chapter describes the Multicast Group Manager (MGM) technology, a feature of Microsoft® Windows® 2000.

Multicasting allows a host to send data to only those destinations that specifically request to receive the data. Multicasting saves network bandwidth because multicast data is sent only to hosts that request the data. In this way, multicasting differs from sending broadcast data, since broadcast data is sent to all hosts. Multicasting also saves network bandwidth because data travels over any link only once. Multicasting saves server bandwidth because a server has to send only one multicast message instead of one unicast message per receiver. Examples of popular multicast applications are online meetings and Internet radio.

The MGM API enables developers to write multicast routing protocols that work within the architecture for Microsoft® Windows® 2000 Routing and RAS (RRAS).

When more than one multicast routing protocol is enabled on a router, the multicast group manager coordinates operations between all routing protocols. The multicast group manager informs each routing protocol when group membership changes occur, and when multicast data from a new source or destined for a new group is received.

The MGM API provides the following features:

- Protocol registration
- Group management
- Multicast forwarding entry enumeration
- Callback definitions for multicast routing protocols

This overview describes the components of the MGM architecture, the client scenarios that are used to implement MGM, and programming issues to consider when using the MGM API.

The multicast group manager is incorporated into Microsoft Windows 2000 as a part of the RRAS technology. It is not available for Microsoft® Windows NT® version 4.0.

Components of the Multicast Architecture

The major components of the multicast routing architecture are explained in the following topics:

- Router
- Multicast Routing Protocol
- Interface
- Multicast Source
- Multicast Group
- (s, g), (*, g), and (*, *) pairs
- Destination
- Next Hop
- Multicast Group Manager Client

The section *How the Multicast Architecture Fits Together* explains how these components interact.

Router

A router is a Windows NT/Windows 2000 server that is running the RRAS service. Such a server handles data forwarding and runs routing protocols.

Multicast Routing Protocol

Router clients are service providers that function within the framework of the router architecture. The Windows NT/Windows 2000 routing architecture is designed to be extended by router client modules. Routing protocols are one type of router client that is supported by the router.

A multicast routing protocol manages group membership and controls the path that multicast data takes over the network. Examples of multicast routing protocols include: Protocol Independent Multicast (PIM), Multicast Open Shortest Path First (MOSPF), and Distance Vector Multicast Routing Protocol (DVMRP). The Internet Group Management Protocol (IGMP) is a special multicast routing protocol that acts as an intermediary between hosts and routers.

Interface

An interface is a logical connection to a network. Each interface is identified by a unique interface *index*. Routing protocols such as MOSPF deal with all types of interfaces similarly.

In the case of a LAN interface, the interface corresponds to an actual physical device in the computer, the LAN adapter. In the case of a WAN interface, the interface is mapped to a port at the time a connection is established. WAN interfaces can be based on tunnels, the port could be a Virtual Private Network (VPN) port.

Windows 2000 supports a "point-to-multipoint" interface. This interface can be viewed as a collection of point-to-point links that share a single termination point. The MGM API has extended interface identification to use a next-hop address. The next-hop address uniquely identifies the exact link in this collection of point-to-point links.

Multicast Source

A multicast source is the IP address of the host from which the multicast data originated. A source is referred to by either of the symbols, "S" or "s".

Multicast Group

A multicast group is a Class D IP address in the range of 224.0.0.0- 239.255.255.255. Messages that are sent to an address in this range are not destined for a single target. Instead, these messages can be received by any host that makes a request to receive data destined for the group the host is interested in receiving messages for. A multicast group is referred to by either of the symbols "G" or "g".

(s, g), (*, g), and (*, *) Pairs

The notation (s, g) represents a specific source and group. The notation (*, g) represents a wildcard source and a specific group. All messages to the group "g" are included.

The notation (*, *) represents a wildcard source and wildcard group. All messages from all sources, and bound to all groups, are included.

These notations are used to describe the addition and removal of group memberships.

Destination

A destination is a host that has joined a multicast group. Such a host has informed the local router (using IGMP) that it is interested in receiving data sent to a specific multicast group.

Next Hop

A next hop is the next router on the path towards a destination. Packets from a source are forwarded to the destination on a hop-by-hop basis.

The address of the router that is the next-hop route is used to uniquely identify links on a point-to-multipoint interface, where all the links share the same interface index.

Multicast Group Manager Client

A client is an entity that calls an MGM function, such as a routing protocol.

The MGM functions are called primarily by multicast routing protocols. Developers of multicast routing protocols use MGM functions to.

- maintain group membership
- control interface ownership
- receive notifications from MGM regarding requests for multicast data generated by other multicast routing protocols

Specific administration applications that must monitor multicast forwarding entries (MFEs) can do so without adding or removing group membership. Administrative program developers use MGM functions to review information in MFEs and the group membership list. MFEs are the cached forwarding information that MGM creates based on group membership. The MFEs that are retrieved from the multicast group manager can provide statistical information. An administrative program can then use this information to determine the appropriate actions (for instance, an administrative program could perform actions that are based on the volume of packets on a specific interface).

How the Multicast Architecture Fits Together

This section describes a sample configuration and how the components fit together.

Figure 12-1 shows the relationship between the various components of a router.

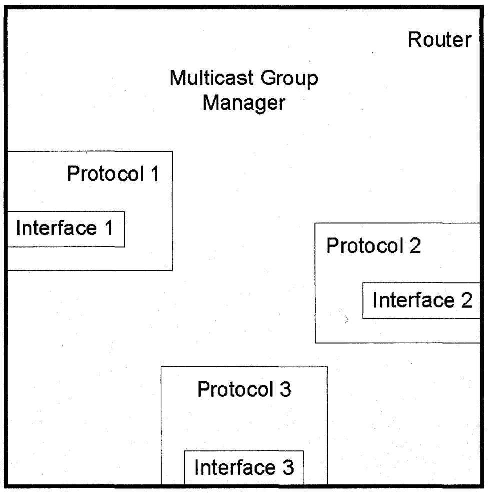


Figure 12-1: Router Components.

The multicast group manager is a part of the RRAS service running on a Windows 2000 server that is operating as a router.

The router shown has three multicast routing protocols (Protocol 1, Protocol 2, Protocol 3) running on it. Each protocol can own one or more interfaces (in this case, Protocol 1 owns Interface 1, Protocol 2 owns Interface 2, and Protocol 3 owns Interface 3). Each interface can be owned by only one routing protocol (in addition to IGMP).

The multicast group manager runs on the router and coordinates group information between the routing protocols.

Figure 12-2 shows the relationship between two routers in a multicast architecture.

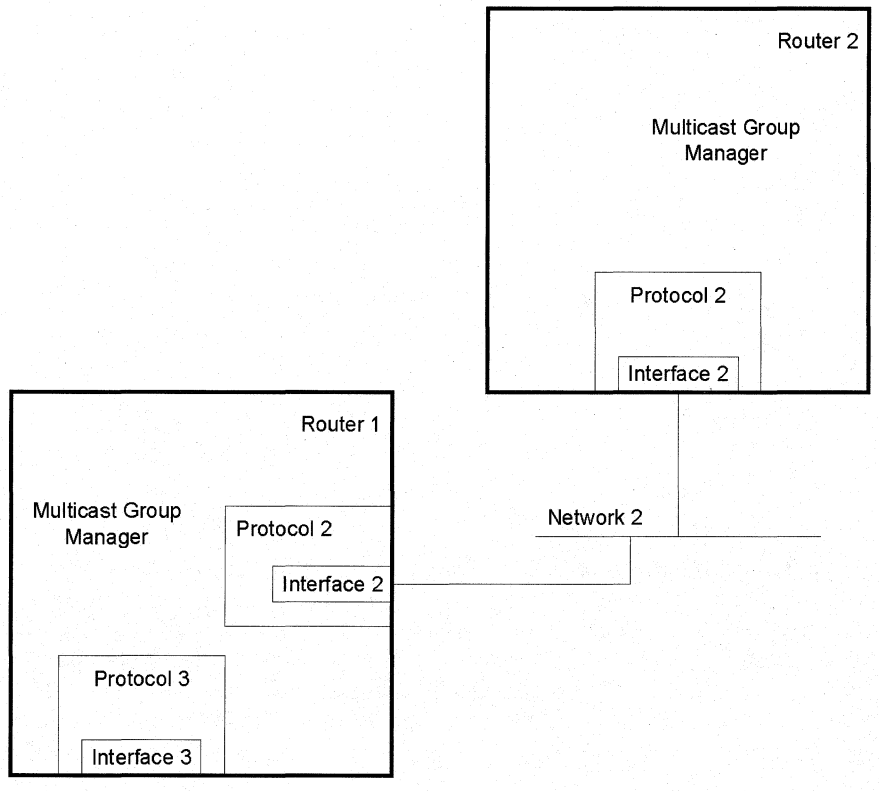


Figure 12-2: Relationship Between Two Routers in a Multicast Architecture.

Router 2 sends multicast data to Network 2 on Interface 2. Router 1 receives multicast data from Network 2 on Interface 2. On both routers, Protocol 2 owns the respective Interface 2.

Figure 12-3 shows the path data from a multicast source (to a multicast group) takes to reach the host that has joined the multicast group. The routers in the illustration use the same configuration as previous illustrations; however, the interface and protocol details are not shown in order to keep the figure simple.

Host 1 joins multicast group G on Network 3. Router 3 learns about G via IGMP. The multicast group manager on Router 3 notifies Protocol 3 on Router 3. Protocol 3 on Router 3 then notifies Protocol 3 on Router 1. In turn, Protocol 3 on Router 1 notifies the multicast group manager on Router 1. The multicast group manager on Router 1 then notifies Protocol 1 and Protocol 2. Protocol 2 may inform Router 2, if the protocol is designed to do so.

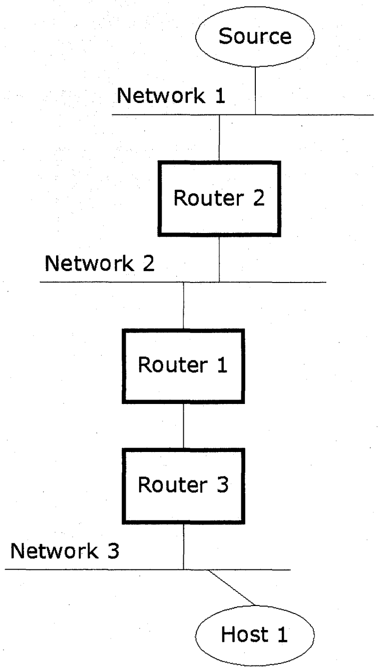


Figure 12-3: Path from a Multicast Source to a Multicast Group.

A source on Network 1 sends data to Group G. Data sent from Source S goes first to Router 2, which then forwards it to Router 1 using Interface 2 (since Router 2 has been informed by Protocol 2 that receivers are present downstream). Then Router 1 forwards the data to Router 3 (since Router 1 has been informed by Protocol 2 that receivers are present downstream). Router 3 forwards the data to Network 3, and therefore it arrives at Host 1.

For further information on multicast protocol interaction, see RFC 2715, *Interoperability Rules for Multicast Routing Protocols*.

Using the Multicast Group Manager

This section contains the following information:

- MGM Programming Issues
- Callbacks
- Multicast Routing Protocol Scenario
- Administration Program Scenario

MGM Programming Issues

Multicast group manager clients should be written based on the following assumptions:

- Function calls must be made from within the routing process. If functions are called from another process, their results will not be valid; the client will not interact with MGM.
- Clients that call MGM functions must provide their own error checking, for validity, of the values of parameters that are passed to the multicast group manager. MGM functions do not return detailed error messages about invalid parameters; an ERROR_INVALID_PARAMETER value is returned without explanation.
- Clients should exercise caution in using locks while calling MGM functions to prevent deadlocks. When calling MGM functions, clients should not hold any locks that might simultaneously be held in a callback from the multicast group manager.

Callbacks

There are two types of callbacks in the MGM API:

- Routing Protocol Callbacks
- IGMP Enable and Disable Callbacks

These callbacks, combined with the MGM API function, create the ongoing notification cycle between routing protocols, IGMP, and the multicast group manager.

Routing Protocol Callbacks

This topic covers the calls into routing protocols.

Join Alert Callbacks

When the multicast group manager is notified that there are new receivers present for a group, the multicast group manager invokes the **PMGM_JOIN_ALERT_CALLBACK** callback to inform the routing protocols of the change. This callback indicates to the routing protocols that they must request multicast data for one or more specified groups.

The multicast group manager uses a predefined set of rules that govern when this callback is invoked. This set of rules is based on both the type of join sent by the client and the order the join requests were received in.

When a wildcard (*, g) join for a group is received from a client, the multicast group manager invokes the **PMGM_JOIN_ALERT_CALLBACK** callback for all other registered clients. When a wildcard join for a group is received from a second client, the multicast group manager invokes this callback for the first client to join the group. The multicast group manager does not invoke this callback for any subsequent joins to the group.

When a source-specific join for a group is received (s, g), the multicast group manager invokes this callback only for the client that owns the incoming interface towards the source “s”.

Prune Alert Callbacks

When the multicast group manager is notified that old receivers are leaving a group, the multicast group manager invokes the **PMGM_JOIN_ALERT_CALLBACK** callback to notify the routing protocols of the change. This callback indicates to the routing protocols that they must stop requesting multicast data for the specified groups.

The multicast group manager has a predefined set of rules that govern when this callback is invoked. These rules are based on both the type of prune request sent by the client and the order the prune requests were received in.

When a wildcard (*, g) prune for a group is received and the final interface is being removed for the second-to-last client (that is, when only the interfaces for a single client remain), the multicast group manager invokes this callback for the last remaining client. After the final interface is removed for the last client for the source and group (that is, when no other interfaces remain), then the callback is invoked for all the other clients that are registered with the multicast group manager.

When a source-specific prune for a group is received (s, g), the multicast group manager invokes this callback only for the client that owns the incoming interface towards the source “s”.

Local Join Callback

After the multicast group manager is notified by IGMP that new receivers are present for a group on an interface, MGM invokes the **PMGM_LOCAL_JOIN_CALLBACK** callback to the routing protocol on that interface (if one exists) to notify the routing protocol of the change. The **PMGM_LOCAL_JOIN_CALLBACK** and **PMGM_LOCAL_LEAVE_CALLBACK** callbacks are used to synchronize forwarding between IGMP and routing protocols.

Local Leave Callback

After the multicast group manager is notified by IGMP that there are no more receivers present for a group on an interface, MGM invokes the **PMGM_LOCAL_LEAVE_CALLBACK** callback to the routing protocol on that interface (if one exists) to notify the routing protocol of the change. This callback and the **PMGM_LOCAL_JOIN_CALLBACK** callback are used to synchronize forwarding between IGMP and routing protocols.

Wrong Interface Alert Callback

After the kernel forwarder receives multicast data from a specific source on the wrong interface, it notifies the multicast group manager. The multicast group manager then invokes this callback to the routing protocol that owns the interface on which the data incorrectly arrived.

This callback is not currently implemented in this version of the MGM API.

RPF Alert Callback

After the multicast group manager receives notification of a packet from a new source or of a packet that is destined for a new group, the multicast group manager looks up the route to the source in the multicast view of the routing table.

The multicast group manager then invokes the **PMGM_RPF_CALLBACK** for the protocol that owns the incoming interface.

When this callback is invoked, the routing protocol can change the incoming interface if the routing protocol must receive the data for the group on another interface.

IGMP Enable and Disable Callbacks

The multicast group manager uses two callbacks to IGMP to coordinate changes in interface ownership from IGMP to a routing protocol, and from a routing protocol to IGMP.

The multicast group manager allows IGMP to coexist on an interface with another routing protocol (such as DVMRP).

After the ownership of an interface changes, the multicast group manager first calls **PMGM_DISABLE_IGMP_CALLBACK**. IGMP must stop adding and deleting group memberships on the specified interface until it receives the **PMGM_ENABLE_IGMP_CALLBACK** callback.

The multicast group manager calls **PMGM_ENABLE_IGMP_CALLBACK** after the change of interface ownership is complete.

Multicast Routing Protocol Scenario

All multicast routing protocols go through three basic phases: startup, operation, and shutdown. The following sections outline a basic set of interactions between a multicast routing protocol and the multicast group manager.

Multicast Routing Protocol Startup Tasks

The following table summarizes the startup interaction between a routing protocol and the multicast group manager. The first column describes actions that the routing protocol performs and the responses of the routing protocol to the multicast group manager. The second column describes the multicast group manager's responses to the routing protocol and any actions the multicast group manager performs (such as callbacks). The third column presents any additional information.

Each row of the table represents one step.

Routing Protocol Action	MGM Action	Notes
Register with the multicast group manager using **MgmRegisterMProtocol**.	Return to the routing protocol a handle that the protocol must use to identify itself in subsequent MGM calls.	
If an interface is already owned, determine the protocol that owns it using **MgmGetProtocolOnInterface**.		
Take ownership of all the interfaces on which the protocol is enabled, using **MgmTakeInterfaceOwnership**.	If IGMP has already taken ownership of an interface and the **MgmTakeInterfaceOwnership** function call is received for the same interface, contact IGMP using the **PMGM_DISABLE_IGMP_ CALLBACK**. Once all internal MGM changesregarding interface ownership have been made, contact IGMP again using **PMGM_ENABLE_IGMP_ CALLBACK**.	Only one protocol can own an interface at a given time, in addition to IGMP.
Determine the current state of group membership on the router. This is done using the group membership enumeration functions: **MgmGroupEnumerationStart**, **MgmGroupEnumerationGetNext**, and **MgmGroupEnumerationEnd**.	Return the list of groups.	Routing protocols can use the results to determine what actions to take based on the groups already joined. See the topic on *Enumerating Groups* for a complete guide to using these functions.

Multicast Routing Protocol Operational Tasks

The following table summarizes the operational interactions between a routing protocol and the multicast group manager. The first column describes the actions that the routing protocol performs and the routing protocol's responses to the multicast group manager. The second column describes the multicast group manager's responses to the routing protocol and any actions the multicast group manager performs (such as callbacks). The third column presents any additional information.

Each row of the table represents one step.

The tasks listed in this table do not occur in any specific order; rather, they occur based on the status of multicast group memberships. The table below is an example order.

Routing Protocol Action	MGM Action	Notes
Manage group memberships based on protocol information received on any interfaces that the protocol owns. Use the following functions to manage group memberships: **MgmAddGroup MembershipEntry** and **MgmDeleteGroup MembershipEntry**.	Add to and delete from the outgoing interface list for the specified (s, g), (*, g), and (*, *) entries. This list represents the set of interfaces on which data for this group is forwarded. The data for this group is from the specified source.	
	Send alerts back to the other routing protocols in the form of callbacks: join/leave groups (**PMGM_JOIN_ALERT_CALLBACK**, **PMGM_PRUNE_ALERT_CALLBACK**), group membership changed by IGMP (**PMGM_LOCAL_JOIN_CALLBACK**, **PMGM_LOCAL_LEAVE_CALLBACK**), data received on wrong interface (**PMGM_WRONG_IF_CALLBACK**), data received from new sources or to a new group (**PMGM_CREATION_ALERT_ CALLBACK** and **PMGM_RPF_CALLBACK**).	Using these callbacks, MGM is able to coordinate packet forwarding when several multicast routing protocols are present on a router.
Enumerate the multicast forwarding entries (MFEs), using the **MgmGetFirstMfe**, **MgmGetNextMfe**, and **MgmGetMfe** Make make decisions about multicast data based on the enumeration results.	Return the requested MFEs. Return ERROR_NO_MORE ITEMS when there are no more MFEs to return.	Use the **MgmGetFirstMfeStats**, **MgmGetNextMfeStats**, **MgmGetMfeStats** to enumerate MFE statistics. See the *Enumerating MFEs* topic for the complete enumeration procedure.
Modify the upstream neighbor in an MFE using **MgmSetMfe**.		

Multicast Routing Protocol Shutdown Tasks

The following table summarizes the interactions between the multicast group manager and the routing protocol when the routing protocol is shutting down. The first column describes the actions that the routing protocol performs and the routing protocol's responses to the multicast group manager. The second column describes the multicast group manager's responses to the routing protocol and any actions the multicast group manager performs (such as callbacks). The third column presents any additional information.

Each row of the table represents one step.

Routing Protocol Action	MGM Action	Notes
Release ownership of each interface that the routing protocol owns using **MgmReleaseInterface Ownership**.	If IGMP is also running on the interface that was just released by a routing protocol, contact IGMP using the **PMGM_DISABLE_IGMP_CALLBACK** callback. Once all internal (to MGM) changes regarding interface ownership have been made, contact IGMP again using **PMGM_ENABLE_IGMP_CALLBACK**. Delete all the forwarding entries associated with this interface.	
Unregister with MGM using **MgmDeRegister MProtocol**.	Destroy the handle that was returned o the routing protocol by the call to **MgmDeRegisterMProtocol**routing protocol's handle.	The routing protocol can no longer use this handle to call MGM functions.

Administration Program Scenario

Administration programs call a subset of MGM functions that are related to enumerating groups and multicast forwarding entries (MFEs). These functions do not need to register with the multicast group and receive a handle. The following sections outline a basic set of interactions between an administration program and the multicast group manager.

Enumerating Groups

The following table summarizes the interactions between an administration program and the multicast group manager. The first column describes the actions that the administration program performs and the administration program's responses to the multicast group manager. The second column describes the multicast group manager's responses to the administration program. The third column presents any additional information.

Each row of the table represents one step.

Administration Program Action	MGM Action	Notes
Call **MgmGroupEnumerationStart** to obtain a handle to an enumeration.	Return a handle.	
Call **MgmGroupEnumeration GetNext** to obtain groups.	Return as many groups as fit in the buffer supplied by the client. If no groupscan be returned in the supplied buffer, return ERROR_INSUFFICIENT_BUFFER and the size of the buffer that is needed to return one group. Return ERROR_NO_MORE_ITEMS when there are no more groups.	
If ERROR_INSUFFICIENT_ BUFFER is received, call **MgmGroupEnumerationGetNext** again using a buffer of the size indicated.		
Continue the enumeration until ERROR_NO_MORE_ITEMS is received.		
Call **MgmGroupEnumerationEnd** to destroy the handle to the enumeration.	Destroy the handle.	

Enumerating MFEs

The following table summarizes the interactions between an administration programand the multicast group manager. The first column describes the actions that the administration program performs and the administration program's responses to the multicast group manager. The second column describes the multicast group manager's responses to the administration program. The third column presents any additional information.

Each row of the table represents one step.

Routing Protocol Action	MGM Action	Notes
Call **MgmGetFirstMfe** to obtain MFEs.	Return as many MFEs as fit in the buffer supplied by the client. If no MFEs can be returned in the supplied buffer, return ERROR_INSUFFICIENT_BUFFER and the size of the buffer that is needed to return one MFE.	Clients can also retrieve MFE statistics using the corresponding statistics functions, **MgmGetFirstMfeStats** and **MgmGetNextMfeStats**.
If ERROR_ INSUFFICIENT_BUFFER is received, call **MgmGetFirstMfe** again using a buffer of the size indicated.		
Call **MgmGetNextMfe**, supplying as one of the parameters the last MFE that was returned by the previous call to **MgmGetFirstMfe**.	Return as many MFEs as fit in the buffer supplied by the client. If no MFEs can be returned in the supplied buffer, return ERROR_INSUFFICIENT_BUFFER and the size of the buffer that is needed for one MFE. Return ERROR_NO_MORE_ITEMS when no more MFEs remain.	
If ERROR_ INSUFFICIENT_BUFFER is received, call **MgmGetNextMfe** again using a buffer of the size indicated. Continue the enumeration until ERROR_NO_MORE_ ITEMS is received.		

Note Use the **MgmGetMfe** and **MgmGetMfeStats** functions to retrieve a specific MFE or specific set of MFE statistics.

Multicast Group Manager Reference

The following documentation describes the functions, callbacks, structures, and enumeration types to use when working with the multicast group manager.

Multicast Group Manager Functions

The following functions are used to control group membership and work with the MFE cache:

Protocol Registration Functions

MgmRegisterMProtocol
MgmDeRegisterMProtocol

Interface Ownership Functions

MgmGetProtocolOnInterface
MgmTakeInterfaceOwnership
MgmReleaseInterfaceOwnership

Group Membership Functions

MgmAddGroupMembershipEntry
MgmDeleteGroupMembershipEntry

Multicast Forwarding Entry Enumeration Functions

MgmGetFirstMfe
MgmGetNextMfe
MgmGetMfe
MgmGetFirstMfeStats
MgmGetNextMfeStats
MgmGetMfeStats

Multicast Forwarding Entry Update Functions

MgmSetMfe

Group Membership Enumeration Functions

MgmGroupEnumerationStart
MgmGroupEnumerationGetNext
MgmGroupEnumerationEnd

MgmAddGroupMembershipEntry

The **MgmAddGroupMembershipEntry** function notifies the multicast group manager that there are receivers for the specified groups on the specified interface. The receivers can restrict the set of sources from which they should receive multicast data by specifying a source range.

A multicast routing protocol calls this function when it is notified that there are receivers for a multicast group on an interface. The protocol must call this function so that multicast data can be forwarded out over an interface.

```
DWORD
MgmAddGroupMembershipEntry(
  HANDLE hProtocol,
  DWORD dwSourceAddr,
  DWORD dwSourceMask,
  DWORD dwGroupAddr,
  DWORD dwGroupMask,
  DWORD dwIfIndex,
  DWORD dwIfNextHopIPAddr
);
```

Parameters

hProtocol
[in] Handle to the protocol obtained from a previous call to **MgmRegisterMProtocol**.

dwSourceAddr
[in] Specifies the range of source addresses from which to receive group data. Specify zero to receive data from all sources (a wildcard receiver for a group); otherwise, specify the IP address of the source or source network.

dwSourceMask
[in] Specifies the subnet mask that corresponds to *dwSourceAddr*. The *dwSourceAddr* and *dwSourceMask* parameters are used together to define a range of sources from which to receive data. Specify zero for this parameter if zero was specified for *dwSourceAddr* (a wildcard receiver).

dwGroupAddr
[in] Specifies the range of multicast groups for which to receive data. Specify zero to receive all groups (a wildcard receiver); otherwise, specify the IP address of the group.

dwGroupMask
[in] Specifies the subnet mask that corresponds to dwGroupAddr. The dwGroupAddr and dwGroupMask parameters are used together to define a range of multicast groups. Specify zero for this parameter if zero was specified for dwGroupAddr (a wildcard receiver).

dwIfIndex

[in] Specifies the interface on which to add the group membership. Multicast packets are forwarded out of this interface.

dwIfNextHopIPAddr

[in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as non-broadcast multiple access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, specify zero.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INVALID_PARAMETER	Invalid handle to a client.
ERROR_NOT_ENOUGH_MEMORY	Not enough memory to complete this operation.

Remarks

This version of the Multicast Group Manager API supports only wildcard sources or specific sources, not source ranges. The same restriction applies to groups, that is, no group ranges are permitted.

When this function is called, the multicast group manager may invoke **PMGM_JOIN_ALERT_CALLBACK** to notify other routing protocols that there are new receivers.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

+ See Also

MgmDeleteGroupMembershipEntry, **PMGM_JOIN_ALERT_CALLBACK**

MgmDeleteGroupMembershipEntry

The **MgmDeleteGroupMembershipEntry** function notifies the multicast group manager that there are no more receivers present for the specified groups on the specified interface.

A multicast routing protocol calls this function after it is notified that there are no more receivers for a multicast group on an interface. The protocol must call this function to stop multicast data from being forwarded out over an interface.

```
DWORD
MgmDeleteGroupMembershipEntry(
  HANDLE hProtocol,
  DWORD dwSourceAddr,
  DWORD dwSourceMask,
  DWORD dwGroupAddr,
  DWORD dwGroupMask,
  DWORD dwIfIndex,
  DWORD dwIfNextHopIPAddr
);
```

Parameters

hProtocol
[in] Handle to the protocol obtained from a previous call to **MgmRegisterMProtocol**.

dwSourceAddr
[in] Specifies the range of source addresses from which to stop receiving group data. Specify zero to stop receiving data from all sources (a wildcard receiver for a group); otherwise, specify the IP address of the source or source network.

dwSourceMask
[in] Specifies the subnet mask that corresponds to *dwSourceAddr*. The *dwSourceAddr* and *dwSourceMask* parameters are used together to define a range of sources from which to stop receiving data. Specify zero for this parameter if zero was specified for *dwSourceAddr* (a wildcard receiver).

dwGroupAddr
[in] Specifies the range of multicast groups for which to stop receiving data. Specify zero to stop receiving all groups (a wildcard receiver); otherwise, specify the IP address of the group.

dwGroupMask
[in] Specifies the subnet mask that corresponds to *dwGroupAddr*. The *dwGroupAddr* and *dwGroupMask* parameters are used together to define a range of multicast groups. Specify zero for this parameter if zero was specified for *dwGroupAddr* (a wildcard receiver).

dwIfIndex
[in] Specifies the interface on which to delete the group membership. Multicast packets for the specified groups will no longer be forwarded out over this interface.

dwIfNextHopIPAddr

[in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as non-broadcast multiple access (NBMA) interfaces, or the internal interface on which all dial-up clients connect).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, specify zero.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INVALID_PARAMETER	Invalid handle to a client, or the interface is owned by another protocol.
ERROR_NOT_FOUND	The specified interface was not found.

Remarks

This version of the Multicast Group Manager API supports only wildcard sources or specific sources, not source ranges. The same restriction applies to groups (that is, no group ranges are permitted).

When this function is called, the multicast group manager may invoke **PMGM_PRUNE_ALERT_CALLBACK** to notify other routing protocols that no more receivers are present.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MgmAddGroupMembershipEntry, **PMGM_PRUNE_ALERT_CALLBACK**

MgmDeRegisterMProtocol

The **MgmDeRegisterMProtocol** function unregisters a client handle obtained from a call to **MgmRegisterMProtocol**.

```
DWORD
MgmDeRegisterMProtocol(
  HANDLE hProtocol
);
```

Parameters

hProtocol

[in] Handle obtained from a previous call to **MgmRegisterMProtocol**.

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_CAN_NOT_COMPLETE	Could not complete the call to this function. Client did not first release the interfaces it owns.
ERROR_INVALID_PARAMETER	Invalid handle to a client.

Remarks

A multicast protocol must deregister only after releasing interface ownership for all interfaces that it owns.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MgmRegisterMProtocol, **MgmReleaseInterfaceOwnership**

MgmGetFirstMfe

The **MgmGetFirstMfe** function retrieves MFEs starting at the beginning of the MFE list. The function can retrieve zero, one, or more MFEs. The number of MFEs returned depends on the size of the MFEs and the size of the buffer supplied when the function is called.

The data returned in the buffer is ordered first by group, and then by the sources within a group.

```
DWORD
MgmGetFirstMfe(
  PDWORD pdwBufferSize,
  PBYTE pbBuffer,
  PDWORD pdwNumEntries
);
```

Parameters

pdwBufferSize

[in, out] On input, *pdwBufferSize* is a pointer to a **DWORD** value containing the size, in bytes, of *pbBuffer*. On output, if the return value of **MgmGetFirstMfe** is ERROR_INSUFFICIENT_BUFFER, *pdwBufferSize* receives the minimum size *pbBuffer* must be to hold the MFE; otherwise *pdwBufferSize* remains unchanged.

pbBuffer

[out] On input, the client must supply a pointer to a buffer. On output, *pbBuffer* receives one or more MFEs. Each MFE is a **MIB_IPMCAST_MFE** structure.

pdwNumEntries

[out] On input, the client must supply a pointer to a **DWORD** value. On output, *pdwNumEntries* receives the number of MFEs in *pbBuffer*.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INSUFFICIENT_BUFFER	The specified buffer is too small for even one MFE. The client should check *pdwBufferSize* for the minimum buffer size required to retrieve one MFE.
ERROR_MORE_DATA	More MFEs are available.
ERROR_NO_MORE_ITEMS	No more MFEs are available. Zero or more MFEs were returned; check *pdwNumEntries* to verify how many were returned.

Remarks

This function is used to begin sequential retrieval of MFEs; use **MgmGetNextMfe** to continue the retrieval process.

Note The minimum size of *pbBuffer* is not fixed; it is different for each MFE. Use the **SIZEOF_MIB_MFE** macro to determine the size of each MFE returned in the buffer.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

+ See Also

MgmGetFirstMfeStats, **MgmGetMfe**, **MgmGetMfeStats**, **MgmGetNextMfe**, **MgmGetNextMfeStats**, **MIB_IPMCAST_MFE**, **SIZEOF_MIB_MFE**

MgmGetFirstMfeStats

The **MgmGetFirstMfeStats** function retrieves MFE statistics starting at the beginning of the MFE list. The function can retrieve zero, one, or more MFE statistics. The number of entries returned depends on the size of the entries and the size of the buffer supplied when the function is called.

The data returned in the buffer is ordered first by group, and then by the sources within a group. The statistics returned include the packets and bytes received, as well as the packets forwarded, on each outgoing interface.

```
DWORD
MgmGetFirstMfeStats(
  PDWORD pdwBufferSize,
  PBYTE pbBuffer,
  PDWORD pdwNumEntries
);
```

Parameters

pdwBufferSize
[in, out] On input, *pdwBufferSize* is a pointer to a **DWORD** value containing the size, in bytes, of *pbBuffer*. On output, if the return value of **MgmGetFirstMfeStats** is ERROR_INSUFFICIENT_BUFFER, *pdwBufferSize* receives the minimum size *pbBuffer* must be to hold statistics for the MFE; otherwise *pdwBufferSize* remains unchanged.

pbBuffer
[out] On input, the client must supply a pointer to a buffer. On output, *pbBuffer* receives statistics for one or more MFEs. Each set of statistics is returned in a **MIB_IPMCAST_MFE_STATS** structure.

pdwNumEntries
[out] On input, the client must supply a pointer to a **DWORD** value. On output, *pdwNumEntries* receives the number of MFEs for which statistics are returned in *pbBuffer*.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INSUFFICIENT_BUFFER	The specified buffer is too small to hold the statistics for even one MFE. The client should check *pdwBufferSize* for the minimum buffer size required to retrieve statistics for one MFE.
ERROR_MORE_DATA	More MFE statistics are available.
ERROR_NO_MORE_ITEMS	No more MFE statistics are available. Zero or more sets of MFE statistics were returned; check *pdwNumEntries* to verify how many were returned.

Remarks

This function is used to begin sequential retrieval of MFE statistics; use **MgmGetNextMfeStats** to continue the retrieval process.

Note The minimum size of *pbBuffer* is not fixed; it is different for each MFE for which statistics are returned. Use the **SIZEOF_MIB_MFE_STATS** macro to determine the size of each group of statistics returned in the buffer.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

+ See Also

MgmGetFirstMfe, **MgmGetMfe**, **MgmGetMfeStats**, **MgmGetNextMfe**, **MgmGetNextMfeStats**, **MIB_IPMCAST_MFE_STATS**, **SIZEOF_MIB_MFE_STATS**

MgmGetMfe

The **MgmGetMfe** function retrieves a specific MFE.

```
DWORD
MgmGetMfe(
  PMIB_IPMCAST_MFE pimm,
  PDWORD pdwBufferSize,
  PBYTE pbBuffer
);
```

Parameters

pimm

[in] Pointer to a **MIB_IPMCAST_MFE** structure specifying the MFE to retrieve. The information to be returned is specified by the **dwSource** and **dwGroup** members of the **MIB_IPMCAST_MFE** structure.

pdwBufferSize

[in, out] On input, *pdwBufferSize* is a pointer to a **DWORD** value that contains the size, in bytes, of *pbBuffer*. On output, if the return value of **MgmGetMfe** is ERROR_INSUFFICIENT_BUFFER, *pdwBufferSize* receives the minimum size *pbBuffer* must be to hold the MFE; otherwise *pdwBufferSize* remains unchanged.

pbBuffer

[out] On input, the client must supply a pointer to a buffer. On output, *pbBuffer* receives the specified MFE. The MFE is a **MIB_IPMCAST_MFE** 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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INSUFFICIENT_BUFFER	The specified buffer is too small to hold the MFE. The client should check *pdwBufferSize* for the minimum buffer size required to retrieve one MFE.
ERROR_NOT_FOUND	The specified MFE was not found.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MgmGetFirstMfe, **MgmGetFirstMfeStats**, **MgmGetMfeStats**, **MgmGetNextMfe**, **MgmGetNextMfeStats**, **MIB_IPMCAST_MFE**

MgmGetMfeStats

The **MgmGetMfeStats** function retrieves the statistics for a specific MFE. The statistics returned include the packets and bytes received, and the packets forwarded, on each outgoing interface.

```
DWORD
MgmGetMfeStats(
  PMIB_IPMCAST_MFE pimm,
  PDWORD pdwBufferSize,
  PBYTE pbBuffer
);
```

Parameters

pimm
[in] Pointer to a **MIB_IPMCAST_MFE** structure specifying the MFE to retrieve. The information to be returned is specified by the **dwSource** and **dwGroup** members of the **MIB_IPMCAST_MFE** structure.

pdwBufferSize
[in, out] On input, *pdwBufferSize* is a pointer to a **DWORD** value that contains the size, in bytes, of *pbBuffer*. On output, if the return value of **MgmGetMfeStats** is ERROR_INSUFFICIENT_BUFFER, *pdwBufferSize* receives the minimum size *pbBuffer* must be to hold the MFE; otherwise *pdwBufferSize* remains unchanged.

pbBuffer
[out] On input, the client must supply a pointer to a buffer. On output, *pbBuffer* receives one or more sets of MFE statistics. Each set of statistics is returned in a **MIB_IPMCAST_MFE_STATS** 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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INSUFFICIENT_BUFFER	The specified buffer is too small for the statistics for even one MFE. The client should check *pdwBufferSize* for the minimum buffer size required to retrieve statistics for one MFE.
ERROR_NOT_FOUND	The specified MFE was not found.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MgmGetFirstMfe, **MgmGetFirstMfeStats**, **MgmGetMfe**, **MgmGetNextMfe**, **MgmGetNextMfeStats**, **MIB_IPMCAST_MFE_STATS**

MgmGetNextMfe

The **MgmGetNextMfe** function retrieves one or more MFEs. The routing table manager retrieves the MFE that follows the specified MFE. The function can retrieve zero, one, or more MFEs. The number of MFEs returned depends on the size of the MFEs and the size of the buffer supplied when the function is called.

The data returned in the buffer is ordered first by group, and then by the sources within a group.

```
DWORD
MgmGetNextMfe(
  PMIB_IPMCAST_MFE pimmStart,
  PDWORD pdwBufferSize,
  PBYTE pbBuffer,
  PDWORD pdwNumEntries
);
```

Parameters

pimmStart
[in] Pointer to a **MIB_IPMCAST_MFE** structure that specifies from where to begin retrieving MFEs. Use the **dwSource** and **dwGroup** members from the last MFE that was returned by the previous call to **MgmGetFirstMfe** or **MgmGetNextMfe**.

pdwBufferSize
[in, out] On input, *pdwBufferSize* is a pointer to a **DWORD** value that contains the size, in bytes, of *pbBuffer*. On output, if the return value of **MgmGetNextMfe** is ERROR_INSUFFICIENT_BUFFER, *pdwBufferSize* receives the minimum size *pbBuffer* must be to hold the MFE; otherwise *pdwBufferSize* remains unchanged.

pbBuffer
[out] On input, the client must supply a pointer to a buffer. On output, *pbBuffer* receives one or more MFEs. Each MFE is a **MIB_IPMCAST_MFE** structure.

pdwNumEntries
[out] On input, the client must supply a pointer to a **DWORD** value. On output, *pdwNumEntries* receives the number of MFEs in *pbBuffer*.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INSUFFICIENT_BUFFER	The specified buffer is too small for even one MFE. The client should check *pdwBufferSize* for the minimum buffer size required to retrieve one MFE.
ERROR_MORE_DATA	More MFEs are available.
ERROR_NO_MORE_ITEMS	No more MFEs are available. Zero or more MFEs were returned; check *pdwNumEntries* to verify how many were returned.

Remarks

This function is used to continue the sequential retrieval of MFEs; use **MgmGetFirstMfe** to start the retrieval process.

In general, to retrieve MFEs, first call **MgmGetFirstMfe**. Then, call **MgmGetNextMfe** one or more times, until there are no more MFEs to return. Each call to **MgmGetNextMfe** should start after the last MFE returned by the previous call to **MgmGetNextMfe** (or the initial call to **MgmGetFirstMfe**) , by specifying the last source and group in the buffer returned by a previous call.

Note The minimum size of *pbBuffer* is not fixed; it is different for each MFE. Use the **SIZEOF_MIB_MFE** macro to determine the size of each MFE returned in the buffer.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MgmGetFirstMfe, **MgmGetFirstMfeStats**, **MgmGetMfe**, **MgmGetMfeStats**, **MgmGetNextMfeStats**, **MIB_IPMCAST_MFE**, **SIZEOF_MIB_MFE**

MgmGetNextMfeStats

The **MgmGetNextMfeStats** function retrieves one or more sets of MFE statistics. The routing table manager retrieves the set of statistics that follows the specified MFE. The function can retrieve zero, one, or more sets MFE statistics. The number of entries returned depends on the size of the entries and the size of the buffer supplied when the function is called.

The data returned in the buffer is ordered first by group, then by the sources within a group. The statistics returned include the packets and bytes received, as well as the packets forwarded, on each outgoing interface.

```
DWORD
MgmGetNextMfeStats(
  PMIB_IPMCAST_MFE pimmStart,
  PDWORD pdwBufferSize,
  PBYTE pbBuffer,
  PDWORD pdwNumEntries
);
```

Parameters

pimmStart
[in] Pointer to a **MIB_IPMCAST_MFE** structure that specifies from where to begin retrieving MFE statistics. Use the **dwSource** and **dwGroup** members from the last MFE returned by the previous call to **MgmGetFirstMfeStats** or **MgmGetNextMfeStats**.

pdwBufferSize
[in, out] On input, *pdwBufferSize* is a pointer to a **DWORD** value that contains the size, in bytes, of *pbBuffer*. On output, if the return value of **MgmGetNextMfeStats** is ERROR_INSUFFICIENT_BUFFER, *pdwBufferSize* receives the minimum size *pbBuffer* must be to hold the MFE; otherwise *pdwBufferSize* remains unchanged.

pbBuffer
[out] On input, the client must supply a pointer to a buffer. On output, *pbBuffer* receives one or more sets of MFE statistics. Each set of statistics is returned in a **MIB_IPMCAST_MFE_STATS** structure.

pdwNumEntries
[out] On input, the client must supply a pointer to a **DWORD** value. On output, *pdwNumEntries* receives the number of MFE statistics in *pbBuffer*.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INSUFFICIENT_BUFFER	The specified buffer is too small to hold the statistics for even one MFE. The client should check *pdwBufferSize* for the minimum buffer size required to retrieve statistics for one MFE.
ERROR_MORE_DATA	More MFE statistics are available.
ERROR_NO_MORE_ITEMS	No more MFE statistics are available. Zero or more sets of MFE statistics were returned; check *pdwNumEntries* to verify how many were returned.

Remarks

This function is used to continue the sequential retrieval of MFE statistics; use **MgmGetFirstMfeStats** to start the retrieval process.

In general, to retrieve MFE statistics, first call **MgmGetFirstMfeStats**. Then, call **MgmGetNextMfeStats** one or more times, until there are no more MFEs to return. Each call to **MgmGetNextMfeStats** start after the last MFE returned by **MgmGetNextMfeStats** (or the initial call to **MgmGetFirstMfeStats)** , by specifying the last source and group in the buffer returned by a previous call.

Note The minimum size of *pbBuffer* is not fixed; it is different for each MFE for which statistics are returned. Use the **SIZEOF_MIB_MFE_STATS** macro to determine the size of each group of statistics returned in the buffer.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

+ See Also

MgmGetFirstMfe, **MgmGetFirstMfeStats**, **MgmGetMfe**, **MgmGetMfeStats**, **MgmGetNextMfe**, **MIB_IPMCAST_MFE_STATS**, **SIZEOF_MIB_MFE**

MgmGetProtocolOnInterface

The **MgmGetProtocolOnInterface** function retrieves the protocol identifier of the protocol that owns the specified interface.

```
DWORD
MgmGetProtocolOnInterface(
  DWORD dwIfIndex,
  DWORD dwIfNextHopAddr,
  PDWORD pdwIfProtocolId,
  PDWORD pdwIfComponentId
);
```

Parameters

dwIfIndex

[in] Specifies the index of the interface for which to retrieve the protocol identifier.

dwIfNextHopAddr

[in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as non-broadcast multiple access (NBMA) interfaces, or the internal interface on which all dial-up clients connect).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, specify zero.

pdwIfProtocolId

[out] On input, the client must supply a pointer to a **DWORD** value. On output, *pdwIfProtocolId* receives the identifier of the protocol on the interface specified by *dwIfIndex*.

pdwIfComponentId

[out] On input, the client must supply a pointer to a **DWORD** value. On output, *pdwIfComponentId* receives the component identifier for the instance of the protocol on the interface. This parameter is used with *pdwIfProtocolId* to uniquely identify an instance of a routing protocol.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_NOT_FOUND	The specified interface was not found in the multicast group manager.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MgmReleaseInterfaceOwnership, **MgmTakeInterfaceOwnership**

MgmGroupEnumerationEnd

The **MgmGroupEnumerationEnd** function releases the specified enumeration handle that was obtained from a previous call to **MgmGroupEnumerationStart**.

```
DWORD
MgmGroupEnumerationEnd(
  HANDLE hEnum
);
```

Parameters

hEnum

[in] Specifies the enumeration handle to release.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INVALID_PARAMETER	Invalid enumeration handle.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MgmGroupEnumerationGetNext, **MgmGroupEnumerationStart**

MgmGroupEnumerationGetNext

The **MgmGroupEnumerationGetNext** function retrieves the next set of group entries. The information that is returned by this function lists the groups that have been joined. For source-specific joins, the sources for those groups are also returned. The groups are not returned in any particular order.

```
DWORD
MgmGroupEnumerationGetNext(
  HANDLE hEnum,
  PDWORD pdwBufferSize,
  PBYTE pbBuffer,
  PDWORD pdwNumEntries
);
```

Parameters

hEnum

[in] Handle to the enumeration that was obtained from a previous call to **MgmGroupEnumerationStart**.

pdwBufferSize

[in, out] On input, *pdwBufferSize* is a pointer to a **DWORD** value that contains the size, in bytes, of *pbBuffer*. On output, if the return value of **MgmGroupEnumerationGetNext** is ERROR_INSUFFICIENT_BUFFER, *pdwBufferSize* receives the minimum size that *pbBuffer* must be to hold the group entry; otherwise *pdwBufferSize* remains unchanged.

pbBuffer

[out] On input, the client must supply a pointer to a buffer. On output, *pbBuffer* receives one or more group entries. Each group entry is a **SOURCE_GROUP_ENTRY** structure.

pdwNumEntries

[out] On input, the client must supply a pointer to a **DWORD** value. On output, *pdwNumEntries* receives the number of groups in *pbBuffer*.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INSUFFICIENT_BUFFER	The specified buffer is too small to hold even one group. The client should check *pdwBufferSize* for the minimum buffer size required to retrieve one group.
ERROR_INVALID_PARAMETER	Invalid handle to an enumeration.
ERROR_MORE_DATA	More groups are available.
ERROR_NO_MORE_ITEMS	No more groups are available. Zero or more groups were returned; check *pdwNumEntries* to verify how many were returned.
ERROR_NOT_ENOUGH_MEMORY	Not enough memory to complete this operation.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MgmGroupEnumerationEnd, **MgmGroupEnumerationStart**, **SOURCE_GROUP_ENTRY**

MgmGroupEnumerationStart

The **MgmGroupEnumerationStart** function obtains an enumeration handle that is later used to list the groups that have been joined. After the client obtains the handle, it should use the **MgmGroupEnumerationGetNext** function to enumerate the groups.

```
DWORD
MgmGroupEnumerationStart(
  HANDLE hProtocol,
  MGM_ENUM_TYPES metEnumType,
  HANDLE *phEnumHandle
);
```

Parameters

hProtocol
[in] Handle to the protocol obtained from a previous call to **MgmRegisterMProtocol**.

metEnumType
[in] Specifies the type of enumeration. The following enumerations are available.

Enumeration	Meaning
ALL_SOURCES	Retrieves wildcard joins (*, g) and source-specific joins (s, g).
ANY_SOURCE	Retrieves group entries that have at least one source specified.

phEnumHandle
[out] Returns the handle to the enumeration. Use this handle in calls to **MgmGroupEnumerationGetNext** and **MgmGroupEnumerationEnd**.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INVALID_PARAMETER	Invalid handle to a protocol.
ERROR_NOT_ENOUGH_MEMORY	Not enough memory to complete this operation.

! Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

+ See Also

MGM_ENUM_TYPES, **MgmGroupEnumerationEnd**, **MgmGroupEnumerationGetNext**

MgmRegisterMProtocol

The **MgmRegisterMProtocol** function is used by clients to register with the multicast group manager. When the registration is complete, the multicast group manager returns a handle to the client. The client must supply this handle in subsequent MGM function calls.

```
DWORD
MgmRegisterMProtocol(
  PROUTING_PROTOCOL_CONFIG prpiInfo,
  DWORD dwProtocolId,
  DWORD dwComponentId,
  HANDLE *phProtocol
);
```

Parameters

prpiInfo
[in] Pointer to a structure that contains callbacks into the client that is registering.

dwProtocolId
[in] Specifies the identifier of the client. The identifier is unique for each client.

dwComponentId
[in] Specifies the component identifier for the instance of the client. This parameter is used with *dwProtocolId* to uniquely identify an instance of a client.

phProtocol

[out] In input, the client must supply a pointer to a handle. On output, *phProtocol* receives the registration handle for the client. This handle must be used in subsequent calls to the multicast group manager.

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_ALREADY_EXISTS	Cannot register the specified client because an entry with the same protocol and component identifier already exists.
ERROR_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_NOT_ENOUGH_MEMORY	Not enough memory to complete this operation.

Remarks

Registering a protocol is the first operation any multicast routing protocol or other client should perform. After registration, the protocol should take ownership of the appropriate interfaces before adding or deleting group memberships.

Only one client may take ownership of an interface at any given time. Multiple routing protocols may be registered with the multicast group manager, each protocol owning different interfaces.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MgmDeRegisterMProtocol, **MgmTakeInterfaceOwnership**, **ROUTING_PROTOCOL_CONFIG**

MgmReleaseInterfaceOwnership

The **MgmReleaseInterfaceOwnership** function is used by a client to relinquish ownership of an interface. When this function is called, all group MFEs maintained by the multicast group manager on behalf of the client for the specified interface are deleted.

```
DWORD
MgmReleaseInterfaceOwnership(
  HANDLE hProtocol,
  DWORD dwIfIndex,
  DWORD dwIfNextHopAddr
);
```

Parameters

hProtocol
[in] Handle to the protocol obtained from a previous call to **MgmRegisterMProtocol**.

dwIfIndex
[in] Specifies the index of the interface to release.

dwIfNextHopAddr
[in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as non-broadcast multiple access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, specify zero.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INVALID_PARAMETER	Invalid handle to a client, or the interface was not found.

Remarks

A client must release ownership of all the interfaces it owns before unregistering itself with the **MgmDeRegisterMProtocol** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MgmDeRegisterMProtocol, **MgmGetProtocolOnInterface**, **MgmTakeInterfaceOwnership**

MgmSetMfe

The **MgmSetMfe** function changes the upstream neighbor for an MFE. An MFE contains the information about which interface is receiving, and which interfaces are forwarding, multicast data.

```
DWORD
MgmSetMfe(
  HANDLE hProtocol,
  PMIB_IPMCAST_MFE pmimm
);
```

Parameters

hProtocol
[in] Handle to the protocol obtained from a previous call to **MgmRegisterMProtocol**.

pmimm
[in] Pointer to a **MIB_IPMCAST_MFE** structure that specifies the MFE to change. Specify the new neighbor in the **dwUpstreamNeighbor** member.

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_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INVALID_PARAMETER	Invalid handle to a client.
ERROR_NOT_FOUND	The specified MFE was not found.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MIB_IPMCAST_MFE

MgmTakeInterfaceOwnership

The **MgmTakeInterfaceOwnership** function is called by a client (such as a routing protocol) when it is enabled on an interface.

Only one client can take ownership of a given interface at any time. The only exception to this rule is the IGMP. IGMP can coexist with another client on an interface.

```
DWORD
MgmTakeInterfaceOwnership(
  HANDLE hProtocol,
  DWORD dwIfIndex,
  DWORD dwIfNextHopAddr
);
```

Parameters

hProtocol
: [in] Handle to the protocol obtained from a previous call to **MgmRegisterMProtocol**.

dwIfIndex
: [in] Specifies the index of the interface of which to take ownership.

dwIfNextHopAddr
: [in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as non-broadcast multiple access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

  For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, specify zero.

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_ALREADY_EXISTS	The specified interface is already owned by another routing protocol.
ERROR_CAN_NOT_COMPLETE	Could not complete the call to this function.
ERROR_INVALID_PARAMETER	Invalid handle to a client.
ERROR_NOT_ENOUGH_MEMORY	Not enough memory to complete this operation.

Remarks

The client must take ownership of an interface only after registering itself with the multicast group manager and before it adds group membership entries.

Requirements

Windows NT/2000: Requires Windows 2000.
Header: Declared in Mgm.h.
Library: Use Rtm.lib.

See Also

MgmGetProtocolOnInterface, **MgmRegisterMProtocol**, **MgmReleaseInterfaceOwnership**

Multicast Group Manager Callbacks

The multicast group manager uses the following callbacks to notify clients (typically, routing protocols) of events and state changes:

Routing Protocol Callbacks

PMGM_CREATION_ALERT_CALLBACK
PMGM_JOIN_ALERT_CALLBACK
PMGM_PRUNE_ALERT_CALLBACK
PMGM_LOCAL_JOIN_CALLBACK
PMGM_LOCAL_LEAVE_CALLBACK
PMGM_RPF_CALLBACK
PMGM_WRONG_IF_CALLBACK

IGMP-Only Callbacks

PMGM_DISABLE_IGMP_CALLBACK
PMGM_ENABLE_IGMP_CALLBACK

PMGM_CREATION_ALERT_CALLBACK

The **PMGM_CREATION_ALERT_CALLBACK** is a call into a routing protocol. This call determines the subset of interfaces owned by the routing protocol on which a multicast packet from a "new" source should be forwarded.

When a packet sent from a new source, or destined for a new group, arrives on an interface, the multicast group manager creates a new MFE. The multicast group manager then issues this callback to those routing protocols that have outgoing interfaces in this new MFE. A routing protocol can choose to disable the forwarding of data from the source to the group on specific interfaces.

```
typedef DWORD(*PMGM_CREATION_ALERT_CALLBACK)(
  DWORD dwSourceAddr,
  DWORD dwSourceMask,
  DWORD dwGroupAddr,
    DWORD dwGroupMask,
    DWORD dwInIfIndex,
    DWORD dwInIfNextHopAddr,
    DWORD dwIfCount,
    PMGM_IF_ENTRY pmieOutIfList
);
```

Parameters

dwSourceAddr

[in] Specifies the address of the source from which the multicast data was received.

dwSourceMask

[in] Specifies the subnet mask that corresponds to *dwSourceAddr*. The *dwSourceAddr* and *dwSourceMask* parameters are used together to define a range of sources from which to receive data. This parameter is set to zero if *dwSourceAddr* was also set to zero. This parameter is not currently used.

dwGroupAddr

[in] Specifies the multicast group for which the data is destined.

dwGroupMask

[in] Specifies the subnet mask that corresponds to *dwGroupAddr*. The *dwGroupAddr* and *dwGroupMask* parameters are used together to define a range of multicast groups. This parameter is set to zero if *dwGroupAddr* was also set to zero. This parameter is currently not used.

dwInIfIndex

[in] Specifies the interface on which the multicast data from the source should arrive.

dwInIfNextHopAddr

[in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as non-broadcast multiple access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, MGM sets *dwInIfNextHopAddr* to zero.

dwIfCount

[in] Specifies the number of interfaces in *pmieOutIfList*.

pmieOutIfList

[in, out] On input, a pointer to the set of interfaces owned by the protocol on which that data will be forwarded. On return, the protocol prevents forwarding (if the prevention of forwarding is required) on any of its interfaces by setting the **bEnabled** member of the corresponding **MGM_IF_ENTRY** structure to FALSE.

! Requirements

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

See Also

MGM_IF_ENTRY

PMGM_DISABLE_IGMP_CALLBACK

The **PMGM_DISABLE_IGMP_CALLBACK** is a call into the IGMP to notify that a routing protocol is taking or releasing ownership of an interface on which IGMP is enabled.

When this callback is invoked, IGMP should stop adding and deleting group memberships on the specified interface:

```
typedef DWORD(*PMGM_DISABLE_IGMP_CALLBACK)(
  DWORD dwIfIndex,
  DWORD dwIfNextHopAddr
);
```

Parameters

dwIfIndex

[in] Specifies the interface on which to disable IGMP.

dwIfNextHopAddr

[in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as non-broadcast multiple access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, MGM sets *dwInIfNextHopAddr* to zero.

Requirements

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

See Also

PMGM_ENABLE_IGMP_CALLBACK

PMGM_ENABLE_IGMP_CALLBACK

The **PMGM_ENABLE_IGMP_CALLBACK** is a call into the IGMP to notify IGMP that a routing protocol has finished taking or releasing ownership of an interface.

When this callback is invoked, IGMP should add all its group memberships on the specified interface using calls to **MgmAddGroupMembershipEntry**.

```
typedef DWORD(*PMGM_ENABLE_IGMP_CALLBACK)(
  DWORD dwIfIndex,
  DWORD dwIfNextHopAddr
);
```

Parameters

dwIfIndex

[in] Specifies the index of the interface on which to enable IGMP.

dwIfNextHopAddr

[in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as non-broadcast multiple access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, MGM sets *dwInIfNextHopAddr* to zero.

Remarks

IGMP must not add group memberships in the context of this callback. The multicast group manager and IGMP will become deadlocked.

Requirements

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

See Also

PMGM_DISABLE_IGMP_CALLBACK

PMGM_JOIN_ALERT_CALLBACK

The **PMGM_JOIN_ALERT_CALLBACK** is a call into a routing protocol to notify the protocol that receivers are present for one or more groups on interfaces that are owned by other routing protocols.

```
typedef DWORD(*PMGM_JOIN_ALERT_CALLBACK)(
  DWORD dwSourceAddr,
  DWORD dwSourceMask,
  DWORD dwGroupAddr,
  DWORD dwGroupMask,
  BOOL bMemberUpdate
);
```

Parameters

dwSourceAddr

[in] Specifies the range of source addresses from which to receive group data. The multicast group manager sets *dwSourceAddr* to zero to indicate a wildcard receiver for a group); otherwise, the multicast group manager specifies the IP address of the source or source network.

dwSourceMask

[in] Specifies the subnet mask that corresponds to *dwSourceAddr*. The *dwSourceAddr* and *dwSourceMask* parameters are used together to define a range of sources from which to receive data. This parameter is not currently used.

dwGroupAddr

[in] Specifies the range of multicast groups for which to receive data. The multicast group manager sets *dwGroupAddr* to zero to receive all groups (a wildcard receiver); otherwise, the multicast group manager specifies the IP address of the group.

dwGroupMask

[in] Specifies the subnet mask that corresponds to dwGroupAddr. The *dwGroupAddr* and *dwGroupMask* parameters are used together to define a range of multicast groups. The multicast group manager sets *dwGroupMask* to zero if *dwGroupAddr* is set to zero (a wildcard receiver).

bMemberUpdate

[in] Specifies whether the callback was invoked because **MgmAddGroupMembershipEntry** was called by a client (the multicast group manager sets this parameter to TRUE), or because an MFE was created or updated (the multicast group manager sets this parameter to FALSE).

Remarks

The multicast group manager sets the *bMemberDelete* parameter to TRUE if both of the following conditions are met:

- A client calls **MgmAddGroupMembershipEntry** for a (*, g) entry.
- The interoperability rules between multicast routing protocols specify that other clients must be informed.

The multicast group manager sets the *bMemberDelete* parameter to FALSE if an **MgmAddGroupMembershipEntry** call causes the MFE for group “g” to contain an entry (that is, if the MFE is caused to leave the negative state). The action taken by the routing protocol when this callback is received is protocol-specific. The protocol may ignore the callback if the *bMemberDelete* parameter is set to FALSE, if the protocol specification indicates that this is the correct behavior.

This version of the Multicast Group Manager API supports only wildcard sources (*, g) or specific sources (s, g), not source ranges. The same restriction applies to groups (that is, no group ranges are permitted).

When **MgmAddGroupMembershipEntry** is called, the multicast group manager uses this callback to notify other multicast group manager clients that there are receivers for the specified source and group.

The multicast group manager uses the following rules to determine when to invoke this callback for wildcard (*, g) joins:

- If this is the first client to inform MGM that there are receivers on an interface for a group, the callback is invoked for all other clients registered with MGM.
- If this is the second client to inform MGM that there are receivers on an interface for a group, MGM invokes this callback for the first client that called **MgmAddGroupMembershipEntry**.

The multicast group manager uses the following rule to determine when to invoke this callback for source-specific (s, g) joins:

- If this is the first client to inform MGM that there are receivers on an interface for a source and group, MGM invokes this callback only for the client that owns the incoming interface towards the specified source.

Requirements

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

PMGM_LOCAL_JOIN_CALLBACK

The **PMGM_LOCAL_JOIN_CALLBACK** is a call into a routing protocol to notify the protocol that IGMP has detected receivers for a group on an interface that is currently owned by the routing protocol.

This callback is invoked when **MgmAddGroupMembershipEntry** is called by IGMP.

```
typedef DWORD(*PMGM_LOCAL_JOIN_CALLBACK)(
  DWORD dwSourceAddr,
  DWORD dwSourceMask,
  DWORD dwGroupAddr,
  DWORD dwGroupMask,
  DWORD dwIfIndex,
  DWORD dwIfNextHopAddr
);
```

Parameters

dwSourceAddr

[in] Specifies the range of source addresses from which to receive group data. The multicast group manager sets *dwSourceAddr* to zero to indicate a wildcard receiver for a group; otherwise, the multicast group manager specifies the IP address of the source or source network.

dwSourceMask

[in] Specifies the subnet mask that corresponds to *dwSourceAddr*. The *dwSourceAddr* and *dwSourceMask* parameters are used together to define a range of sources. This parameter is set to zero if *dwSource Addr* was also set to zero. This parameter is currently unused.

dwGroupAddr

[in] Specifies the range of multicast groups for which to receive data. The multicast group manager sets *dwGroupAddr* to zero to receive all groups (a wildcard receiver); otherwise, the multicast group manager specifies the IP address of the group.

dwGroupMask

[in] Specifies the subnet mask that corresponds to *dwGroupAddr*. The *dwGroupAddr* and *dwGroupMask* parameters are used together to define a range of multicast groups. This parameter is set to zero if *dwGroupAddr* was also set to zero. This parameter is currently unused.

dwIfIndex

[in] Specifies the interface on which to add the group membership.

dwIfNextHopAddr

[in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as Non-Broadcast Multiple Access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, the multicast group manager sets *dwInIfNextHopAddr* to zero.

Remarks

This version of the Multicast Group Manager API supports only wildcard sources (*, g) or specific sources (s, g), not source ranges. The same restriction applies to groups (that is, no group ranges are permitted).

! Requirements

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

See Also

PMGM_LOCAL_LEAVE_CALLBACK

PMGM_LOCAL_LEAVE_CALLBACK

The **PMGM_LOCAL_LEAVE_CALLBACK** is a call into a routing protocol to notify the routing protocol that the IGMP has detected that it no longer has receivers for a group on an interface that is currently owned by the routing protocol.

This callback is invoked when **MgmDeleteGroupMembershipEntry** is called by IGMP.

```
typedef DWORD(*PMGM_LOCAL_LEAVE_CALLBACK)(
  DWORD dwSourceAddr,
  DWORD dwSourceMask,
  DWORD dwGroupAddr,
  DWORD dwGroupMask,
  DWORD dwIfIndex,
  DWORD dwIfNextHopAddr
);
```

Parameters

dwSourceAddr
: [in] Specifies the range of source addresses from which to stop receiving group data. The multicast group manager sets *dwSourceAddr* to zero to indicate a wildcard receiver for a group; otherwise, the multicast group manager specifies the IP address of the source or source network.

dwSourceMask
: [in] Specifies the subnet mask that corresponds to *dwSourceAddr*. The *dwSourceAddr* and *dwSourceMask* parameters are used together to define a range of sources. This parameter is set to zero if *dwSource Addr* was also set to zero. This parameter is currently not used.

dwGroupAddr
: [in] Specifies the range of multicast groups for which to stop receiving data. The multicast group manager sets *dwGroupAddr* to zero to stop receiving all groups (a wildcard receiver); otherwise, the multicast group manager specifies the IP address of the group.

dwGroupMask
: [in] Specifies the subnet mask that corresponds to *dwGroupAddr*. The *dwGroupAddr* and *dwGroupMask* parameters are used together to define a range of multicast groups. This parameter is set to zero if *dwGroupAddr* was also set to zero. This parameter is currently not used.

dwIfIndex
: [in] Specifies the interface on which to remove the group membership.

dwIfNextHopAddr

[in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as Non-Broadcast Multiple Access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, the multicast group manager sets *dwInIfNextHopAddr* to zero.

Remarks

This version of the Multicast Group Manager API supports only wildcard sources (*, g) or specific sources (s, g), not source ranges. The same restriction applies to groups (that is, no group ranges are permitted).

! Requirements

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

+ See Also

PMGM_LOCAL_JOIN_CALLBACK

PMGM_PRUNE_ALERT_CALLBACK

The **PMGM_PRUNE_ALERT_CALLBACK** is a call into a routing protocol to notify the protocol that receivers are no longer present on interfaces owned by other routing protocols.

```
typedef DWORD(*PMGM_PRUNE_ALERT_CALLBACK)(
  DWORD dwSourceAddr,
  DWORD dwSourceMask,
  DWORD dwGroupAddr,
  DWORD dwGroupMask,
  DWORD dwIfIndex,
  DWORD dwIfNextHopAddr,
  BOOL bMemberDelete,
  PDWORD pdwTimeout
);
```

Parameters

dwSourceAddr

[in] Specifies the range of source addresses from which to stop receiving group data. The multicast group manager sets *dwSourceAddr* to zero to indicate a wildcard receiver for a group; otherwise, the multicast group manager specifies the IP address of the source or source network.

dwSourceMask

[in] Specifies the subnet mask that corresponds to *dwSourceAddr*. The *dwSourceAddr* and *dwSourceMask* parameters are used together to define a range of sources from which to stop receiving data. This parameter is set to zero if *dwSourceAddr* was also set to zero.

dwGroupAddr

[in] Specifies the range of multicast groups for which to stop receiving data. The multicast group manager sets *dwGroupAddr* to zero to stop receiving all groups (a wildcard receiver); otherwise, the multicast group manager specifies the IP address of the group.

dwGroupMask

[in] Specifies the subnet mask that corresponds to *dwGroupAddr*. The *dwGroupAddr* and *dwGroupMask* parameters are used together to define a range of multicast groups. This parameter is set to zero if *dwGroupAddr* was also set to zero.

dwIfIndex

[in] Specifies the interface on which to remove the group membership.

dwIfNextHopAddr

[in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as Non-Broadcast Multiple Access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, the multicast group manager sets *dwInIfNextHopAddr* to zero.

bMemberDelete

[in] Specifies whether the callback was invoked because **MgmDeleteGroupMembershipEntry** was called by a client (the multicast group manager sets this parameter to TRUE), or because an MFE was updated (the multicast group manager sets this parameter to FALSE).

pdwTimeout

[out] Pointer to a **DWORD** value that on return contains the time-out value, in seconds, for this MFE. If *bMemberDelete* is FALSE, this parameter can be used to specify how long the corresponding MFE should remain in the multicast forwarding cache. If the client does not specify a value, the default is 900 seconds.

Remarks

The multicast group manager sets the *bMemberDelete* parameter to TRUE if both of the following conditions are met:

- A client calls **MgmDeleteGroupMembershipEntry** for a (*, g) entry.
- The interoperability rules between multicast routing protocols specify that other clients must be informed.

The multicast group manager sets the *bMemberDelete* parameter to FALSE if a **MgmDeleteGroupMembershipEntry** call causes the MFE for group "g" to become negative. The action taken by the routing protocol when this callback is received is protocol-specific. The protocol may ignore the callback if the *bMemberDelete* parameter is set to FALSE, if the protocol specification indicates that this is the correct behavior.

This version of the Multicast Group Manager API supports only wildcard sources (*, g) or specific sources (s, g), not source ranges. The same restriction applies to groups (that is, no group ranges are permitted).

When **MgmDeleteGroupMembershipEntry** is called, the multicast group manager uses **PMGM_PRUNE_ALERT_CALLBACK** to notify other multicast group manager clients that receivers no longer exist for the specified source and group.

The multicast group manager uses the following rules to determine when to invoke this callback for wildcard (*, g) prunes:

- If this is the final interface being removed for the second-to-last client (that is, there are only interfaces remaining for a single client), the multicast group manager invokes this callback for the last remaining client.
- If this is the final interface being removed for the last client for the group (that is, if no other interfaces remain), then the callback is invoked for all other clients registered with the multicast group manager.

The multicast group manager uses the following rule to determine when to invoke this callback for source-specific (s, g) prunes:

- If this is the final interface being removed for the last client, this callback is invoked only for the client that owns the incoming interface towards the specified source.

Requirements

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

See Also

PMGM_CREATION_ALERT_CALLBACK

PMGM_RPF_CALLBACK

The **PMGM_RPF_CALLBACK** is a call into a routing protocol to determine if a given packet was received on the correct interface.

This callback is invoked when a packet from a new source or destined for a new group is received. The multicast group manager calls into the routing protocol that owns the incoming interface towards the source.

```
typedef DWORD(*PMGM_RPF_CALLBACK)(
  DWORD dwSourceAddr,
  DWORD dwSourceMask,
  DWORD dwGroupAddr,
  DWORD dwGroupMask,
  PDWORD pdwInIfIndex,
  PDWORD pdwInIfNextHopAddr,
  PDWORD pdwUpStreamNbr,
  DWORD dwHdrSize,
  PBYTE pbPacketHdr,
  PBYTE pbRoute
);
```

Parameters

dwSourceAddr
: [in] Specifies the address of the source that originated the multicast packet.

dwSourceMask
: [in] Specifies the subnet mask that corresponds to *dwSourceAddr*.

dwGroupAddr
: [in] Specifies the multicast group to which the data is to be delivered.

dwGroupMask
: [in] Specifies the subnet mask that corresponds to *dwGroupAddr*.

pdwInIfIndex
: [in, out] On input, a pointer to a **DWORD** value that specifies the index of the interface on which data from the source is expected to be received, based on the multicast view of the routing table. On output, *pdwInIfIndex* points to a **DWORD** value which contains the index of the interface on which the protocol expects to receive packets. The interface index may differ on output from the index specified on input.

pdwInIfNextHopAddr
: [in, out] On input, *pdwInIfNextHopAddr* points to the address of the next hop that corresponds to the interface in *pdwIfIndex*. The *pdwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as Non-Broadcast Multiple Access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, the multicast group manager sets *dwInIfNextHopAddr* to zero.

On output, *pdwInIfNextHopAddr* points to the next hop that corresponds to *pdwInIfIndex*.

pdwUpStreamNbr

[in, out] On input, *pdwUpStreamNbr* points to a **DWORD** value specifying the immediate upstream neighbor towards the source (the source is found in the multicast view of the routing table). On output, *pdwUpStreamNbr* may have been modified by the protocol. This parameter is informational.

dwHdrSize

[in] Specifies, in bytes, the size of the buffer pointed to by *pbPacketHdr*.

pbPacketHdr

[in] Pointer to a buffer that contains the IP header of the packet, including the IP options and a fragment of the data. This parameter is supplied for those protocols that examine the contents of the packet header.

pbRoute

[in] Pointer to a buffer that contains the route towards the source. The buffer contains an **RTM_DEST_INFO** structure.

Remarks

This callback is invoked when an MFE is created. MFEs are created when data from a new multicast source, or destined to a new group, is received.

The multicast group manager invokes this callback for the routing protocol that owns the incoming interface towards the source. The multicast group manager determines the interface by looking up the source of the multicast data in multicast view of the routing table. This interface is not always the same as the interface on which the data was actually received; this condition occurs if multicast data was received on the wrong interface.

When this callback is invoked, the routing protocol can change the incoming interface if the routing protocol behavior requires it to receive the data for the group from another interface.

! Requirements

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

PMGM_WRONG_IF_CALLBACK

The **PMGM_WRONG_IF_CALLBACK** is a call into a routing protocol to notify the protocol that a packet has been received from the specified source and for the specified group on the wrong interface.

```
typedef DWORD(*PMGM_WRONG_IF_CALLBACK)(
  DWORD dwSourceAddr,
  DWORD dwGroupAddr,
  DWORD dwIfIndex,
  DWORD dwIfNextHopAddr,
  DWORD dwHdrSize,
  PBYTE pbPacketHdr
);
```

Parameters

dwSourceAddr
[in] Specifies the source address from which group data was received.

dwGroupAddr
[in] Specifies the multicast group for which the data is destined.

dwIfIndex
[in] Specifies the interface on which the packet arrived.

dwIfNextHopAddr
[in] Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as Non-Broadcast Multiple Access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, the multicast group manager sets *dwInIfNextHopAddr* to zero.

dwHdrSize
[in] Specifies the size, in bytes, of the buffer pointed to by *pbPacketHdr*.

pbPacketHdr
[in] Pointer to a buffer that contains the IP header of the packet, including the IP options and a fragment of the data. This parameter is supplied for those protocols that examine the contents of the packet header.

Remarks

This callback is not currently available.

! Requirements

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

Multicast Group Manager Structures

The Multicast Group Manager functions use the following structures:

MGM_IF_ENTRY
ROUTING_PROTOCOL_CONFIG
SOURCE_GROUP_ENTRY

MGM_IF_ENTRY

The **MGM_IF_ENTRY** structure describes a router interface. This structure is used in the **PMGM_CREATION_ALERT_CALLBACK**. In the context of this callback, the routing protocol must enable or disable multicast forwarding on each interface, notifying the multicast group manager by using the **bIsEnabled** member.

```
typedef struct _MGM_IF_ENTRY {
  DWORD             dwIfIndex;
  DWORD             dwIfNextHopAddr;
  BOOL              bIGMP;
  BOOL              bIsEnabled;
} MGM_IF_ENTRY, *PMGM_IF_ENTRY;
```

Members

dwIfIndex
Specifies the index of the interface.

dwIfNextHopAddr
Specifies the address of the next hop that corresponds to *dwIfIndex*. The *dwIfIndex* and *dwIfNextHopIPAddr* parameters uniquely identify a next hop on point-to-multipoint interfaces, where one interface connects to multiple networks (such as Non-Broadcast Multiple Access (NBMA) interfaces, or the internal interface all dial-up clients connect on).

For broadcast interfaces (such as Ethernet interfaces) or point-to-point interfaces, which are identified by only *dwIfIndex*, specify zero.

bIGMP
Indicates whether or not IGMP is enabled on this interface. If TRUE, IGMP is enabled on this interface. If FALSE, IGMP is not enabled on this interface.

bIsEnabled

Indicates whether or not multicast forwarding is enabled on this interface. If TRUE, multicast forwarding is enabled on this interface. If FALSE, multicast forwarding is disabled on this interface.

Requirements

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

See Also

PMGM_CREATION_ALERT_CALLBACK

ROUTING_PROTOCOL_CONFIG

The **ROUTING_PROTOCOL_CONFIG** structure describes the routing protocol configuration information that is passed to the multicast group manager when a protocol is registered.

```
typedef struct _ROUTING_PROTOCOL_CONFIG {
  DWORD                            dwCallbackFlags;
  PMGM_RPF_CALLBACK                pfnRpfCallback;
  PMGM_CREATION_ALERT_CALLBACK     pfnCreationAlertCallback;
  PMGM_PRUNE_ALERT_CALLBACK        pfnPruneAlertCallback;
  PMGM_JOIN_ALERT_CALLBACK         pfnJoinAlertCallback;
  PMGM_WRONG_IF_CALLBACK           pfnWrongIfCallback;
  PMGM_LOCAL_JOIN_CALLBACK         pfnLocalJoinCallback;
  PMGM_LOCAL_LEAVE_CALLBACK        pfnLocalLeaveCallback;
  PMGM_DISABLE_IGMP_CALLBACK       pfnDisableIgmpCallback;
  PMGM_ENABLE_IGMP_CALLBACK        pfnEnableIgmpCallback;
} ROUTING_PROTOCOL_CONFIG, *PROUTING_PROTOCOL_CONFIG;
```

Members

dwCallbackFlags

Reserved for future use.

pfnRpfCallback

Callback into a routing protocol to perform an RPF check.

pfnCreationAlertCallback

Callback into a routing protocol to determine the subset of interfaces owned by the routing protocol on which a multicast packet from a new source or to a new group should be forwarded.

pfnPruneAlertCallback
Callback into a routing protocol to notify the protocol that receivers are no longer present on an interface owned by other routing protocols for the specified source and group.

pfnJoinAlertCallback
Callback into a routing protocol to notify the protocol that receivers are present on an interface owned by another routing protocol for the specified source and group.

pfnWrongIfCallback
Callback into a routing protocol to notify the protocol that a packet has been received from the specified source and for the specified group on the wrong interface.

pfnLocalJoinCallback
Callback into a routing protocol to notify the protocol that IGMP has detected receivers for a group on an interface.

pfnLocalLeaveCallback
Callback into a routing protocol to notify the protocol that IGMP has detected that there are no receivers for a group on an interface.

pfnDisableIgmpCallback
Callback into IGMP to notify IGMP that a protocol is taking or releasing ownership of an interface on which IGMP is enabled.

pfnEnableIgmpCallback
Callback into IGMP to notify IGMP that a protocol has finished taking or releasing ownership of an interface on which IGMP is enabled.

Requirements

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

See Also

MgmRegisterMProtocol

SOURCE_GROUP_ENTRY

The **SOURCE_GROUP_ENTRY** structure describes the entry returned by the multicast group manager group enumeration functions.

```
typedef struct _SOURCE_GROUP_ENTRY {
  DWORD              dwSourceAddr;
  DWORD              dwSourceMask;
  DWORD              dwGroupAddr;
  DWORD              dwGroupMask;
} SOURCE_GROUP_ENTRY, *PSOURCE_GROUP_ENTRY;
```

Members

dwSourceAddr
Specifies the range of source addresses for the membership entry. Zero indicates a wildcard source.

dwSourceMask
Specifies the subnet mask that corresponds to **dwSourceAddr**. The **dwSourceAddr** and **dwSourceMask** members are used together to define a range of sources.

dwGroupAddr
Specifies the range of multicast groups for the membership entry. Zero indicates a wildcard group.

dwGroupMask
Specifies the subnet mask that corresponds to *dwGroupAddr*. The **dwGroupAddr** and **dwGroupMask** members are used together to define a range of multicast groups.

Requirements

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

See Also

MGM_ENUM_TYPES, **MgmGroupEnumerationEnd**, **MgmGroupEnumerationGetNext**, **MgmGroupEnumerationStart**

Multicast Group Manager Enumerations

The Multicast Group Manager functions use the following enumeration:

MGM_ENUM_TYPES

MGM_ENUM_TYPES

The **MGM_ENUM_TYPES** enumeration lists the types of group enumerations that the multicast group manager uses. It is used by the **MgmGroupEnumerationStart** function.

```
typedef enum _MGM_ENUM_TYPES {
  ANY_SOURCE = 0,
  ALL_SOURCES
} MGM_ENUM_TYPES;
```

Values

ANY_SOURCE
Enumerate group entries that have at least one source.

ALL_SOURCES
Enumerate all source entries for a group.

! Requirements

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

+ See Also

MgmGroupEnumerationStart

I N D E X

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.

Routing

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.*

Routing

This volume provides reference materials about the Routing and Remote Access Service (RRAS), which is available as an add-on for Microsoft Windows NT® Server 4.0 and included in Microsoft Windows® 2000 Server. The RRAS API lets you create applications to administer routing and remote access services, to implement your own routing protocols, or even to turn a computer into a fully functioning network router. RRAS can run many of the most popular routing protocols and provides the capability to deploy economical, high-performance midrange routers on computers that run Windows NT 4.0 or Windows 2000 Server.

Part No. 097-0002787

Microsoft