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

RPC and Windows® Networking

BASED ON
msdn™ library

Microsoft®

David Iseminger
Series Editor

RPC and Windows® Networking

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

Copyright © 2000 by Microsoft Corporation; portions © 2000 by David Iseminger.

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

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

Printed and bound in the United States of America.

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

Distributed in Canada by Penguin Books Canada Limited.

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

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

Intel is a registered trademark of Intel Corporation. Active Directory, BackOffice, FrontPage, Microsoft, Microsoft Press, MSDN, MS-DOS, Visual Basic, Visual C++, Visual FoxPro, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, Win32, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person, or event is intended or should be inferred.

Acquisitions Editor: Ben Ryan
Project Editor: Wendy Zucker

Part No. 097-0002785

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

Acknowledgements iii

Part 1

Chapter 1: Getting Around in the Networking Services Library 1

- How the Networking Services Library Is Structured 2
- How the Networking Services Library Is Designed 3

Chapter 2: What's In This Volume? 5

- Microsoft RPC Model 5
- Installing The RPC Programming Environment 6
- Building RPC Applications 6
- Connecting the Client and the Server 6
- Tutorial 6
- IDL and ACF Files 6
- Data and Language Features 6
- Arrays and Pointers 6
- Pipes 6
- Binding and Handles 6
- Memory Management 6
- Serialization Services 6
- Security 7
- Installing and Configuring RPC Applications 7
- Asynchronous RPC 7
- RPC Message Queuing 7
- Remote Procedure Calls Using HTTP 7
- Samples 7
- RPC Reference 7

Chapter 3: Using Microsoft Reference Resources 9

The Microsoft Developer Network 10

- Comparing MSDN with MSDN Online 11
- MSDN Subscriptions 13
  - MSDN Library Subscription 13
  - MSDN Professional Subscription 14
  - MSDN Universal Subscription 14

Purchasing an MSDN Subscription 14
Using MSDN 15
Navigating MSDN 16
Quick Tips 18
Using MSDN Online 20
Navigating MSDN Online 22
MSDN Online Features 23
MSDN Online Registered Users 29
The Windows Programming Reference Series 30

Chapter 4: Finding the Developer Resources You Need 31
Developer Support 31
Online Resources 33
Internet Standards 34
Learning Products 35
Conferences 37
Other Resources 37

Chapter 5: Avoiding Common RPC Programming Errors 39
Solution Summary 39
Common RPC Programming Errors 40
Pointer_default(unique) and embedded pointers 40
A valid switch_is value in an RPC-capable structure doesn't ensure a non-NULL pointer 41
A NULL DACL affords no protection 41
Call RpcImpersonateClient() before any security-relevant operation 42
Starting and stopping impersonation 43
Strings are only zero-terminated when declared with string in the .idl 44
Don't copy arbitrary length data into independently sized buffers 45
size_is may result in a zero-length structure 45
Calculations in a size_is or length_is specification are susceptible to overflow 46
Strict context handles 46

Part 2

Chapter 6: Microsoft RPC Model 47
The Programming Model 47
The Client-Server Model 49
The Compute-Server Model 51
How RPC Works 51

OSF Standards for RPC 53
Microsoft RPC Components 54
RPC Extends Client-Server Computing 55

Chapter 7: Installing the RPC Programming Environment 57
Developing 32-Bit Windows Applications 57
Developing Macintosh Client Applications 59

Chapter 8: Building RPC Applications 61
General Build Procedure 61
  Developing the Interface 62
    Generating Interface UUIDs 63
    Using MIDL 64
  Developing the Server 66
  Developing the Client 67
Environment, Compiler, and API Set Choices 68
Exception Handling 69

Chapter 9: Connecting the Client and the Server 71
Essential RPC Binding Terminology 71
How the Server Prepares for a Connection 72
  Registering the Interface 73
  Creating Binding Information 73
  Advertising the Server Program 74
  Registering Endpoints 75
  Listening for Client Calls 75
How the Client Establishes a Connection 76
  Making a Remote Procedure Call 78
  Finding the Server Host Computer 78
  Finding the Server Program 79
  Creating a Binding 79

Chapter 10: An RPC Tutorial 81
The Stand-Alone Application 81
Defining the Interface 82
Generating the UUID 82
The IDL File 83
The ACF File 83
Generating the Stub Files 84
The Client Application 86
The Server Application 88

Stopping the Server Application 90
Compiling and Linking 91
Running the Application 92

Chapter 11: The IDL and ACF Files 93
The Interface Definition Language (IDL) File 93
The IDL Interface Header 94
The IDL Interface Body 95
The Application Configuration File (ACF) 96
The ACF Header 96
The ACF Body 96
MIDL Compiler Output 97

Chapter 12: Data and Language Features 99
Strong Typing 99
Base Types 100
Signed and Unsigned Types 101
Wide-Character Types 101
Structures 101
Unions 101
Enumerated Types 103
Arrays 103
Function Attributes 103
Field Attributes 103
Three Pointer Types 104
Type Attributes 105
Directional (Parameter) Attributes 106
Data Representation 108
The transmit_as and represent_as Attributes 109
The transmit_as Attribute 109
The type_to_xmit Function 111
The type_from_xmit Function 112
The type_free_xmit Function 114
The type_free_inst Function 114
The represent_as Attribute 115
The named_type_from_local Function 118
The named_type_to_local Function 118
The named_type_free_local Function 118
The named_type_free_inst Function 118
The wire_marshal and user_marshal Attributes 119

The wire_marshal Attribute 119
The user_marshal Attribute 120
The type_UserSize Function 121
The type_UserMarshal Function 122
The type_UserUnmarshal Function 123
The type_UserFree Function 124
Marshaling Rules for user_marshal and wire_marshal 124

Chapter 13: Arrays and Pointers 127
Arrays and RPC 127
Kinds of Arrays 127
Fixed Arrays 127
Varying Arrays 128
Conformant Arrays 129
Array Attributes 130
MIDL Array Attributes Used in RPC 132
The [size_is] Attribute 132
The [length_is] Attribute 133
The [first_is] and [last_is] Attributes 133
The [max_is] Attribute 134
Combining Array Attributes 134
The [string] Attribute in Arrays 135
Multidimensional Arrays 135
Pointers and RPC 137
Kinds of Pointers 137
Reference Pointers 137
Unique Pointers 139
Full Pointers 140
Pointers and Memory Allocation 141
Default Pointer Types 142
Pointer-Attribute Type Inheritance 143
Using Arrays, Strings, and Pointers 144
Counted Character Arrays 145
[in, out, size_is] Prototype 145
[in, size_is and out, size_is] Prototype 146
Strings 147
[in, out, string] Prototype 148
[in, string] and [out, string] Prototype 149
Multiple Levels of Pointers 149

Chapter 14: Pipes **151**
Essential Pipe Terminology 151
The Pipe State 152
Defining Pipes in IDL Files 152
Client-Side Pipe Implementation 153
  Implementing Input Pipes on the Client 154
  Implementing Output Pipes on the Client 157
Server-Side Pipe Implementation 159
  Implementing Input Pipes on the Server 159
  Implementing Output Pipes on the Server 160
Rules for Multiple Pipes 161
Combining Pipe and Nonpipe Parameters 162

Chapter 15: Binding and Handles **163**
Binding Handles 163
  Types of Binding Handles 163
    Automatic Binding Handles 164
    Implicit Binding Handles 165
    Explicit Binding Handles 167
    Primitive and Custom Binding Handles 167
  Client-Side Binding 170
    Selecting a Protocol Sequence 171
    Finding Server Host Systems 172
    Finding Endpoints 176
  Server-Side Binding 177
    Registering Interfaces 177
    Specifying Protocol Sequences 186
    Specifying Endpoints 187
    Advertising Server Interfaces 189
    Listening for Remote Procedure Calls 190
  Fully and Partially Bound Handles 191
  Interpreting Binding Information 191
  Microsoft RPC Binding-Handle Extensions 193
  Binding-Handle Functions 194
  The RPC Name-Service Database 195
    Name-Service Application Guidelines 196
    An Overview of the Name Service Entry 197
    Criteria for Name Service Entries 197
    Name Service Entry Cleanup 198

What Happens During a Query 198
Using Microsoft Locator 199
Using the Cell Directory Service (CDS) 200
Name Syntax 201
Context Handles 201
Interface Development Using Context Handles 202
Server Development Using Context Handles 203
Client Development Using Context Handles 205
Server Context Rundown Routine 207
Client Context Reset 208
Multi-Threaded Clients and Context Handles 208

Chapter 16: Memory Management 209
Introduction to RPC Memory Management 209
How Memory Is Allocated and Deallocated 210
The midl_user_allocate Function 210
The midl_user_free Function 211
RpcSs Memory Management Package 212
Memory-Management Models 213
Node-by-Node Allocation and Deallocation 213
Stub-Allocated Buffers 214
Application-Allocated Buffer 214
Persistent Storage on the Server 215
Who Manages Memory? 216
Top-Level and Embedded Pointers 216
Directional Attributes Applied to the Parameter 216
Length, Size, and Directional Attributes 217
Pointer Attributes Applied to the Parameter 219
Combining Pointer and Directional Attributes 219
Embedded Out-Only Reference Pointers 219
Out-Only Unique or Full Pointer Parameters Not Accepted 220
Function Return Values 220
Memory Orphaning 221
Summary of Memory Allocation Rules 221

Chapter 17: Serialization Services 223
Using Serialization Services 224
Procedure Serialization 225
Type Serialization 225
Serialization Handles 226

Implicit Versus Explicit Handles 227
Serialization Styles 227
Fixed Buffer Serialization 228
Dynamic Buffer Serialization 230
Incremental Serialization 230
Obtaining an Encoding Identity 233

Chapter 18: Security 235
RPC Security Essentials 235
Principal Names 235
Authentication Levels 236
Authentication Services 237
Client Authentication Credentials 237
Authorization Services 237
Quality of Service 237
Authorization Functions 238
Key Acquisition Functions 239
Client Impersonation 239
Security Methods 240
Security Support Provider Interface (SSPI) 240
SSPI Architectural Overview 240
Security Support Providers (SSPs) 241
Writing an Authenticated SSPI Client 242
Writing an Authenticated SSPI Server 245
Windows NT and Windows 2000 Transport Security 246
Using Transport-Level Security on the Server 247
Using Transport-Level Security on the Client 247

Chapter 19: Installing and Configuring RPC Applications 249
Configuring the Name Service Provider 249
Configuring the Name Service for Windows 95 250
Editing the Windows 95 Registry 250
Configuring the Name Service for Windows NT or Windows 2000 251
Configuring the Name Service for Windows 3.*x* or MS-DOS 252
Starting and Stopping Microsoft Locator 253
Registry Information 253
Using RPC Registry Entries 253
Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding 255
Using RPC with Winsock Proxy 257

SPX/IPX Installation 258
  Configuring RPC for SPX/IPX 258
  Configuring SAP and RPC 260
Configuring the Security Server 261

Chapter 20: Asynchronous RPC 263
Declaring Asynchronous Functions 264
Client-Side Asynchronous RPC 264
  Making the Asynchronous Call 265
  Waiting for the Asynchronous Reply 267
  Receiving the Asynchronous Reply 268
Server-Side Asynchronous RPC 269
  Handling Asynchronous Calls 269
  Receiving Cancellations 269
  Sending the Asynchronous Reply 269
  Asynchronous I/O and Asynchronous RPC 271
Causal Ordering of Asynchronous Calls 272
Error Handling 272
Asynchronous RPC Over the Named-Pipe Protocol 273
Using Asynchronous RPC with DCE Pipes 274
  Asynchronous Pipes 274
  Declaring Asynchronous Pipes 274
  Client-Side Asynchronous Pipe Handling 275
  Server-Side Asynchronous Pipe Handling 276
Asynchronous DCOM 280

Chapter 21: RPC Message Queuing 281
Overview of Message Queuing Services Architecture 281
Message and Message Queue Properties 283
Using MSMQ as an RPC Transport 283
System Requirements for RPC-MQ Applications 284
Developing RPC-MQ Applications 284
MSMQ Security Services 286

Chapter 22: Remote Procedure Calls Using HTTP 287
Using HTTP as an RPC Transport 287
HTTP RPC Security 290
System Requirements for HTTP RPC 291
Configuring Computers for HTTP RPC 291

Chapter 23: RPC Samples 293

Chapter 24: RPC Data Types, Structures, and Constants 295
RPC Structures 295
RPC Enumerated Types 315
Other RPC Types 316
RPC Constants 329
RPC Return Values 340

Chapter 25: RPC Function Reference 347
RPC Functions 347

Chapter 26: RPC Callback and Notification Functions 575

Chapter 27: RPC Macros 583

Chapter 28: Windows Networking (WNet) 591
About Windows Networking 591
- WNet Functions 592
- Windows Networking Operations 594

Using Windows Networking 595
- Using the Connections Dialog Box 595
- Enumerating Network Resources 595
- Adding a Network Connection 599
- Assigning a Drive to a Share 600
- Determining the Location of a Share 601
- Retrieving the Connection Name 603
- Retrieving the User Name 604
- Canceling a Network Connection 605
- Retrieving Network Errors 606

Windows Networking Reference 608
- Windows Networking Functions 608
- Obsolete Functions 608
- Windows Networking Structures 656

Part 3

Glossary 669

Index: Networking Services Programming Elements – Alphabetical Listing 677

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 3 of the *Networking Services Developer's Reference Library* provides in-depth information about the world of Remote Procedure Calls (RPC), as well as detailed information about Microsoft Windows Networking (WNet) programming.

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, chapters common to all WPRS volumes. By including this information in each library and 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.
- You can access this important and useful information regardless of which volume you have in your hand. You don't have to (nor *should* you have to) fumble with another physical book to access information about how to get the most out of MSDN or where to get support for questions you have about a particular Windows development problem you're having.
- Each volume becomes more useful, more portable, and more complete in and of itself. This goal of the WPRS makes it easier for you to grab one of its libraries' volumes and take it with you, rather than feeling like you must bring multiple volumes with you to have access to the library's important overview and usability information.

These goals have 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 is broken into two sections:

- RPC programmer's guide and reference information, in Chapters 6–27
- WNet information, all wrapped up in Chapter 28

The following provides information about what you will find in this volume's treatment of RPC:

Microsoft RPC Model

Provides an overview of the client-server programming model, standards for distributed application programming, and a description of how Microsoft RPC works.

Installing The RPC Programming Environment

Tells how to install the files and tools needed to develop distributed applications with Microsoft RPC.

Building RPC Applications

Describes the MIDL compiler and the necessary environment for building distributed applications with Microsoft RPC.

Connecting the Client and the Server

Provides an overview of the process of initializing and running distributed applications.

Tutorial

Provides an overview of the development of a small distributed application. This example demonstrates all the steps in developing a distributed application, the tools you use, and the components that make up the executable programs.

IDL and ACF Files

Describes the IDL and ACF files used to specify the interface to the remote procedure call and the MIDL compiler switches that control how these files are processed.

Data and Language Features

Demonstrates the use of standard data types.

Arrays and Pointers

Explains how to pass arrays pointers as parameters.

Pipes

Describes how to use named pipes as the transport mechanism for remote procedure calls.

Binding and Handles

Describes the binding handle—the data structure that allows the developer to bind the calling application to the remote procedure.

Memory Management

Offers ideas about how to manage memory on the client and server when performing remote procedure calls.

Serialization Services

Describes the methods for encoding or decoding data.

Security

Describes the methods for implementing security features in your distributed applications.

Installing and Configuring RPC Applications

Discusses installing your client and server applications in the MS-DOS, Microsoft Windows 3.*x*, Windows 95, and Windows NT/Windows 2000 environments. Describes how to configure the name service provider and the security service. This section also contains network transport information for RPC.

Asynchronous RPC

Presents information on the Microsoft asynchronous extensions to the RPC definition. Asynchronous remote procedure calls return immediately without waiting for output. When the remote procedure finishes executing on the server, it transfers return data to the client.

RPC Message Queuing

Describes the use of the Message Queuing Service (MSMQ), which lets users communicate across networks and systems regardless of the current state of the communicating applications and systems.

Remote Procedure Calls Using HTTP

Provides RPC clients with the ability to securely connect across the Internet to RPC server programs and execute remote procedure calls.

Samples

Contains a description of the example RPC programs shipped with the Microsoft Platform Software Developer's Kit.

RPC Reference

This collection of chapters provides a complete treatment of RPC programming reference elements.

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.

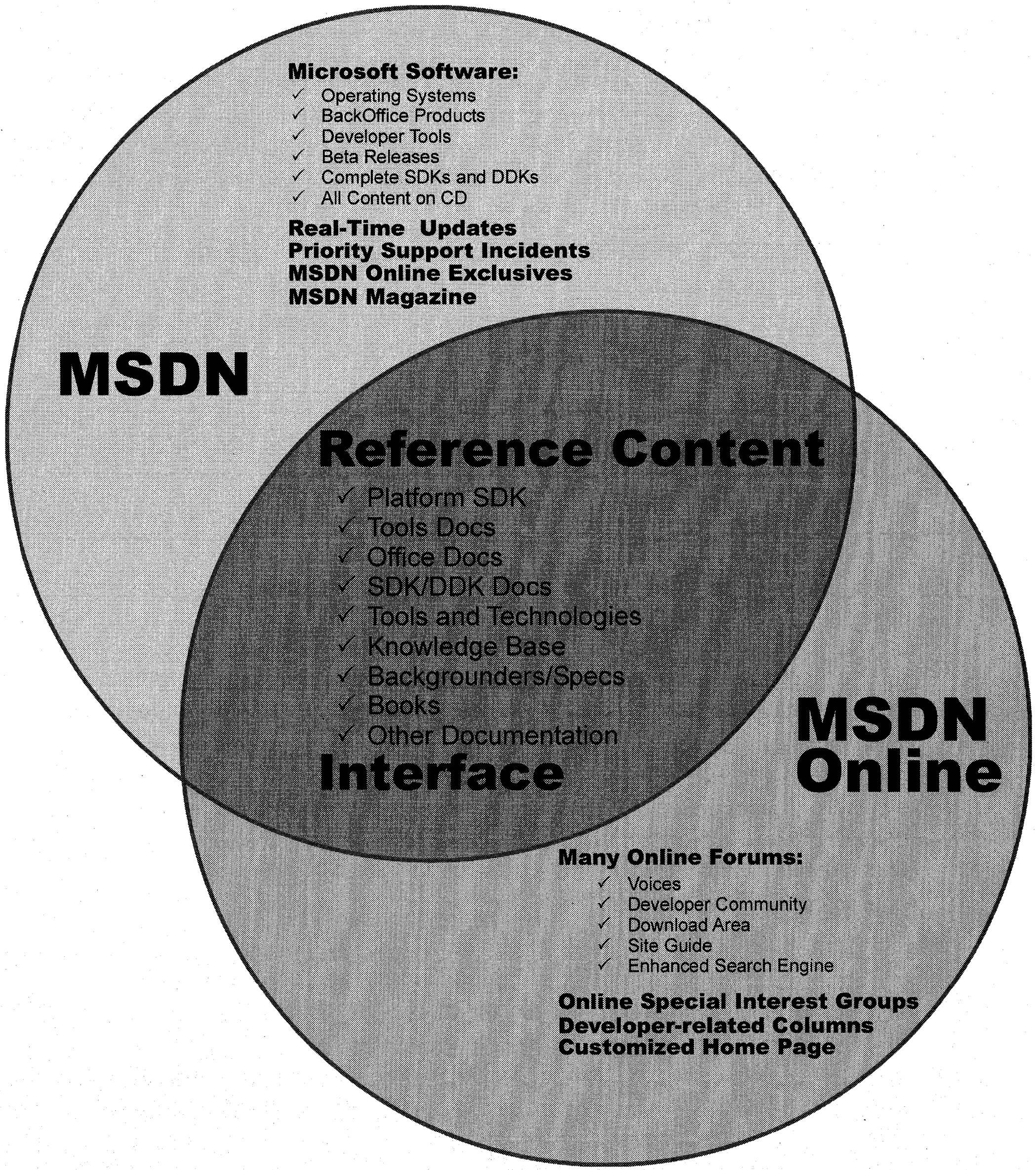


Figure 3-1: The similarities and differences in coverage between MSDN and MSDN Online.

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

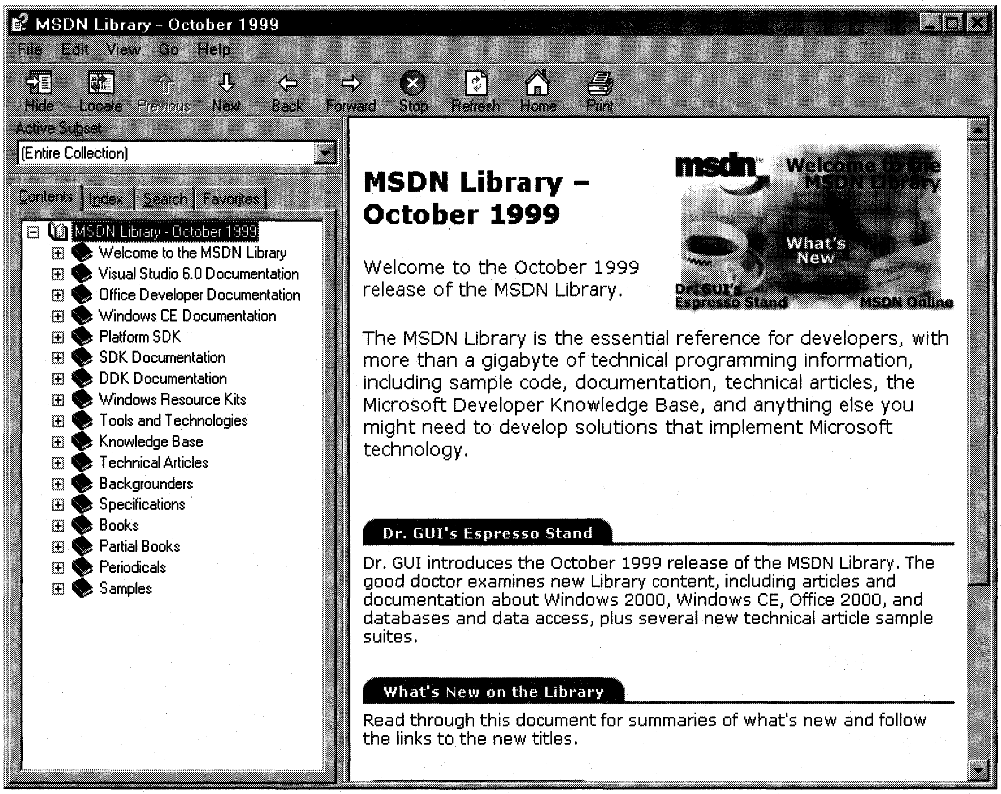
MSDN subscriptions come with an installable interface, and the Professional and Universal subscriptions also come with a bunch of Microsoft product software such as Windows platform versions and BackOffice applications. There's no need to tell you how to use Microsoft product software, but there's a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

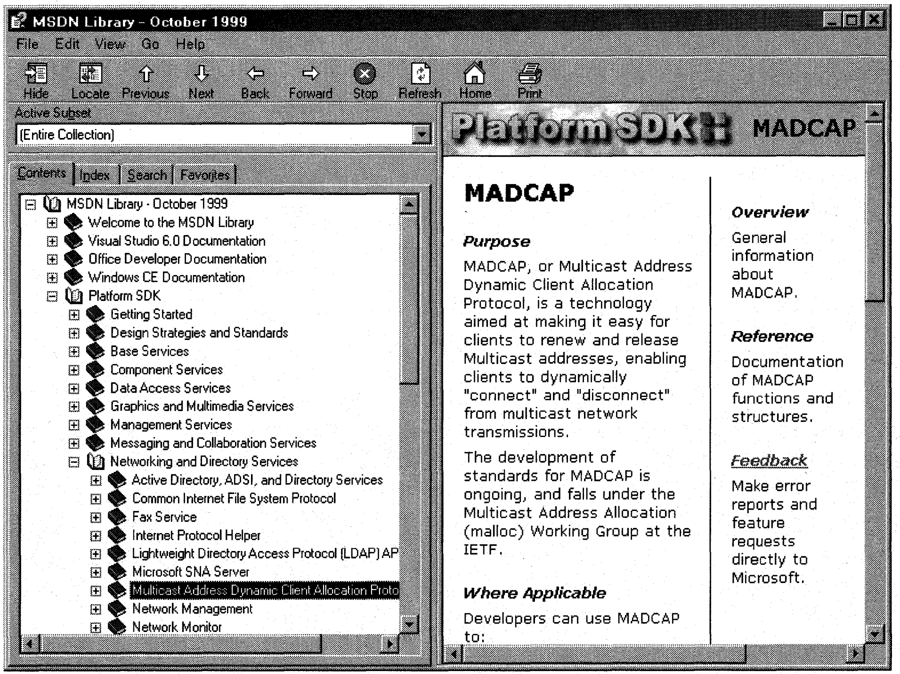
What's New on the Library

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.

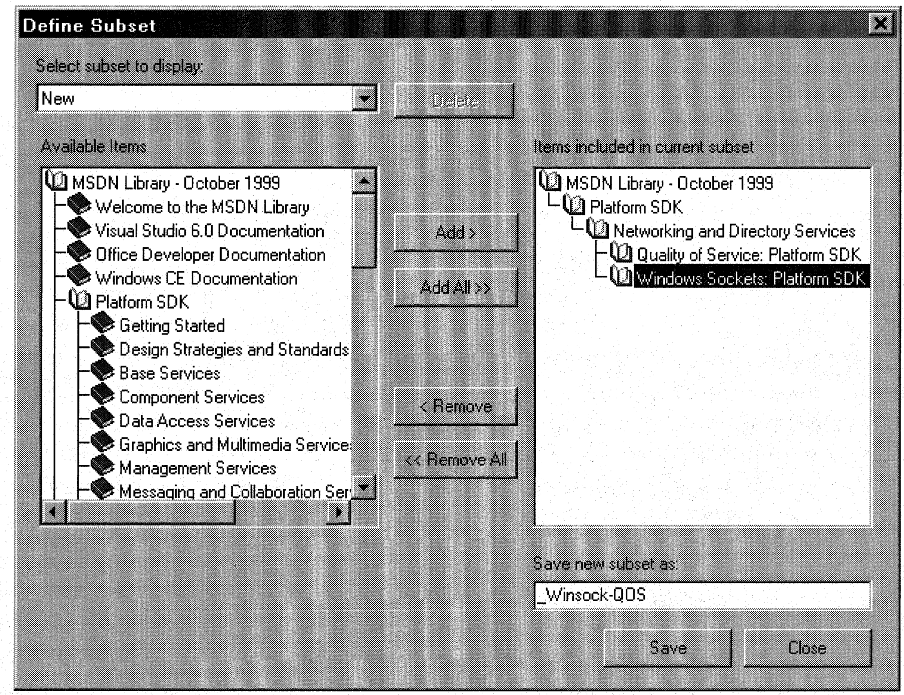

Define Subset
Select subset to display:
New
Delete
Available Items
MSDN Library - October 1999
Welcome to the MSDN Library
Visual Studio 6.0 Documentation
Office Developer Documentation
Windows CE Documentation
Platform SDK
Getting Started
Design Strategies and Standards
Base Services
Component Services
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.

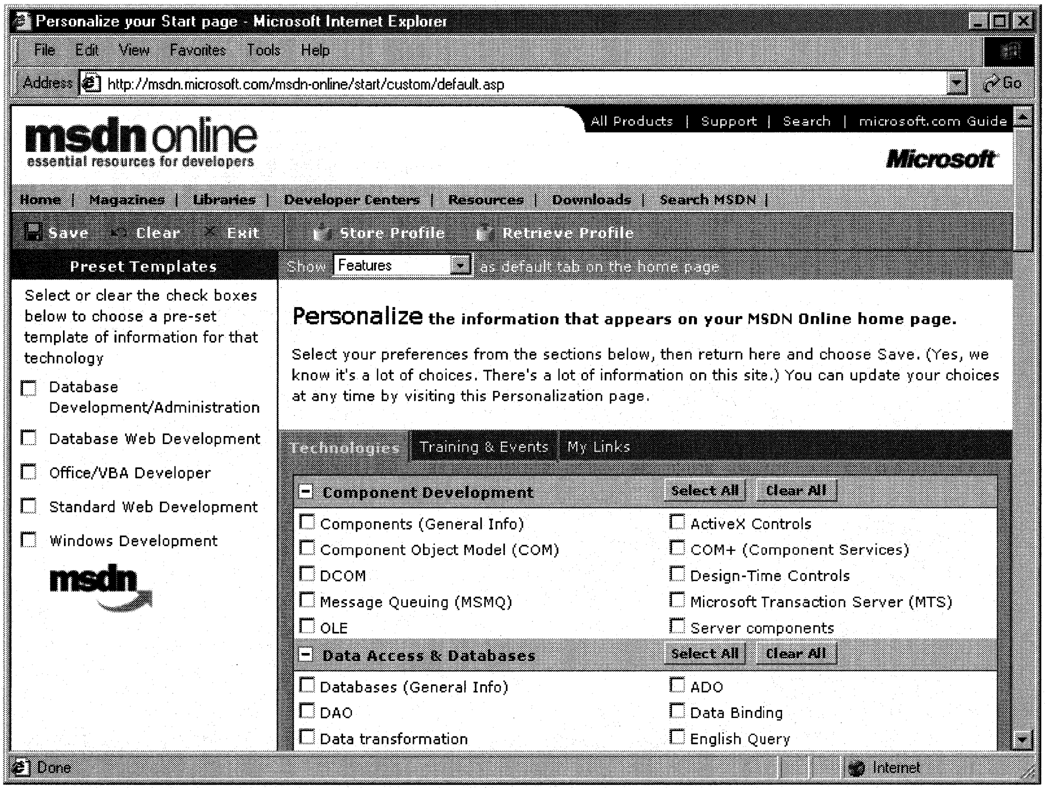
If you've used MSN's home page before (*www.msn.com*), you're familiar with the fact that you can customize the page to your liking; choose from an assortment of available national news, computer news, local news, local weather, stock quotes, and other collections of information or news that suit your 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 http://msdn.microsoft.com/msdn-online/start/custom/default.asp
Go
msdn online
essential resources for developers
All Products | Support | Search | microsoft.com Guide
Microsoft
Home | Magazines | Libraries | Developer Centers | Resources | Downloads | Search MSDN |
Save Clear Exit Store Profile Retrieve Profile
Preset Templates
Show Features as default tab on the home page
Technologies Training & Events My Links
Done
Internet

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 Select All Clear All

☐ 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 Select All Clear All

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

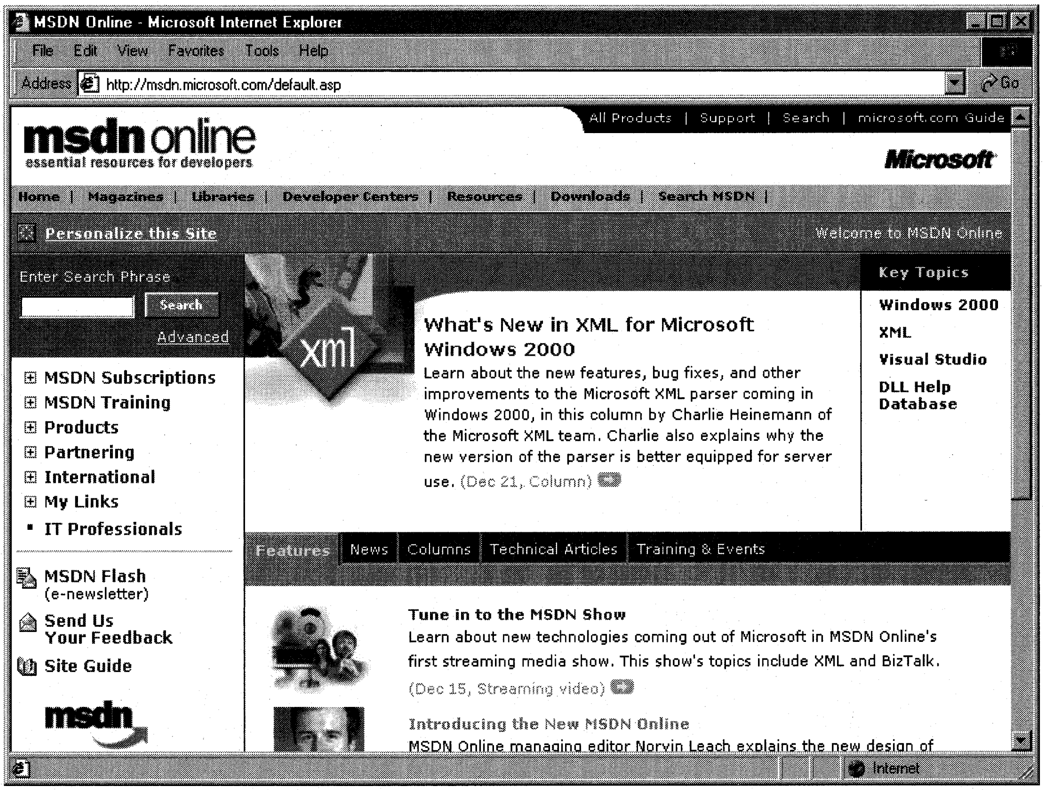
Figure 3-5: The MSDN Online Personalize Page.

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.



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	

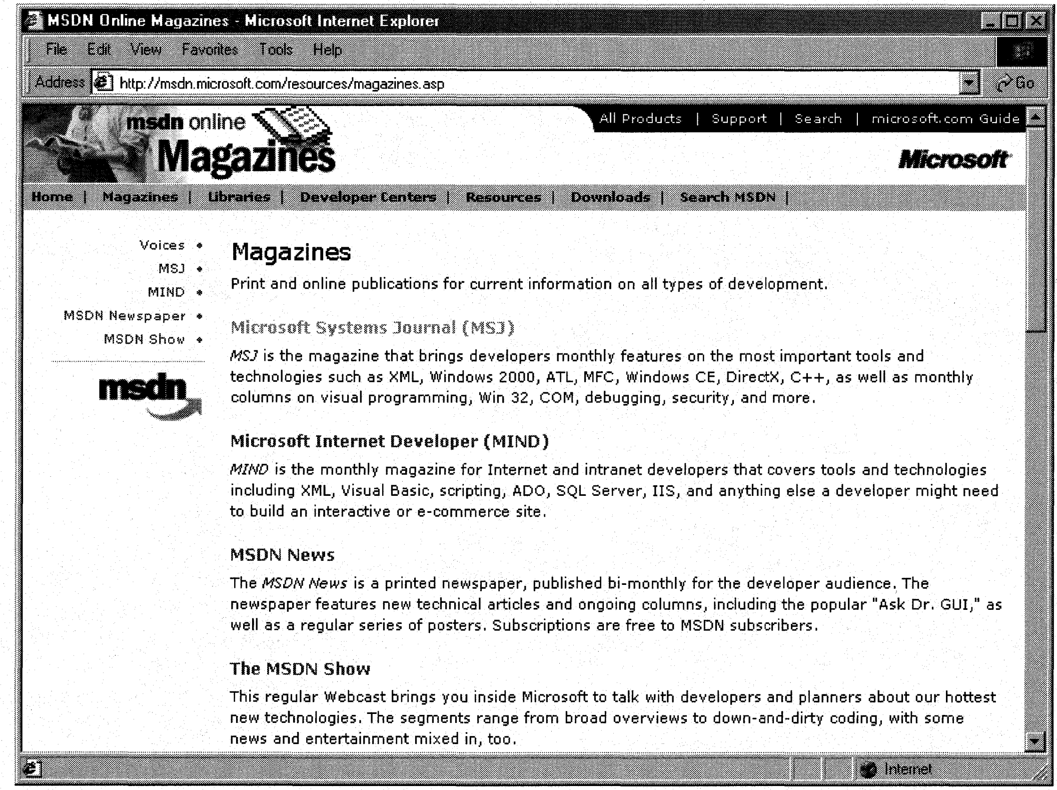
The navigation bar is available regardless of where you are in MSDN Online, so the capability to navigate the site from this familiar menu is always available, leaving you a click away from any area on MSDN Online. These menu categories create a functional and logical grouping of MSDN Online's feature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

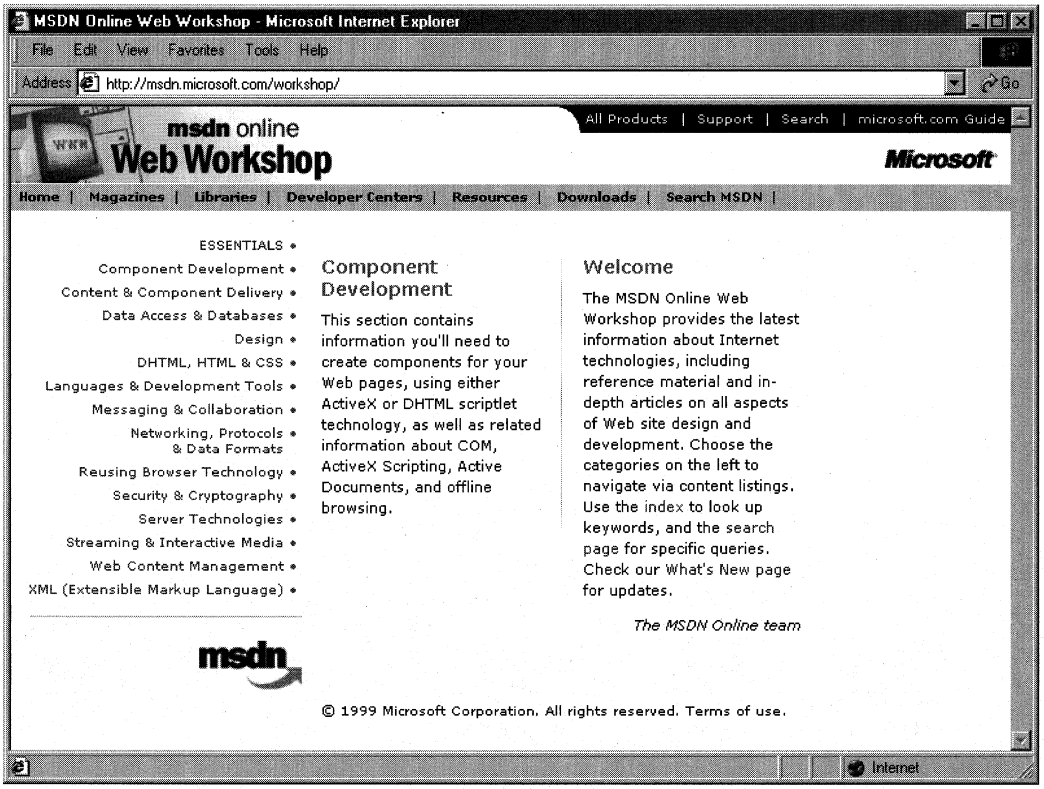
The MSDN Show

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/
Go
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)
msdn
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

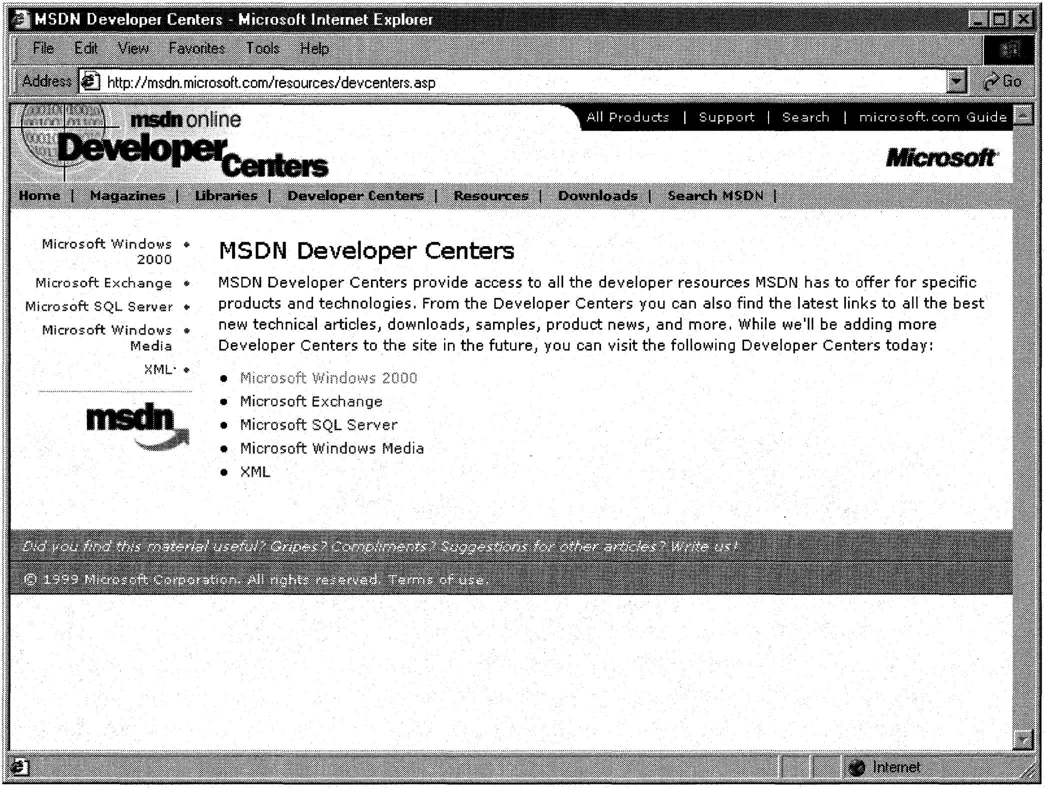
© 1999 Microsoft Corporation. All rights reserved. Terms of use.

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:

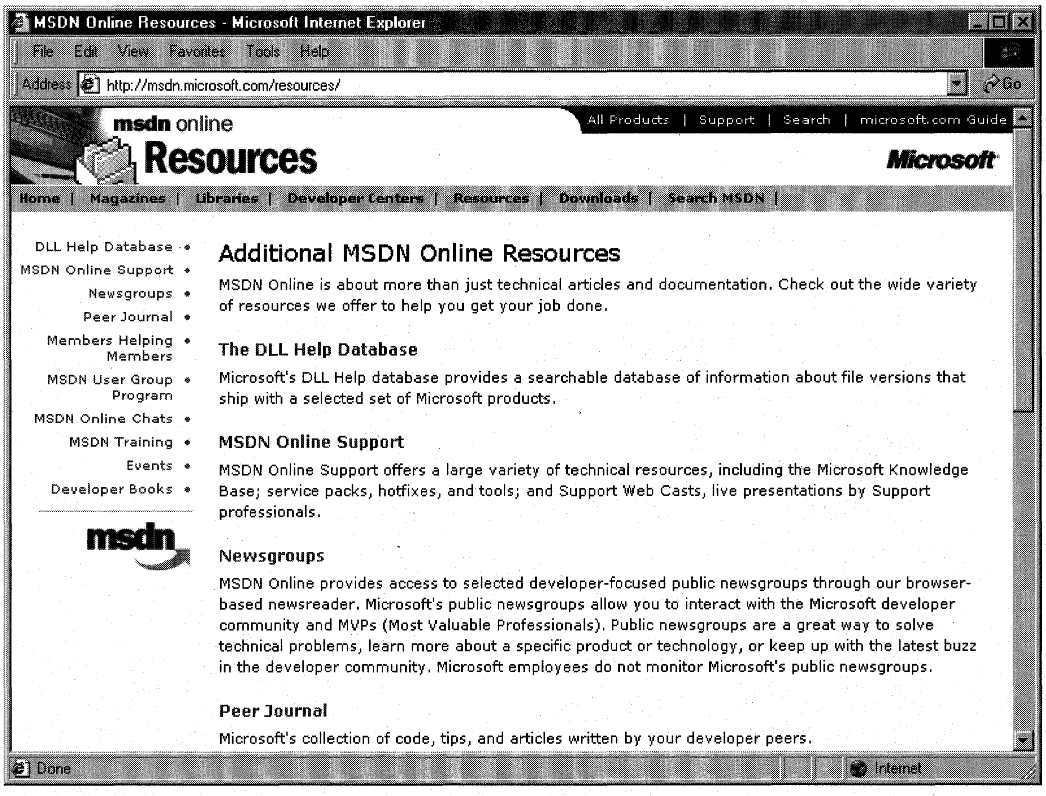
- Microsoft Windows 2000
- Microsoft Exchange
- Microsoft SQL Server
- Microsoft Windows Media
- XML

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
Resources
All Products | Support | Search | microsoft.com Guide
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

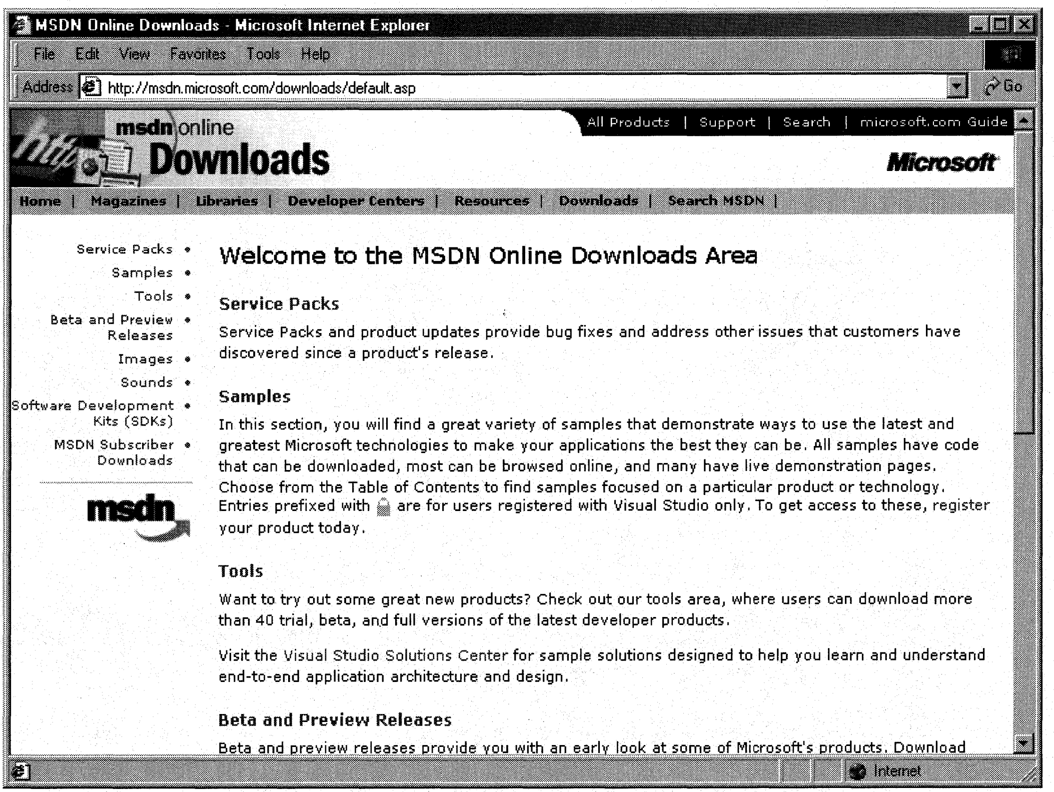
MSDN Online provides access to selected developer-focused public newsgroups through our browser-based newsreader. Microsoft's public newsgroups allow you to interact with the Microsoft developer community and MVPs (Most Valuable Professionals). Public newsgroups are a great way to solve technical problems, 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.

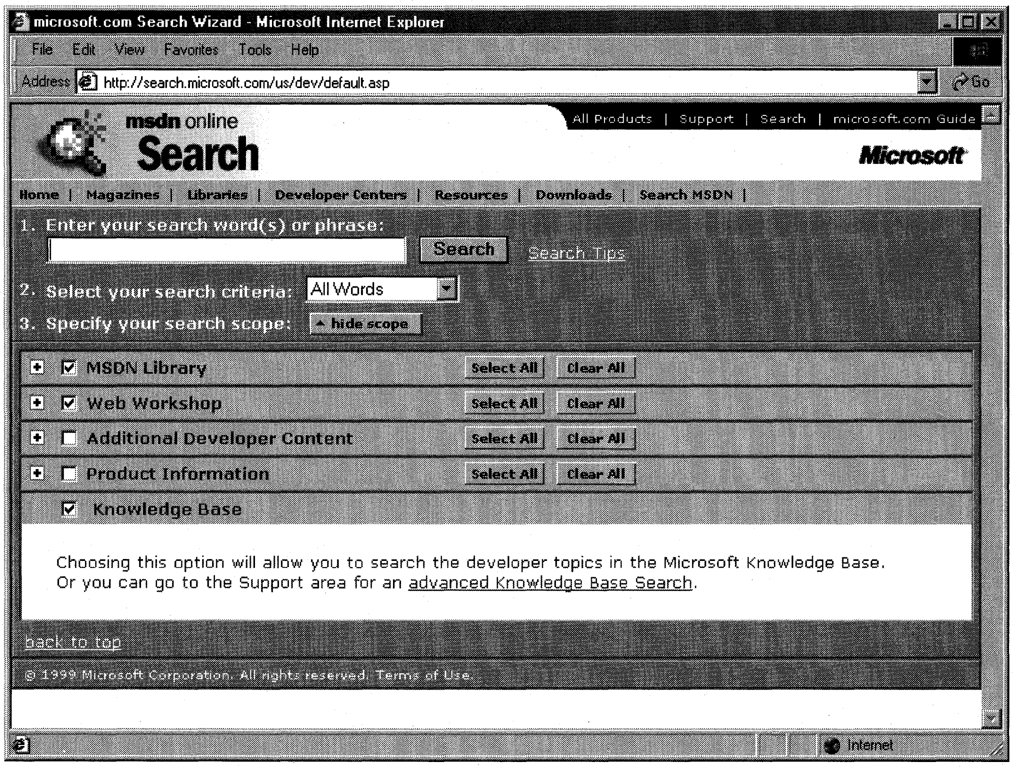
Beta and Preview Releases

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**.

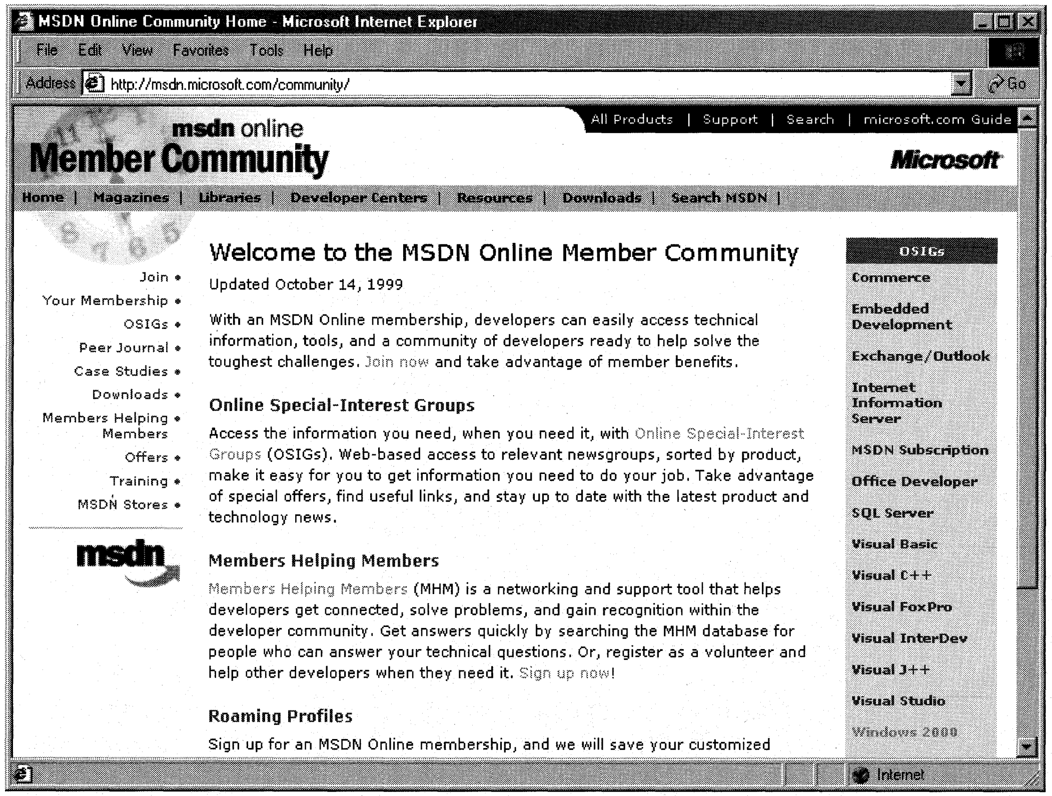

microsoft.com Search Wizard - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://search.microsoft.com/us/dev/default.asp
Go
msdn online
Search
All Products | Support | Search | microsoft.com Guide
Microsoft
Home | Magazines | Libraries | Developer Centers | Resources | Downloads | 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
Web Workshop
Additional Developer Content
Product Information
Knowledge Base
Select All
Clear All
back to top
© 1999 Microsoft Corporation. All rights reserved. Terms of Use.
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.

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.

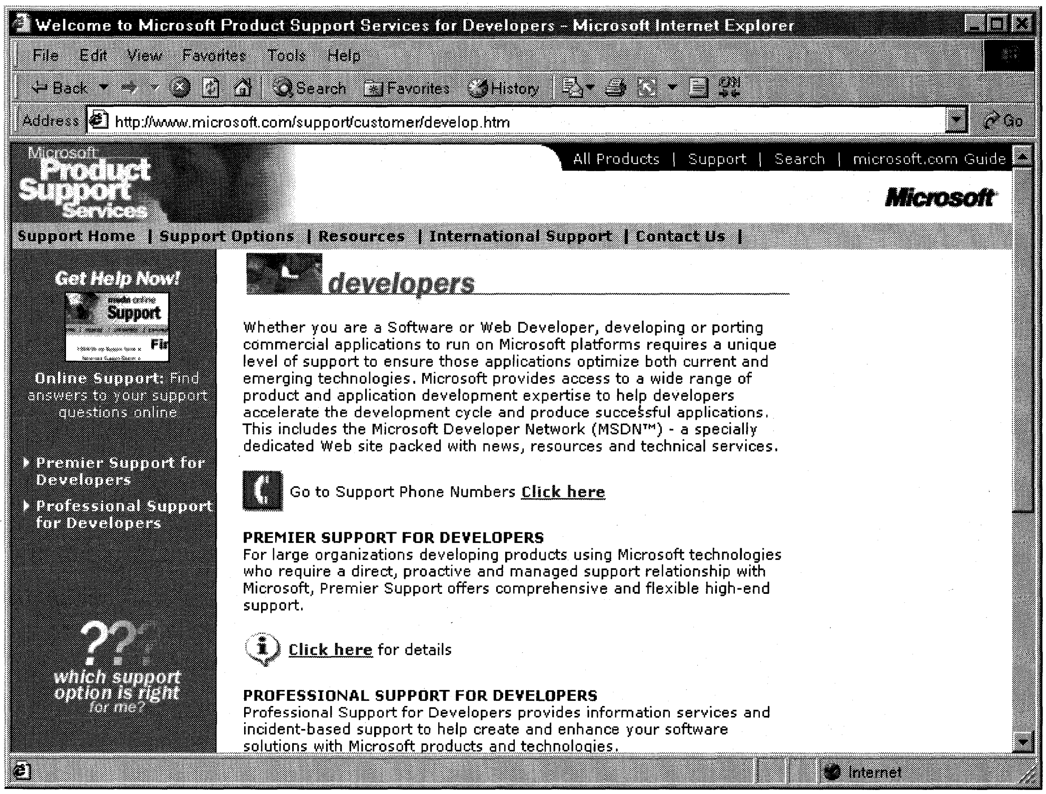
Microsoft provides plenty of resource material through MSDN and MSDN Online, and the WPRS provides a great filtered version of focused reference material and development knowledge. However, there is a lot more information to be had. Some of that information comes from Microsoft, some of it from the general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

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

Developer Support

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

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


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?
Internet

developers

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 as follows.

Desktop Applications Development (one required):

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

Distributed Applications Development (one required):

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

Solutions Architecture:

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

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

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

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

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

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

Conferences

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

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

Other Resources

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

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

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

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

CHAPTER 5

Avoiding Common RPC Programming Errors

This chapter presents a series of simple but common RPC programming errors that developers of Microsoft Windows applications should look out for during the development process.

If you own the *Microsoft Win32 Developer's Reference Library*, some of this information might look familiar, since common RPC programming pitfalls are also included in that library. However, I thought this information was pertinent to this *Networking Services Developer's Reference Library* and especially to this volume, so I've included it here for your reading and pitfall-avoiding pleasure.

The chapter begins with the Solution Summary, which presents you with the short-version solutions to each of the numbered, underlined common coding errors you should look out for. The chapter then moves into common RPC programming errors, which provides further detail on the problems and pitfalls you should take special care to avoid when using RPC.

Solution Summary

This section provides short answer listings for each of the problems explained in the rest of the chapter. For more information about any of these issues, read the corresponding explanation provided later in this chapter.

1. **pointer_default(unique)** and embedded pointers: Check unique pointers for NULL before dereferencing.
2. A valid **switch_is** value in an RPC-capable structure doesn't ensure a non-NULL pointer: When using a **switch_is** construct that has a default clause:
   - Verify that the value switching on is within expected range.
   - Verify that pointers within the switched object are not null before dereferencing them.
3. A NULL DACL affords no protection: Don't use NULL DACLs—they don't protect anything.
4. Call **RpcImpersonateClient()** before any security relevant operation: Impersonate before acting on behalf of the caller and check the result.
5. Starting and stopping impersonation: Stop impersonating when finished acting on behalf of the caller and then check the result.

6. Strings are only zero-terminated when declared with **string** in the .idl: Don't expect strings to be zero-terminated unless **string** is specified in the .idl file.
7. Don't copy arbitrary length data into independently sized buffers: This one's self-answering!
8. **size_is** may result in a zero-length structure; it is not safe to dereference this without first checking its length: Check the length of **size_is**-specified data before dereferencing corresponding pointers.
9. Calculations in a **size_is** or **length_is** specification are susceptible to overflow: Be aware that calculations in MIDL definitions using **size_is** and **length_is** can overflow and that it may be impossible for the server to detect this.
10. Strict context handles: Use strict context handles.

Common RPC Programming Errors

The following sections explain common RPC programming errors in detail, and provide you with pointers (pardon the pun) on how to avoid them.

Pointer_default(unique) and embedded pointers

When an RPC structure contains pointers, its pointers default to the default pointer type (typically set by **pointer_default(unique)**). Under such circumstances, **unique** pointers can be NULL and must be verified to be non-NULL before being dereferenced.

Example

```
[
    pointer_default(unique)
]

typedef struct _RPC_STRUCTURE {
    INSTANCE_DATA *Instance;
} RPC_STRUCTURE;

NTSTATUS
RpcInterface(
    [in] RPC_STRUCTURE *s
)
{
    INSTANCE_DATA *i;

    i = s -> Instance;

    if (*i) {  // error:  i may be NULL.
        [...]
```

A valid switch_is value in an RPC-capable structure doesn't ensure a non-NULL pointer

A valid value for the **switch** field does not change the default of embedded pointers from **unique**. Thus, even when it is valid, the pointer must still be verified to be non-NULL before being dereferenced.

Example

```
typedef struct _rpc_structure {
    ULONG type;
    [switch_is(type)] union {
        [case(1)]  INSTANCE_DATA *Instance;
    } field;
} RPC_STRUCTURE;

NTSTATUS
RpcInterface(
    [in] RPC_STRUCTURE *s
)
{
    INSTANCE_DATA *i;

    switch (s -> type) {
        case 1:
            i = s->Field.Instance;

            if (*i) {  // error:  i may be NULL.
                [...]
```

A NULL DACL affords no protection

A NULL DACL grants access to everyone and protects nothing; it doesn't even protect an object from having its DACL changed to deny access to everyone. In general, an untrusted user should not be granted access to change a security-descriptor's Owner or DACL fields (unless they own the object, in which case no one else should be granted such access).

Example

```
InitializeSecurityDescriptor(pSecurityDescriptor,
                             SECURITY_DESCRIPTOR_REVISION);

fBool = SetSecurityDescriptorDacl(pSecurityDescriptor,
                                  TRUE,   // Dacl present
                                  NULL,   // NULL Dacl
                                  FALSE); // Not defaulted
```

(continued)

(continued)

```
if (fBool) {
    RpcStatus = RpcServerUseProtseqEp(TEXT("ncacn_np"),
                                      10, // max concurrent calls
                                      "PipeName",
                                      pSecurityDescriptor);
```

Remarks

This example exposes this error for RPC, but the error's scope goes beyond RPC. If you create a publicly accessible securable object and do not secure it against unauthorized users' changing of the DACL, *anyone* can lock the object such that *no one* can access it.

Allowing "all" access—for example, applying a DACL granting EVENT_ALL_ACCESS to everyone who accesses an event object—is an equally bad idea, because "all" access typically grants WRITE_DAC and WRITE_OWNER permissions. Granting either of these permissions explicitly enables objects to be locked up. Use (GENERIC_READ | GENERIC_WRITE | GENERIC_EXECUTE) when it's necessary to grant broad access to an object to any non-administrative-level user.

Call RpcImpersonateClient() before any security-relevant operation

The purpose of many RPC servers is to act on behalf of a client, but they must protect system integrity while doing so. Many RPC servers run in the system context; impersonating the caller enables the server to use the user's credentials to access some objects, while otherwise being a part of the secure side of the system.

Example

```
BOOLEAN
RpcInterface(
    [in] ULONG Pid
)
{
    HANDLE Process;

    Process = OpenProcess(PROCESS_VM_WRITE,
                          FALSE,
                          Pid);
```

Remarks

Opening a process by pid *without* first impersonating can provide a caller with access to the process that it normally would not have. The server now has a handle to a process—LSASS for example—allowing it to scribble in the address of a process the user would not have been allowed on its own.

Starting and stopping impersonation

There are a handful of issues that programmers should be on the lookout for when starting and/or stopping impersonation.

Always check the result of RpcImpersonateClient() before a security relevant operation.

The **RpcImpersonateClient()** function returns an indication of success or failure; skip the check and you may as well have skipped the call (which, as we saw previously in this chapter, can be dangerous).

Call RpcRevertToSelf() after security relevant operations

Once a server has acted on behalf of the user by impersonating, it should revert to its own security context by calling **RpcRevertToSelf()**. Although the consequences of failing to undo impersonation are typically not as drastic as failing to impersonate, it can result in failure to function correctly, and cause spurious behavior such as extra audits.

Example

```
BOOLEAN
RpcFunction(
    [in] ULONG pid
)
{
    HANDLE Process;
    RPC_STATUS Status;

    Status = RpcImpersonateClient(NULL);

    if (Status != RPC_S_OK) {
        return FALSE;
    }

    Process = OpenProcess(PROCESS_VM_WRITE,
                    FALSE,
                          Pid);

    [...]

    if (Process) {
        CloseHandle(Process);
    }

    if (RpcRevertToSelf() != RPC_S_OK) {
        // Something bad happened, might need
        // to do more than just clean up this call.
```

(continued)

(continued)

```
        return FALSE;
    }

    return TRUE;
}
```

Remarks

This example shows how to avoid this programming error in RPC, the scope of this error extends beyond RPC. Impersonation is possible over LPC, Named Pipes, and when using Tokens. In all cases, a decision must be made as to whose context (typically System versus untrusted user) should be used for various operations, and impersonation used where appropriate.

Strings are only zero-terminated when declared with string in the .idl

Variably sized RPC buffers can be tricky to deal with. For the most part, variably sized RPC buffers consist of either character strings (which should contain NULL termination defining the size) or amorphous buffers for which there is a corresponding size value passed to the function. The examples that follow document some of the common errors involved in dealing with such buffers.

A buffer that hasn't been explicitly declared as a **string** type cannot be assumed to contain a NULL terminator, and thus must not be passed to C runtime string functions prior to verification of zero termination. This cannot be done by touching a byte outside the valid length of your buffer.

Example

```
BOOLEAN
RpcFunction(
    [in] DWORD NameSize,
    [in, size_is(NameSize)] PWCHAR Name
)
{
    if (Name && wcslen(Name) == 0) {
        return STATUS_CTX_WINSTATION_NAME_INVALID;
    }

    [...]
```

Remarks

The NameSize parameter should be checked and used to bound any operations, either by explicitly attaching a NULL terminator (on the server side), or by using bounded string operations with the size of the buffer specified.

Don't copy arbitrary length data into independently sized buffers

Data buffers should not be assumed to be bound by an arbitrary size limit. An explicit check of the size of the indicated data must be made prior to copying to local fixed-size buffers.

Example

```
HRESULT
RpcSetInfo(
    [in, string] LPCWSTR pwszName
);
{
    WCHAR wszPath[MAX_PATH + 1];

    wcscpy(wszJobPath, "\\");
    wcscat(wszJobPath, pwszName);
```

Remarks

`string` guarantees that the pwszName parameter is zero terminated, not that its length is less than MAX_PATH.

size_is may result in a zero-length structure

A **size_is** specifier can result in a zero-length buffer but a non-NULL buffer pointer (as reference pointers, such as passed parameters, cannot be NULL). A **unique** pointer can always be NULL. The best practice is to verify both the pointer as non-NULL and the buffer size as non-zero to avoid problems.

Example 1

```
ULONG
RpcServerSideRoutine(
    [in] ULONG StructureSize,
    [in, size_is(StructureSize)] PSTRUCTURE Structure
)
{
    ULONG NameLength = 0;

    if (Structure) {
        NameLength = Structure->NameLength;
[…]
```

Remarks

There is no guarantee in this example that the *StructureSize* parameter is sufficient to cover the *NameLength* member, and in fact, the Structure pointer may be non-NULL, while *StructureSize*, and thus the allocated buffer, indicate a zero length.)

Example 2

```
ULONG
RpcServerSideRoutine2(
    [in] ULONG StructureSize,
    [in, unique, size_is(StructureSize)] PSTRUCTURE Structure
)
{
    ULONG NameLength = 0;

    if (StructureSize) {
        NameLength = Structure->NameLength;
[…]
```

Remarks

This example presents a similar problem. In this case, the *StructureSize* parameter could be non-zero, but Structure—being defined as `unique`—could contain a NULL.)

Calculations in a size_is or length_is specification are susceptible to overflow

Calculations in the MIDL definition for a **size_is** or **length_is** specification are subject to overflow problems. If you perform a calculation in a **size_is** or **length_is** specification, consider what difficulties overflow (or rounding) might cause.

Strict context handles

Context handles enable RPC servers to associate information with calls. RPC looks up context handles in a linked list associated with each binding handle. If you have more than one interface accessible from a single binding handle, then the code must be prepared to reject invalid handles or use strict context handles. Interfaces end up being accessible from a single binding handle if they share things like the same named pipe. Using the [strict_context_handle] on the interface definition in the .acf file causes RPC to only allow context handles to be used against interfaces that created them.

CHAPTER 6

Microsoft RPC Model

Microsoft® Remote Procedure Call (RPC) for the C and C++ programming languages is designed to help meet the needs of developers working on the next generation of software for the Microsoft® family of operating systems: MS-DOS®, Windows®, Windows 95, and Windows NT®/Windows 2000.

Microsoft RPC represents the convergence of three powerful programming models:

- The familiar model of developing C applications by writing procedures and libraries
- The model that uses powerful computers as network servers to perform specific tasks for their clients
- The client-server model, in which the client usually manages the user interface while the server handles data storage, queries, and manipulation

This section explains the convergence of these three models in distributed computing, which delivers the ability to share computational power among the computers on a network. It also describes the industry standard for RPC and provides an overview of Microsoft RPC components and their operation.

The Programming Model

In the early days of computing, each program was written as a large monolithic chunk, filled with **goto** statements. Each program had to manage its own input and output to different hardware devices. As the programming discipline matured, this monolithic code was organized into procedures, with the commonly used procedures packed in libraries for sharing and reuse. Today, RPC takes the next step in the development of procedure libraries. Now, procedure libraries can run on other remote computers. (See Figure 6-1.)

The C programming language supports procedure-oriented programming. In C, the main procedure relates to all other procedures as black boxes. For example, the main procedure cannot find out how procedures A, B, and X do their work. The main procedure only calls another procedure; it has no information about how that procedure is implemented. (See Figure 6-2.)

Procedure-oriented programming languages provide simple mechanisms for specifying and writing procedures. For example, the ANSI-standard C-function prototype is a construct used to specify the name of a procedure, the type of the result it returns (if any) and the number, sequence, and type of its parameters. Using the function prototype is a formal way to specify an interface between procedures.

In this topic, the term *procedure* is synonymous with the terms *subroutine* and *subprocedure* and refers to any sequence of computer instructions that accomplishes a functional purpose. In this topic, the term *function* refers to a procedure that returns a value.

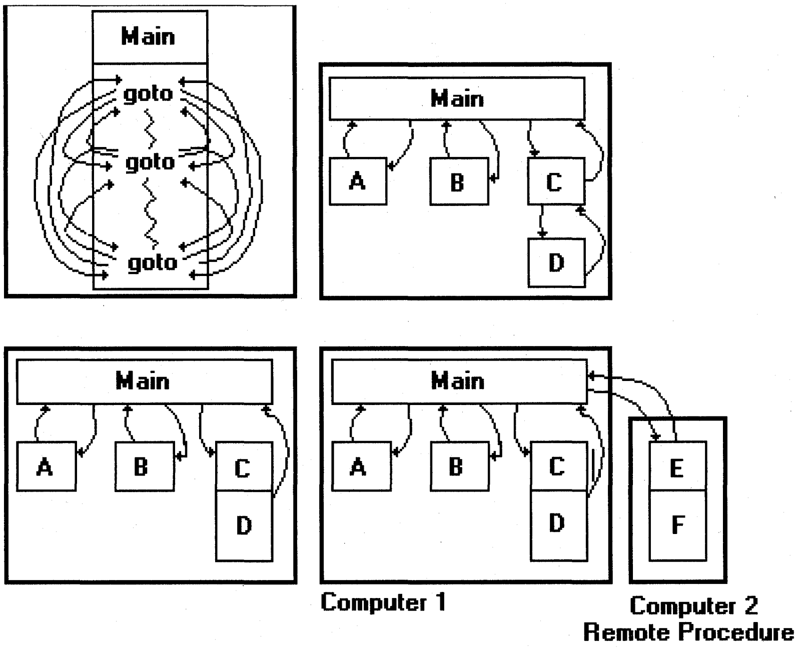


Figure 6-1: Procedure Libraries Running on Remote Computers.

Related procedures are often grouped in libraries. For example, a procedure library can include a set of procedures that performs tasks common to a single domain such as floating-point math operations, formatted input and output, and network functions.

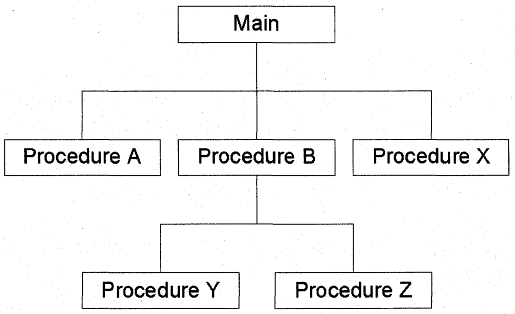


Figure 6-2: Main Procedure Calling Another Procedure.

The procedure library is another level of packaging that makes it easy to develop applications. Procedure libraries can be shared among many applications. Libraries developed in C are usually accompanied by header files. Each program that uses the library is compiled with the header files that formally define the interface to the library's procedures.

The Microsoft RPC tools represent a general approach in which procedure libraries written in C can run on other computers. In fact, an application can link with libraries implemented using RPC without indicating to the user that the application is using RPC.

The Client-Server Model

Client-server architecture is an effective and popular design for distributed applications. In the client-server model, an application is split into two parts: a front-end client that presents information to the user, and a back-end server that stores, retrieves, and manipulates data, and generally handles the bulk of the computing tasks for the client. In this model, the server is usually a more powerful computer than the client and works as a central data store for many client computers, thus making the system easy to administer.

Typical examples of client-server applications include shared databases, remote file servers, and remote printer servers. Figure 6-3 illustrates the client-server model.

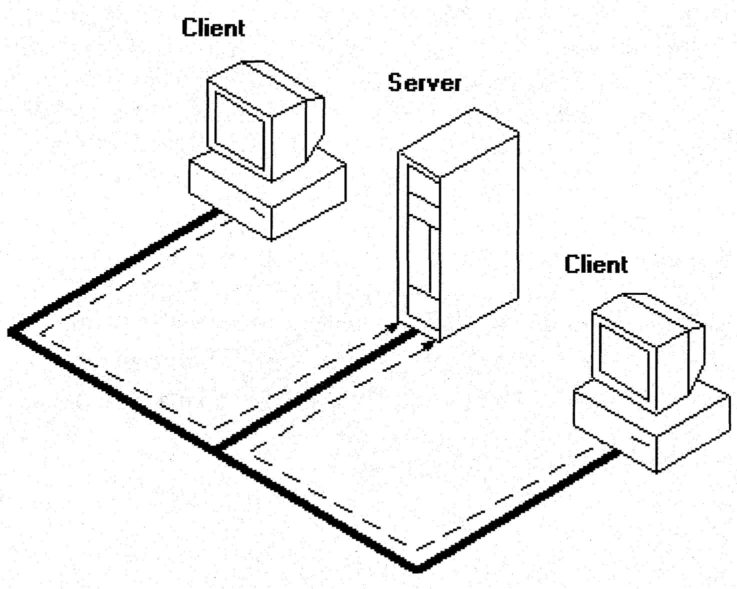


Figure 6-3: The Client-Server Model.

Network systems support the development of client-server applications through an Interprocess Communication (IPC) facility in which the client and server can communicate and coordinate their work. You can use NetBIOS NCBs (Network Control Blocks), mailslots, or named pipes to transfer information between two or more computers.

For example, the client can use an IPC mechanism to send an opcode and data to the server requesting that a particular procedure be called. The server receives and decodes the request and calls the appropriate procedure. The server then performs all the computations needed to satisfy the request and returns the result to the client. Client-server applications are usually designed to minimize the amount of data transmitted over the network.

Using NetBIOS, mailslots, or named pipes to implement interprocess communication means learning specific details relating to network communication. Each application must manage the network-specific conditions. To write this network-specific level of code, you must:

- Learn details relating to network communications and how to handle error conditions.
- Translate data to different internal formats, when the network includes different kinds of computers.
- Support communications using multiple transport interfaces.

In addition to all the possible errors that can occur on a single computer, the network has its own error conditions. For example, a connection can be lost, a server can disappear from the network, the network security service can deny access to system resources, or users can compete for and tie up system resources. Because the state of the network is always changing, an application can fail in new and interesting ways that are difficult to reproduce. For these reasons, each application must rigorously handle all possible error conditions.

When you write a client-server application, you must provide the layer of code that manages network communication. The advantage of using Microsoft RPC is that the RPC tools provide this layer for you. RPC virtually eliminates the need to write network-specific code, thus making it easier to develop distributed applications.

Using the remote procedure call model, RPC tools manage many of the details relating to network protocols and communication. This allows you to focus on the details of the application rather than the details of the network.

The Compute-Server Model

Networking software for personal computers has been built on the model of a powerful computer—the server—that provides specialized services to workstations, or client computers. In this model, servers are designated as file servers, print servers, or communications (modem) servers, depending on whether they are assigned to file sharing or are connected to printers or modems.

RPC represents an evolutionary step in this model. In addition to its traditional roles, a server using RPC can be designated as a computational server or a compute server. In this role, the server shares its own computational power with other computers on the network. A workstation can ask the compute server to perform computations and return the results. The client not only uses files and printers, it also uses the central processing units of other computers.

How RPC Works

The RPC tools make it appear to users as though a client directly calls a procedure located in a remote server program. The client and server each have their own address spaces; that is, each has its own memory resource allocated to data used by the procedure. Figure 6-4 illustrates the RPC architecture.

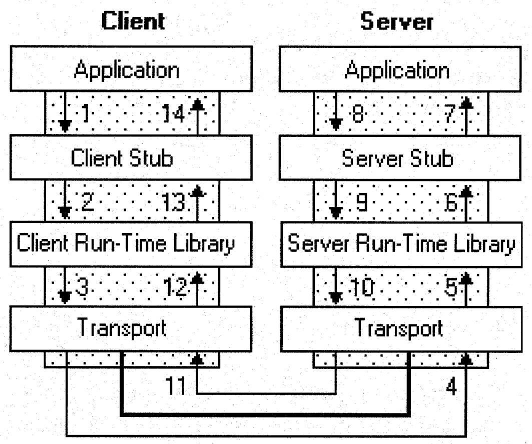


Figure 6-4: RPC Architecture.

As the illustration shows, the client application calls a local stub procedure instead of the actual code implementing the procedure. Stubs are compiled and linked with the client application. Instead of containing the actual code that implements the remote procedure, the client stub code:

- Retrieves the required parameters from the client address space.
- Translates the parameters as needed into a standard network data representation (NDR) format for transmission over the network.
- Calls functions in the RPC client run-time library to send the request and its parameters to the server.

The server performs the following steps to call the remote procedure.

1. The server RPC run-time library functions accept the request and call the server stub procedure.
2. The server stub retrieves the parameters from the network buffer and converts them from the network transmission format to the format the server needs.
3. The server stub calls the actual procedure on the server.

The remote procedure then runs, possibly generating output parameters and a return value. When the remote procedure is complete, a similar sequence of steps returns the data to the client.

1. The remote procedure returns its data to the server stub.
2. The server stub converts output parameters to the format required for transmission over the network and returns them to the RPC run-time library functions.
3. The server RPC run-time library functions transmit the data on the network to the client computer.

The client completes the process by accepting the data over the network and returning it to the calling function.

1. The client RPC run-time library receives the remote-procedure return values and returns them to the client stub.
2. The client stub converts the data from its network data representation to the format used by the client computer. The stub writes data into the client memory and returns the result to the calling program on the client.
3. The calling procedure continues as if the procedure had been called on the same computer.

For Microsoft® Windows® 3.*x*, Windows 95, and Windows NT®/Windows 2000, the run-time libraries are provided in two parts: an import library, which is linked with the application and the RPC run-time library, which is implemented as a Dynamic-Link Library (DLL).

The server application contains calls to the server run-time library functions which register the server's interface and allow the server to accept remote procedure calls. The server application also contains the application-specific remote procedures that are called by the client applications.

OSF Standards for RPC

The design and technology behind Microsoft® RPC is just one part of a complete environment for distributed computing defined by the Open Software Foundation (OSF), a consortium of companies formed to define that environment. The OSF requests proposals for standards, accepts comments on the proposals, votes on whether to accept the standards, and then promulgates them. The components of the OSF Distributed Computing Environment (DCE) are shown in Figure 6-5.

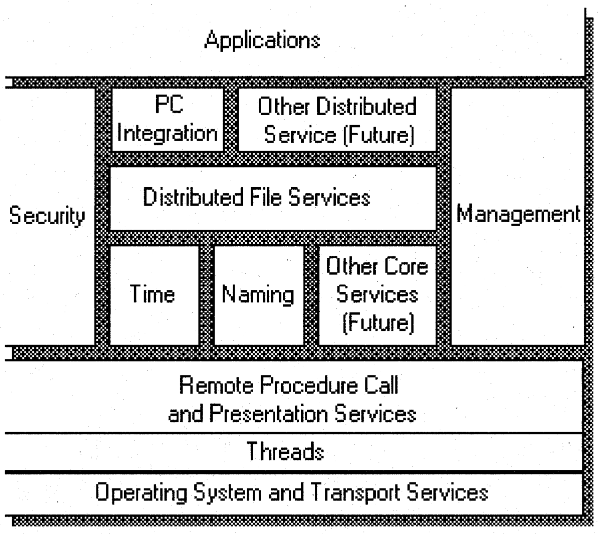

Applications
PC Integration
Other Distributed Service (Future)
Distributed File Services
Security
Management
Time
Naming
Other Core Services (Future)
Remote Procedure Call and Presentation Services
Threads
Operating System and Transport Services

Figure 6-5: Components of the OSF Distributed Computing Environment.

In selecting the RPC standard, the OSF cited the following rationale:

- The three most important properties of a remote procedure call are simplicity, transparency, and performance.
- The selected RPC model adheres to the local procedure model as closely as possible. This requirement minimizes the amount of time developers spend learning the new environment.

- The selected RPC model permits interoperability; its core protocol is well defined and cannot be modified by the user.
- The selected RPC model allows applications to remain independent of the transport and protocol on which they run, while supporting a variety of other transports and protocols.
- The selected RPC model can be easily integrated with other components of the DCE.

The OSF-DCE remote procedure call standards define not only the overall approach, but the language and the specific protocols to use for communications between computers as well, down to the format of data as it is transmitted over the network.

The Microsoft implementation of RPC is compatible with the OSF standard with some minor exceptions. Client or server applications written using Microsoft RPC will interoperate with any DCE RPC client or server whose run-time libraries run over a supported protocol. For a list of supported protocols, see *Building RPC Applications*.

Microsoft RPC Components

The Microsoft® RPC product includes the following major components:

- MIDL compiler
- Run-time libraries and header files
- Transport interface modules
- Name service provider
- Endpoint supply service

In the RPC model, you can formally specify an interface to the remote procedures using a language designed for this purpose. This language is called the Interface Definition Language, or IDL. The Microsoft implementation of this language is called the Microsoft Interface Definition Language, or MIDL.

After you create an interface, you must pass it through the MIDL compiler. This compiler generates the stubs that translate local procedure calls into remote procedure calls. Stubs are placeholder functions that make the calls to the run-time library functions, which manage the remote procedure call. The advantage of this approach is that the network becomes almost completely transparent to your distributed application. Your client program calls what appear to be local procedures; the work of turning them into remote calls is done for you automatically. All the code that translates data, accesses the network, and retrieves results is generated for you by the MIDL compiler and is invisible to your application.

RPC Extends Client-Server Computing

Microsoft® RPC is an evolution of the procedural programming model familiar to all developers. It also represents a new category of specialized server and extends the model of client-server computing. Developers can use Microsoft RPC as a tool to leverage the power of the single personal computer by expanding its computational capacity far beyond its own resources. With RPC, you can harness all of the CPU horsepower available on the network.

Microsoft RPC allows a process running in one address space to make a procedure call that is executed in another address space. The call looks like a standard local procedure call but is actually made to a stub that interacts with the run-time library and performs all the steps necessary to execute the call in the remote address space.

As a tool for creating distributed applications, Microsoft RPC provides the following benefits:

- The RPC programming model is already familiar. You can easily turn functions into remote procedures that simplify development and test cycles.
- RPC hides many details of the network interface from the developer. You do not have to understand specific network functions or low-level network protocols to implement powerful distributed applications.
- RPC solves the data-translation problems that crop up in heterogeneous networks. Individual applications can ignore this problem.
- The RPC approach is scalable. As a network grows, applications can be distributed to more than one computer on the network.
- The RPC model is an industry standard. The Microsoft implementation is compatible with both client and server.

CHAPTER 7

Installing the RPC Programming Environment

You develop RPC distributed applications, for all supported platforms, on the 32-bit Microsoft® Windows NT®/Windows® 2000 platform. This section describes the process of installing the RPC application development environment in the following topics:

- Developing 32-bit Windows Applications
- Developing Macintosh Client Applications

Note See *Building RPC Applications* for information about various build environments.

Developing 32-Bit Windows Applications

When the Platform SDK is installed, the RPC development environment and the run-time libraries are automatically installed. For 32-bit Windows platforms, no additional installation is required. The Microsoft Platform SDK contains the Microsoft® Windows NT®/Windows® 2000 and Windows 95 APIs. When you install the Platform SDK, you install the following RPC tools and files:

- C/C++ language header (.H) files for the RPC run-time libraries and run-time library (.lib and .dll) files for 32-bit Windows platforms
- 32-bit sample programs
- RPC reference Help files
- The **uuidgen** utility

When you install Windows NT/Windows 2000 or Windows 95, you install the following:

- RPC Run-time DLLs
- Microsoft Locator (Windows NT/Windows 2000 only) and RPC Endpoint-mapping services

The following RPC import libraries are included for Microsoft 32-bit Windows clients and servers.

Import library	Description	Platform
Rpcndr.lib	Helper functions	Windows 95/98, Windows NT version 4 and earlier.
Rpcns4.lib	Name-service functions	Windows 95/98, Windows NT version 4 and earlier, Windows 2000.
Rpcrt4.lib	32-bit Windows run-time functions	Windows 95/98, Windows NT version 4 and earlier, Windows 2000.

The following RPC libraries are included for Microsoft 32-bit Windows clients and servers:

Dynamic-link library	Description	Platform
Rpcltc1.dll	Client named-pipe transport	Windows 95/98, Windows NT version 4 and earlier.
Rpclts1.dll	Server named-pipe transport	Windows 95/98, Windows NT version 4 and earlier.
Rpcltc3.dll	Client TCP/IP transport	Windows 95/98, Windows NT version 4 and earlier.
Rpclts3.dll	Server TCP/IP transport	Windows 95/98, Windows NT version 4 and earlier.
Rpcltc5.dll	Client NetBIOS transport	Windows 95/98, Windows NT version 4 and earlier.
Rpclts5.dll	Server NetBIOS transport	Windows 95/98, Windows NT version 4 and earlier.
Rpcltc6.dll	Client SPX transport	Windows 95/98, Windows NT version 4 and earlier.
Rpclts6.dll	Server SPX transport	Windows 95/98, Windows NT version 4 and earlier.
Rpcdgc6.dll	Client IPX transport (Windows NT/ Windows 2000 only)	Windows 95/98, Windows NT version 4 and earlier.
Rpcdgs6.dll	Server IPX transport (Windows NT/ Windows 2000 only)	Windows 95/98, Windows NT version 4 and earlier.

Dynamic-link library	Description	Platform
Rpcdgc3.dll	Client UDP transport (Windows NT/ Windows 2000 only)	Windows 95/98, Windows NT version 4 and earlier.
Rpcdgs3.dll	Server UDP transport (Windows NT/ Windows 2000 only)	Windows 95/98, Windows NT version 4 and earlier.
Rpcns4.dll	Name service	Windows 95/98, Windows NT version 4 and earlier, Windows 2000.
Rpcrt4.dll	32-bit Windows run-time library	Windows 95/98, Windows NT version 4 and earlier, Windows 2000.

You will also need the Microsoft Interface Definition Language (MIDL) compiler. For more information, see *Using The MIDL Compiler*.

Developing Macintosh Client Applications

To develop client-side applications for the Macintosh, you must have the following:

- The Microsoft® Visual C++® development system for the Macintosh. The RPC runtime has been compiled using Visual C++ cross-development tools. In order to use Rpc.lib, you must link against the C run-time and swapper library (Swap.lib) provided with Visual C++, version 2.0 or later.
- The Macintosh RPC SDK, which is contained in a disk image in the Platform SDK directory \rpc_sdk. Run SETUP.EXE from Disk 1 to install the Macintosh header and library files. Note that the current Rpc.lib is native 68K. We currently do not provide a native Power Mac library. RPC runs in emulation on Power Macs.
- The target computer must have a microprocessor of 68020 or later, and it must be running System 7.0 or later.

▶ To connect to the Windows NT/Windows 2000 or Windows 95 server

- Current Windows NT/Windows 2000-supported protocols for the Macintosh are ADSP and TCP/IP. In order to use ADSP, the Windows NT/Windows 2000 server must have both the AppleTalk protocol and Services for Macintosh. Windows 95 supports only the TCP/IP protocol for the Macintosh.

▶ To write an RPC client

1. If you use **atexit** to perform cleanup during shutdown, do not call any RPC APIs in your exit processing function.
2. If a yielding function is not registered, an RPC will not yield on the Macintosh. Register a yielding function by calling **RpcMacSetYieldInfo**.

```
void RPC_ENTRY MacCallbackFunc(short *pStatus)
{
  MSG msg;
  while (*pStatus == 1)
  {
    if(PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }
  }
}
```

3. Most client-side APIs that are supported by Windows 3.*x* are also supported by the Macintosh. The Macintosh does not support the following APIs:
   - **RpcNs*** APIs
   - **RpcMgmt*** APIs
   - **RpcWinSetYieldInfo** (replaced by **RpcMacSetYieldInfo**)

The only authentication service currently supported for the Macintosh is RPC_C_AUTHN_WINNT.

The following protocol sequences are supported:

- ADSP:ncacn_at_dsp
- TCP:ncacn_ip_tcp

CHAPTER 8

Building RPC Applications

The exact procedure for building a distributed Remote Procedure Call (RPC) application varies slightly, depending on:

- the operating-system platform you are developing on
- the target platform
- the version of the MIDL and C or C++ compiler you use
- the API libraries you use

For details, see *Environment, Compiler, and API Set Choices*. This section discusses the process of building client/server applications with Microsoft® Remote Procedure Call.

General Build Procedure

The process for creating a client/server application using Microsoft® RPC is:

- Develop the interface.
- Develop the server that implements the interface.
- Develop the client that uses the interface.

Figure 8-1 illustrates these steps.

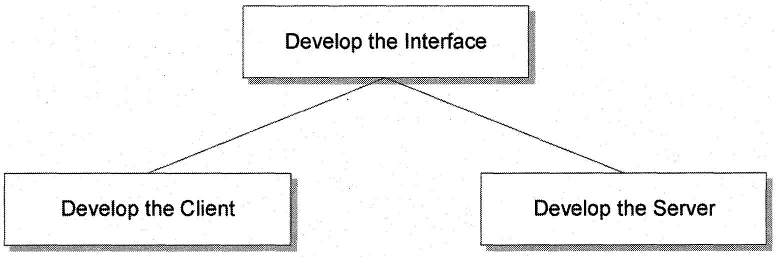


Figure 8-1: Creating a Client/Server Application Using RPC.

Note that it is not only feasible to develop the client and server applications concurrently, it is likely that you will do so. However, since both the client and server programs are dependent on the interface, the interface between them must be developed before the client and server are developed, as shown in the preceding diagram.

This section discusses the steps required for building a client/server application with RPC.

Developing the Interface

An RPC interface describes the remote functions that the server program implements. The interface ensures that the client and server programs communicate using the same rules when the client invokes a remote procedure that the server offers. An interface consists of an interface name, some attributes, optional type or constant definitions, and a set of procedure declarations. Each procedure declaration must contain a procedure name, return type, and parameter list.

Interfaces are defined in the Microsoft® Interface Definition Language (MIDL). If you are familiar with C or C++, MIDL interface definitions will seem fairly straightforward. MIDL resembles C and C++ in many ways.

When developing an RPC application, you use a text editor to define the interface and store it in a text file with an .idl extension. For more information, see *The IDL and ACF Files*. You use the MIDL compiler to generate a header file that your program includes in the client and server source files. The MIDL compiler also generates two C source files. You compile and link one of these to your client program, and the other to your server program. These two C source files are the client and server stubs. For an overview of the client and server stubs, see *How RPC Works*. For an overview on the MIDL compiler, see *Compiling a MIDL File*.

Figure 8-2 shows the process of creating an interface.

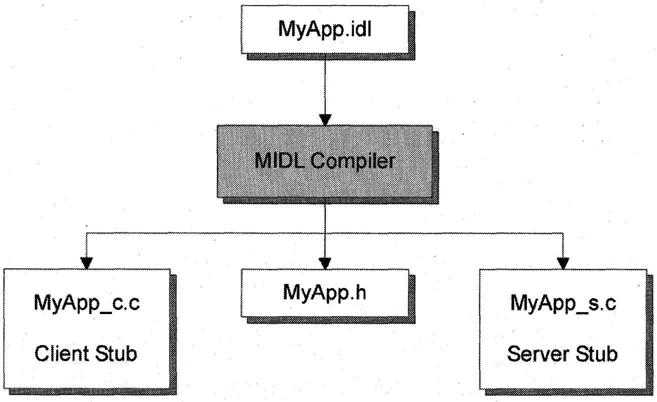


Figure 8-2: Creating an Interface.

It is possible that you will also need to specify an application configuration file (ACF) for input to the MIDL compiler as well. For more information on application configuration files, see *The IDL and ACF Files*.

In addition to the MIDL compiler, you will typically need to use the Uuidgen utility to generate a Universal Unique Identifier (UUID). This section presents information on both of these tools, divided into the following topics:

- Generating Interface UUIDs
- Using MIDL

Generating Interface UUIDs

This section presents information on Universal Unique Identifiers (UUIDs) and the Uuidgen utility in the following topics:

- What is a UUID?
- Using Uuidgen

What is a UUID?

All interfaces must be uniquely identified on a network so that clients can find them. On small networks, the interface's name alone may be sufficient to identify it. However, that is usually not feasible on large networks. Therefore, developers typically assign a Universal Unique Identifier (UUID) to each interface. A UUID is a string that contains a set of hexadecimal digits. Each interface has a different UUID. For details, see *String UUID*.

The textual representation of a UUID is a string consisting of 8 hexadecimal digits followed by a hyphen, followed by three hyphen-separated groups of 4 hexadecimal digits, followed by a hyphen, followed by 12 hexadecimal digits. The following example is a valid UUID string:

```
ba209999-0c6c-11d2-97cf-00c04f8eea45
```

Empty UUIDs are referred to as nil UUIDs rather than NULL UUIDs. The term nil indicates anything that is zero, blank, empty, or uninitialized. An empty string, an empty database record, or an uninitialized UUID are all examples of nil values.

Note The value NULL is the specific value zero. It is often used in C and C++ programming in conjunction with pointers. Nil is a more general term than NULL. Uninitialized object interface UUIDs should always be referred to as nil UUIDs rather than NULL UUIDs.

Using Uuidgen

Microsoft provides a utility program called Uuidgen, that you can use to generate your UUIDs. The Uuidgen program is a command-line utility that creates unique identifiers in the required format using both a time identifier and a computer identifier. It guarantees that any two UUIDs produced on the same computers are unique because they are produced at different times, and that any two UUIDs produced at the same time are unique because they are produced on different machines. The Uuidgen utility generates the UUID in IDL file format or C-language format.

When you run the Uuidgen utility from the command line, you can use the following command switches.

Uuidgen switch	Description
/I	Outputs UUID to an IDL interface template.
/s	Outputs UUID as an initialized C structure.
/o<*filename*>	Redirects output to a file; specified immediately after the **/o** switch.
/n<*number*>	Specifies the number of UUIDs to generate.
/v	Displays version information about Uuidgen.
/h or ?	Displays command-option summary.

Typically, you will use the Uuidgen utility as shown in the following example:

```
uuidgen -i -oMyApp.idl
```

This command generates a UUID and stores it in a MIDL file that you can use as a template. When the preceding command is executed, the contents of MyApp.idl are similar to the following:

```
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(1.0)
]
interface INTERFACENAME
{

}
```

The next step would be to replace the placeholder name, INTERFACENAME, with the actual name of your interface.

Using MIDL

All interfaces for programs using RPC must be defined in Microsoft Interface Definition Language (MIDL) and compiled with the MIDL compiler. The following topics present a brief overview of creating and compiling a MIDL interface:

- Defining an Interface with MIDL
- Compiling a MIDL File

For a detailed discussion of these topics, see *The IDL and ACF Files.*

Defining an Interface with MIDL

MIDL files are text files that you can create and edit with a text editor. If you generate a UUID for your interface, you will typically store the output in a template MIDL file. For more information on UUIDs, see *Generating Interface UUIDs.*

All interfaces in MIDL follow the same format. They begin with a header that contains a list of interface attributes and the interface name. The attributes are enclosed in square brackets. The interface header is followed by its body, which is enclosed in curly brackets. A simple interface is shown in the following example:

```
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(1.0)
]
interface MyInterface
{
  const unsigned short INT_ARRAY_LEN = 100;

  void MyRemoteProc(
      [in] int param1,
      [out] int outArray[INT_ARRAY_LEN]
  );
}
```

Some of the attributes that typically appear in a MIDL interface definition are the UUID and the interface version number. The body of the interface definition must contain the procedure declarations of all of the remote procedures in the interface. It can also contain the declarations of data types and constants that the interface requires.

All parameters in the remote procedure declarations must be declared as **[in]**, **[out]**, or **[in,out]**. These declarations specify that the client program passes data into a remote procedure, gets data out of a remote procedure, or both. For more detailed information about interface parameter declarations, see *The IDL Interface Body.*

Compiling a MIDL File

The MIDL compiler is a command-line tool that is automatically installed with the Platform SDK. Invoke it in an MS-DOS® window by typing the command **midl**, followed by the name of a MIDL file, at the command line. Make sure that the directory containing the MIDL compiler is in your path. The following example illustrates its use:

```
midl MyApp.idl
```

Note that you do not have to include the extension if the file name has the .idl extension. You can also use the MIDL compiler command-line switches by inserting them between the **midl** command and the file name. This is demonstrated in the following example:

```
midl /acf MyApp.acf MyApp.idl
```

In this example, the MIDL compiler is executed using the file MyApp.idl as the input file. The command line switch **/acf** instructs the compiler to use an application configuration file (ACF) for input as well. Application configuration files are discussed more thoroughly in *The IDL and ACF Files.*

For more detailed information on using the MIDL compiler, see the *MIDL Programmer's Guide and Reference* on MSDN, which contains information on the following topics:

- C-Preprocessor Requirements and Options
- C-Compiler Requirements and Options
- Files Generated for an RPC Interface
- MIDL Command-line Reference
- MIDL Language Reference
- MIDL Compiler Errors and Warnings

Developing the Server

When you create a server program for a distributed application, you must use the header file and server stub that the MIDL compiler generates. For details, see *Developing the Interface*. Include the header file in your server C program file. Compile the server stub with the C source files that compose your application. Link the resulting object files together with the RPC run-time library. This process is illustrated in Figure 8-3.

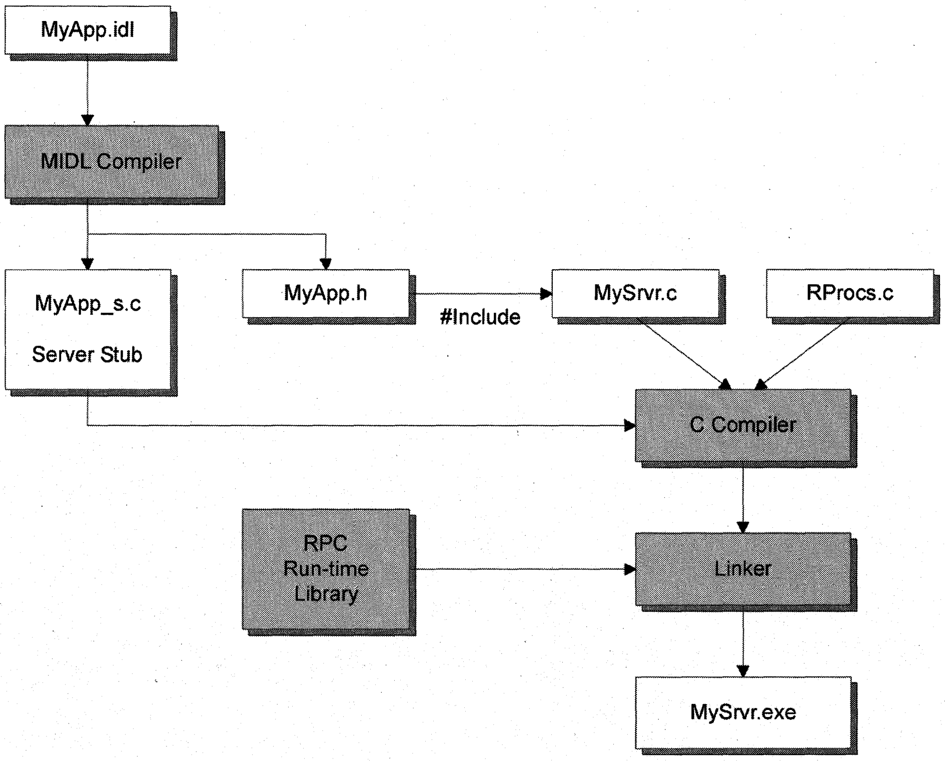


Figure 8-3: Creating a Server Program for a Distributed Application.

As you can see from the example in the illustration, a MIDL file called MyApp.idl was used to define the interface. The MIDL compiler used MyApp.idl to produce MyApp_s.c and MyApp.h. It also produces a C source file for the client stub, but that is not relevant to this particular discussion. The C source file for the server program (in this case, Mysrvr.c) must include the file MyApp.h. It will also need to include the files RPC.h and RPCNDR.h.

The server application was developed in two files, MySrvr.c and RProcs.c. The file MySrvr.c contains the functions necessary for getting the server program up and running. The remote procedures that the server program offers are contained in the file RProcs.c.

The files MySrvr.c and RProcs.c were compiled together with MyApp_s.c to produce object files. The object files were then linked with the RPC run-time library, and any other libraries that they might need. The result is an executable server program named MySrvr.exe.

If you do not compile your IDL file in Open Software Foundation (OSF) compatibility mode (**/osf**), your server program must provide a function for allocating memory and a function for deallocating it. For details, see *How Memory Is Allocated and Deallocated*, and *Pointers and Memory Allocation*.

Developing the Client

Developing an RPC client program is similar to developing the server program. For information on developing an RPC server program, see *Developing the Server*.

As in server development, your client program must include the header file that the MIDL compiler generates from your .idl file. The MIDL compiler also generates a C source file containing the client stub. You must compile this C source file and link it to your client program. (In addition, the MIDL compiler generates a C source file containing the server stub, but that is not relevant to this discussion.)

In addition to compiling and linking the server stub with your program files, you must link the RPC run-time library (and any other libraries your client program needs) to your client program. The process of creating an RPC client program is illustrated in Figure 8-4.

The example in the preceding illustration demonstrates the creation of an RPC client program called MyClnt.exe. The first step is to define the interface in the file MyApp.idl. The MIDL compiler uses MyApp.idl to generate the file MyApp_c.c, which contains the client stub. It also generates the file MyApp.h, which the client program must include. The client program will also need to include the files RPC.h and RPCNDR.h.

The client program itself is created in the file MyClnt.c. In a real project, the client program would typically be composed of several C source files. All of them would need to be compiled and linked together. However, this example uses only one file for simplicity.

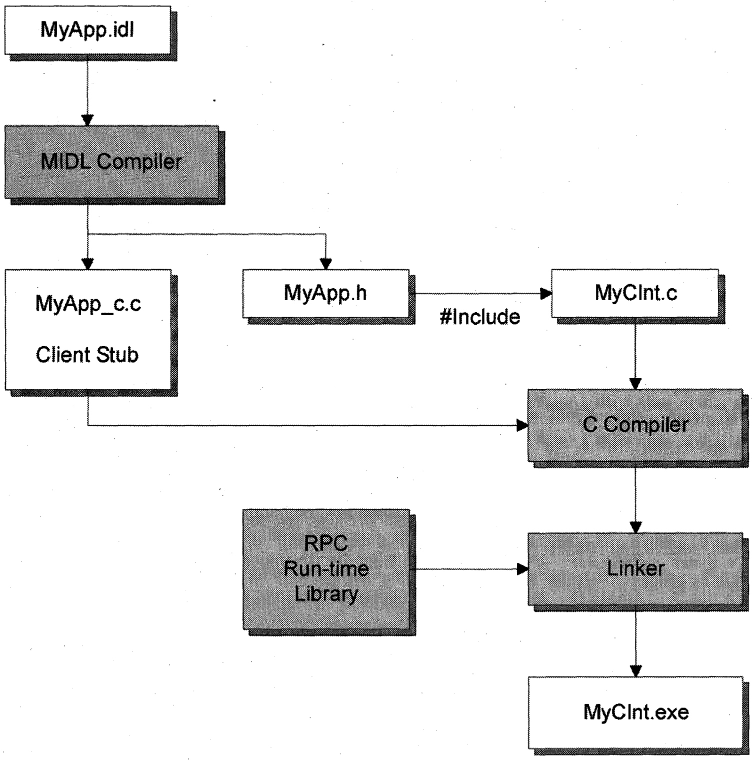


Figure 8-4: Creating an RPC Client Program.

The files MyClnt.c and MyApp_c.c are compiled and linked together with the RPC run-time library, and any other libraries that the client program needs. The result is an executable client program named MyClnt.exe.

If you do not compile your IDL file in OSF compatibility mode (**/osf**), your client program must provide a function for allocating memory and a function for deallocating it. For details, see *How Memory Is Allocated and Deallocated*, and *Pointers and Memory Allocation*.

Environment, Compiler, and API Set Choices

You can develop RPC applications for different target environments: Microsoft® MS-DOS®, Microsoft® Windows® 3.x, Windows 95/98, and Microsoft® Windows NT®/Windows® 2000. You can also develop the executable applications for these target environments using different build environments. Accordingly, you can choose among several development environments, MIDL and C compilers, and API sets.

Available tools and libraries are described in the following table.

Development tool	Description
MIDL 3.0 for 32-bit environment	Produces C source code for 16-bit or 32-bit environments.
C and MSVC for 16-bit environment	Produces 16-bit object files only.
C and MSVC for 32-bit environment (Platform SDK)	Produces 32-bit object files only.
Microsoft® Win32® API	Provided for 32-bit environment only (RPC functions are provided as 32-bit DLLs).
Windows 3.x API	Provided for 16-bit environment only (RPC functions are provided as 16-bit Windows DLLs).

Exception Handling

RPC uses the same approach to exception handling as the Microsoft® Win32® API.

With Microsoft® Windows® 95 and Windows NT®/Windows 2000, the **RpcTryFinally** / **RpcFinally** / **RpcEndFinally** structure is equivalent to the Win32 **try-finally** statement. The RPC exception construct **RpcTryExcept** / **RpcExcept** / **RpcEndExcept** is equivalent to the Win32 **try-except** statement.

The exception-handler structures in RPC are provided so they can also be supported on computers with the Microsoft® MS-DOS® and Windows 3.*x* operating systems. When you use the RPC exception handlers, your client-side source code is portable to Windows NT/Windows 2000, Windows 95/98, Windows 3.*x*, and MS-DOS. The different RPC header files provided for each platform resolve the **RpcTry** and **RpcExcept** structures for each platform. In the Win32 environment, these macros map directly to the Win32 **try-finally** and **try-except** statements. In other environments, these macros map to other platform-specific implementations of exception handlers.

Potential exceptions raised by these structures include the set of error codes returned by the RPC functions with the prefixes RPC_S_ and RPC_X and the set of exceptions returned by Win32. For details, see *RPC Return Values*.

Exceptions that occur in the server application, server stub, and server run-time library (above the transport layer) are propagated to the client. This propagation feature includes multiple layers of callbacks. No exceptions are propagated from the server transport level. Figure 8-5 shows how exceptions are returned from the server to the client.

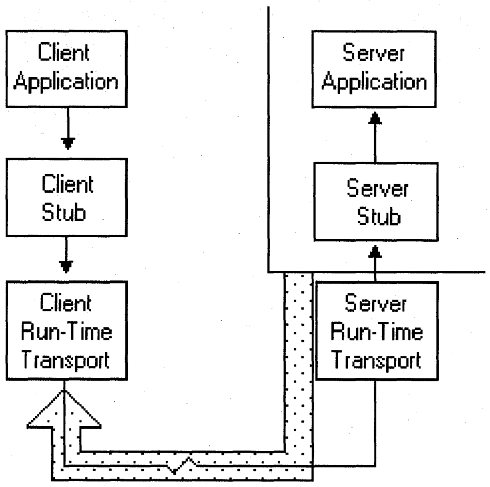


Figure 8-5: Returning Exceptions from Server to Client.

The RPC exception handlers differ slightly from the Open Software Foundation-Distributed Computing Environment (OSF-DCE) exception-handling macros **TRY**, **FINALLY**, and **CATCH**. Various vendors provide include files that map the OSF-DCE RPC functions to the Microsoft RPC functions, including **TRY**, **CATCH**, **CATCH_ALL**, and **ENDTRY**. These header files also map the RPC_S_* error codes onto the OSF-DCE exception counterparts, rpc_s_*, and map RPC_X_* error codes to rpc_x_*. For OSF-DCE portability, use these include files.

For more information about the RPC exception handlers, see ***RpcExcept*** and ***RpcFinally***. For more information about the Win32 exception handlers, see the *Platform SDK* documentation.

CHAPTER 9

Connecting the Client and the Server

To communicate, client and server programs must establish a communication session across the network or networks that connect them. Once they establish the connection, the client can call remote procedures in the server program as if they were local to the client program.

This section provides a conceptual overview of how to establish a connection between clients and servers for remote procedure calls. It does not provide an in-depth discussion of this topic. All of the concepts in this section are presented in detail in later chapters and in the RPC Function Reference section.

Note that the discussion assumes automatic binding handles for the sake of simplicity. However, if your application uses implicit or explicit binding handles, you must take some extra steps in addition to what is presented in this section. For details, see *Binding and Handles*.

Essential RPC Binding Terminology

To better aid in a discussion of the client/server connection process, it is helpful to know the following terms.

Protocol Sequence
When network operating systems communicate with each other, they must listen and speak the same language. These languages are called *protocol sequences*. Client and server programs must use protocol sequences that the network connecting them supports. Microsoft® RPC supports a variety of protocol sequences. For details, see *Selecting a Protocol Sequence*, *Specifying Protocol Sequences*, and *endpoint*.

Server Host Computer or Server Host System
The server program runs on the server host computer. However, much literature on client/server computing refers to both the server program and the server host computer as the "server." The result is that it is not always clear which is being discussed.

Endpoint
Server programs listen to a port or a group of ports on the server host computer for client requests. Server host systems maintain a database of these ports, which are called endpoints in RPC. The database is called the endpoint map.

Binding

Client and server programs create a binding to each other to establish a communication session. A binding contains all of the information the client and server applications need to create the session.

Name Service

A name service is a distributed database containing server program identification information.

How the Server Prepares for a Connection

When a server program begins execution, it must first register the interface or interfaces it contains with the RPC run-time library. It then creates the necessary binding information, and advertises its presence in a name-service database. The server program must also register the endpoint or endpoints it listens to. It can then begin listening for client calls. This process is illustrated in Figure 9-1.

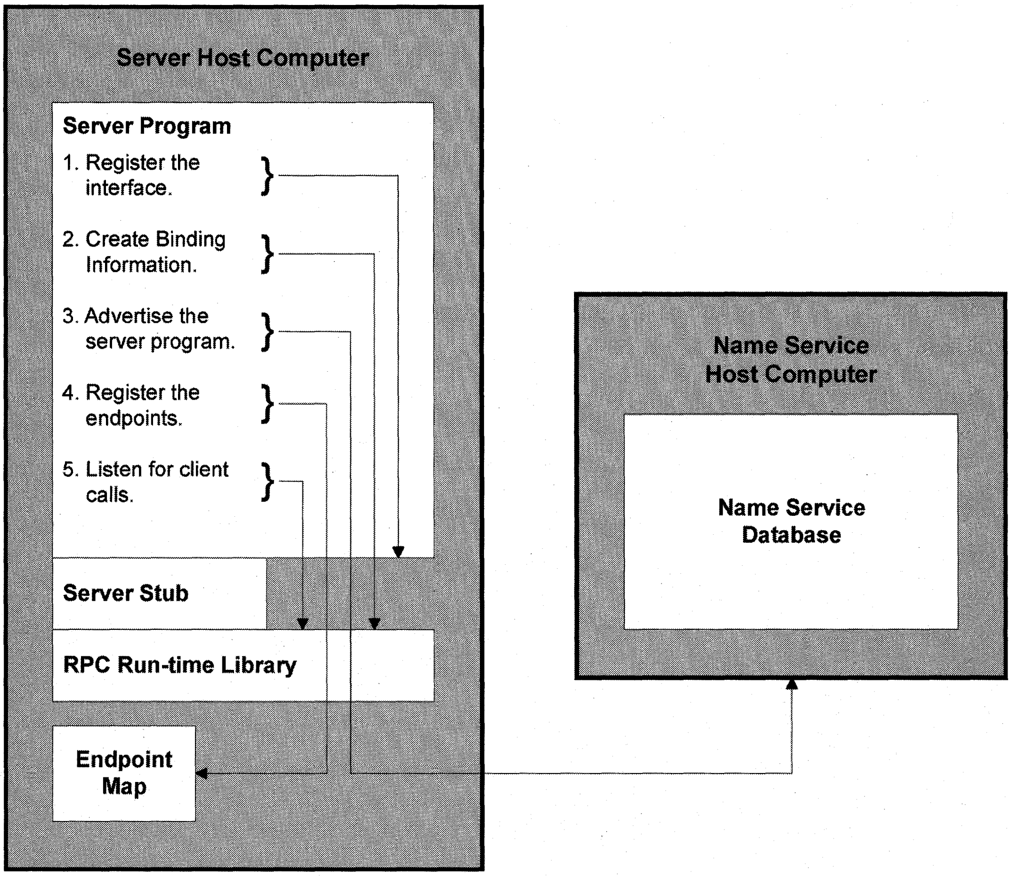

Server Host Computer
Server Program
1. Register the interface.
2. Create Binding Information.
3. Advertise the server program.
4. Register the endpoints.
5. Listen for client calls.
Server Stub
RPC Run-time Library
Endpoint Map
Name Service Host Computer
Name Service Database

Figure 9-1: A Server Preparing for a Connection.

This section presents information on the steps that a server process must take to prepare for a connection.

Registering the Interface

Registering the interface that a server program supports enables client programs to find out on which server host computer the server program runs. Server programs call **RpcServerRegisterIf** to register their interfaces. The following code fragment demonstrates its use:

```
RPC_STATUS status;
status = RpcServerRegisterIf(MyInterface_v1_0_s_ifspec, NULL, NULL);
```

The first parameter to the **RpcServerRegisterIf** function is a structure the MIDL compiler generates from the IDL file that defines the interface (or interfaces) for the server. The second and third parameters are a UUID and an entry-point vector, respectively. They are set to NULL in this example. In many instances, your server program will set these parameter values to NULL. Server programs use the second and third parameters when they provide multiple implementations of the same procedures in an interface. For more information, see *Entry-Point Vectors*.

Server programs can also use **RpcServerRegisterIfEx** to register an interface. One advantage of using this function is that it provides your application with the ability to set a security-callback function. Using security-callback functions is more secure than using entry-point vectors.

Creating Binding Information

To register an endpoint and advertise itself in a name-service database, a server program must create binding information. However, before creating the binding information, your server application must select one or more protocol sequences. Most server programs, use all of the protocol sequences that are available on the network. To do this, they invoke the **RpcServerUseAllProtseqs** function, as shown in the following code fragment:

```
RPC_STATUS status;
status = RpcServerUseAllProtseqs(
    RPC_C_PROTSEQ_MAX_REQS_DEFAULT,     // Default max number of calls.
    NULL);                              // No authentication.
```

The first parameter to the **RpcServerUseAllProtseqs** function is the maximum number of remote procedure calls that the server program will accept at one time. As shown in the code fragment, most server programs set this parameter to RPC_C_PROTSEQ_MAX_REQS_DEFAULT. This sets the RPC library to use the default maximum value. The second parameter is a security descriptor for secure RPC bindings. See *Security* for details.

If you want your application to use just one protocol sequence, call **RpcServerUseProtseq**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqEp**, or **RpcServerUseProtseqEpEx**.

After it selects at least one protocol sequence, a server application must create binding information for each protocol sequence that it uses. It stores the binding information in a binding handle. For details, see *Binding and Handles*. If the server program uses more than one protocol sequence, it must create more than one binding handle. A set of binding handles is called a binding vector.

Use the **RpcServerInqBindings** function to obtain a binding vector for the server application as shown in the following code fragment:

```
RPC_STATUS status;
RPC_BINDING_VECTOR *rpcBindingVector;

status = RpcServerInqBindings(&rpcBindingVector);
```

The only parameter to the **RpcServerInqBindings** function is a pointer to a pointer to an **RPC_BINDING_VECTOR** structure. The RPC run-time library will dynamically allocate an array of binding vectors and store the address of the array in the parameter variable (in this case, *rpcBindingVector*). Before it terminates, your server application must call the **RpcBindingVectorFree** function to free the memory that the **RpcServerInqBindings** function allocates.

Advertising the Server Program

After a server registers all interfaces it supports and creates the binding information, it can advertise its presence in a name-service database. Use the **RpcNsBindingExport** function to accomplish this, as shown in the following code fragment:

```
// This example assumes that MyInterface_v1_0_s_ifspec is a valid data
// structure that represents the interface being registered. The
// variable is a valid pointer to a binding vector.
RPC_STATUS status;
status = RpcNsBindingExport(
    RPC_C_NS_SYNTAX_DEFAULT,
    /.:/servers/our_dept/iface_aserver,
    MyInterface_v1_0_s_ifspec,
    rpcBindingVector,
    NULL);
```

The first parameter to the **RpcNsBindingExport** function specifies the syntax of the second parameter, *EntryName*. Microsoft® RPC currently supports only one name-service syntax. Therefore, applications should set the first parameter of **RpcNsBindingExport** to the value RPC_C_NS_SYNTAX_DEFAULT.

Microsoft RPC does not force an entry name format on the application. However, the client and server programs must use the same format. Set the *IfSpec* parameter to a structure that specifies the interface to export to the name-service database. The MIDL compiler generates this data structure. You will find it in the header file that the MIDL compiler produces from your MIDL file.

The fourth parameter is a pointer to the binding vector that your server program obtained from the **RpcServerInqBindings** function.

In addition to exporting binding handles to the name-service database, your program can export object UUIDs. In this example, only binding handles are exported. Therefore, the last parameter to **RpcNsBindingExport** is set to NULL.

Registering Endpoints

Registering the server program in the endpoint map of the server host computer enables client programs to determine which endpoint (usually a TCP/IP port or a named pipe) the server program is listening to. To register itself in the server host system's endpoint map, a server program calls the **RpcEpRegister** function as shown in the following code fragment:

```
// This example assumes that MyInterface_v1_0_s_ifspec is a valid data
// structure that represents the interface being registered. The
// variable is a valid pointer to a binding vector.
RPC_STATUS status;
status = RpcEpRegister(
    MyInterface_v1_0_s_ifspec,
    rpcBindingVector,
    NULL,
    NULL);
```

The first parameter to **RpcEpRegister** is the structure that represents the interface. You can find it in the header file that the MIDL compiler generated from your MIDL file for this distributed application. See *Developing the Interface*. Next, **RpcEpRegister** needs your application to pass a set of binding handles that are stored in a binding vector.

In addition to registering interface names, your server application can also register object UUIDs in the endpoint map. In this example, there are no object UUIDs to register, so the third parameter to **RpcEpRegister** is set to NULL.

The last parameter is a comment string. The RPC run-time library does not use this string. Your client programs read this string from the endpoint map and, if you set it up to do so, display it to users.

Listening for Client Calls

After your server application has registered its interfaces, created the necessary binding information, advertised its interfaces in the name-service database, and registered its endpoints, it is ready to begin listening for remote procedure calls from client programs.

To listen for remote procedure calls, your server program must call **RpcServerListen**, as shown in the following code fragment:

```
RPC_STATUS status;
status = RpcServerListen(
    1,
    RPC_C_LISTEN_MAX_CALLS_DEFAULT,
    0);
```

The **RpcServerListen** function can create multiple threads to listen for concurrent remote procedure calls. The first parameter to **RpcServerListen** is the minimum number of threads to create.

The second parameter to **RpcServerListen** is the maximum number of concurrent remote procedure calls to handle. If you want your application to use the default maximum value, pass RPC_C_LISTEN_MAX_CALLS_DEFAULT as the value for this parameter.

The DCE specification calls for **RpcServerListen** to keep running until it receives a signal to stop. One Microsoft extension to this function is to enable it to begin listening and return immediately. If you want your application to use the default DCE behavior, set the third parameter to zero. See ***RpcServerListen***, ***RpcMgmtStopServerListening***, and ***RpcMgmtWaitServerListen*** for details.

How the Client Establishes a Connection

To establish a client/server communication session with a server program, client applications with automatic binding handles can simply call remote procedures. When they do, the RPC run-time library finds the computer that hosts the server program. It then finds the endpoint that the server program is listening to and creates a binding handle. Once the client has a binding handle to the server program, it can execute any remote procedures that the server program offers. Figure 9-2 illustrates this process.

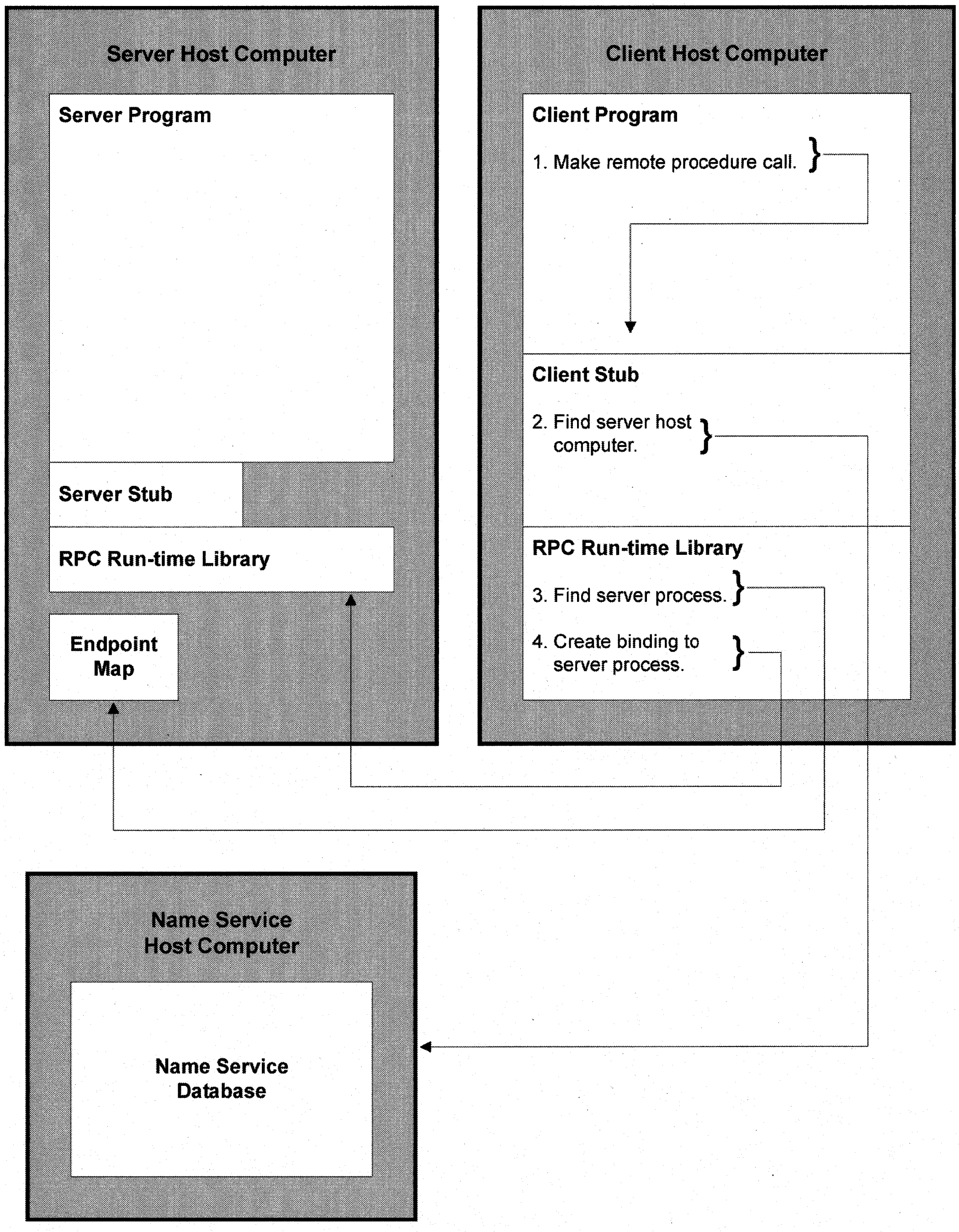


Figure 9-2: Establishing a Communication Session with a Server Program.

This section presents information about how the client connects to the server program and executes remote procedures that it offers.

Making a Remote Procedure Call

The client program of distributed applications that use automatic binding handles can simply execute remote procedures as if they were procedures local to the client program.

Microsoft® RPC also offers implicit and explicit binding handles. These binding handles offer your client and server programs more control over the process of executing remote procedures. However, with increased flexibility and control also comes increased complexity. Implicit and explicit binding handles require that your application manage all or part of the binding process. You must decide what features and levels of control are appropriate for your application. For an in-depth discussion of the different handle types and the flexibility they offer, see *Binding and Handles*.

Finding the Server Host Computer

When a client program invokes a remote procedure, it is actually calling a procedure in the client stub that you compiled and linked to it. For more information on the client stub, see *How RPC Works*.

Clients must be able to find server programs. Most networks are large enough to require the use of a name-service database. Server programs register their interfaces in the name server database. See *Registering the Interface*. The client stub uses the name-service database to find the computer that currently hosts the interface.

It is possible for multiple computers to host the same interface. With automatic binding handles, your client program has no control over which server host computer it connects to, as long as the server host computer offers the interface the client requires. Other types of binding handles offer your client programs the ability to connect to specific server programs. Your server applications can also selectively refuse to allow clients to connect. This lets you specify which servers clients can and can't connect to when requesting services. For more information, see *Binding and Handles*.

Finding the Server Program

After the client stub uses the name-service database to find a server host system that offers the interface it is looking for, the client RPC run-time library finds the server process. To do this, it queries the endpoint map on the server host system. The endpoint map contains information about which endpoint the server is listening to.

Creating a Binding

When the RPC client run-time library finds a server host system that provides the interface the client program needs and the endpoint that the server application is listening to, it creates a binding handle.

A binding handle is a structure that contains the information that client and server programs need to establish a communication session between the client and the server. After they have created the binding handles they need, client and server program do not need to continue to query the name-service database or endpoint map to maintain the communication session.

Client and server programs must not try to access the contents of the binding handle structure. The RPC run-time library functions use and manage the information in binding handles. The most that client and server applications need to do is pass the binding handle to RPC library calls and remote procedures. This is only the case if the application does not use automatic binding handles. See *Binding and Handles* for details.

CHAPTER 10

An RPC Tutorial

This chapter's tutorial takes you through the steps required to create a simple, single-client, single-server distributed application from an existing stand-alone application. These steps are:

- Create interface definition and application configuration files.
- Use the MIDL compiler to generate C-language client and server stubs and headers from those files.
- Write a client application that manages its connection to the server.
- Write a server application that contains the actual remote procedures.
- Compile and link these files to the RPC run-time library to produce the distributed application.

The client application passes a character string to the server in a remote procedure call, and the server prints the string "Hello, World" to its standard output.

The complete source files for this example application, with additional code to handle command-line input and to output various status messages to the user, are in the Platform SDK directory \mstools\samples\rpc\hello and in the Code Samples, RPC section of the Platform SDK documentation.

The Stand-Alone Application

This stand-alone application, which consists of a call to a single function, forms the basis of our distributed application. The function, HelloProc, is defined in its own source file so that it can be compiled and linked with either a stand-alone application or a distributed application.

```
/* file hellop.c */
#include <stdio.h>
void HelloProc(unsigned char * pszString)
{
    printf("%s\n", pszString);
}

/* file: hello.c, a stand-alone application */
#include "hellop.c"
void main(void)
{
```

(continued)

(continued)

```
    unsigned Char * pszString = "Hello, World";
    HelloProc(pszString);
}
```

Defining the Interface

An interface definition is a formal specification for how a client application and a server application communicate with each other. The interface defines how the client and server "recognize" each other, the remote procedures that the client application can call, and the data types for those procedures' parameters and return values. It also specifies how the data is transmitted between client and server.

You define this interface in the Microsoft® Interface Definition Language (MIDL) which consists of C-language definitions augmented with keywords, called attributes, which describe how the data is transmitted over the network.

The interface definition (.IDL) file contains type definitions, attributes and function prototypes that describe how data is transmitted on the network. The application configuration (.ACF) file contains attributes that configure your application for a particular operating environment without affecting its network characteristics.

Generating the UUID

The first step in defining the interface is to use the **uuidgen** utility to generate a universally unique identifier (UUID). A UUID enables the client and server applications identify each other. The **uuidgen** utility (UUIDGEN.EXE) is automatically installed when you install the Platform SDK. The following command generates a UUID and creates a template file called hello.idl:

```
C:\>uuidgen /i /ohello.idl
```

Your hello.idl template will look like this (with a different UUID, of course):

```
[
    uuid(7a98c250-6808-11cf-b73b-00aa00b677a7),
    version(1.0)
]
interface INTERFACENAME
{

}
```

The IDL File

The IDL file consists of one or more interface definitions, each of which has a header and a body. The header contains information that applies to the entire interface, such as the UUID. This information is enclosed in square brackets and is followed by the keyword **interface** and the interface name. The body contains C-style data type definitions and function prototypes, augmented with attributes that describe how the data is transmitted over the network.

In this example, the interface header contains only the UUID and the version number. The version number ensures that when there are multiple versions of an RPC interface, only compatible versions of the client and server will be connected.

The interface body contains the function prototype for **HelloProc**. In this prototype, the function parameter *pszString* has the attributes **[in]** and **[string]**. The **[in]** attribute tells the run-time library that the parameter is passed only from the client to the server. The **[string]** attribute specifies that the stub should treat the parameter as a C-style character string.

The client application should be able to shut down the server application, so the interface contains a prototype for another remote function, **Shutdown**, that will be implemented later in this tutorial.

```
//file hello.idl
[
    uuid(7a98c250-6808-11cf-b73b-00aa00b677a7),
    version(1.0)
]
interface hello
{
    void HelloProc([in, string] unsigned char * pszString);
    void Shutdown(void);
}
```

The ACF File

The ACF file enables you to customize your client and/or server applications' RPC interface without affecting the network characteristics of the interface. For example, if your client application contains a complex data structure that only has meaning on the local machine, you can specify in the ACF file how the data in that structure can be represented in a machine-independent form for remote procedure calls.

This tutorial demonstrates another use of the ACF file—specifying the type of binding handle that represents the connection between client and server. The **[implicit_handle]** attribute in the ACF header allows the client application to select a server for its remote procedure call. The ACF defines the handle to be of the type **handle_t** (a MIDL primitive data type). The MIDL compiler will put the binding handle name that the ACF specified, *hello_IfHandle* into the header file it generates. Notice that this particular ACF file has an empty body.

```
//file: hello.acf
[
    implicit_handle (handle_t hello_IfHandle)
]
interface hello
{
}
```

The MIDL compiler has an option, **/app_config**, that lets you include certain ACF attributes, such as **implicit_handle**, in the IDL file, rather than creating a separate ACF file. Consider using this option if your application doesn't require a lot of special configuration and if strict OSF compatibility is not an issue. For more information, see *OSF Standards for RPC.*

Generating the Stub Files

After defining the client/server interface, you usually develop your client and server source files. Next use a single makefile to generate the stub and header files. Compile and link the client and server applications. However, if this is your first exposure to the distributed computing environment, you may want to invoke the MIDL compiler now to see what MIDL generates before you continue. The MIDL compiler (MIDL.EXE) is automatically installed when you install the Platform SDK.

When you compile these files, make sure that hello.idl and hello.acf are in the same directory. The following command will generate the header file hello.h, and the client and server stubs, hello_c.c and hello_s.c:

```
C:\> midl hello.idl
```

Notice that hello.h contains function prototypes for **HelloProc** and **Shutdown**, as well as forward declarations for two memory management functions,**midl_user_allocate** and **midl_user_free**. You will provide these two memory management functions in the server application. If data were being transmitted from the server to the client (by means of an **[out]** parameter) you would also need to provide these two memory management functions in the client application.

Note the definitions for the global handle variable, *hello_IfHandle*, and the client and server interface handle names, *hello_v1_0_c_ifspec* and *hello_v1_0_s_ifspec*. The client and server applications will use the interface handle names in run-time calls.

At this point, you don't need to do anything with the stub files hello_c.c and hello_s.c.

```
/*file: hello.h */
/* this ALWAYS GENERATED file contains the definitions for the interfaces */
/* File created by MIDL compiler version 3.00.06
/* at Tue Feb 20 11:33:32 1996 */
/* Compiler settings for hello.idl:
    Os, W1, Zp8, env=Win32, ms_ext, c_ext
    error checks: none */
//@@MIDL_FILE_HEADING(  )
#include "rpc.h"
#include "rpcndr.h"

#ifndef __hello_h__
#define __hello_h__

#ifdef __cplusplus
extern "C"{
#endif

/* Forward Declarations */

void __RPC_FAR * __RPC_USER MIDL_user_allocate(size_t);
void __RPC_USER MIDL_user_free( void __RPC_FAR * );

#ifndef __hello_INTERFACE_DEFINED__
#define __hello_INTERFACE_DEFINED__

/****************************************
 * Generated header for interface: hello
 * at Tue Feb 20 11:33:32 1996
 * using MIDL 3.00.06
 ****************************************/
/* [implicit_handle][version][uuid] */

            /* size is 0 */
void HelloProc(
    /* [string][in] */ unsigned char __RPC_FAR *pszString);
    /* size is 0 */
void Shutdown( void);
extern handle_t hello_IfHandle;

extern RPC_IF_HANDLE hello_v1_0_c_ifspec;
extern RPC_IF_HANDLE hello_v1_0_s_ifspec;
#endif /* __hello_INTERFACE_DEFINED__ */

/* Additional Prototypes for ALL interfaces */
/* end of Additional Prototypes */
#ifdef __cplusplus
}
#endif
#endif
```

The Client Application

The helloc.c source file contains a directive to include the MIDL-generated header file, hello.h. Within hello.h are directives to include rpc.h and rpcndr.h, which contain the definitions for the RPC run-time routines and data types that the client and server applications use.

Because the client is managing its connection to the server, the client application calls run-time functions to establish a handle to the server and to release this handle after the remote procedure calls are complete. The function **RpcStringBindingCompose** combines the components of the binding handle into a string representation of that handle and allocates memory for the string binding. The function **RpcBindingFromStringBinding** creates a server binding handle, *hello_IfHandle*, for the client application from that string representation.

In the call to **RpcStringBindingCompose**, the parameters do not specify the UUID because this tutorial assumes there is just one implementation of the interface "hello." In addition, the call does not specify a network address because the application will use the default, which is the local host machine. The protocol sequence is a character string that represents the underlying network transport. The endpoint is a name which is specific to the protocol sequence. This example uses named pipes (a native Microsoft® Windows NT®/Windows® 2000 protocol) for its network transport, so the protocol sequence is "ncacn_np". The endpoint name is "\pipe\hello".

The actual remote procedure calls, **HelloProc** and **Shutdown**, take place within the RPC exception handler—a set of macros that let you control exceptions that occur outside the application code. If the RPC run-time module reports an exception, control passes to the **RpcExcept** block. This is where you would insert code to do any needed cleanup and then exit gracefully. This example program simply informs the user that an exception ocurred. If you do not want to use exceptions, you can use the ACF attributres comm_status and fault_status to report errors.

After the remote procedure calls are completed, the client first calls **RpcStringFree** to free the memory that was allocated for the string binding. Note that once the binding handle has been created, a client program can free a string binding at any time. The client next calls **RpcBindingFree** to release the handle.

```
/* file: helloc.c */
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "hello.h"

void main()
{
    RPC_STATUS status;
    unsigned char * pszUuid             = NULL;
unsigned char * pszProtocolSequence = "ncacn_np";
unsigned char * pszNetworkAddress   = NULL;
unsigned char * pszEndpoint    = "\\pipe\\hello";
unsigned char * pszOptions          = NULL;
unsigned char * pszStringBinding    = NULL;
unsigned char * pszString      = "hello, world";
unsigned long ulCode;

status = RpcStringBindingCompose(pszUuid,
                                 pszProtocolSequence,
                                 pszNetworkAddress,
                                 pszEndpoint,
                                 pszOptions,
                                 &pszStringBinding);
if (status)
{
    exit(status);
}
status = RpcBindingFromStringBinding(pszStringBinding,
                                     &hello_IfHandle);

if (status)
{
    exit(status);
}


RpcTryExcept
{
    HelloProc(pszString);
    Shutdown();
}
RpcExcept(1)
{
    ulCode = RpcExceptionCode();
    printf("Runtime reported exception 0x%lx = %ld\n", ulCode, ulCode);
}
RpcEndExcept

status = RpcStringFree(&pszStringBinding);

if (status)
{
    exit(status);
```

(continued)

(continued)

```
    }

    status = RpcBindingFree(&hello_IfHandle);

    if (status)
    {
        exit(status);
    }

    exit(0);

}  // end main()
```

The Server Application

The server side of the distributed application informs the system that its services are available. It then waits for client requests.

Depending on the size of your application and your coding preferences, you can choose to implement remote procedures in one or more separate files. In this tutorial program, the source file hellos.c contains the main server routine. The file hellop.c contains the remote procedure.

The benefit of organizing the remote procedures in separate files is that the procedures can be linked with a stand-alone program to debug the code before it is converted to a distributed application. After the procedures work in the stand-alone program, you can compile and link the the source files containing the remote procedures with the server application. As with the client-application source file, the server-application source file must include the hello.h header file.

The server calls the RPC run-time functions **RpcServerUseProtseqEp** and **RpcServerRegisterIf** to make binding information available to the client. This example program passes the interface handle name to **RpcServerRegisterIf**. The other parameters are set to NULL. The server then calls the **RpcServerListen** function to indicate that it is waiting for client requests.

The server application must also include the two memory management functions that the server stub calls:**midl_user_allocate** and **midl_user_free**. These functions allocate and free memory on the server when a remote procedure passes parameters to the server. In this example program, **midl_user_allocate** and **midl_user_free** are simply wrappers for the C-library functions **malloc** and **free**. (Note that, in the MIDL compiler- generated forward declarations, "midl" is uppercase. The header file rpcndr.h defines midl_user_free and midl_user_allocate to be MIDL_user_free and MIDL_user_allocate, respectively).

```
/* file: hellos.c */
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "hello.h"

void main()
{
    RPC_STATUS status;
    unsigned char * pszProtocolSequence = "ncacn_np";
    unsigned char * pszSecurity     = NULL; /*Security not implemented*/
    unsigned char * pszEndpoint    = "\\pipe\\hello";
    unsigned int    cMinCalls      = 1;
    unsigned int    cMaxCalls      = 20;
    unsigned int    fDontWait      = FALSE;

    status = RpcServerUseProtseqEp(pszProtocolSequence,
                                   cMaxCalls,
                                   pszEndpoint,
                                   pszSecurity);

    if (status)
    {
        exit(status);
    }

    status = RpcServerRegisterIf(hello_v1_0_s_ifspec,
                                 NULL,
                                 NULL);

    if (status)
    {
        exit(status);
    }

    status = RpcServerListen(cMinCalls,
                             cMaxCalls,
                             fDontWait);

    if (status)
    {
        exit(status);
    }

 }  // end main()
```

(continued)

(continued)

```
/****************************************************/
/*        MIDL allocate and free                   */
/****************************************************/

void __RPC_FAR * __RPC_USER midl_user_allocate(size_t len)
{
    return(malloc(len));
}

void __RPC_USER midl_user_free(void __RPC_FAR * ptr)
{
    free(ptr);
}
```

Stopping the Server Application

A robust server application should stop listening for clients and clean up after itself before shutting down. The two core server functions that accomplish this are **RpcMgmtStopServerListening** and **RpcServerUnregisterIf**.

The server function **RpcServerListen** doesn't return to the calling program until an exception occurs or until a call to **RpcMgmtStopServerListening** occurs. By default, only another server thread is allowed to halt the RPC server by using **RpcMgmtStopServerListening**. Clients who try to halt the server will receive the error RPC_S_ACCESS_DENIED. However, it is possible to configure RPC to allow some or all clients to stop the server. See ***RpcMgmtStopServerListening*** for details.

You can also have the client application make a remote procedure call to a shutdown routine on the server. The shutdown routine calls **RpcMgmtStopServerListening** and **RpcServerUnregisterIf**. This tutorial's example program application uses this approach by adding a new remote function, **Shutdown**, to the file hellop.c.

In the **Shutdown** function, the single NULL parameter to **RpcMgmtStopServerListening** indicates that the local application should stop listening for remote procedure calls. The two NULL parameters to **RpcServerUnregisterIf** are wildcards, indicating that all interfaces should be unregistered. The FALSE parameter indicates that the interface should be removed from the registry immediately, rather than waiting for pending calls to complete.

```
/* add this function to hellop.c */
void Shutdown(void)
{
    RPC_STATUS status;

    status = RpcMgmtStopServerListening(NULL);

    if (status)
    {
       exit(status);
    }

    status = RpcServerUnregisterIf(NULL, NULL, FALSE);

    if (status)
    {
       exit(status);
    }
} //end Shutdown
```

Compiling and Linking

The following makefile shows the dependencies among the files needed to compile the client and server applications and link them to the RPC run-time library and the standard C run-time library.

This makefile can be used to build client and server applications from the source code in this tutorial. The stubs and headers shown here were generated with MIDL version 2.0. The compiler and linker commands and arguments may be different for your computer configuration. See your compiler documentation for more information.

```
#makefile for helloc.exe and hellos.exe
#link refers to the linker
#conflags refers to flags for console applications
#conlibs refers to libraries for console applications

!include <ntwin32.mak>

all : helloc hellos

# Make the client side application helloc
helloc : helloc.exe
helloc.exe : helloc.obj hello_c.obj
    $(link) $(linkdebug) $(conflags) -out:helloc.exe \
        helloc.obj hello_c.obj \
        rpcrt4.lib $(conlibs)

# helloc main program
helloc.obj : helloc.c hello.h
    $(cc) $(cdebug) $(cflags) $(cvars) $*.c

# helloc stub
hello_c.obj : hello_c.c hello.h
```

(continued)

(continued)

```
    $(cc) $(cdebug) $(cflags) $(cvars) $*.c

# Make the server side application
hellos : hellos.exe
hellos.exe : hellos.obj hellop.obj hello_s.obj
    $(link) $(linkdebug) $(conflags) -out:hellos.exe \
        hellos.obj hello_s.obj hellop.obj \
        rpcrt4.lib $(conlibsmt)

# hello server main program
hellos.obj : hellos.c hello.h
    $(cc) $(cdebug) $(cflags) $(cvarsmt) $*.c

# remote procedures
hellop.obj : hellop.c hello.h
    $(cc) $(cdebug) $(cflags) $(cvarsmt) $*.c

# hellos stub file
hello_s.obj : hello_s.c hello.h
    $(cc) $(cdebug) $(cflags) $(cvarsmt) $*.c

# Stubs and header file from the IDL file
hello.h hello_c.c hello_s.c : hello.idl hello.acf
    midl hello.idl
```

Running the Application

To run the application on a single computer with Microsoft® Windows NT®/Windows® 2000, open two console windows. In the first window, type:

```
C:\> hellos
```

and in the second window, type:

```
C:\> helloc
```

Because the distributed application uses named pipes as the transport protocol, the server-side application will not run on Windows 95. To experiment with different protocol sequences, endpoints, and other options, build the sample hello application from the source files in \mstools\samples\rpc\hello on the Platform SDK CD.

CHAPTER 11

The IDL and ACF Files

The syntax of the Microsoft® Interface Definition Language (MIDL) is based on the syntax of the C programming language. When a language concept in this description of MIDL is not fully defined, the C-language definition of that term is implied.

The MIDL design specifies two distinct files: the Interface Definition Language (IDL) file and the Application Configuration File (ACF). These files contain attributes that direct the generation of the C-language stub files that manage the remote procedure call (RPC). The IDL file contains a description of the interface between the client and the server programs. RPC applications use the ACF file to describe the characteristics of the interface that are specific to the hardware and operating system that make up a particular operating environment. The purpose of dividing this information into two files is to keep the software interface separate from characteristics that affect only the operating environment.

The IDL file specifies a network contract between the client and server—that is, the IDL file specifies what is transmitted between the client and the server. Keeping this information distinct from the information about the operating environment makes the IDL file portable to other environments. The IDL file consists of two parts: an interface header and an interface body.

The ACF specifies attributes that affect only local performance rather than the network contract. Microsoft RPC allows you to combine the ACF and IDL attributes in a single IDL file. You can also combine multiple interfaces in a single IDL file (and its ACF).

This section summarizes the attributes that are specified in the IDL and ACF files. It is intended to only provide an overview. For more detailed information, see the *MIDL Language Reference*, and the *MIDL Command-Line Reference*. The discussion in this section is presented in the following topics:

- The Interface Definition Language (IDL) File
- The Application Configuration File (ACF)
- MIDL Compiler Output

The Interface Definition Language (IDL) File

An IDL file contains one or more interface definitions. Each interface definition is composed of an interface header and an interface body. The interface header is demarcated by square brackets. The interface body is contained in curly brackets. This is illustrated in the example interface on the following page.

```
[
  /*Interface attributes go here. */
]
interface INTERFACENAME
{
  /*The interface body goes here. */
}
```

This section gives an overview of the components of an interface. It is organized into the following topics:

- The IDL Interface Header
- The IDL Interface Body

See Also

IDL Attributes

The IDL Interface Header

The IDL interface header specifies information about the interface as a whole. Unlike the ACF, the interface header contains attributes that are platform-independent.

Attributes in the interface header are global to the entire interface. That is, they apply to the interface and all of its parts. These attributes are enclosed in square brackets at the beginning of the interface definition. An example is shown in the following interface definition:

```
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(1.0)
]
interface INTERFACENAME
{

}
```

Notice that the interface header contains the **[uuid]** and **[version]** attributes. Since these represent the UUID and version number of the interface respectively, they are attributes of the entire interface.

The interface body can also contain attributes. However, they are not applicable to the entire interface. They refer to specific items in the interface such as remote procedure parameters.

For a complete discussion of the IDL header attributes, see the *MIDL Language Reference*.

The IDL Interface Body

The IDL interface body contains data types used in remote procedure calls and the function prototypes for the remote procedures. The interface body can also contain imports, pragmas, constant declarations, and type declarations. In Microsoft-extensions mode, the MIDL compiler also allows implicit declarations in the form of variable definitions.

The following example shows an IDL file containing the definition of an interface. The body of the interface definition, which occurs between the curly brackets, contains the definition of a constant (BUFSIZE), a type (**PCONTEXT_HANDLE_TYPE**), and some remote procedures (**RemoteOpen**, **RemoteRead**, **RemoteClose**, and **Shutdown**).

```
[
  uuid (ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(1.0),
  pointer_default(unique)
]
interface cxhndl
{

  const short BUFSIZE = 1024;

  typedef [context_handle] void *PCONTEXT_HANDLE_TYPE;

  short RemoteOpen(
      [out] PCONTEXT_HANDLE_TYPE *pphContext,
      [in, string] unsigned char *pszFile
  );

   hort RemoteRead(
      [in]  PCONTEXT_HANDLE_TYPE phContext,
      [out] unsigned char achBuf[BUFSIZE],
      [out] short *pcbBuf
  );

  short RemoteClose( [in, out] PCONTEXT_HANDLE_TYPE *pphContext );

  void Shutdown(void);

}
```

For more information see the *MIDL Language Reference*.

The Application Configuration File (ACF)

The Application Configuration File (ACF) has two parts: an interface header, similar to the interface header in the IDL file, and a body, which contains configuration attributes that apply to types and functions defined in the interface body of the IDL file.

The ACF Header

The ACF header contains platform-specific attributes that apply to the interface as a whole. Attributes applied to individual types and functions in the ACF body override the attributes in the ACF header. No attributes are required in the ACF header.

The ACF header can include one of the following attributes: **[auto_handle]**, **[implicit_handle]**, or **[explicit_handle]**. These handle attributes specify the type of handle used for implicit binding when a remote function does not have an explicit binding-handle parameter. When the ACF is not present or does not specify an automatic, implicit, or explicit binding handle, MIDL uses **[auto_handle]** for implicit binding.

Either **[code]** or **[nocode]** can appear in the interface header, but the one you choose can appear only once. When neither attribute is present, the compiler uses **[code]** as a default.

For more information, see *ACF Attributes*.

The ACF Body

The ACF body contains configuration attributes that apply to types and functions defined in the interface body of the IDL file. The body of the ACF can be empty or it can contain ACF **include**, **typedef**, function, and parameter attributes. All of these items are optional. Attributes applied to individual types and functions in the ACF body override attributes in the ACF header.

The ACF specifies behavior on the local computer and does not affect the data transmitted over the network. It is used to specify details of a stub to be generated. In DCE-compatibility mode (**/osf**), the ACF does not affect interaction between stubs, but between the stub and application code.

A parameter specified in the ACF must be one of the parameters specified in the IDL file. The order of specification of the parameter in the ACF is not significant because the matching is by name, not by position. The parameter list in the ACF can be empty, even when the parameter list in the corresponding IDL signature is not (but this is not recommended). Abstract declarators (unnamed parameters) in the IDL file cause the MIDL compiler to report errors while processing the ACF because the parameter is not found.

The ACF **include** directive specifies the header files to appear in the generated header as part of a standard C-preprocessor **#include** statement. The ACF keyword **include** differs from an **#include** directive. The ACF keyword **include** causes the line "**#include** *filename*" to appear in the generated header file, while the C-language directive "**#include** *filename*" causes the contents of that file to be placed in the ACF.

The ACF **typedef** statement lets you apply ACF type attributes to types previously defined in the IDL file. The ACF **typedef** syntax differs from the C **typedef** syntax.

The ACF function attributes let you specify attributes that apply to the function as a whole. For more information, see ***[code], [optimize],*** and ***0***

The ACF parameter attributes let you specify attributes that apply to individual parameters of the function. For more information, see ***[byte_count].***

See Also

/app_config, **/osf**, **[auto_handle]**, **[code]**, **[explicit_handle]**, The Interface Definition Language (IDL) File, **[implicit_handle]**, **include**, **midl**, **[nocode]**, **[optimize]**, **[represent_as]**, **typedef**

MIDL Compiler Output

With the IDL and ACF files as input, the MIDL compiler generates up to five C-language source files. By default, the MIDL compiler uses the base file name of the IDL file as part of the generated stub files. When more than six characters are present in the base file name, some file systems may not accept the full stub name. The following table shows conventions used for file names.

File	Default portion of base file name	Example
IDL file	---	Abcdefgh.idl
Header	.h	Abcdef.h
Client stub	_c.c	Abcdef_c.c
Server stub	_s.c	Abcdef_s.c

CHAPTER 12

Data and Language Features

The Microsoft® Interface Definition Language (MIDL) provides the set of features that extend the C programming language to support remote procedure calls. MIDL is not a variation of C; it is a strongly typed formal language through which you can control the data transmitted over a network. MIDL is designed so that developers familiar with C can learn it quickly.

Strong Typing

C is a weakly typed language, that is, the compiler allows operations such as assignment and comparison among variables of different types. For example, C allows the value of a variable to be cast to another type. The ability to use variables of different types in the same expression promotes flexibility as well as efficiency.

A strongly typed language imposes restrictions on operations among variables of different types. In those cases, the compiler issues an error prohibiting the operation. These strict guidelines regarding data types are designed to avoid potential errors.

The difficulty with using a weakly typed language such as C for remote procedure calls is that distributed applications can run on several different computers with different C compilers and different architectures. When an application runs on only one computer, you don't have to be concerned with the internal data format because the data is handled in a consistent manner. However, in a distributed computing environment, different computers can use different definitions for their base data types. For example, some computers define the **int** type, so its internal representation is 16 bits, while other computers use 32 bits. One computer architecture, known as "little endian," assigns the least significant byte of data to the lowest memory address and the most significant byte to the highest address. Another architecture, known as "big endian," assigns the least significant byte to the highest memory address associated with that data.

Remote procedure calls require strict control over parameter types. To handle data transmission and conversion over the network, MIDL strictly enforces type restrictions for data transferred over the network. For this reason, MIDL includes a set of well-defined base types. MIDL enforces strong typing by mandating the use of keywords that unambiguously define the size and type of data. The most visible effect of strong typing is that MIDL does not allow variables of the type **void ***.

In the following topics, this section discusses the MIDL language features that enforce strong data typing.

- Base Types
- Signed and Unsigned Types
- Wide-Character Types
- Structures
- Unions
- Enumerated Types
- Arrays
- Function Attributes
- Field Attributes
- Three Pointer Types
- Type Attributes

Base Types

To prevent the problems that implementation-dependent data types can cause on different computer architectures, MIDL defines its own base data types.

Base type	Description
boolean	A data item that can have the value TRUE or FALSE.
byte	An 8-bit data item guaranteed to be transmitted without any change.
char	An 8-bit unsigned character data item.
double	A 64-bit floating-point number.
float	A 32-bit floating-point number.
handle_t	A primitive handle that can be used for RPC binding or data serializing.
hyper	A 64-bit integer that can be declared as either **signed** or **unsigned** Can also be referred to as **_int64**.
int	A 32-bit integer that can be declared as either **signed** or **unsigned**.
long	A modifier for **int** that indicates a 64-bit integer. Can be declared as either **signed** or **unsigned**.
short	A 16-bit integer that can be declared as either **signed** or **unsigned**.
small	A modifier for **int** that indicates an 8-bit integer. Can be declared as either **signed** or **unsigned**.
wchar_t	Wide-character type that is supported as a Microsoft® extension to IDL. Therefore, this type is not available if you compile using the **/osf** switch.

The header file Rpcndr.h provides definitions for most of these base data types. The keyword **int** is recognized and is transmittable on 32-bit platforms. On 16-bit platforms, the **int** data type requires a modifier, such as **short** or **long**, to specify its length.

Although **void** * is recognized as a generic pointer type by the ANSI C standard, MIDL restricts its usage. Each pointer used in a remote or serializing operation must point to either base types or types constructed from base types. (There is an exception: context handles are defined as **void** * types. For more information see *Context Handles.*)

Signed and Unsigned Types

Compilers that use different defaults for signed and unsigned types can cause software errors in your distributed application. You can avoid these problems by explicitly declaring your character types as **signed** or **unsigned**.

MIDL defines the **small** type to take the same default sign as the **char** type in the target C compiler. If the compiler assumes that **char** is unsigned, **small** will also be defined as unsigned. Many C compilers let you change the default as a command-line option. For example, the Microsoft C compiler **/J** command-line option changes the default sign of **char** from signed to unsigned.

You can also control the sign of variables of type **char** and **small** with the MIDL compiler command-line switch **/char**. This switch allows you to specify the default sign used by your compiler. The MIDL compiler explicitly declares the sign of all **char** types that do not match your C-compiler default type in the generated header file.

Wide-Character Types

Microsoft RPC supports the wide-character type **wchar_t**. The wide-character type uses 2 bytes for each character. The ANSI C-language definition allows you to initialize long characters and long strings as:

```
wchar_t wcInitial = L'a';
wchar_t * pwszString = L"Hello, world";
```

Structures

Normal C semantics apply to the fields of base types. Fields of more complex types, such as pointers, arrays, and other constructed types, can be modified by type or **field_attributes**. For more information, see ***struct***.

Unions

Some features of the C language, such as *unions*, require special MIDL keywords to support their use in remote procedure calls. A union in the C language is a variable that holds objects of different types and sizes. The developer usually creates a variable to keep track of the types stored in the union. To operate correctly in a distributed environment, the variable that indicates the type of the union, or the *discriminant*, must also be available to the remote computer. MIDL provides the **[switch_type]** and **[switch_is]** keywords to identify the discriminant type and name.

MIDL requires that the discriminant be transmitted with the union in one of two ways:

- The union and the discriminant must be provided as parameters.
- The union and the discriminant must be packaged in a structure.

Two fundamental types of discriminated unions are provided by MIDL: **nonencapsulated_union** and **encapsulated_union**. The discriminant of a nonencapsulated union is another parameter if the union is a parameter. It is another field if the union is a field of a structure. The definition of an encapsulated union is turned into a structure definition whose first field is the discriminant and whose second and last fields are the union. The following example demonstrates how to provide the union and discriminant as parameters:

```
typedef [switch_type(short)] union
{
    [case(0)]    short     sVal;
    [case(1)]    float     fVal;
    [case(2)]    char      chVal;
    [default]    ;
} DISCRIM_UNION_PARAM_TYPE;

short UnionParamProc(
    [in, switch_is(sUtype)] DISCRIM_UNION_PARAM_TYPE Union,
    [in] short sUtype);
```

The union in the preceding example can contain a single value: either **short**, **float**, or **char**. The type definition for the union includes the MIDL **switch_type** attribute that specifies the type of the discriminant. Here, [switch_type(short)] specifies that the discriminant is of type **short**. The switch must be an integer type.

If the union is a member of a structure, then the discriminant must be a member of the same structure. If the union is a parameter, then the discriminant must be another parameter. The prototype for the function **UnionParamProc** in the preceding example shows the discriminant *sUtype* as the last parameter of the call. (The discriminant can appear in any position in the call.) The type of the parameter specified in the **[switch_is]** attribute must match the type specified in the **[switch_type]** attribute.

The following example demonstrates the use of a single structure that packages the discriminant with the union:

```
typedef struct
{
    short utype;  /* discriminant can precede or follow union */
    [switch_is(utype)] union
    {
        [case(0)]   short     sVal;
        [case(1)]   float     fVal;
        [case(2)]   char      chVal;
        [default]   ;
    } u;
} DISCRIM_UNION_STRUCT_TYPE;

short UnionStructProc(
    [in] DISCRIM_UNION_STRUCT_TYPE u1);
```

The Microsoft RPC MIDL compiler allows union declarations outside of **typedef** constructs. This feature is an extension to DCE IDL. For more information, see ***union***.

Enumerated Types

The **enum** declaration is not translated into **#define** statements as it is by some DCE compilers, but is reproduced as a C-language **enum** declaration in the generated header file.

Arrays

For information on arrays, see ***MIDL Arrays***.

Function Attributes

The **[callback]** and **[local]** attributes can be applied as function attributes.

A callback is a remote call from server to client that executes as part of a conceptual single-execution thread. A callback is always issued in the context of a remote call (or callback) and is executed by the thread that issued the original remote call (or callback).

It is often desirable to place a local procedure declaration in the IDL file, since this is the logical place to describe interfaces to a package. The **[local]** attribute indicates that a procedure declaration is not actually a remote function, but a local procedure. The MIDL compiler does not generate any stubs for functions with the **[local]** attribute.

Field Attributes

Field attributes are the attributes that can be applied to fields of an array, structure, union, or character array:

- **[ignore]**, **[size_is]**
- **[max_is]**
- **[length_is]**
- **[first_is]**
- **[last_is]**
- **[switch_is]**
- **[string]**
- pointer attributes

For example, field attributes are used in conjunction with array declarations to specify either the size of the array or the portion of the array that contains valid data. This is done by associating another parameter, structure field, or a constant expression with the array.

The **[ignore]** attribute designates pointer fields to be ignored during the marshaling process. Such an ignored field is set to NULL on the receiver side.

MIDL provides *conformant*, *varying*, and *open* arrays. An array is called conformant if its bounds are determined at run time. The **[size_is]** attribute designates the upper bound on the allocation size of the array and the **[max_is]** attribute designates the upper bound on the value of a valid array index. For more information, see ***[arrays]***.

An array is called varying if its bounds are determined at compile time, but the range of transmitted elements is determined at run time. An open array (also called a conformant varying array) is an array whose upper bound and range of transmitted elements are determined at run time. To determine the range of transmitted elements of an array, the array declaration must include a **[length_is]**, **[first_is]**, or **[last_is]** attribute.

The **[length_is]** attribute designates the number of array elements to be transmitted and the **[first_is]** attribute designates the index of the first array element to be transmitted. The **[last_is]** attribute designates the index of the last array element to be transmitted.

The **[switch_is]** field attribute designates a union discriminator. When the union is a procedure parameter, the union discriminator must be another parameter of the same procedure. When the union is a field of a structure, the discriminator must be another field of the structure at the same level as the union field. The discriminator must be a **Boolean**, **char**, **int**, or **enum** type, or a type that resolves to one of these types. For more information, see *Nonencapsulated Unions* and ***[switch_is]***.

The **[string]** field attribute designates that a one-dimensional character or byte array, or a pointer to a zero-terminated character or byte stream, is to be treated as a string. The string attribute applies only to one-dimensional arrays and pointers. The element type is limited to **char**, **byte**, **wchar_t**, or a named type that resolves to one of these types.

For information about the context in which field attributes appear, see *MIDL*, *MIDL Structures*, and *MIDL Unions*.

Three Pointer Types

MIDL supports three types of pointers to accommodate a wide range of applications. The three different levels are called reference, unique, and full pointers, and are indicated by the attributes **[ref]**, **[unique]**, and **[ptr]**, respectively. The pointer classes described by these attributes are mutually exclusive. Pointer attributes can be applied to pointers in type definitions, function return types, function parameters, members of structures or unions, or array elements.

Embedded pointers are pointers that are members of structures or unions. They can also be elements of arrays. In the **[in]** direction, embedded **[ref]** pointers are assumed to be pointing to valid storage and must not be null. This situation is recursively applicable to any **[ref]** pointers they are pointing to. In the **[in]** direction, embedded **[unique]** and full pointers (pointers with the **[ptr]** attribute) may or may not be null.

Any pointer attribute placed on a parameter in the syntax of a function declaration affects only the rightmost pointer declarator for that parameter. To affect other pointer declarators, intermediate named types must be used.

Functions that return a pointer can have a pointer attribute as a function attribute. The **[unique]** and **[ptr]** attributes must be applied to function return types. Member declarations that are pointers can specify a pointer attribute as a field attribute. A pointer attribute can also be applied as a type attribute in **typedef** constructs.

When no pointer attribute is specified as a field or type attribute, pointer attributes are applied according to the rules for an unattributed pointer declaration as follows.

In DCE-compatibility mode, pointer attributes are determined in the defining IDL file. If there is a **[pointer_default]**attribute specified in the defining interface, that attribute is used. If no **[pointer_default]** attribute is present, all unattributed pointers are full pointers.

In Microsoft-extensions mode, pointer attributes can be determined by importing IDL files and are applied in the following order:

1. An explicit pointer attribute applied at the use site.
2. The **[ref]** attribute, when the unattributed pointer is a top-level pointer parameter.
3. A **[pointer_default]** attribute specified in the defining interface.
4. A **[pointer_default]** attribute specified in the base interface.
5. The **[unique]** attribute.

The **[pointer_default]** interface attribute specifies the default pointer attributes to be applied to a pointer declarator in a type, parameter, or return type declaration when that declaration does not have an explicit pointer attribute applied to it. The **[pointer_default]** interface attribute does not apply to an unattributed top-level pointer of a parameter, which is assumed to be **[ref]**.

Type Attributes

Type attributes are the MIDL attributes that can be applied to type declarations:

- **[handle]**
- **[context_handle]**
- **[switch_type]**
- pointer type attributes

The **[switch_type]** attribute designates the type of a union discriminator. This attribute applies only to a nonencapsulated union.

A context handle is a pointer with a **[context_handle]** attribute. The **[context_handle]** attribute allows you to write procedures that maintain state information between remote procedure calls. A context handle with a non-null value represents saved context and serves two purposes:

- On the client side, it contains the information needed by the RPC run-time library to direct the call to the server.
- On the server side, it serves as a handle on active context.

The **[handle]** attribute specifies that a type can occur as a user-defined (generic) handle. This feature permits the design of handles that are meaningful to the application. The user must provide binding and unbinding routines to convert between the user-defined handle type and the RPC primitive handle type, **handle_t**. A primitive handle contains destination information meaningful to the RPC run-time libraries. A user-defined handle can only be defined in a type declaration, not in a function declaration. A parameter with the **[handle]** attribute has a double purpose. It is used to determine the binding for the call, and it is transmitted to the called procedure as a normal data parameter.

Directional (Parameter) Attributes

Directional attributes describe whether the data is transmitted from client to server, server to client, or both. All parameters in the function prototype must be associated with directional attributes. The three possible combinations of directional attributes are: 1) **[in]**, 2) **[out]**, and 3) **[in, out]**. These describe the way parameters are passed between calling and called procedures. When you compile in the default (Microsoft-extended mode) and you omit a directional attribute for a parameter, the MIDL compiler assumes a default value of **[in]**.

An **[out]** parameter must be a pointer. In fact, the **[out]** attribute is not meaningful when applied to parameters that do not act as pointers because C function parameters are passed by value. In C, the called function receives a private copy of the parameter value; it cannot change the calling function's value for that parameter. If the parameter acts as a pointer, however, it can be used to access and modify memory. The **[out]** attribute indicates that the server function should return the value to the client's calling function, and that memory associated with the pointer should be returned in accordance with the attributes assigned to the pointer.

The following interface demonstrates the three possible combinations of directional attributes that can be applied to a parameter. The function **InOutProc** is defined in the IDL file as:

```
void InOutProc ([in]        short     s1,
                [in, out]   short *   ps2,
                [out]       float *   pf3);
```

The first parameter, *s1*, is **[in]** only. Its value is transmitted to the remote computer, but is not returned to the calling procedure. Although the server application can change its value for *s1*, the value of *s1* on the client is the same before and after the call. (See Figure 12-1.)

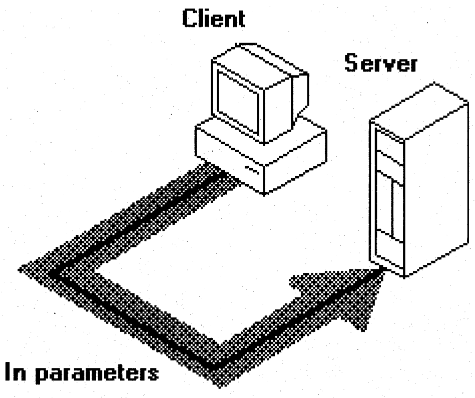


Figure 12-1: In Parameter.

The second parameter, *ps2*, is defined in the function prototype as a pointer with both **[in]** and **[out]** attributes. The **[in]** attribute indicates that the value of the parameter is passed from the client to the server. The **[out]** attribute indicates that the value pointed to by *ps2* is returned to the client.

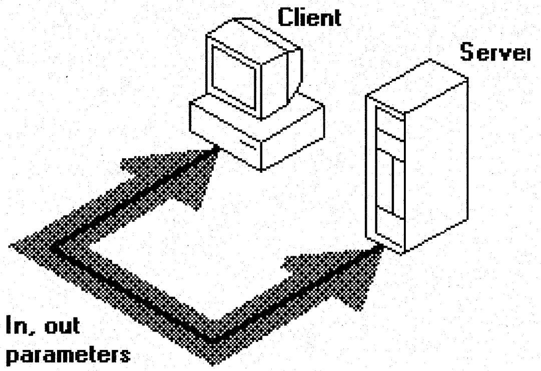


Figure 12-2: Parameter with In and Out Attributes.

The third parameter is **[out]** only. Space is allocated for the parameter on the server, but the value is undefined on entry. As mentioned above, all **[out]** parameters must be pointers.

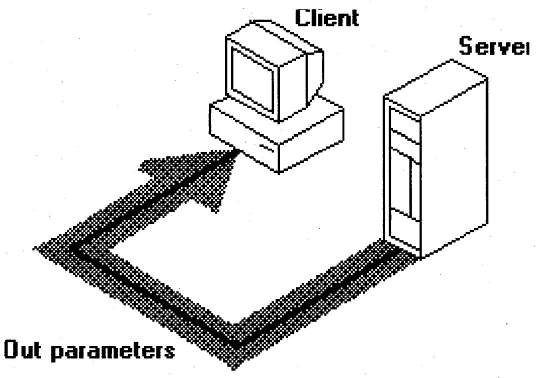


Figure 12-3: Out Parameter.

The remote procedure changes the value of all three parameters, but only the new values of the **[out]** and **[in, out]** parameters are available to the client.

```
#define MAX 257

void InOutProc(short    s1,
               short * ps2,
               float * pf3)
{
    *pf3 = (float) s1 / (float) *ps2;
    *ps2 = (short) MAX - s1;
    s1++;  // in only; not changed on the client side
    return;
}
```

On return from the call to **InOutProc**, the second and third parameters are modified. The first parameter, which is **[in]** only, is unchanged.

Data Representation

Computing environments can differ significantly, as can network architectures. To accommodate these differences, MIDL enables you to modify the way you represent data. You can sometimes simplify development by converting data into a format that your application can more easily handle. You can alter your application's data format so that it can be more efficiently transmitted over the network.

The **[transmit_as]** and **[represent_as]** attributes instruct the compiler to associate a transmissible type that the stub passes between client and server, with a user type that the client and server applications use. You must supply the routines that carry out the conversion between the user type and the transmissible type, and the routines to release the memory that was used to hold the converted data. Using the **[transmit_as]** IDL attribute or the **[represent_as]** ACF attribute instructs the stub to call these conversion routines before and after transmission. The **[transmit_as]** attribute lets you convert one data type to another data type for transmission over the network. The **[represent_as]** attribute lets you control the way data from the network is presented to the application.

The **[wire_marshal]** and **[user_marshal]** attributes are Microsoft extensions to the OSF-DCE IDL. Their syntax and functionality are similar to that of the DCE-specified **[transmit_as]** and **[represent_as]** attributes, respectively. The difference is that, instead of converting the data from one type to another, you marshal the data directly. To do this, you must supply the external routines for sizing the data buffer on the client and server sides, marshaling and unmarshaling the data on the client and server sides, and freeing the data on the server side. The MIDL compiler generates format codes that instruct the NDR engine to call these external routines when needed.

The **[wire_marshal]** and **[user_marshal]** attributes make it possible to marshal data types that otherwise could not be transmitted across process boundaries. Also, because there is less overhead associated with the type conversion, **[wire_marshal]** and **[user_marshal]** provide improved performance at run time, when compared to **[transmit_as]** and **[represent_as]**. The **[wire_marshal]** and **[user_marshal]** attributes are mutually exclusive with respect to each other and with respect to the **[transmit_as]** and **[represent_as]** attributes for a given type.

The transmit_as and represent_as Attributes

This section discusses the implementation of programmer data type conversion using the MIDL **[transmit_as]** and **[represent_as]** attributes.

The transmit_as Attribute

The **[transmit_as]** attribute offers a way to control data marshaling without worrying about marshaling data at a low level—that is, without worrying about data sizes or byte swapping in a heterogeneous environment. By letting you reduce the amount of data transmitted over the network, the **[transmit_as]** attribute can make your application more efficient.

You use the **[transmit_as]** attribute to specify a data type that the RPC stubs will transmit over the network instead of using the data type provided by the application. You supply routines that convert the data type to and from the type that is used for transmission. You must also supply routines to free the memory used for the data type and the transmitted type. For example, the following defines **xmit_type** as the data type transmitted for all application data specified as being of type **type_spec**:

```
typedef [transmit_as (xmit_type)] type_spec type;
```

The following table describes the four programmer-supplied routine names. **Type** is the data type known to the application, and **xmit_type** is the data type used for transmission.

Routine	Description
type_to_xmit	Allocates an object of the transmitted type and converts from application type to type transmitted over the network (caller and object called).
Type_from_xmit	Converts from transmitted type to application type (caller and object called).
Type_free_inst	Frees resources used by the application type (object called only).
Type_free_xmit	Frees storage returned by the **type_to_xmit** routine (caller and object called).

Other than by these four programmer-supplied functions, the transmitted type is not manipulated by the application. The transmitted type is defined only to move data over the network. After the data is converted to the type used by the application, the memory used by the transmitted type is freed.

These programmer-supplied routines are provided by either the client or the server application based on the directional attributes. If the parameter is **[in]** only, the client transmits to the server. The client needs the **type_to_xmit** and **type_free_xmit** functions. The server needs the **type_from_xmit** and **type_free_inst** functions. For an **[out]**-only parameter, the server transmits to the client. The server application must implement the **type_to_xmit** and **type_free_xmit** functions, while the client program must supply the **type_from_xmit** function. For the temporary **xmit_type** objects, the stub will call **type_free_xmit** to free any memory allocated by a call to **type_to_xmit**.

Certain guidelines apply to the application type instance. If the application type is a pointer or contains a pointer, then the **type_from_xmit** routine must allocate memory for the data that the pointers point to (the application type object itself is manipulated by the stub in the usual way).

For **[out]** and **[in, out]** parameters, or one of their components, of a type that contains the **[transmit_as]** attribute, the **type_free_inst** routine is automatically called for the data objects that have the attribute. For **in** parameters, the **type_free_inst** routine is called only if the **[transmit_as]** attribute has been applied to the parameter. If the attribute has been applied to the components of the parameter, the **type_free_inst** routine is not called. There are no freeing calls for the embedded data and at-most-one call (related to the top-level attribute) for an **in** only parameter.

Effective with MIDL 2.0, both client and server must supply all four functions. For example, a linked list can be transmitted as a sized array. The **type_to_xmit** routine walks the linked list and copies the ordered data into an array. The array elements are ordered so that the many pointers associated with the list structure do not have to be transmitted. The **type_from_xmit** routine reads the array and puts its elements into a linked-list structure.

The double-linked list (DOUBLE_LINK_LIST) includes data and pointers to the previous and next list elements:

```
typedef struct _DOUBLE_LINK_LIST
{
    short sNumber;
    struct _DOUBLE_LINK_LIST * pNext;
    struct _DOUBLE_LINK_LIST * pPrevious;
} DOUBLE_LINK_LIST;
```

Rather than shipping the complex structure, the **[transmit_as]** attribute can be used to send it over the network as an array. The sequence of items in the array retains the ordering of the elements in the list at a lower cost:

```
typedef struct _DOUBLE_XMIT_TYPE
{
    short sSize;
    [size_is(sSize)] short asNumber[];
} DOUBLE_XMIT_TYPE;
```

The **[transmit_as]** attribute appears in the IDL file:

```
typedef [transmit_as(DOUBLE_XMIT_TYPE)]  DOUBLE_LINK_LIST DOUBLE_LINK_TYPE;
```

In the following example, **ModifyListProc** defines the parameter of type DOUBLE_LINK_TYPE as an **[in, out]** parameter:

```
void ModifyListProc([in, out] DOUBLE_LINK_TYPE * pHead)
```

The four programmer-defined functions use the name of the type in the function names, and use the presented and transmitted types as parameter types, as required:

```
void __RPC_USER DOUBLE_LINK_TYPE_to_xmit (
    DOUBLE_LINK_TYPE __RPC_FAR * pList,
    DOUBLE_XMIT_TYPE __RPC_FAR * __RPC_FAR * ppArray);

void __RPC_USER DOUBLE_LINK_TYPE_from_xmit (
    DOUBLE_XMIT_TYPE __RPC_FAR * pArray,
    DOUBLE_LINK_TYPE __RPC_FAR * pList);

void __RPC_USER DOUBLE_LINK_TYPE_free_inst (
    DOUBLE_LINK_TYPE __RPC_FAR * pList);

void __RPC_USER DOUBLE_LINK_TYPE_free_xmit (
    DOUBLE_XMIT_TYPE __RPC_FAR * pArray);
```

The type_to_xmit Function

The stubs call the **type_to_xmit** function to convert the type that is presented by the application into the transmitted type. The function is defined as:

```
void __RPC_USER <type>_to_xmit (
     <type> __RPC_FAR *, <xmit_type> __RPC_FAR *     __RPC_FAR *);
```

The first parameter is a pointer to the application data. The second parameter is set by the function to point to the transmitted data. The function must allocate memory for the transmitted type.

In the following example, the client calls the remote procedure that has an **[in, out]** parameter of type DOUBLE_LINK_TYPE. The client stub calls the **type_to_xmit** function, here named **DOUBLE_LINK_TYPE_to_xmit**, to convert double-linked list data into a sized array.

The function determines the number of elements in the list, allocates an array large enough to hold those elements, then copies the list elements into the array. Before the function returns, the second parameter, *ppArray*, is set to point to the newly allocated data structure.

```
void __RPC_USER DOUBLE_LINK_TYPE_to_xmit (
    DOUBLE_LINK_TYPE __RPC_FAR * pList,
    DOUBLE_XMIT_TYPE __RPC_FAR * __RPC_FAR * ppArray)
{
    short cCount = 0;
    DOUBLE_LINK_TYPE * pHead = pList;  // save pointer to start
    DOUBLE_XMIT_TYPE * pArray;

    /* count the number of elements to allocate memory */
    for (; pList != NULL; pList = pList->pNext)
        cCount++;

    /* allocate the memory for the array */
    pArray = (DOUBLE_XMIT_TYPE *) MIDL_user_allocate
         (sizeof(DOUBLE_XMIT_TYPE) + (cCount * sizeof(short)));
    pArray->sSize = cCount;

    /* copy the linked list contents into the array */
    cCount = 0;
    for (i = 0, pList = pHead; pList != NULL; pList = pList->pNext)
        pArray->asNumber[cCount++] = pList->sNumber;

    /* return the address of the pointer to the array */
    *ppArray = pArray;
}
```

The type_from_xmit Function

The stubs call the **type_from_xmit** function to convert data from its transmitted type to the type that is presented to the application. The function is defined as:

```
void __RPC_USER <type>_from_xmit (
    <xmit_type> __RPC_FAR *,
    <type> __RPC_FAR *);
```

The first parameter is a pointer to the transmitted data. The function sets the second parameter to point to the presented data.

The **type_from_xmit** function must manage memory for the presented type. The function must allocate memory for the entire data structure that starts at the address indicated by the second parameter, except for the parameter itself (the stub allocates memory for the root node and passes it to the function). The value of the second parameter cannot change during the call. The function can change the contents at that address.

In this example, the function **DOUBLE_LINK_TYPE_from_xmit** converts the sized array to a double-linked list. The function retains the valid pointer to the beginning of the list, frees memory associated with the rest of the list, then creates a new list that starts at the same pointer. The function uses a utility function, **InsertNewNode**, to append a list node to the end of the list and to assign the *pNext* and *pPrevious* pointers to appropriate values.

```
void __RPC_USER DOUBLE_LINK_TYPE_from_xmit(
     DOUBLE_XMIT_TYPE __RPC_FAR * pArray,
     DOUBLE_LINK_TYPE __RPC_FAR * pList)
{
    DOUBLE_LINK_TYPE *pCurrent;
    int i;

    if (pArray->sSize <= 0)
    {
        // error checking
        return;
    }

    if (pList == NULL) // if invalid, create the list head
        pList = InsertNewNode(pArray->asNumber[0], NULL);
    else
    {
        DOUBLE_LINK_TYPE_free_inst(pList);  // free all other nodes
        pList->sNumber = pArray->asNumber[0];
        pList->pNext = NULL;
    }

    pCurrent = pList;
    for (i = 1; i < pArray->sSize; i++)
        pCurrent = InsertNewNode(pArray->asNumber[i], pCurrent);

    return;
}
```

The type_free_xmit Function

The stubs call the **type_free_xmit** function to free memory associated with the transmitted data. After the **type_from_xmit** function converts the transmitted data to its presented type, the memory is no longer needed. The function is defined as:

```
void __RPC_USER <type>_free_xmit(<xmit_type> __RPC_FAR *);
```

The parameter is a pointer to the memory that contains the transmitted type.

In this example, the memory contains an array that is in a single structure. The function **DOUBLE_LINK_TYPE_free_xmit** uses the user-supplied function **midl_user_free** to free the memory:

```
void __RPC_USER DOUBLE_LINK_TYPE_free_xmit(
     DOUBLE_XMIT_TYPE __RPC_FAR * pArray)
{
     midl_user_free(pArray);
}
```

The type_free_inst Function

The stubs call the **type_free_inst** function to free memory associated with the presented type. The function is defined as:

```
void __RPC_USER <type>_free_inst(<type> __RPC_FAR *)
```

The parameter points to the presented type instance. This object should not be freed. For a discussion on when to call the function, see *The transmit_as Attribute*.

In the following example, the double-linked list is freed by walking the list to its end, then backing up and freeing each element of the list.

```
void __RPC_USER DOUBLE_LINK_TYPE_free_inst(
     DOUBLE_LINK_TYPE __RPC_FAR * pList)
{
    while (pList->pNext != NULL)  // go to end of the list
        pList = pList->pNext;

    pList = pList->pPrevious;
    while (pList != NULL)
    {
        // back through the list
        midl_user_free(pList->pNext);
        pList = pList->pPrevious;
    }
}
```

The represent_as Attribute

The **[represent_as]** attribute lets you specify how a particular transmittable data type is represented to the application. This is done by specifying the name of the represented type for a known transmittable type and supplying the conversion routines. You must also supply the routines to free the memory used by the data type objects.

Use the **[represent_as]** attribute to present an application with a different, possibly untransmittable, data type rather than the type that is actually transmitted between the client and server. It is also possible that the type the application manipulates can be unknown at the time of MIDL compilation. When you choose a well-defined transmittable type, you need not be concerned about data representation in the heterogeneous environment. The **[represent_as]** attribute can make your application more efficient by reducing the amount of data transmitted over the network.

The **[represent_as]** attribute is similar to the **[transmit_as]** attribute. However, while **[transmit_as]** lets you specify a data type that will be used for transmission, **[represent_as]** lets you specify how a data type is represented for the application. The represented type need not be defined in the MIDL processed files; it can be defined at the time the stubs are compiled with the C compiler. To do this, use the include directive in the application configuration file (ACF) to compile the appropriate header file. For example, the following ACF defines a type local to the application, **repr_type**, for the transmittable type **named_type:**

```
typedef [represent_as(repr_type) [, type_attribute_list] named_type;
```

The following table describes the four programmer-supplied routines.

Routine	Description
named_type_from_local	Allocates an instance of the network type and converts from the local type to the network type.
named_type_to_local	Converts from the network type to the local type.
named_type_free_local	Frees memory allocated by a call to the **named_type_to_local** routine, but not the type itself.
named_type_free_inst	Frees storage for the network type (both sides).

Other than by these four programmer-supplied routines, the named type is not manipulated by the application. The only type visible to the application is the represented type. The application uses the represented type name instead of the transmitted type name in the prototypes and stubs generated by the compiler. You must supply the set of routines for both sides.

For temporary **named_type** objects, the stub will call **named_type_free_inst** to free any memory allocated by a call to **named_type_from_local**.

If the represented type is a pointer or contains a pointer, the **named_type_to_local** routine must allocate memory for the data to which the pointers point (the represented type object itself is manipulated by the stub in the usual way). For **[out]** and **[in, out]** parameters of a type that contain **[represent_as** or one of its components, the **named_type_free_local** routine is automatically called for the data objects that contain the attribute. For **[in]** parameters, the **named_type_free_local** routine is called only if the **[represent_as]** attribute has been applied to the parameter. If the attribute has been applied to the components of the parameter, the ***_free_local** routine is not called. Freeing routines are not called for the embedded data and at-most-once call (related to the top-level attribute) for an **[in]** only parameter.

Note It is possible to apply both the **[transmit_as]** and **[represent_as]** attributes to the same type. When marshaling data, the **[represent_as]** type conversion is applied first and then the **[transmit_as]** conversion is applied. The order is reversed when unmarshaling data. Thus, when marshaling, ***_from_local** allocates an instance of a named type and translates it from a local type object to the temporary named type object. This object is the presented type object used for the ***_to_xmit** routine. The ***_to_xmit** routine then allocates a transmitted type object and translates it from the presented (named) object to the transmitted object.

An array of long integers can be used to represent a linked list. In this way, the application manipulates the list, and the transmission uses an array of long integers when a list of this type is transmitted. You can begin with an array, but using a construct with an open array of long integers is more convenient. The following example shows how to do this.

```
/* IDL definitions */

typedef struct_lbox
{
    long        data;
    struct_lbox *       pNext
} LOC_BOX, * PLOC_BOX;

/* The definition of the local type visible to the application,
as shown above, can be omitted in the IDL file. See the include
in the ACF file. */

typedef struct_xmit_lbox
{
    short       Size;
    [size_is(Size)] long DaraArr[];
} LONGARR;
```

```
void WireTheList( [in,out] LONGARR * pData );

/* ACF definitions */

/* If the IDL file does not have a definition for PLOC_BOX, you
can still ready it for C compilation with the following include
statement (notice that this is not a C include):
include "local.h";*/

typedef [represent_as(PLOC_BOX)] LONGARR;
```

Note that the prototypes of the routines that use the **LONGARR** type are actually displayed in the Stub.h files as **PLOC_BOX** in place of the **LONGARR** type. The same is true of the appropriate stubs in the Stub_c.c file.

You must supply the following four functions:

```
void __RPC_USER
LONGARR_from_local(
    PLOC_BOX __RPC_FAR * pList,
    LONGARR __RPC_FAR * _RPC_FAR * ppDataArr );

void __RPC_USER
LONGARR_to_local(
    LONGARR __RPC_FAR * _RPC_FAR * ppDataArr,
    PLOC_BOX __RPC_FAR * pList );

void __RPC_USER
LONGARR_free_inst(
    LONGARR __RPC_FAR * pDataArr);

void __RPC_USER
LONGARR_free_local(
    PLOC_BOX __RPC_FAR * pList );
```

The routines shown above do the following:

- The **LONGARR_from_local** routine counts the nodes of the list, allocates a LONGARR object with the size **sizeof**(**LONGARR**) + Count***sizeof**(**long**), sets the *Size* field to Count, and copies the data to the *DataArr* field.
- The **LONGARR_to_local** routine creates a list with Size nodes and transfers the array to the appropriate nodes.
- The **LONGARR_free_inst** routine frees nothing in this case.
- The **LONGARR_free_local** routine frees all the nodes of the list.

The named_type_from_local Function

The stubs call the **named_type_from_local** function. It converts the type that the application uses into the type the stubs transmit across the network. The function is defined as:

```
void __RPC_USER <named_type>_from_local (
    <local_type> __RPC_FAR *, <named_type> __RPC_FAR * __RPC_FAR *);
```

The first parameter is a pointer to the application data. The second parameter is a pointer to a pointer. The function points it to the transmitted data. The function must allocate memory for the transmitted type.

The named_type_to_local Function

The stubs call the **named_type_to_local** function to convert data from a transmitted type to the type that they present to the application. The function is defined as:

```
void __RPC_USER <named_type>_to_local(
    <named_type> __RPC_FAR * _RPC_FAR * ,
    <local_type> __RPC_FAR * );
```

The first parameter points to the transmitted data. The function sets the second parameter to point to the presented data.

The **named_type_to_local** function must manage memory for the presented type. The function must allocate memory for the entire data structure that starts at the address indicated by the second parameter, except for the parameter itself (the stub allocates memory for the root node and passes it to the function). The value of the second parameter cannot change during the call. The function can change the contents at that address.

The named_type_free_local Function

The stubs call the **type_free_local** function to free the memory allocated by a call to the **named_type_to_local** routine. It does not free the memory allocated by the stub. The function prototype is defined as:

```
void __RPC_USER <local_type>_free_local(<named_type> __RPC_FAR *);
```

The parameter is a pointer to the memory allocated by **named_type_to_local**.

The named_type_free_inst Function

The stubs call the *named_type*_**free_inst** function to free memory associated with the transmitted type. The function is defined as:

```
void __RPC_USER <named_type>_free_inst(<type> __RPC_FAR *)
```

The parameter points to the instance of the transmitted type. This object should not be freed. For a discussion on when to call the function, see *The represent_as Attribute*.

The wire_marshal and user_marshal Attributes

This section discusses the implementation of programmer data type conversion using the MIDL **[wire_marshal]** and **[user_marshal]** attributes.

The wire_marshal Attribute

The **[wire_marshal]** attribute is an IDL-type attribute similar in syntax to **[transmit_as]**, but providing a more efficient way to marshal data across a network.

You use the **[wire_marshal]** attribute to specify a data type that will be transmitted in place of the application-specific data type. Each application-specific type has a corresponding transmittable type that defines the wire representation (the representation used on the network).The application-specific type need not be transmittable, but it must be a type that MIDL recognizes. To marshal a type unknown to MIDL, use the ACF attribute **[user_marshal]**.

Your application-specific type can be a simple, composite, or pointer type. The main restriction is that the type instance must have a fixed, well-defined memory size. If the size of your type instance needs to change, use a pointer field rather than a conformant array. Alternatively, you can define a pointer to the changeable type.

You must supply the routines for sizing, marshaling, and unmarshaling the data as well as freeing the associated memory. The following table describes the four user-supplied routine names. The <type> is the **userm-type** specified in the **[wire_marshal]** type definition.

Routine	Description
<type>_**UserSize**	Sizes the RPC data buffer before marshaling on the client or server side.
<type>_**UserMarshal**	Marshals the data on the client or server side.
<type>_**UserUnmarshal**	Unmarshals the data on the client or server side.
<type>_**UserFree**	Frees the data on the server side.

These programmer-supplied routines are provided by either the client or the server application based on the directional attributes.

If the parameter is **[in]** only, the client transmits to the server. The client needs the <type>_**UserSize** and <type>_**UserMarshal** functions. The server needs the <type>_**UserUnmarshal**, and <type>_**UserFree** functions.

For an **[out]**-only parameter, the server transmits to the client. The server needs the <type>_**UserSize** and <type>_**UserMarshal** functions, while the client needs the <type>_**UserMarshal** function.

See Also

The user_marshal Attribute, Marshaling Rules for user_marshal and wire_marshal, wire_marshal, user_marshal, NdrGetUserMarshalInfo

The user_marshal Attribute

The **[user_marshal]** attribute is an ACF-type attribute similar in syntax to **[represent_as]**. As with the IDL attribute, **[wire_marshal]**, it offers a more efficient way to marshal data across a network. As an ACF attribute, **[user_marshal]** lets you marshal custom data types that are unknown to MIDL. Each application-specific type has a corresponding transmittable type that defines the wire representation.

Your application-specific type can be a simple, composite, or pointer type. The main restriction is that the type instance must have a fixed, well-defined memory size. If the size of your type instance needs to change, use a pointer field rather than a conformant array. Alternatively, you can define a pointer to the changeable type.

As with the **[wire_marshal]** attribute, you supply routines for the sizing, marshaling, unmarshaling, and freeing passes. The following table describes the four user-supplied routine names. The <type> is the userm-type specified in the **[user_marshal]** type definition.

Routine	Description
<type>_**UserSize**	Sizes the RPC data buffer before marshaling on the client or server side.
<type>_**UserMarshal**	Marshals the data on the client or server side.
<type>_**UserUnmarshal**	Unmarshals the data on the client or server side.
<type>_**UserFree**	Frees the data on the server side.

These user-supplied routines are provided by either the client or the server application, based on the directional attributes.

If the parameter is **[in]** only, the client transmits to the server. The client needs the <type>_**UserSize** and <type>_**UserMarshal** functions. The server needs the <type>_**UserUnmarshal** and <type>_**UserFree** functions.

For an **[out]**-only parameter, the server transmits to the client. The server needs the <type>_**UserSize** and <type>_**UserMarshal** functions, while the client needs the <type>_**UserMarshal** function.

See Also

The wire_marshal Attribute, Marshaling Rules for user marshal and wire_marshal, [user_marshal], [wire_marshal], NdrGetUserMarshalInfo

The type_UserSize Function

The <type>_**UserSize** function is a helper function for the **[wire_marshal]** and **[user_marshal]** attributes. The stubs call this function to size the RPC data buffer for the user data object before the data is marshaled on the client or server side. The function is defined as:

```
unsigned long __RPC_USER  <type>_UserSize(
    unsigned long __RPC_FAR * pFlags,
    unsigned long StartingSize,
    <type>  __RPC_FAR *pMyObj);
```

The <type> in the function name means the userm-type, as specified in the **[wire_marshal]** or **[user_marshal]** type definition. This type may be untransmittable or even—when used with the **[user_marshal]** attribute— unknown to the MIDL compiler. The wire type name (the name of the type transmitted across the network) is not used in the function prototype. Note, however, that the wire type defines the layout for the data as specified by OSF DCE. All data must be converted to Network Data Representation (NDR) format.

The *pFlags* argument is a pointer to an **unsigned long** flag field. The upper word of the flag contains NDR format flags as defined by OSF DCE for floating point, byte order, and character representations. The lower word contains a marshaling context flag as defined by the COM channel. The exact layout of the flags within the field is shown in the following table.

Bits	Flag	Value
31-24	Floating-point representation	0 = IEEE 1 = VAX 2 = Cray 3 = IBM
23-20	Integer and floating-point byte order	0 = Big-endian 1 = Little-endian
19-16	Character representation	0 = ASCII 1 = EBCDIC
15-0	Marshaling context flag	0 = MSHCTX_LOCAL 1 = MSHCTX_NOSHAREDMEM 2 = MSHCTX_DIFFERENTMSCHINE 3 = MSHCTX_INPROC

The marshaling context flag makes it possible to alter the behavior of your routine depending on the context for the RPC call. For example, if you have a handle (**long**) to a block of data, you could send the handle for an in-process call, but you would send the actual data for a call to a different machine. The marshaling context flag and its values are defined in the Wtypes.h and Wtypes.idl files in the Platform SDK.

Note When the wire type is properly defined, you do not have to use the NDR format flags, as the NDR engine performs the necessary conversions.

The *StartingSize* argument is the current buffer offset. The starting size indicates the buffer offset for the user object, and it may or may not be aligned properly. Your routine should account for whatever padding is necessary.

The *pMyObj* argument is a pointer to a user type object.

The return value is the new offset or buffer position. The function should return the cumulative size, which is the starting size plus possible padding plus the data size.

The <type>_**UserSize** function can return an overestimate of the size needed. The actual size of the sent buffer is defined by the data size, not by the buffer allocation size.

The <type>_**UserSize** function is not called if the wire size can be computed at compile time. Note that for most unions, even if there are no pointers, the actual size of the wire representation can be determined only at run time.

See Also

Marshaling rules for user_marshal and wire_marshal, **[user_marshal]**, **[wire_marshal]**

The type_UserMarshal Function

The <*type*>_**UserMarshal** function is a helper function for the **[wire_marshal]** and **[user_marshal]** attributes. The stubs call this function to marshal data on the client or server side. The function is defined as:

```
unsigned char __RPC_FAR * __RPC_USER  <type>_UserMarshal(
    unsigned long __RPC_FAR * pFlags,
    unsigned char __RPC_FAR * pBuffer,
    <type>  __RPC_FAR *       pMyObj);
```

The <type> in the function name means the userm-type specified in the **[wire_marshal]** or **[user_marshal]** type definition. This type may be untransmittable or even—when used with the **[user_marshal]** attribute—a type unknown to the MIDL compiler. The wire type name (the name of transmissible type) is not used in the function prototype. Note, however, that the wire type defines the wire layout for the data as specified by OSF DCE.

The *pFlags* argument is a pointer to an unsigned long flag field. The upper word of the flag contains NDR data representation flags as defined by OSF DCE for floating point, byte order, and character representations. The lower word contains a marshaling context flag as defined by the COM channel. The exact layout of the flags within the field is described in The type_UserSize Function.

The *pBuffer* argument is the current buffer pointer. This pointer may or may not be aligned on entry. Your <type>_**UserMarshal** function should align the buffer pointer appropriately, marshal the data, and return the new buffer position, which is the address of the first byte after the marshaled object. Keep in mind that the wire type specification determines the actual layout of the data in the buffer.

The *pMyObj* argument is a pointer to a user type object.

The return value is the new buffer position, which is the address of the first byte after the unmarshaled object.

Buffer overflow can occur when you incorrectly calculate the size of the data and attempt to marshal more data than expected. You should be careful to avoid this situation. You can check against it by using the pointer that <type>_**UserMarshal** returns. Otherwise, you risk having the NDR engine raise a buffer-overflow exception later.

See Also

Marshaling Rules for user_marshal and wire_marshal, **[wire_marshal]**, **[user_marshal]**

The type_UserUnmarshal Function

The <type>_**UserUnmarshal** function is a helper function for the **[wire_marshal]** and **[user_marshal]** attributes. The stubs call this function to unmarshal data on the client or server side. The function is defined as:

```
unsigned char __RPC_FAR * __RPC_USER  <type>_UserUnmarshal(
    unsigned long __RPC_FAR * pFlags,
    unsigned char __RPC_FAR * pBuffer,
    <type>  __RPC_FAR *       pMyObj);
```

The <type> in the function name means the userm-type specified in the **[wire_marshal]** or **[user_marshal]** type definition. This type may be untransmittable or even—when used with the **[user_marshal]** attribute—unknown to the MIDL compiler. The wire type name (the name of transmissible type) is not used in the function prototype. Note, however, that the wire type defines the wire layout for the data as specified by OSF DCE.

The *pFlags* argument is a pointer to an **unsigned long** flag field. The upper word of the flag contains NDR data representation flags as defined by OSF DCE for floating point, byte order, and character representations. The lower word contains a marshaling context flag as defined by the COM channel. The exact layout of the flags within the field is described in The type_UserSize Function.

The *pBuffer* argument is the current buffer pointer. This pointer may or may not be aligned on entry. Your <type>_**UserUnmarshal** function should align the buffer pointer appropriately, unmarshal the data, and return the new buffer position, which is the address of the first byte after the unmarshaled object.

The *pMyObj* argument is a pointer to a user-defined type object.

In a heterogeneous environment, the NDR engine performs any data conversion necessary before calling the <type>_**UserUnmarshal** function. Note that the NDR engine carries out this data conversion according to the wire-type definition supplied for this user data type. The flag indicates the data representation of the sender.

See Also

Marshaling Rules for user_marshal and wire_marshal, **[wire_marshal]**, **[user_marshal]**

The type_UserFree Function

The <type>_**UserFree** function is a helper function for the **[wire_marshal]** and **[user_marshal]** attributes. The stubs call this function to free the data on the server side. The function is defined as:

```
void __RPC_USER  <type>_UserFree(
    unsigned long __RPC_FAR * pFlags,
    <type_name>  __RPC_FAR *  pMyObj );
```

The <type> in the function name means the userm-type specified in the **[wire_marshal]** or **[user_marshal]** type definition.

The *pFlags* argument is a pointer to an **unsigned long** flag field. The upper word of the flag contains NDR data representation flags as defined by OSF DCE for floating point, byte order, and character representations. The lower word contains a marshaling context flag as defined by the COM channel. The exact layout of the flags within the field is described in The **type_UserSize** Function.

The *pMyObj* argument is a pointer to a user type object. The NDR engine frees the top-level object. You are responsible for freeing any objects to which the top-level object may point.

See Also

Marshaling Rules for user_marshal and wire_marshal, **[wire_marshal]**, **[user_marshal]**

Marshaling Rules for user_marshal and wire_marshal

The OSF-DCE specification for marshaling embedded pointer types requires that you observe the following restrictions when implementing the <type>_**UserSize**, <type>_**UserMarshal**, and <type>_**UserUnMarshal** functions. (The rules and examples given here are for marshaling. However, your sizing and unmarshaling routines must follow the same restrictions):

- If the wire-type is a flat type with no pointers, your marshaling routine for the corresponding userm-type should simply marshal the data according to the layout of the wire-type. For example:

  ```
  typedef [wire_marshal (long)] void * HANDLE_HANDLE
  ```

  Note that the wire type, **long**, is a flat type. Your HANDLE_HANDLE_UserMarshal function marshals a **long** whenever a HANDLE_HANDLE object is passed to it.
- If the wire-type is a pointer to another type, your marshaling routine for the corresponding userm-type should marshal the data according to the layout for the type that the wire-type points to. The NDR engine takes care of the pointer. For example:

  ```
  typedef struct HDATA
  {
      long size;
      [size_is(size)] long * pData;
  } HDATA;

  typedef HDATA * WIRE_TYPE;
  typedef [wire_marshal(WIRE_TYPE)] void * HANDLE_DATA
  ```

  Note that the wire type, **WIRE_TYPE**, is a pointer type. Your HANDLE_DATA_UserMarshal function marshals the data related to the handle, using the HDATA layout, rather than the HDATA * layout.
- A wire-type must be either a flat data type or a pointer type. If your transmissible type must be something else (a structure with pointers, for example), use a pointer to your desired type as the wire-type.

The effect of these restrictions is that the types defined with the **[wire_marshal]** or **[user_marshal]** attributes can be freely embedded in other types.

See Also

[wire_marshal], **[user_marshal]**, **The type_UserSize Function**, **The type_UserMarshal Function**, **The type_UserUnMarshal Function**, **The type_UserFree Function**

CHAPTER 13

Arrays and Pointers

Remote Procedure Call (RPC) is designed to be mostly transparent to developers. To achieve this transparency, the client stub transmits to the server both the pointer and the data object to which it points. If the remote procedure changes the data, the server must transmit the new data back to the client so that the client can copy the new data over the original data.

In general, a remote procedure call behaves just like a local procedure call. That is, when a pointer is a parameter, the remote procedure can access the data object the pointer refers to in the same way that a local procedure can.

Since client and server programs run in different address spaces, developers must use Microsoft Interface Definition Language (MIDL) attributes to describe how array and pointer data is transmitted between the client and the server. This chapter presents an overview of how to use arrays and pointers in distributed applications.

Arrays and RPC

The C and C++ programming languages provide essentially two types of arrays: single-dimensional and multidimensional. RPC enables developers to specify additional array types using MIDL attributes to describe the characteristics of arrays in distributed applications.

This section describes the types of arrays available under RPC. It also discusses the MIDL attributes that developers can use to describe single-dimensional and multidimensional arrays.

Kinds of Arrays

MIDL provides the ability to specify the following types of arrays in your RPC application:

- Fixed Arrays
- Varying Arrays
- Conformant Arrays

All three array types can be used as **[in]**, **[out]**, or **[in,out]** parameters.

Fixed Arrays

If your interface specifies an array with a specific number of elements as a parameter, it is using a fixed array. When using MIDL, you define fixed arrays in the same way you define them in C. You specify the array's type, name, and size.

The following example demonstrates how to define a fixed array.

```
[
    /*Attributes are defined here. */
]
interface MyInterface
{
    const long ARRAY_SIZE = 1000;

    MyRemoteProc(char achArray[ARRAY_SIZE]);

    /* Other interface procedures are defined here. */
}
```

When a client program passes a fixed array to a server program, the client stub sends the entire array to the server stub. The server stub allocates memory for the array and stores the array data it receives across the network into the allocated memory. It then passes the array to the remote procedure on the server. The server may modify the data in the array.

When the remote procedure terminates, the server stub sends the contents of the array back to the client. The client stub copies the data it received from the server stub into the original array. The client program can then use the data as it would if it received the data from a local procedure call.

Varying Arrays

In MIDL, varying arrays are fixed in size. They allow clients to pass different portions of arrays from clients to servers. The size of the array portion can vary from invocation to invocation. However, the size of the overall array is fixed.

For instance, the following example shows the definition of a remote procedure in an interface in a MIDL file. The size of the array that the client passes to the server is fixed by the constant ARRAY_SIZE. The interface specifies the portion of the array that the client passes to the server in the parameters *firstElement* and *chunkSize*.

```
[
    /*Attributes are defined here. */
]
interface MyInterface
{
    const long ARRAY_SIZE = 1000;

    MyRemoteProc(
        [in] long lFirstElement,
        [in] long lChunkSize,
        [in, first_is(lFirstElement),
          length_is(lChunkSize)] char achArray[ARRAY_SIZE]
    );

    /* Other interface procedures are defined here. */
}
```

The interface definition uses the MIDL attribute **[first_is]** to specify the index number of the first element in the portion of the array that the client passes to the server. The **[length_is]** attribute specifies the total number of array elements that the client passes. For more information on these MIDL attributes, see *Array Attributes*.

The following code fragment illustrates how a client might invoke the remote procedure defined in the preceding MIDL file.

```
long lFirstArrayElementNumber = 20;
long lTotalElementsPassed = 100;
char achCharArray[ARRAY_SIZE];

// Code to store chars in the array goes here.

MyRemoteProc(
    lFirstArrayElementNumber ,
    lTotalElementsPassed ,
    achCharArray);

firstArrayElementNumber = 120;
totalElementsPassed = 200;

MyRemoteProc(
    lFirstArrayElementNumber ,
    lTotalElementsPassed ,
    achCharArray);
```

This fragment calls the remote procedure MyRemoteProc twice. On the first invocation it passes the array elements numbered 20 through 119, as indicated by the values in the variables firstArrayElementNumber and totalElementsPassed. On the second call, the client passes the array elements numbered 120 through 319.

Conformant Arrays

The size of a conformant array can vary or conform each time the client passes it to a remote procedure on the server. The interface definition in the application's MIDL file enables the client to specify the size of the array each time it invokes the remote procedure. Use empty square brackets ([ ]) or an asterisk in the square brackets ([*]) in the array definition to indicate a conformant array.

The following sample contains the definition of a remote procedure in an interface in a MIDL file. The client specifies the size of the array that it passes to the server by the parameter *arraySize*.

```
[
    /*Attributes are defined here. */
]
interface MyInterface
{
    MyRemoteProc(
         long lArraySize,
         [size_is(lArraySize)] char achArray[*]
    );

    /* Other interface procedures are defined here. */
}
```

The interface definition uses the MIDL attribute **[size_is]** to specify the size of the array that the client passes to the server. If you would rather indicate the maximum value of the array's index numbers, use the **[max_is]** attribute instead. For more information on these MIDL attributes, see *Array Attributes*.

The following code fragment illustrates how a client might invoke the remote procedure defined in the preceding MIDL file.

```
long lArrayLength = 20;
char achCharArray[20], achAnotherCharArray[200];

// Code to store 20 chars in achCharArray goes here.

MyRemoteProc(
    lArrayLength ,
    achCharArray);

lArrayLength = 200;

// Code to store 200 chars in achAnotherCharArray goes here.

MyRemoteProc(
    lArrayLength ,
    achAnotherCharArray);
```

This fragment calls the remote procedure MyRemoteProc twice. On the first invocation it passes an array of 20 elements. On the second call, the client passes an array of 200 elements.

Array Attributes

There is a close relationship between arrays and pointers in the C language. When passed as a parameter to a function, an array name is treated as a pointer to the first element of the array, as shown in the following example:

```
/* fragment */
extern void f1(char * p1);

void main(void)
{
    char chArray[MAXSIZE];

    fLocal1(chArray);
}
```

In a local call, you can use the pointer parameter to march through memory and examine the contents of other addresses:

```
/* dump memory (fragment) */
void fLocal1(char * pch1)
{
    int i;

    for (i = 0; i < MAXSIZE; i++)
       printf("%c ", *pch1++);
}
```

When a client passes a pointer to a remote procedure, the client stub transmits both the pointer and the data it points to. Unless the pointer is restricted to its corresponding data, all the client's memory must be transmitted with every remote call. By enforcing strong typing in the interface definition, MIDL limits client-stub processing to the data that corresponds to the specified pointer.

The size of the array and the range of array elements transmitted to the remote computer can be constant or variable. When these values are variable, and thus determined at run time, you must use attributes in the IDL file to specify how many array elements to transmit. The following MIDL attributes support array bounds.

Attribute	Description	Default
[first_is]	Index of the first array element transmitted.	0
[last_is]	Index of the last array element transmitted.	-
[length_is]	Total number of array elements transmitted.	-
[max_is]	Highest valid array index value.	-
[min_is]	Lowest valid array index value.	0
[size_is]	Total number of array elements allocated for the array.	-

Note The **min_is** attribute is not implemented in RPC. The minimum array index is always treated as zero.

MIDL Array Attributes Used in RPC

This section discusses the MIDL array attributes in the following topics:

- The [size_is] Attribute
- The [length_is] Attribute
- The [first_is] and [last_is] Attributes
- The [max_is] Attribute
- Combining Array Attributes
- The [string] Attribute in Arrays

The [size_is] Attribute

The **[size_is]** attribute is associated with an integer constant, expression, or variable that specifies the allocation size of the array. Consider a character array whose length is determined by user input:

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(2.0)
]
interface arraytest
{
  void fArray2([in] short sSize,
               [in, out, size_is(sSize)] char achArray[*]);
}
```

Note The asterisk (*) that marks the placeholder for the variable-array dimension is optional.

The server stub must allocate memory on the server that corresponds to the memory on the client for that parameter. The variable that specifies the size must always be at least an **[in]** parameter. The **[in]** directional attribute is required so that the size value is defined on entry to the server stub. This size value provides information that the server stub requires to allocate the memory.

The size parameter can also be **[in, out]**. This is useful if, for instance, the array the client sends is not large enough for the data that the server needs to store in it. You can use an **[in, out]** size parameter to send the required size back to the client program.

See Also

Multiple Levels of Pointers

The [length_is] Attribute

The **[size_is]** attribute lets you specify the maximum size of the array. When this is the only attribute, all elements of the array are transmitted. Instead of sending all elements of the array, you can specify the transmitted elements using the **[length_is]** attribute, as follows:

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(3.0)
]
interface arraytest
{
  void fArray3([in] short sSize,
               [in] short sLength
               [in, out, size_is(sSize),
                 length_is(sLength)] char achArray[*]);
}
```

Size describes allocation while length describes transmission. The number of elements transmitted must always be less than or equal to the number of elements allocated. The value associated with **length_is** is always less than or equal to **size_is**.

The [first_is] and [last_is] Attributes

You can determine the number of transmitted elements by specifying the first and last elements. Use the **[first_is]** and **[last_is]** attributes as shown:

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(4.0)
]
interface arraytest
{

  void fArray4([in] short sSize,
               [in] short sLast,
               [in] short sFirst,
               [in, out,
                 size_is(sSize),
                 first_is(sFirst),
                 last_is(sLast)]   char achArray[]) ;
}
```

The [max_is] Attribute

You can specify the valid bounds of the index numbers of an array using the **[max_is]** attribute.

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(5.0)
]
interface arraytest
{
  void fArray5([in] short sMax,
               [in, out, max_is(sMax)]  char achArray[]);
}
```

Combining Array Attributes

Field attributes can be supplied in various combinations as long as the stub can use the information to determine the size of the array and the number of bytes to transmit to the server. The relationships between the attributes are defined using the following formulas:

```
size_is = max_is + 1;
length_is = last_is - first_is + 1;
```

The values associated with the attributes must obey several common-sense rules based on those formulas. These rules are:

- Do not specify a **[first_is]** index value smaller than zero or a **[last_is]** value greater than **[max_is]**.
- Do not specify a negative size for an array. Define the first and last elements so that they result in a length value of zero or greater. Define the **[max_is]** value so that the size is zero or greater. If MIDL was invoked with the **/error bounds_check** option, then the stub raises an exception when the size is less than zero, or the transmitted length is less than zero.
- Do not use the **[length_is]** and **[last_is]** attributes at the same time, nor the **[size_is]** and **[max_is]** attributes at the same time.

Because of the close relationship in C between arrays and pointers, MIDL also lets you declare arrays in parameter lists using pointer notation. MIDL treats a parameter that is a pointer to a type as an array of that type if the parameter has any of the attributes commonly associated with arrays.

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45)
  version(6.0)
]
```

```
interface arraytest
{
  void fArray6([in] short sSize,
               [in, out, size_is(sSize)] char * p1);
  void fArray7([in] short sSize,
               [in, out, size_is(sSize)] char achArray[]);
}
```

In the preceding example, the array and pointer parameters in the functions **fArray6** and **fArray7** are equivalent.

The [string] Attribute in Arrays

You can use the **[string]** attribute for one-dimensional character arrays, wide-character arrays, and byte arrays that represent text strings. If you use the **[string]** attribute, the client stub uses the C-library functions **strlen** or **wstrlen** to count the number of characters in the string. To avoid possible inconsistencies, MIDL does not let you use the **[string]** attribute at the same time as the **[first_is]**, **[last_is]**, and **[size_is]** attributes.

With null-terminated strings in C, you must allow space for the null character at the end of the string. For example, when declaring a string that will hold up to 80 characters, allocate 81 characters. The following sample IDL file demonstrates how to declare arrays with the **[string]** attribute.

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(8.0)
]
interface arraytest
{
  void fArray8([in, out, string] char achArray[]);
}
```

Multidimensional Arrays

Array attributes can also be used with multidimensional arrays. However, be careful to ensure that every dimension of the array has a corresponding attribute. For example:

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(2.0)
]
interface multiarray
{
```

(continued)

(continued)

```
 void arr2d( [in] short        d1size,
             [in] short        d2len,
             [in,size_is( d1size, ),
               length_is ( , d2len) ] long    array2d[*][30] ) ;
}
```

The preceding array is a conformant array (of size *d1size*) of 30 element arrays (with *d2len* elements shipped for each). The comma in the parentheses of the **[size_is]** attribute specifies that value in *d1size* is applied to the first dimension of the array. Likewise, the command in the parentheses of the **[length_is]** attribute indicates that the value in *d2len* is applied to the second dimension of the array.

The MIDL 2.0 compiler provides two methods for marshaling parameters: mixed-mode (**/Os**) and fully-interpreted (**/Oif** or **/Oicf**). By defaut, the MIDL compiler compiles interfaces in mixed mode. You do not need to explicitly specify the **/Os** switch to get mixed-mode marshaling.

The fully-interpreted method marshals data completely offline. This reduces the size of the stub code considerably, but it also results in decreased performance. In mixed-mode marshaling, the stubs marshals some parameters online. While this results in a larger stub size, it also offers increased performance.

Tip Caution needs to be exercised when compiling IDL files in this mode. Using multidimenstional arrays in mixed mode can result in parameters that are not marshaled correctly. The **/Oicf** command line switch is recommended when your interface defines parameters that are multidimensional arrays.

The **[string]** attribute can also be used with multidimensional arrays. The attribute applies to the least significant dimension, such as a conformant array of strings. You can also use multidimensional pointer attributes. For example:

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(2.0)
]
interface multiarray
{
  void arr2d([in] short  d1len,
             [in] short  d2len,
             [in] size_is(d1len, d2len) ] long  ** ptr2d) ;
}
```

In the preceding example, the variable *ptr2d* is a pointer to a *d1len*-sized block of pointers, each of which points to *d2len* pointers to **long**.

Multidimensional arrays are not equivalent to arrays of pointers. A multidimensional array is a single, large block of data in memory. An array of pointers only contains a block of pointers in the array. The data that the pointers point to can be anywhere in memory. Also, ANSI C syntax allows only the most significant (leftmost) array dimension to be unspecified in a multidimensional array. Therefore, the following is a valid statement:

```
long a1[] [20]
```

Compare this to the following invalid statement:

```
long a1[20] []
```

Pointers and RPC

It is very efficient to use pointers as C-function parameters. The pointer costs only a few bytes and can be used to access a large amount of memory. However, in a distributed application, the client and server procedures reside in different address spaces—they can be on different computers. Therefore, the client and the server usually do not have access to the same memory space.

When one of the remote procedure's parameters is a pointer to an object, the client must transmit a copy of that object and its pointer to the server. If the remote procedure modifies the object through its pointer, the server returns the pointer and its modified copy.

MIDL offers pointer attributes to minimize the amount of required overhead and the size of your application. This section discusses the purpose and uses of MIDL pointer attributes. It also presents information on pointer handling in RPC applications.

Kinds of Pointers

MIDL enables RPC applications to define the following pointer types:

- Reference Pointers
- Unique Pointers
- Full Pointers

Reference Pointers

Reference pointers are the simplest pointers and require the least amount of processing by the client stub. When a client program passes a reference pointer to a remote procedure, the reference pointer always contains the address of a valid block of memory. It will still be pointing to the same memory block when the remote procedure completes. These pointers are mainly used to implement reference semantics, and to allow for **[out]** parameters in C.

In Figure 13-1, the value of the pointer does not change during the call, although the contents of the data at the address indicated by the pointer can change.

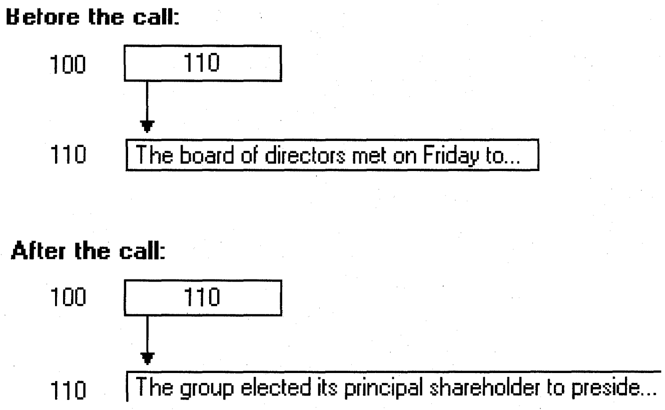

Before the call:
100
110
110
The board of directors met on Friday to...
After the call:
100
110
110
The group elected its principal shareholder to preside...

Figure 13-1: Reference Pointer Before and After a Call.

A reference pointer has the following characteristics:

- It always points to valid storage and never has the value NULL.
- It never changes during a call and always points to the same storage before and after the call.
- Data returned from the remote procedure is written into the existing storage.
- The storage pointed to by a reference pointer cannot be accessed by any other pointer or any other name in the function.

Use the **[ref]** attribute to specify reference pointers in interface definitions, as shown in the following example:

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(1.0)
]
interface RefPtrInterface
{
  void RemoteFn([out, ref] char *pChar);
}
```

This example defines the parameter *pChar* as a pointer to a single character, not an array of characters. It is an **[out]** parameter and a reference pointer that points to memory that the server routine **RemoteFn** will fill with data.

Unique Pointers

In C programs, more than one pointer can contain the address of data. The pointers are said to create an alias for the data. Aliases are also created when pointers point at declared variables. The following code fragment illustrates both of these methods of aliasing:

```
int iAnInteger=50;

// The next statement makes ipAnIntegerPointer an
// alias for iAnInteger.
int *ipAnIntegerPointer = &iAnInteger;

// This statement creates an alias for ipAnIntegerPointer.
int *ipAnotherIntegerPointer = ipAnIntegerPointer;
```

In a typical C program, you might specify a binary tree using the following definition:

```
typedef struct _treetype
{
    long               lValue;
    struct _treetype * left;
    struct _treetype * right;
} TREETYPE;

TREETYPE * troot;
```

More than one pointer can access the contents of a tree node. This is generally fine for nondistributed applications. However, this style of programming generates more complicated RPC support code. The client and server stubs require the additional code to manage the data and the pointers. The underlying stub code must resolve the various pointers to the addresses and determine which copy of the data represents the most recent version.

The amount of processing can be reduced if you guarantee that your pointer is the only way the application can access that area of memory. The pointer can still have many of the features of a C pointer. For example, it can change between null and non-null values or stay the same. The following example illustrates this. The pointer is null before the call and points to a valid string after the call, as shown in Figure 13-2.

By default, the MIDL compiler applies the **[unique]** pointer attribute to all pointers that are not parameters. This default setting can be changed with the **[pointer_default]** attribute.

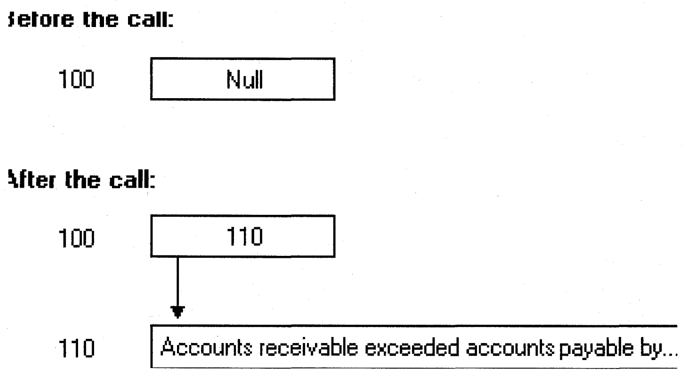


Figure 13-2: Null and Non-Null Values Before and After a Call.

A unique pointer has the following characteristics:

- It can have the value null.
- It can change from null to non-null during the call. When the value changes to non-null, new memory is allocated on return.
- It can change from non-null to null during the call. When the value changes to NULL, the application is responsible for freeing the memory.
- The value can change from one non-null value to another.
- The storage that a unique pointer points to cannot be accessed by any other pointer or name in the operation.
- Return data is written into existing storage if the pointer does not have the value null.

The following example demonstrates how to define a unique pointer:

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(1.0)
]
interface RefPtrInterface
{
  void RemoteFn([out, unique] char *achArray);
}
```

Full Pointers

Unlike unique pointers, full pointers support aliasing. This means that multiple pointers can refer to the same data, as shown in Figure 13-3.

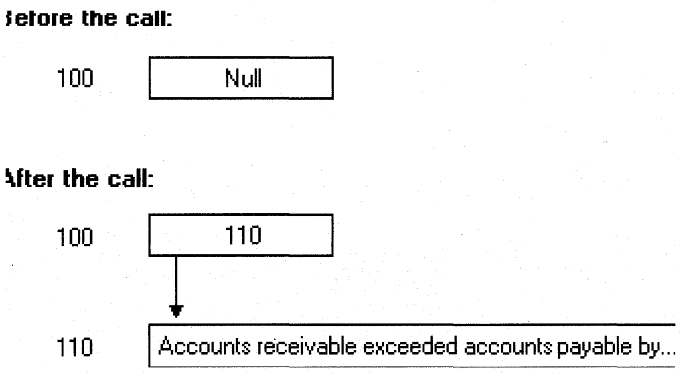


Figure 13-3: Multiple Pointers Referring to the Same Data.

A full pointer has the following characteristics:

- It can have the value null.
- It can change from null to non-null during the call. When the value changes to non-null, the client stub allocates new memory allocated on return. The client program should free this memory before it terminates.
- It can change from non-null to null during the call. When the value changes to null, the application is responsible for freeing the memory.
- The value can change from one non-null value to another.
- The storage that a unique pointer points to may be accessed by another pointer or name in the operation.
- Return data is written into existing storage if the pointer does not have the value null.

Use the **[ptr]** attribute to specify a full pointer, as shown in the following example:

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(1.0)
]
interface FullPtrInterface
{
  void RemoteFn([in,out,ptr] char *achArray);
}
```

Pointers and Memory Allocation

The ability to change memory through pointers often requires that the server and the client allocate enough memory for the elements in the array.

When a stub must allocate or free memory, it calls run-time library functions that in turn call the functions **midl_user_allocate** and **midl_user_free**. These functions are not included as part of the run-time library. You need to write your own versions of these functions and link them with your application. In this way, you can decide how to manage memory. When compiling your IDL file in OSF-compatibility (/**osf**) mode, you do not need to implement these functions. You must write these functions to the following prototypes:

```
void __RPC_FAR * __RPC_API midl_user_allocate(size_t len)

void __RPC_API midl_user_free(void __RPC_FAR * ptr)
```

For example, the versions of these functions for an application can simply call standard library functions:

```
void __RPC_FAR * __RPC_API midl_user_allocate(size_t len)
{
    return(malloc(len));
}

void __RPC_API midl_user_free(void __RPC_FAR * ptr)
{
    free(ptr);
}
```

Default Pointer Types

For a variety of reasons, pointers in interface definitions may not have an attribute specification. When they don't, the MIDL compiler must use a default pointer attribute. MIDL allows you to specify which pointer attribute you want as the default by using the **[pointer_default]** attribute. The following example illustrates its use:

```
/* IDL file */
[
  uuid(ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(1.0),
  pointer_default(ref)
]
interface MyInterface
{
    // Types, constants, and procedure definitions go here.
}
```

This example sets the default so that the MIDL compiler will treat all unattributed pointers as reference pointers.

The MIDL compiler offers three different default cases for unattributed pointers. Note that these cases use the terms top-level pointer and embedded pointer. A top-level pointer is a parameter that is a pointer. An embedded pointer is a pointer in a structure or union. The default cases for unattributed pointers are:

- Function parameters that are top-level pointers default to **[ref]** pointers.
- Pointers embedded in structures and pointers to other pointers default to the type specified by the **[pointer_default]** attribute.
- When no **[pointer_default]** attribute is supplied, pointers to pointers default to the **[unique]** attribute if the MIDL compiler is in Microsoft-extensions mode. If the compiler is set to DCE-compatible mode the default is **[ptr]**.

Remote procedures always return a **[unique]** or a full pointer. The MIDL compiler reports an error if a function return value is, either explicitly or by default, a reference pointer.

Functions that return a pointer value can specify a pointer attribute as a function attribute. They cannot, however, specify a reference pointer. If a pointer attribute is not present, the function return pointer uses the value that the **[pointer_default]** attribute specifies.

Note To ensure unambiguous pointer-attribute behavior, always use explicit pointer attributes when defining a pointer.

Pointer-Attribute Type Inheritance

According to the DCE specification, each IDL file must define attributes for its pointers. If an explicit attribute is not assigned to a pointer, the pointer uses the value specified by the **[pointer_default]** keyword. Some DCE implementations do not allow unattributed pointers. If a pointer does not have an explicit attribute, the IDL file must have a **[pointer_default]** specification so that the pointer attribute can be set.

In default (Microsoft-extensions) mode, you can specify a pointer's attribute in the IDL file that imports the defining IDL file. Pointers defined in one IDL file can inherit attributes that are specified in other IDL files. Also, in default mode, IDL files can include unattributed pointers. If neither the base nor the imported IDL files specify a pointer attribute or **[pointer_default]**, unattributed pointers are interpreted as unique pointers.

The MIDL compiler assigns pointer attributes to pointers using the following priority rules (1 is highest).

Priority	Description
1	Explicit pointer attributes are applied to the pointer at the definition or use site.
2	The default is the **[pointer_default]** attribute in the IDL file that defines the type.
3	The default is the **[pointer_default]** attribute in the IDL file that imports the type.
4	The default is **[ptr]** in DCE-compatibility mode, or **[unique]** in Microsoft-extensions mode.

Using Arrays, Strings, and Pointers

The Doctor program (see \samples\rpc\doctor in the Platform SDK) is a good example of an application that demonstrates the design trade-offs involved with array and string attributes. The Doctor program is a small sample version of a better-known program, Eliza, developed by artificial intelligence researcher Joseph Weizenbaum at MIT in the 1960s. The program mimics a therapist by using simple pattern-matching to respond to natural language input.

The Eliza program recorded the following session:

User	Eliza
Men are all alike.	In what way?
They're always bugging us about something or other.	Can you think of a specific example?
Well, my boyfriend made me come here.	Your boyfriend made you come here.
He says I'm depressed much of the time.	I am sorry to hear that you are depressed.
It's true, I am unhappy.	Do you think coming here will help you not to be unhappy?
I need some help; that much seems certain.	What would it mean to you if you got some help?
Perhaps I could learn to get along with my mother.	Tell me more about your family.
My mother takes care of me.	Who else in your family takes care of you?
My father.	Your father.
You are like my father in some ways.	What resemblance do you see?

The Doctor program can be split into client-side and server-side applications. The client side prompts the patient for input and displays the doctor's response. The server side processes the patient input and generates the doctor's response. This is a classic example of a client-server application: the client is responsible for user interaction while the server handles the extensive computational load. Not much data is passed to and returned by the function but, because the data can require a significant amount of processing, the server processes it.

The Doctor program uses a character array for input and returns another character array as output. The table below lists four ways to pass character arrays between the client and server, and the attributes and functions needed to implement each approach.

Approach	Attributes or functions
Counted character arrays	**[size_is]**, **[length_is]**, **[ref]**
Stub-managed strings	**[string]**, **[ref]**, **midl_user_allocate** on server
Stub-managed strings	**[string]**, **[unique]**, **midl_user_allocate** on client and server
Function that returns a string	**[unique]**

Within the constraints associated with these combinations of attributes, there are alternative ways of sending one character array from client to server and of returning another character array from server to client.

Counted Character Arrays

The **[size_is]** attribute indicates the upper bound of the array while the **[length_is]** attribute indicates the number of array elements to transmit. In addition to the array, the remote procedure prototype must include any variables representing length or size that determine the transmitted array elements (they can be separate parameters or bundled with the string in a structure). These attributes can be used with wide-character or single-byte character arrays just as they would be with arrays of other types.

The information in this section describes remote procedure parameter prototypes for character arrays. It is divided into the following topics:

- [in, out, size_is] Prototype
- [in, size_is and out, size_is] Prototype

[in, out, size_is] Prototype

The following function prototype uses a single-counted character array that is passed both ways: from client to server and from server to client.

```
#define STRSIZE 500 //maximum string length

void Analyze(
    [in, out, length_is(*pcbSize), size_is(STRSIZE)] char  achInOut[],
    [in, out]  long *pcbSize);
```

As an **[in]** parameter, *achInOut* must point to valid storage on the client side. The developer allocates memory associated with the array on the client side before making the remote procedure call.

The stubs use the **[size_is]** parameter *strsize* to allocate memory on the server and then use the **[length_is]** parameter *pcbSize* to transmit the array elements into this memory. The developer must make sure the client code sets the **[length_is]** variable before calling the remote procedure:

```
/* client */
char achInOut[STRSIZE];
long cbSize;
...
gets(achInOut);                  // get patient input
cbSize = strlen(achInOut) + 1;   // transmit '\0' too
Analyze(achInOut, &cbSize);
```

In the previous example, the character array achInOut is also used as an **[out]** parameter. In C, the name of the array is equivalent to the use of a pointer. By default, all pointers are reference pointers—they do not change in value and they point to the same area of memory on the client before and after the call. All memory that the remote procedure accesses must fit the size that the client specifies before the call, or the stubs will generate an exception.

Before returning, the **Analyze** function on the server must reset the *pcbSize* parameter to indicate the number of elements that the server will transmit to the client as shown:

```
/* server */
Analyze(char * str, long * pcbSize)
{
   ...
   *pcbSize = strlen(str) + 1; // transmit '\0' too
   return;
}
```

Instead of using a single string for both input and output, you may find it more efficient and flexible to use separate parameters.

[in, size_is and out, size_is] Prototype

The following function prototype uses two counted strings. The developer must write code on both client and server to keep track of the character array lengths and pass parameters that tell the stubs how many array elements to transmit.

```
void Analyze(
    [in,  length_is(cbIn), size_is(STRSIZE)]    char  achIn[],
    [in]                                        long  cbIn,
    [out, length_is(*pcbOut), size_is(STRSIZE)] char  achOut[],
    [out]                                       long  *pcbOut);
```

Note the parameters that describe the array length are transmitted in the same direction as the arrays: *cbIn* and *achIn* are **[in]** parameters while *pcbOut* and *achOut* are **[out]** parameters. As an **[out]** parameter, the parameter *pcbOut* must follow C convention and be declared as a pointer.

The client code counts the number of characters in the string, including the trailing zero, before calling the remote procedure as shown:

```
/* client */
char achIn[STRSIZE], achOut[STRSIZE];
long cbIn, cbOut;
...
gets(achIn);                   // get patient input
cbIn = strlen(achIn) + 1;      // transmitted elements
Analyze(achIn, cbIn, achOut, &cbOut);
```

The remote procedure on the server supplies the length of the return buffer in *cbOut* as shown:

```
/* server */
void Analyze(char *pchIn,
             long cbIn,
             char *pchOut,
             long *pcbOut)
{
   ...
   *pcbOut = strlen(pchOut) + 1; // transmitted elements
   return;
}
```

Knowing that the parameter is a string allows us to use the **[string]** attribute. This attribute directs the stub to calculate the string size, thus eliminating the overhead associated with the **[size_is]** parameters.

Strings

The **[string]** attribute indicates the parameter is a pointer to an array of type **char**, **byte**, or **w_char**. As with a conformant array, the size of a **[string]** parameter is determined at run time. Unlike a conformant array, the developer does not have to provide the length associated with the array—the **[string]** attribute tells the stub to determine the array size by calling **strlen**. A **[string]** attribute cannot be used at the same time as the **[length_is]** or **[last_is]** attributes.

The **[in, string]** attribute combination directs the stub to pass the string from client to server only. The amount of memory allocated on the server is the same as the transmitted string size plus one.

The **[out, string]** attributes direct the stub to pass the string from server to client only. The call-by-value design of the C language insists that all **[out]** parameters must be pointers.

The **[out]** parameter must be a pointer and, by default, all pointer parameters are reference pointers. The reference pointer does not change during the call—it points to the same memory as before the call. For string pointers, the additional constraint of the reference pointer means the client must allocate sufficient valid memory before making the remote procedure call. The stubs transmit the string that the **[out, string]** attributes indicate into the memory already allocated on the client side.

The following topics describe the remote procedure parameter prototypes for strings:

- [in, out, string] Prototype
- [in, string] and [out, string] Prototype

[in, out, string] Prototype

The following function prototype uses a single **[in, out, string]** parameter for both the input and output strings. The string first contains patient input and is then overwritten with the doctor response as shown:

```
void Analyze([in, out, string, size_is(STRSIZE)] char  achInOut[]);
```

This example is similar to the one that employed a single-counted string for both input and output. As with that example, the **[size_is]** attribute determines the number of elements allocated on the server. The **[string]** attribute directs the stub to call **strlen** to determine the number of transmitted elements.

The client allocates all memory before the call as:

```
/* client */
char achInOut[STRSIZE];
...
gets(achInOut);            // get patient input
Analyze(achInOut);
printf("%s\n", achInOut);  // display doctor response
```

Note that the **Analyze** function no longer must calculate the length of the return string as it did in the counted-string example where the **[string]** attribute was not used. Now the stubs calculate the length as shown:

```
/* server */
void Analyze(char *pchInOut)
{
   ...
   Respond(response, pchInOut); // don't need to call strlen
   return;                      // stubs handle size
}
```

[in, string] and [out, string] Prototype

The following function prototype uses two parameters: an **[in, string]** parameter and an **[out, string]** parameter:

```
void Analyze(
    [in, string]                       *pszInput,
    [out, string, size_is(STRSIZE)]    *pszOutput);
```

The first parameter is **[in]** only. This input string is only transmitted from the client to the server. The server uses it as the basis for further processing. The string is not modified and is not required again by the client, so it does not have to be returned to the client.

The second parameter, representing the doctor's response, is **[out]** only. This response string is only transmitted from the server to the client. The allocation size is provided so that the server stubs can allocate memory for it. Since *pszOutput* is a **[ref]** pointer, the client must have sufficient memory allocated for the string before the call. The response string is written into this area of memory when the remote procedure returns.

Multiple Levels of Pointers

You can use multiple pointers such as a **[ref]** pointer to another **[ref]** pointer that points to the character array as shown:

```
void Analyze(
   [in, string]                     char  *pszInput,
   [out, string, size_is(STRSIZE)]  char **ppszOutput);
```

When there are multiple levels of pointers, the attributes are associated with the pointer closest to the variable name. The client is still responsible for allocating any memory associated with the response.

The following example allows the stub to call the server without knowing in advance how much data will be returned:

```
[
    uuid( ...),
    version(3.3),
    pointer_default(unique)   // Required whenever you
                              // have pointers to pointers.
```

(continued)

(continued)

```
                                // Pointer has to be unique so
                                // that it can be NULL if
                                // necessary.
]
interface AnInterface
{
    HRESULT GetBars([out] long * pSize,
             [out, size_is( , *pSize)]
             BAR ** ppBar);//BAR type defined elsewhere
}
```

In this example, the stub passes the server a unique pointer, which the server initializes to NULL. The server then allocates a block of BARs, sets the pointer, sets the size argument and returns. Note that in order for the server to have an effect on the caller you must pass a **[ref]** pointer to a **[unique]** pointer to your data. Also note the comma in **[size_is**(, *pSize)**]**, which indicates that the top-level pointer is not a sized pointer, but that the lower-level pointer is.

On the client side, the stub allocates the block, assigns the address to the *ppBar* argument and unmarshals BAR objects. The size argument indicates the size of the block (and the number of unmarshaled BARs).

See Also

size_is

CHAPTER 14

Pipes

The pipe type constructor is a highly efficient mechanism for passing large amounts of data, or any quantity of data that is not all available in memory at one time. By using a pipe, RPC run time handles the actual data transfer, eliminating the overhead associated with repeated remote procedure calls.

After a client invokes a remote procedure that has a pipe parameter, the client and server enter loops to transfer data. The data can be produced on the client or the server. Either way, the amount of data (in bytes) does not have to be known in advance. The data can be produced or consumed incrementally. While in the data-transfer loop, the server calls stub routines that load or unload a buffer of data. The client calls programmer-defined procedures to allocate buffers, load data into and unload data from the buffers.

For more information on pipe syntax and restrictions, see *pipe* in the MIDL Language Reference. The PIPES sample program in the Platform SDK samples\rpc directory demonstrates how to use **[in,out]** pipes to transfer data between a client and a server.

Essential Pipe Terminology

Like other types of parameters to remote procedure calls, pipes can be **[in]** or **[out]** parameters. Since the server controls the transfer of data through a pipe, pipes with the **[in]** attribute are said to *pull* data to the server. Similarly, output pipes *push* data from the server to the client. The procedures that do the data transfer are called the *pull procedure* and the *push procedure,* respectively.

The MIDL compiler generates the push and pull procedures for the server. In addition, it manages the allocation of data buffers in memory. However, the client must provide its own push and pull procedures. It must also provide a procedure for allocating the memory buffers used by the pipe. These are automatically called at the appropriate time by the client stub. The allocation procedure is often called the **alloc** procedure or the **alloc** function.

The Pipe State

On the server, the MIDL compiler creates a *state* variable that coordinates push, pull, and alloc procedures. On the client side, the developer must create the *state* variable. Therefore, the *state* variable is local to both sides—that is, the client and server each maintain their own pipe state. The server stub code maintains the state variable on the server. You should not attempt to modify its contents directly. The client must initialize the fields in the pipe control structure and maintain its *state* variable. It uses the *state* variable to identify where to locate or place data.

The client *state* variable can be as simple as a file handle, if you are transferring data from one file to another. It can also be an integer that points to an element in an array. Or you can define a fairly complex state structure to perform additional tasks, such as coordinating the push and pull routines on an **[in, out]** parameter.

Defining Pipes in IDL Files

When a pipe is defined in an IDL file, the MIDL compiler generates a pipe control structure whose members are pointers to push, pull, and alloc procedures as well as a state variable that coordinates these procedures. The client application initializes the fields in the pipe control structure, maintains its state variable, and manages the data transfer with its own push, pull, and alloc functions. The client stub code calls these application functions in loops during data transfer. For an input pipe, the client stub marshals the transfer data and transmits it to the server stub. For an output pipe, the client stub unmarshals the data into a buffer and passes a pointer to that buffer back to the client application.

The server stub code initializes the fields of the pipe control structure to a state variable, as well as pointers to push and pull routines. The server stub maintains the state and manages its private storage for the transfer data. The server application calls the pull and push routines in loops during the remote procedure call as it receives and unmarshals data from the client stub, or marshals and transmits data to the client stub.

The following example IDL file defines a pipe type LONG_PIPE, whose element size is defined as **long**. It also declares function prototypes for the remote procedure calls **InPipe** and **OutPipe**, to send and receive data, respectively. When the MIDL compiler processes the IDL file, it generates the header file shown in the example:

Example

```
// File: pipedemo.idl
typedef pipe long LONG_PIPE;
void InPipe( [in] LONG_PIPE pipe_data );
void OutPipe( [out] LONG_PIPE *pipe_data );
//end pipedemo.idl
```

```
// File: pipedemo.h (fragment)
// Generated by the MIDL compiler from pipedemo.idl
typedef struct pipe_LONG_PIPE
{
    void (__RPC_FAR * pull) (
        char __RPC_FAR * state,
        long __RPC_FAR * buf,
        unsigned long esize,
        unsigned long __RPC_FAR * ecount );
    void (__RPC_FAR * push) (
        char __RPC_FAR * state,
        long __RPC_FAR * buf,
        unsigned long ecount );
    void (__RPC_FAR * alloc) (
        char __RPC_FAR * state,
        unsigned long bsize,
        long __RPC_FAR * __RPC_FAR * buf,
        unsigned long __RPC_FAR * bcount );
    char __RPC_FAR * state;
} LONG_PIPE;

void InPipe(
    /* [in] */ LONG_PIPE pipe_data);
void OutPipe(
    /* [out] */ LONG_PIPE __RPC_FAR *pipe_data);
//end pipedemo.h
```

Client-Side Pipe Implementation

The client application must implement the following procedures, which the client stub will call during data transfer:

- A pull procedure (for an input pipe)
- A push procedure (for an output pipe)
- An alloc procedure to allocate a buffer for the transfer data

All of these procedures must use the arguments specified by the MIDL-generated header file. In addition, the client application must have a state variable to identify where to locate or place data.

The alloc procedure can also be as simple or as complex as needed. For example, it can return a pointer to the same buffer every time the stub calls the function, or it can allocate a different amount of memory each time. If your data is already in the proper form (an array of pipe elements, for example) you can coordinate the alloc procedure with the pull procedure to allocate a buffer that already contains the data. In that case, your pull procedure could be an empty routine.

The buffer allocation must be in bytes. The push and pull procedures, on the other hand, manipulate elements, whose size in bytes depends on how they were defined.

Implementing Input Pipes on the Client

When using an input pipe to transfer data from the client to the server, you must implement a pull procedure. The pull procedure must find the data to be transferred, read the data into the buffer, and set the number of elements to send. Not all of the data has to be in the buffer when the server begins to pull data to itself. The pull procedure can fill the buffer incrementally.

When there is no more data to send, the procedure sets its last argument to zero. When all the data is sent, the pull procedure should do any needed cleanup before returning. For a parameter that is an **[in**, **out]** pipe, the pull procedure must reset the client's state variable after all the data has been transmitted, so that the push procedure can use it to receive data.

The following example is extracted from the Pipedemo program included with the Platform SDK.

```
//file: client.c (fragment)
#include "pipedemo.h"
long *globalPipeData;
long    globalBuffer[BUF_SIZE];

ulong   pipeDataIndex; /* state variable */

void SendLongs()
{
    LONG_PIPE inPipe;
    int i;
    globalPipeData =
        (long *)malloc( sizeof(long) * PIPE_SIZE );

    for (i=0; i<PIPE_SIZE; i++)
        globalPipeData[i] = IN_VALUE;

    pipeDataIndex = 0;
    inPipe.state =  (rpc_ss_pipe_state_t )&pipeDataIndex;
    inPipe.pull  = PipePull;
    inPipe.alloc = PipeAlloc;

    InPipe( inPipe ); /* Make the rpc */

    free( (void *)globalPipeData );
```

```
}//end SendLongs

void PipeAlloc( rpc_ss_pipe_state_t stateInfo,
                ulong requestedSize,
                long **allocatedBuffer,
                ulong *allocatedSize )
{
    ulong *state = (ulong *)stateInfo;
    if ( requestedSize > (BUF_SIZE*sizeof(long)) )
    {
       *allocatedSize = BUF_SIZE * sizeof(long);
    }
    else
    {
       *allocatedSize = requestedSize;
    }
    *allocatedBuffer = globalBuffer;
} //end PipeAlloc

void PipePull( rpc_ss_pipe_state_t stateInfo,
               long *inputBuffer,
               ulong maxBufSize,
               ulong *sizeToSend )
{
    ulong currentIndex;
    ulong i;
    ulong elementsToRead;
    ulong *state = (ulong *)stateInfo;

    currentIndex = *state;
    if (*state >=  PIPE_SIZE )
    {
        *sizeToSend = 0; /* end of pipe data */
        *state = 0; /* Reset the state = global index */
    }
    else
    {
        if ( currentIndex + maxBufSize > PIPE_SIZE )
            elementsToRead = PIPE_SIZE - currentIndex;
        else
            elementsToRead = maxBufSize;

        for (i=0; i < elementsToRead; i++)
        {
            /*client sends data */
```

(continued)

(continued)

```
            inputBuffer[i] = globalPipeData[i + currentIndex];
        }

        *state +=   elementsToRead;
        *sizeToSend = elementsToRead;
    }
}//end PipePull
```

This example includes the header file generated by the MIDL compiler. For details see *Defining Pipes in IDL Files*. It also declares a variable it uses as the data source called *globalPipeData*. The variable *globalBuffer* is a buffer that the pull procedure uses to send the blocks of data it obtains from *globalPipeData*.

The **SendLongs** function declares the input pipe, and allocates memory for the data source variable *globalPipeData*. In your client/server program, the data source can be a file or structure that the client creates. You can also have your client program obtain data from the server, process it, and return it to the server using an input pipe. In this simple example, the data source is a dynamically allocated buffer of long integers.

Before the transfer can begin, the client must set pointers to the state variable, the pull procedure, and the alloc procedure. These pointers are kept in the pipe variable the client declares. In this case, **SendLongs** declares ***inPipe***. You can use any appropriate data type for your state variable.

Clients initiate data transfers across a pipe by invoking a remote procedure on the server. Calling the remote procedure tells the server program that the client is ready to transmit. The server can then pull the data to itself. This example invokes a remote procedure called **InPipe**. After the data is transferred to the server, the **SendLongs** function frees the dynamically allocated buffer.

Rather than allocate memory each time a buffer is needed. the alloc procedure in this example simply sets a pointer to the variable globalBuffer. The pull procedure reuses this buffer each time it transfers data. More complex client programs may need to allocate a new buffer each time the server pulls data from the client.

The client stub calls the pull procedure. The pull procedure in this example uses the state variable to track the next position in the global data source buffer to read from. It reads data from the source buffer into the pipe buffer. The client stub transmits the data to the server. When all the data has been sent, the pull procedure sets the buffer size to zero. This tells the server to stop pulling data.

Implementing Output Pipes on the Client

When using an output pipe to transfer data from the server to the client, you must implement a push procedure in your client. The push procedure takes a pointer to a buffer and an element count from the client stub and, if the element count is greater than 0, processes the data. For example, it could copy the data from the stub's buffer to its own memory. Alternately, it could process the data in the stub's buffer and save it to a file. When the element count equals zero, the push procedure completes any needed cleanup tasks before returning.

In the following example, the client function **ReceiveLongs** allocates a pipe structure and a global memory buffer. It initializes the structure, makes the remote procedure call, and then frees the memory.

Example

```
//file: client.c (fragment)
#include "pipedemo.h"
long *  globalPipeData;
long    globalBuffer[BUF_SIZE];

ulong   pipeDataIndex; /* state variable */

void ReceiveLongs()
{
    LONG_PIPE *outputPipe;
    idl_long_int i;

    globalPipeData =
        (long *)malloc( sizeof(long) * PIPE_SIZE );

    pipeDataIndex = 0;
    outputPipe.state =  (rpc_ss_pipe_state_t )&pipeDataIndex;
    outputPipe.push  = PipePush;
    outputPipe.alloc = PipeAlloc;

    OutPipe( &outputPipe ); /* Make the rpc */

    free( (void *)globalPipeData );

}//end ReceiveLongs()

void PipeAlloc( rpc_ss_pipe_state_t stateInfo,
                ulong requestedSize,
                long **allocatedBuffer,
                ulong *allocatedSize )
```
(continued)

(continued)

```
{
    ulong *state = (ulong *)stateInfo;
    if ( requestedSize > (BUF_SIZE*sizeof(long)) )
    {
       *allocatedSize = BUF_SIZE * sizeof(long);
    }
    else
    {
       *allocatedSize = requestedSize;
    }
    *allocatedBuffer = globalBuffer;
} //end PipeAlloc

void PipePush( rpc_ss_pipe_state_t stateInfo,
               long *buffer,
               ulong numberOfElements )
{
    ulong elementsToCopy, i;
    ulong *state = (ulong *)stateInfo;

    if (numberOfElements == 0)/* end of data */
    {
        *state = 0; /* Reset the state = global index */
    }
    else
    {
        if (*state + numberOfElements > PIPE_SIZE)
            elementsToCopy = PIPE_SIZE - *state;
        else
            elementsToCopy = numberOfElements;

        for (i=0; i <elementsToCopy; i++)
        {
            /*client receives data */
            globalPipeData[*state] = buffer[i];
            (*state)++;
        }
    }
}//end PipePush
```

This example includes the header file generated by the MIDL compiler. For details see *Defining Pipes in IDL File*. It also declares a variable, *globalPipeData*, that it uses as the data sink. The variable *globalBuffer* is a buffer that the push procedure uses to receive blocks of data it stores in *globalPipeData*.

The **ReceiveLongs** function declares a pipe and allocates memory space for the global data sink variable. In your client/server program, the data sink can be a file or data structure the client creates. In this simple example, the data source is a dynamically allocated buffer of long integers.

Before the data transfer can begin, your client program must initialize the output pipe structure. It must set pointers to the state variable, the push procedure, and the alloc procedure. In this example, the output pipe variable is called *outputPipe*.

Clients signal servers that they are ready to receive data by invoking a remote procedure on the server. In this example, the remote procedure is called **OutPipe**. When the client calls the remote procedure, the server begins the data transfer. Each time data arrives, the client stub calls the client's alloc and push procedures as needed.

Rather than allocate memory each time a buffer is needed, the alloc procedure in this example simply sets a pointer to the variable *globalBuffer*. The pull procedure then reuses this buffer each time it transfers data. More complex client programs may need to allocate a new buffer each time the server pulls data from the client.

The push procedure in this example uses the state variable to track the next position where it will store data in the global data sink buffer. It writes data from the pipe buffer into sink buffer. The client stub then receives the next block of data from the server and stores it in the pipe buffer. When all the data has been sent, the server transmits a zero-sized buffer. This cues the push procedure to stop receiving data.

Server-Side Pipe Implementation

Server programs for distributed applications that use pipes need not implement any push, pull, or alloc functions. They do need to contain procedures that clients can invoke remotely to initiate data transfers.

Implementing Input Pipes on the Server

To begin sending data to a server, a client calls one of the server's remote procedures. This procedure must repeatedly call the pull procedure in the server's stub. The MIDL compiler uses the application's IDL file to automatically generate the server's pull procedure.

Each time the server program invokes the **pull** procedure in its stub, the **pull** procedure receives blocks of data from the client. It unmarshals the data into the server's buffer. The server's remote procedure can then process this data in any way required. The loop continues until the server receives a buffer of zero length.

The following example is from the Pipedemo program contained in the samples that come with the Platform SDK. It illustrates a remote server procedure that uses a pipe to pull data from the client to the server.

```
//file: server.c (fragment)
uc_server.c
#define PIPE_TRANSFER_SIZE 100
    /* Transfer 100 pipe elements at one time */

void InPipe(LONG_PIPE     long_pipe )
{
    long local_pipe_buf[PIPE_TRANSFER_SIZE];
    ulong actual_transfer_count = PIPE_TRANSFER_SIZE;

    while(actual_transfer_count > 0) /* Loop to get all
                                        the pipe data elements */
    {
        long_pipe.pull( long_pipe.state,
                        local_pipe_buf,
                        PIPE_TRANSFER_SIZE,
                        &actual_transfer_count);
        /* process the elements */
    } // end while
} //end InPipe
```

Implementing Output Pipes on the Server

To begin receiving data from a server, a client calls one of the server's remote procedures. This procedure must repeatedly call the push procedure in the server's stub. The MIDL compiler uses the application's IDL file to automatically generate the server's push procedure.

The remote server routine must fill the output pipe's buffer with data before it calls the push procedure. Each time the server program invokes the push procedure in its stub, the push procedure marshals the data and transmits it to the client. The loop continues until the server sends a buffer of zero length.

The following example is from the Pipedemo program contained in the samples that come with the Platform SDK. It illustrates a remote server procedure that uses a pipe to push data from the server to the client.

```
void OutPipe(LONG_PIPE *outputPipe )
{
    long *outputPipeData;
    ulong index = 0;
    ulong elementsToSend = PIPE_TRANSFER_SIZE;

    /* Allocate memory for the data to be passed back in the pipe */
    outputPipeData = (long *)malloc( sizeof(long) * PIPE_SIZE );
    while(elementsToSend >0) /* Loop to send pipe data elements */
    {
        if (index >= PIPE_SIZE)
            elementsToSend = 0;
        else
        {
            if ( (index + PIPE_TRANSFER_SIZE) > PIPE_SIZE )
                elementsToSend = PIPE_SIZE - index;
            else
                elementsToSend = PIPE_TRANSFER_SIZE;
        }

        outputPipe->push( outputPipe->state,
                          &(outputPipeData[index]),
                          elementsToSend );
        index += elementsToSend;

    } //end while

    free((void *)outputPipeData);
}
```

Rules for Multiple Pipes

You can combine **[in]**, **[out]**, and **[in**, **out]** pipe parameters in any combination in a single call, but you must process the pipes in a specific order, as shown in the following pseudocode example:

- Get the data from every input pipe, starting with the first (leftmost) **[in]** parameter, and continuing in order, draining each pipe before beginning to process the next.
- After every input pipe has been completely processed, send the data for the output pipes, again starting with the first **[out]** parameter, and continuing in order, filling each pipe before beginning to process the next.

```
//in .IDL file:
void InOutUCharPipe( [in,out] UCHAR_PIPE *uchar_pipe_1,
                     [out] UCHAR_PIPE * uchar_pipe_2,
                     [in] UCHAR_PIPE uchar_pipe_3);

//remote procedure:
void InOutUCharPipe( UCHAR_PIPE *param1,
                     UCHAR_PIPE *param2,
                     UCHAR_PIPE  param3)
```

(continued)

(continued)

```
{
    while(!END_OF_PIPE1)
    {
        param1->pull (. . .);
        . . .
    };

    while(!END_OF_PIPE3)
    {
        param3.pull (. . .);
        . . .
    };

    while(!END_OF_PIPE1)
    {
        param1->push (. . .);
        . . .
    };

    while(!END_OF_PIPE2)
    {
        param2->push(. . .);
        . . .
    };
} //end InOutUCharPipe
```

Combining Pipe and Nonpipe Parameters

When you combine pipe types and other types in a remote procedure call, the data is transmitted according to the direction of the parameter:

- In the **[in]** direction, the data for all nonpipe arguments is transmitted first, followed by pipe data.
- In the **[out]** direction, the server sends the pipe data first. After the manager routine returns, the server transmits the nonpipe data.
- When there are **[in,out]** pipe arguments combined with **[in,out]** non-pipe arguments, first the input data is transmitted in its entirety, as previously described. Then, the output data is transmitted as previously described.

The following restriction applies to this (MIDL 3.0) implementation of pipes: When you combine pipe types and other types in a single remote procedure call, the nonpipe parameters must have a well-defined size in order to allow the MIDL compiler to calculate the buffer size needed. For example, you cannot combine pipe parameters with a **[unique]** pointer or a conformant structure, since their sizes cannot be determined at compile time.

CHAPTER 15

Binding and Handles

This chapter explains creating and using bindings and binding handles between client and server programs. It also discusses client/server contexts and context handles.

Note In addition to binding and context handles, Microsoft® RPC also supports serialization handles used to encode or decode data. These are used for serialization on a local computer and do not involve remote binding. For additional information on serialization handles, see *Serialization Services*.

Binding Handles

Binding is the process of creating a logical connection between a client program and a server program. The information that composes the binding between client and server is represented by a structure called a binding handle.

A binding handle is analogous to a file handle that the **fopen** C run-time library function returns, or a window handle that the function **CreateWindow** returns. As with these handles, your application cannot directly access and manipulate the information in the binding handle. The information in a binding handle data structure is available only to the RPC run-time libraries. You provide the handle, the run-time libraries access and manipulate the appropriate data.

Types of Binding Handles

Binding handles can be automatic, implicit, or explicit. The difference between these binding handle types is in how much control you want your application to have over the binding process. As the name suggests, automatic binding handles automate binding. The client and server applications do not need code to handle the binding process.

Implicit binding handles allow client programs to configure the binding handle before the binding takes place. After the client establishes a binding, the RPC run-time library handles the rest.

Explicit binding handles move complete control over the binding process into the source code of the client and the server programs. With this control comes increased complexity. Your application must call RPC functions to manage the binding. It does not happen automatically.

Figure 15-1 illustrates the differences between automatic, implicit, and explicit binding handles.

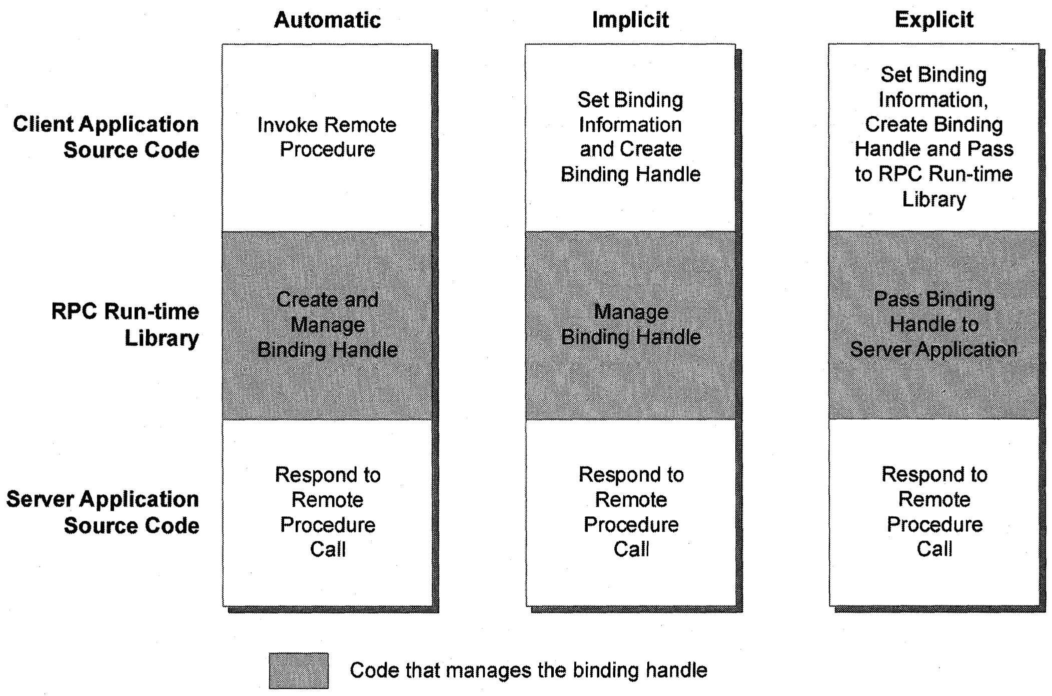

Automatic
Implicit
Explicit
Client Application Source Code
RPC Run-time Library
Server Application Source Code
Invoke Remote Procedure
Set Binding Information and Create Binding Handle
Set Binding Information, Create Binding Handle and Pass to RPC Run-time Library
Create and Manage Binding Handle
Manage Binding Handle
Pass Binding Handle to Server Application
Respond to Remote Procedure Call
Respond to Remote Procedure Call
Respond to Remote Procedure Call
Code that manages the binding handle

Figure 15-1: Differences Between Binding Handles.

In addition, every binding handle is either primitive or custom. Each of these types of binding handles are discussed in the following topics:

- Automatic Binding Handles
- Implicit Binding Handles
- Explicit Binding Handles
- Primitive and Custom Binding Handles

Automatic Binding Handles

Automatic binding handles are useful when the application does not require a specific server and when it does not need to maintain any state information between the client and server. When you use an automatic binding handle, you do not have to write any client application code to deal with binding and handles—you simply specify the use of the automatic binding handle in the Application Configuration File (ACF). The stub then defines the handle and manages the binding.

For example, a time-stamp operation can be implemented using an auto handle. It makes no difference to the client application which server provides it with the time stamp because it can accept the time from any available server.

Note Auto handles are not supported for the Macintosh platform.

You specify the use of auto handles by including the **[auto_handle]** attribute in the ACF. The time-stamp example uses the following ACF:

```
/* ACF file */
[
  auto_handle
]
interface autoh
{
}
```

When the ACF does not include any other handle attribute, and when the remote procedures do not use explicit handles, the MIDL compiler uses automatic handles by default. It also uses automatic handles as the default when the ACF is not present.

The remote procedures are specified in the IDL file. The auto handle must not appear as an argument to the remote procedure. For example:

```
/* IDL file */
[
  uuid (6B29FC40-CA47-1067-B31D-00DD010662DA),
  version(1.0),
  pointer_default(unique)
]
interface autoh
{
  void GetTime([out] long * time);
  void Shutdown(void);
}
```

The benefit of the auto handle is that the developer does not have to write any code to manage the handle; the stubs manage the binding automatically. This is significantly different from the Hello, World example, where the client manages the implicit primitive handle defined in the ACF and must call several run-time functions to establish the binding handle.

Implicit Binding Handles

Implicit binding handles allow your application to select a specific server to execute its remote procedure calls. For details, see *Client-Side Binding*. They also enable your client/server program to use authenticated bindings. That is, the client can specify authentication information in an implicit binding handle. The RPC run-time library uses the authentication information to establish an authenticated RPC session between the client and the server. For more information, see *Security*.

When your application uses implicit bindings, the client must set the binding information so that it can create the binding. After the client creates an implicit binding, it does not need to pass any binding handles to remote procedures. The RPC library handles the rest of the mechanics of the communication session.

The client stores the binding information for an implicit handle in a global variable. When the MIDL compiler generates the client stub and header file from the interface specification in your MIDL file, it also generates code for a global binding handle variable. Your client program initializes the handle and then does not refer to it again until it destroys the binding.

You create an implicit handle by specifying the **[implicit_handle]** attribute in the ACF for an interface as follows:

```
/* ACF file (complete) */

[
  implicit_handle(handle_t hHello)
]
interface hello
{
}
```

The **handle_t** type, which is used in the preceding example, is a MIDL data type used for defining binding handles.

After creating the implicit handle, the application needs to use it as a parameter to the RPC run-time library functions. Do not use the implicit handle as a parameter to remote procedure calls. The following code sample demonstrates the use of implicit binding handles.

```
RPC_STATUS status;
status = RpcBindingFromStringBinding(
            pszStringBinding,
            &hHello);

status = MyRemoteProcedure();

status = RpcBindingFree(hHello);
```

In the preceding example, the RPC run-time library functions **RpcBindingFromStringBinding** and **RpcBindingFree** both required the implicit binding handle to be passed in their parameter lists. However, the remote procedure **MyRemoteProcedure** did not, since it is not an RPC run-time library function.

Explicit Binding Handles

For maximum control over the binding process, client/server applications may use explicit binding handles. Like implicit handles, explicit binding handles enable your client application to select a server to execute its calls. In addition, explicit binding handles enable your client/server application to create an authenticated RPC communication session. With explicit handles, your client can connect to more than one server and execute remote procedures on multiple servers. Multithreaded and asynchronous client applications can even connect to multiple servers and execute multiple remote procedures at the same time.

Your client application must pass the explicit handle as a parameter to each remote procedure call. To conform to the OSF standard, the handle should be specified as the first parameter on each remote procedure. However, the Microsoft extensions to RPC enable you to specify the binding handle in other positions. For details, see *Microsoft RPC Binding-Handle Extensions*.

To create an explicit handle, declare the handle as a parameter to the remote operations in the IDL file. The Hello, World example can be redefined to use an explicit handle as shown:

```
/* IDL file for explicit handles */

[
  uuid(20B309B1-015C-101A-B308-02608C4C9B53),
  version(1.0)
]
interface hello
{
  void HelloProc([in] handle_t h1,
                 [in, string] char *  pszString);
}
```

You can combine explicit and implicit handles in a single interface. If a function has an explicit handle in its parameter list, that handle will be used. If a function in an interface using implicit handles does not specify an explicit handle, then the default implicit handle will be used.

Primitive and Custom Binding Handles

All handles declared with the **handle_t** or **RPC_BINDING_HANDLE** types are primitive binding handles. You can extend the **handle_t** or **RPC_BINDING_HANDLE** types to include more or different information than the primitive handle type contains. When you do, you create a custom binding handle.

To make a custom binding handle for your distributed application, you will need to create your own data type and specify the **[handle]** attribute on a type definition in your IDL file. Ultimately, the stub files map custom binding handles to primitive handles.

If you do create your own binding handle type, you must also supply bind and unbind routines that the client stub uses to map a custom handle to a primitive handle. The stub calls your bind and unbind routines at the beginning and end of each remote procedure call. The bind and unbind routines must conform to the following function prototypes.

Function prototype	Description
handle_t *type*_**bind**(*type*)	Binding routine
void *type*_**unbind**(*type*, **handle_t**)	Unbinding routine

The following example shows how a custom binding handle can be defined in the IDL file:

```
/* usrdef.idl */
[
  uuid(20B309B1-015C-101A-B308-02608C4C9B53),
  version(1.0),
  pointer_default(unique)
]
interface usrdef
{
  typedef struct _DATA_TYPE
  {
      unsigned char * pszUuid;
      unsigned char * pszProtocolSequence;
      unsigned char * pszNetworkAddress;
      unsigned char * pszEndpoint;
      unsigned char * pszOptions;
  } DATA_TYPE;

  typedef [handle] DATA_TYPE * DATA_HANDLE_TYPE;
  void UsrdefProc([in] DATA_HANDLE_TYPE  hBinding,
                  [in, string] unsigned char *   pszString);

  void Shutdown([in] DATA_HANDLE_TYPE hBinding);
}
```

The programmer-defined bind and unbind routines appear in the client application. In the following example, the bind routine calls **RpcBindingFromStringBinding** to convert the string-binding information to a binding handle. The unbind routine calls **RpcBindingFree** to free the binding handle.

The name of the programmer-defined binding handle, DATA_HANDLE_TYPE, appears as part of the name of the functions. It is also used as the parameter type in the function parameters.

```
/* The client stub calls this _bind routine at the */
/* beginning of each remote procedure call             */

RPC_BINDING_HANDLE __RPC_USER DATA_HANDLE_TYPE_bind(
    DATA_HANDLE_TYPE dh1)
{
    RPC_BINDING_HANDLE hBinding;
    RPC_STATUS status;

    unsigned char *pszStringBinding;

    status = RpcStringBindingCompose(
          dh1.pszUuid,
          dh1.pszProtocolSequence,
          dh1.pszNetworkAddress,
          dh1.pszEndpoint,
          dh1.pszOptions,
          &pszStringBinding);
          ...

    status = RpcBindingFromStringBinding(
          pszStringBinding,
          &hBinding);
          ...

    status = RpcStringFree(&pszStringBinding);
    ...

    return(hBinding);
}

/* The client stub calls this _unbind toutine */
/* after each remote procedure call.                   */
void __RPC_USER DATA_HANDLE_TYPE_unbind(
    DATA_HANDLE_TYPE dh1,
    RPC_BINDING_HANDLE h1)
{
    RPC_STATUS status;
    status = RpcBindingFree(&h1);
    ...
}
```

Both implicit and explicit binding handles can either be primitive or custom handles. That is, a handle may be:

- Primitive and implicit
- Custom and implicit
- Primitive and explicit
- Custom and explicit

Client-Side Binding

Binding handles are composed of a protocol sequence, the name or address of a server program host computer, and a server program endpoint. Therefore, your client program must obtain or provide this information to create a binding.

If your client program uses automatic binding handles, you do not need to write any special source code in it to create or manage the binding. The client stub calls the RPC functions that are required to establish and maintain the binding. All you have to do is specify that your client uses automatic binding handles in the Application Configuration File (ACF), and design the interface accordingly. For details, see *Automatic Binding Handles*.

Suppose, for example, that you were developing a client program that called remote time-stamping functions. Here, the stubs do all the work and the client only needs to include the generated header file Auto.h to obtain the function prototypes for the remote procedures. The client application calls to the remote procedures appear just as if they were calls to local procedures, as shown in the following example:

```
/* auto handle client application (fragment) */

#include <stdio.h>
#include <time.h>
#include "auto.h"    // header file generated by the MIDL compiler


void main(int argc, char **argv)
{
    time_t t1;
    time_t t2;
    char * pszTime;
    ...

    GetTime(&t1);  // GetTime is a remote procedure
    GetTime(&t2);

    pszTime = ctime(&t1);
    printf("time 1= %s\n", pszTime);
    pszTime = ctime(&t2);
    printf("time 2= %s\n", pszTime);

    Shutdown();     // Shutdown is a remote procedure
    exit(0);
}
```

As you can see in the preceding example, the client application does not have to make any explicit calls to the RPC run-time library functions. The client stub manages them.

If your application uses implicit or explicit binding handles, the client must obtain the binding information and call the RPC functions to create the handles. Where the client obtains the binding information from depends on the requirements of your application. The setup program that installs your client application can store binding information in environment variables that it creates. It can also save binding information in an application-specific configuration file. Since binding information in environment variables or configuration files is usually stored as strings, your client application will need to convert the string to a binding. For more information, see *Using String Bindings*.

Most networks have a name service. Server programs can advertise themselves in the name-service database. When a client begins execution, it can obtain its binding information from the name-service database. For details, see *Importing from Name Service Databases*.

The steps required for binding with implicit and explicit handles are discussed in the following topics:

- Selecting a Protocol Sequence
- Finding Server Host Systems
- Finding Endpoints

For a brief overview of these topics, see *Connecting the Client and the Server.*

Selecting a Protocol Sequence

A protocol sequence is the language that a network operating system uses to talk over the network to other computers. In more specific terms, RPC applications must specify a string that represents a combination of an RPC protocol, a transport protocol, and a network protocol.

Microsoft RPC supports three RPC protocols:

- Network Computing Architecture connection-oriented protocol (NCACN)
- Network Computing Architecture datagram protocol (NCADG)
- Network Computing Architecture local remote procedure call (NCALRPC)

RPC applications can use the NCALRPC protocol to invoke procedures offered by server programs running on the same computer that the client program runs on. Developers often use this capability for debugging.

The transport and network protocols that your application uses depend on what protocols the network supports. Many networks today, including the Internet, support TCP/IP. Other common transport and network protocols are IPX/SPX, NetBIOS, and AppleTalk DSP. Microsoft RPC supports these and other transport and network protocols. For a complete list, see ***PROTSEQ***.

When your application uses automatic binding handles, it does not need to specify the protocol sequence. If it uses implicit or explicit handles, it must obtain or specify the protocol sequence. The preferred method is for the server program to advertise its host address and protocol sequence in a name-service database. The client can then query the name service to set up a binding handle. For details, see *Importing from Name Service Databases*.

Clients can also specify protocol sequence information that they obtain from environment variables created and initialized by the setup program, from application-specific configuration files, or from literal strings in the program source code.

In addition, your client program can invoke **RpcNetworkInqProtseqs** to query the RPC run-time library for all of the protocol sequences that it and the network both support. After your client obtains the list of possible protocol sequences, it can call **RpcNetworkIsProtseqValid** to see which protocol sequence it can use to connect to the server.

After your client program has a valid protocol sequence string, it can pass that information to the **RpcStringBindingCompose** and **RpcBindingFromStringBinding** functions to create the binding handle.

Finding Server Host Systems

A server host system is the computer that executes the distributed application's server program. There may be one or many server host systems on a network. How your client program finds a server to connect to depends on the needs of your program.

There are two methods of finding server host systems. The first is to query a name-service database for the location of a server program. The second is to use information stored in strings in the client source code, environment variables, or application-specific configuration files. Your client application can use the data in the string to compose a binding between the client and the server.

This section presents information on both of these techniques in the following topics:

- Importing from Name Service Databases
- Using String Bindings

Importing from Name Service Databases

The preferred method of finding server host systems on a network is to query a name-service database. This gives both the distributed application and the network administrator greater flexibility. If it queries the name-service database, the distributed application can be more easily ported from network to network. Network administrators can more easily move server programs from host to host, as needed.

Name-service databases store, among other things, binding handles and UUIDs. Your client application can search for either or both of these when it needs to bind to the server. For a discussion of the information that a name service stores, and the storage format, see *The RPC Name-Service Database*.

The RPC library provides two sets of functions that your client program can use to search the name-service database. The names of one set begin with **RpcNsBindingImport**. The names of the other set begin with **RpcNsBindingLookup**. The difference between the two groups of functions is that the **RpcNsBindingImport** functions return a single binding handle per call and the **RpcNsBindingLookup** functions return groups of handles per call.

To begin a search with the **RpcNsBindingImport** functions, first call **RpcNsBindingImportBegin**, as shown in the following code fragment.

```
RPC_STATUS status;
RPC_NS_HANDLE hNameServiceHandle;

status = RpcNsBindingImportBegin(
    RPC_C_NS_SYNTAX_DEFAULT,
    NULL,
    MyInterface_v1_0_c_ifspec,
    NULL,
    &hNameServiceHandle);
```

When the RPC functions search the name-service database, they need a place to begin the search. In RPC terminology, this is called the entry name. Your client program passes the entry name as the second parameter to **RpcNsBindingImportBegin**. This parameter can be NULL if you want to search the entire name-service database. Alternatively, you can search the server entry by passing a server-entry name or search the group entry by passing a group-entry name. Passing an entry name restricts the search to the contents of that entry.

In the preceding example, the value RPC_C_NS_SYNTAX_DEFAULT is passed as the first parameter to **RpcNsBindingImportBegin**. This selects the default entry name syntax. Currently, this is the only entry-name syntax that Windows NT/Windows 2000 supports.

Your client application can search the name-service database for an interface name, a UUID, or both. If you want to have it search for an interface by name, pass the global interface variable that the MIDL compiler generates from your IDL file as the third parameter to **RpcNsBindingImportBegin**. You'll find its declaration in the header file that the MIDL compiler generated when it generated the client stub. If you want your client program to search by UUID only, set the third parameter to NULL.

When searching the name-service database for a UUID, set the fourth parameter of **RpcNsBindingImportBegin** to the UUID that you want to search for. If you are not searching for a UUID, set this parameter to NULL.

The **RpcNsBindingImportBegin** function passes the address of a name service–search context handle through its fifth parameter. You pass this parameter to other **RpcNsBindingImport** functions.

In particular, the next function your client application would call is **RpcNsBindingImportNext**. Client programs use this function to retrieve compatible binding handles from the name-service database. The following code fragment demonstrates how this function might be called:

```
RPC_STATUS status;
RPC_BINDING_HANDLE hBindingHandle;
// The variable hNameServiceHandle is a valid name service search
// context handle obtained from the RpcNsBindingBegin function.

status = RpcNsBindingImportNext(hNameServiceHandle, &hBindingHandle);
```

Once it has called the **RpcNsBindingImportNext** function to obtain a binding handle, your client application can determine if the handle it received is acceptable. If not, your client program can execute a loop and call **RpcNsBindingImportNext** again to see if the name service contains a more appropriate handle. For each call to **RpcNsBindingImportNext**, there must be a corresponding call to **RpcNsBindingFree**. When your search is complete, call the **RpcNsBindingImportDone** function to free the lookup context.

After your client application has an acceptable binding handle, it should check to ensure that the server application is running. There are two methods your client can use to perform this verification. The first is to call a function in the client interface. If the server program is running, the call will complete. If not, the call will fail. A better way to verify that the server is running is to invoke **RpcEpResolveBinding**, followed by a call to **RpcMgmtIsServerListening**. For more information on the name-service database, see *The RPC Name-Service Database*.

Using String Bindings

Applications can create bindings from information stored in strings. Your client application composes this information as a string, then calls the **RpcBindingFromStringBinding** function. The client must supply the following information to identify the server:

- The interface name, the Globally Unique Identifier (GUID) of the object, or UUID of the object. For more information, see *Generating Interface UUIDs* and *String UUID*.
- The transport type to communicate over, such as named pipes or TCP/IP. For details, see *Essential RPC Binding Terminology* and *Selecting a Protocol Sequence*.
- The network address or the name of the server host computer.
- The endpoint of the server program on the server host computer. For more information, see *Finding Endpoints*, and *Specifying Endpoints*.

(The object UUID and the endpoint information are optional.)

In the following examples, the *pszNetworkAddress* parameter and other parameters include embedded backslashes. The backslash is an escape character in the C programming language. Two backslashes are needed to represent each single literal backslash character. The string-binding structure must contain four backslash characters to represent the two literal backslash characters that precede the server name.

The following example shows that the server name must be preceded by eight backslashes so that four literal backslash characters will appear in the string-binding data structure after the **sprintf** function processes the string.

```
/* client application */

char * pszUuid = "6B29FC40-CA47-1067-B31D-00DD010662DA";
char * pszProtocol = "ncacn_np";
char * pszNetworkAddress = "\\\\\\\\servername";
char * pszEndpoint = "\\\\pipe\\\\pipename";
char * pszString;

int len = 0;

len  = sprintf(pszString, "%s", pszUuid);
len += sprintf(pszString + len, "@%s:", pszProtocolSequence);
if (pszNetworkAddress != NULL)
    len += sprintf(pszString + len, "%s", pszNetworkAddress);
len += sprintf(pszString + len, "[%s]", pszEndpoint);
```

In the following example, the string binding appears as:

```
6B29FC40-CA47-1067-B31D-00DD010662DA@ncacn_np:\\\\servername[\\pipe\\pipename]
```

The client then calls **RpcBindingFromStringBinding** to obtain the binding handle:

```
RPC_BINDING_HANDLE hBinding;

status = RpcBindingFromStringBinding(pszString, &hBinding);
...
```

A convenience function, **RpcStringBindingCompose** assembles the object UUID, protocol sequence, network address, and endpoint in the correct syntax for the call to **RpcBindingFromStringBinding**. You do not have to worry about putting the ampersand, colon, and the various components for each protocol sequence in the right place; you just supply the strings as parameters to the function. The run-time library even allocates the memory needed for the string binding.

```
char * pszNetworkAddress = "\\\\server";
char * pszEndpoint = "\\pipe\\pipename";
status = RpcStringBindingCompose(
            pszUuid,
```

(continued)

(continued)

```
          pszProtocolSequence,
          pszNetworkAddress,
          pszEndpoint,
          pszOptions,
          &pszString);
...
status = RpcBindingFromStringBinding(
          pszString,
          &hBinding);
...
```

Another convenience function, **RpcBindingToStringBinding**, takes a binding handle as input and produces the corresponding string binding.

Finding Endpoints

Server programs listen to endpoints for client requests. The syntax of the endpoint string depends on the protocol sequence you use. For example, the endpoint for TCP/IP is a port number, and the endpoint syntax for named pipes is a valid pipe name.

There are two types of endpoints: well-known and dynamic. Your choice of which type of endpoint your program uses determines whether the distributed application or the run-time library specifies the endpoint.

This section discusses endpoints and presents information on how to find them. It is organized into the following topics:

- Using Well-Known Endpoints
- Using Dynamic Endpoints

Note that it is possible for your client application to use the endpoint map to determine whether or not a server program is currently running. Your client can call **RpcMgmtInqIfIds**, **RpcMgmtEpEltInqBegin**, and **RpcMgmtEpEltInqDone** to see if the server has registered the particular interface it requires in the endpoint map.

Using Well-Known Endpoints

Well-known endpoints are pre-assigned endpoints that the server program uses every time it runs. Because the server always listens to that particular endpoint, the client always attempts to connect to it. Well-known endpoints are usually assigned by the authority responsible for the transport protocol. Because server host computers have a finite number of available endpoints, well-known endpoints are not recommended for most applications.

A distributed application can specify a well-known endpoint in a string and pass that string as a parameter to the function **RpcServerUseProtseqEp**. Alternatively, the endpoint string can appear in the IDL file interface header as part of the **[endpoint]** interface attribute.

You can use two approaches to implement the well-known endpoint:

- Specify all information in a string binding
- Store the well-known endpoint in the name-service database

You can write all of the information needed to establish a binding into a distributed application when you develop it. The client can specify the well-known endpoint directly in a string, call **RpcStringBindingCompose** to create a string that contains all the binding information, and supply this string to the function **RpcBindingFromStringBinding** to obtain a handle. The client and server can be hard-coded to use a well-known endpoint, or written so that the endpoint information comes from the command line, a data file, a configuration file, or the IDL file.

Your client application can also query a name-service database for well-known endpoint information.

Using Dynamic Endpoints

The number of communication ports for a particular server are usually limited. For example, when you use the **ncacn_nb_nb** protocol sequence, indicating that RPC network communication occurs using NetBIOS over NetBEUI, only 234 ports are available. The RPC run-time libraries allow you to assign endpoints dynamically, as needed.

By default, the RPC run-time library functions search for endpoint information when they query a name-service database. If the endpoint is dynamic, the name-service database will not contain endpoint information. However, the query will give your client program the name of a server. It can then search the server's endpoint map.

The client can instruct the RPC library to search the endpoint map and resolve a binding by invoking the **RpcEpResolveBinding** function. If you need more specific control over endpoint selection, you can make your client search the endpoint map one entry at a time by calling the **RpcMgmtEpEltInqBegin**, **RpcMgmtEpEltInqNext**, and **RpcMgmtEpEltInqDone** functions.

Server-Side Binding

This section presents a discussion of a server program's role in providing binding information to its clients.

Registering Interfaces

This section presents a detailed discussion of the process of registering an RPC interface. For an overview of registering server interfaces, see *Registering the Interface.*

Interface Registration Functions

Servers register their interfaces by calling the **RpcServerRegisterIf** function. Complex server programs often support more than one interface. Server applications must call this function once for each interface they support.

Also, servers can support multiple versions of the same interface, each with its own implementation of the interface's functions. If your server program does this, it must provide a set of entry points. An entry point is a manager routine that dispatches calls for a version of an interface. There must be one entry point for each version of the interface. The group of entry points is called an entry point vector. For details, see *Entry-Point Vectors*.

In addition to the standard function **RpcServerRegisterIf,** Microsoft RPC also supports other interface registration functions. The **RpcServerRegisterIf2** function extends the capabilities of **RpcServerRegisterIf** by enabling you to specify a set of registration flags (see *Interface Registration Flags*), the maximum number of concurrent remote procedure call requests the server can accept, and the maximum size in bytes of incoming data blocks.

The Microsoft RPC library also contains a function called **RpcServerRegisterIfEx**. Like the **RpcServerRegisterIf** function, this function registers an interface. Your server program can also use this function to specify a set of registration flags (see *Interface Registration Flags*), the maximum number of concurrent remote procedure call requests the server can accept, and a security callback function.

The **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, and **RpcServerRegisterIf2** functions set values in the internal interface registry table. This table is used to map the interface UUID and object UUIDs to a manager EPV. The manager EPV is an array of function pointers that contains exactly one function pointer for each function prototype in the interface specified in the IDL file.

For information on supplying multiple EPVs to provide multiple implementations of the interface, see *Multiple Interface Implementations*.

The run-time library uses the interface registry table (set by calls to the function **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, or **RpcServerRegisterIf2**) and the object registry table (set by calls to the function **RpcObjectSetType**) to map interface and object UUIDs to the function pointer.

When you want your server program to remove an interface from the RPC run-time library registry, call the **RpcServerUnregisterIf** function. After the interface is removed from the registry, the RPC run-time library will no longer accept new calls for that interface.

Entry-Point Vectors

The manager Entry-Point Vector (EPV) is an array of function pointers that point to implementations of the functions specified in the IDL file. The number of elements in the array corresponds to the number of functions specified in the IDL file. Microsoft RPC supports multiple entry-point vectors representing multiple implementations of the functions specified in the interface.

The MIDL compiler automatically generates a manager EPV data type for use in constructing manager EPVs. The data type is named *if-name*_**SERVER_EPV**, where *if-name* specifies the interface identifier in the IDL file.

The MIDL compiler automatically creates and initializes a default manager EPV on the assumption that a manager routine of the same name exists for each procedure in the interface and is specified in the IDL file.

When a server offers multiple implementations of the same interface, the server must create one additional manager EPV for each implementation. Each EPV must contain exactly one entry point (address of a function) for each procedure defined in the IDL file. The server application declares and initializes one manager EPV variable of type *if-name*_**SERVER_EPV** for each additional implementation of the interface. To register the EPVs it calls **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, or **RpcServerRegisterIf2** once for each object type it supports.

When the client makes a remote procedure call to the server, the EPV containing the function pointer is selected based on the interface UUID and the object type. The object type is derived from the object UUID by the object-inquiry function or the table-driven mapping controlled by **RpcObjectSetType**.

Manager EPVs

By default, the MIDL compiler uses the procedure names from an interface's IDL file to generate a manager EPV, which the compiler places directly into the server stub. This default EPV is statically initialized using the procedure names declared in the interface definition.

To register a manager using the default EPV, specify NULL as the value of the *MgrEpv* argument (a null EPV).If the routine names used by a manager correspond to those of the interface definition, you can register this manager using the default EPV of the interface generated by the MIDL compiler. You can also register a manager using an EPV that the server application supplies.

A server can (and sometimes must) create and register a non-null manager EPV for an interface. To select a server application–supplied EPV, pass the address of an EPV whose value has been declared by the server as the value of the *MgrEpv* argument. A non-null value for the *MgrEpv* argument always overrides a default EPV in the server stub.

The MIDL compiler automatically generates a manager EPV data type (RPC_MGR_EPV) for a server application to use in constructing manager EPVs. A manager EPV must contain exactly one entry point (function address) for each procedure defined in the IDL file.

A server must supply a non-null EPV in the following cases:

- When the names of manager routines differ from the procedure names declared in the interface definition
- When the server uses the default EPV for registering another implementation of the interface

A server declares a manager EPV by initializing a variable of type *if-name*_**SERVER_EPV** for each implementation of the interface.

Registering a Single Implementation of an Interface

When a server offers only one implementation of an interface, the server calls **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, or **RpcServerRegisterIf2** only once. In the simplest case, the server uses the default manager EPV. (The exception is when the manager uses routine names that differ from those declared in the interface.)

For the simple case, you supply the following values for calls to **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, or **RpcServerRegisterIf2**:

- Manager EPVs

  To use the default EPV, specify a null value for the *MgrEpv* argument.
- Manager type UUID

  When using the default EPV, register the interface with a nil manager type UUID by supplying either a null value or a nil UUID for the *MgrTypeUuid* argument. In this case, all remote procedure calls, regardless of the object UUID in their binding handle, are dispatched to the default EPV, assuming no **RpcObjectSetType** calls have been made.

  You can also provide a non-nil manager type UUID. In this case, you must also call the **RpcObjectSetType** routine.

Registering Multiple Implementations of an Interface

You can supply more than one implementation of the remote procedure(s) specified in the IDL file. The server application calls **RpcObjectSetType** to map object UUIDs to type UUIDs and calls **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, or **RpcServerRegisterIf2** to associate manager EPVs with a type UUID. When a remote procedure call arrives with its object UUID, the RPC server run-time library maps the object UUID to a type UUID. The server application then uses the type UUID and the interface UUID to select the manager EPV.

You can also specify your own function to resolve the mapping from object UUID to manager type UUID. You specify the mapping function when you call **RpcObjectSetInqFn**.

To offer multiple implementations of an interface, a server must register each implementation by calling **RpcServerRegisterIf**, **RpcServerRegisterIfEx** or **RpcServerRegisterIf2** separately. For each implementation a server registers, it supplies the same *IfSpec* argument, but a different pair of *MgrTypeUuid* and *MgrEpv* arguments.

In the case of multiple managers, use **RpcServerRegisterIf**, **RpcServerRegisterIfEx** or **RpcServerRegisterIf2** as follows:

- Manager EPVs

  To offer multiple implementations of an interface, a server must:

  - Create a non-null manager EPV for each additional implementation.
  - Specify a non-null value for the *MgrEpv* argument in **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, or **RpcServerRegisterIf2**.

Please note that the server can also register with the default manager EPV.

- Manager type UUID

  Provide a manager type UUID for each EPV of the interface. The nil type UUID (or null value) for the *MgrTypeUuid* argument can be specified for one of the manager EPVs. Each type UUID must be different.

Rules for Invoking Manager Routines

The RPC run-time library dispatches an incoming remote procedure call to a manager that offers the requested RPC interface. When multiple managers are registered for an interface, the RPC run-time library must select one of them. To select a manager, the RPC run-time library uses the object UUID specified by the call's binding handle.

The run-time library applies the following rules when interpreting the object UUID of a remote procedure call:

- Nil object UUIDs

  A nil object UUID is automatically assigned the nil type UUID (it is illegal to specify a nil object UUID in the **RpcObjectSetType** routine). Therefore, a remote procedure call whose binding handle contains a nil object UUID is automatically dispatched to the manager registered with the nil type UUID, if any.

- Non-nil object UUIDs

  In principle, a remote procedure call whose binding handle contains a non-nil object UUID should be processed by a manager whose type UUID matches the type of the object UUID. However, identifying the correct manager requires that the server has specified the type of that object UUID by calling the **RpcObjectSetType** routine.

  If a server fails to call the **RpcObjectSetType** routine for a non-nil object UUID, a remote procedure call for that object UUID goes to the manager EPV that services remote procedure calls with a nil object UUID (that is, the nil type UUID).

  Remote procedure calls with a non-nil object UUID in the binding handle cannot be executed if the server assigned that non-nil object UUID a type UUID by calling the **RpcObjectSetType** routine, but did not also register a manager EPV for that type UUID by calling **RpcServerRegisterIf**, **RpcServerRegisterIfEx** or **RpcServerRegisterIf2**.

The following table summarizes the actions that the run-time library uses to select the manager routine.

Object UUID of call	Server set type for object UUID?	Server registered EPV type?	Dispatching action
Nil	Not applicable	Yes	Uses the manager with the nil type UUID.
Nil	Not applicable	No	Error (RPC_S_UNSUPPORTED_TYPE); rejects the remote procedure call.

(continued)

(continued)

Object UUID of call	Server set type for object UUID?	Server registered EPV type?	Dispatching action
Non-nil	Yes	Yes	Uses the manager with the same type UUID.
Non-nil	No	Ignored	Uses the manager with the nil type UUID. If no manager with the nil type UUID, error (RPC_S_UNSUPPORTEDTYPE); rejects the remote procedure call.
Non-nil	Yes	No	Error (RPC_S_UNSUPPORTEDTYPE); rejects the remote procedure call.

The object UUID of the call is the object UUID found in a binding handle for a remote procedure call.

The server sets the type of the object UUID by calling **RpcObjectSetType** to specify the type UUID for an object.

The server registers the type for the manager EPV by calling **RpcServerRegisterIf**, **RpcServerRegisterIfEx** or **RpcServerRegisterIf2** using the same type UUID.

Note The nil object UUID is always automatically assigned the nil type UUID. It is illegal to specify a nil object UUID in the **RpcObjectSetType** routine.

Dispatching a Remote Procedure Call to a Server-Manager Routine

The following tables show the steps that the RPC run-time library takes to dispatch a remote procedure call to a server-manager routine.

A simple case where the server registers the default manager EPV, is described in the following tables.

Interface Registry Table

Interface UUID	Manager type UUID	Entry-point vector
uuid1	Nil	Default EPV

Object Registry Table

Object UUID	Object type
Nil	Nil
(Any other object UUID)	Nil

Mapping the Binding Handle to an Entry-Point Vector (EPV)

Interface UUID (from client binding handle)	Object UUID (from client binding handle)	Object type (from object registry table)	Manager EPV (from interface registry table)
uuid1	Nil	Nil	Default EPV
Same as above	*uuidA*	Nil	Default EPV

The following steps describe the actions that the RPC server's run-time library take, as shown in the preceding tables, when a client with interface UUID *uuid1* calls it.

1. The server calls **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, or **RpcServerRegisterIf2** to associate an interface it offers with the nil manager type UUID and with the MIDL-generated default manager EPV. This call adds an entry in the interface registry table. The interface UUID is contained in the *IfSpec* argument.
2. By default, the object registry table associates all object UUIDs with the nil type UUID. In this example, the server does not call **RpcObjectSetType**.
3. The server run-time library receives a remote procedure code containing the interface UUID that the call belongs to and the object UUID from the call's binding handle.

   See the following function reference entries for discussions of how an object UUID is set into a binding handle:

   - **RpcNsBindingImportBegin**
   - **RpcNsBindingLookupBegin**
   - **RpcBindingFromStringBinding**
   - **RpcBindingSetObject**
4. Using the interface UUID from the remote procedure call, the server's run-time library locates that interface UUID in the interface registry table.

   If the server did not register the interface using **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, or **RpcServerRegisterIf2**, then the remote procedure call returns to the caller with an RPC_S_UNKNOWN_IF status code.
5. Using the object UUID from the binding handle, the server's run-time library locates that object UUID in the object registry table. In this example, all object UUIDs map to the nil object type.
6. The server's run-time library locates the nil manager type in the interface registry table.
7. Combining the interface UUID and nil type in the interface registry table resolves to the default EPV, which contains the server-manager routines to be executed for the interface UUID found in the remote procedure call.

Assume that the server offers multiple interfaces and multiple implementations of each interface, as described in the following tables.

Interface Registry Table

Interface UUID	Manager-type UUID	Entry-point vector
uuid1	Nil	*epv1*
uuid1	*uuid3*	*epv4*
uuid2	*uuid4*	*epv2*
uuid2	*uuid7*	*epv3*

Object Registry Table

Object UUID	Object type
uuidA	*uuid3*
uuidB	*uuid7*
uuidC	*uuid7*
uuidD	*uuid3*
uuidE	*uuid3*
uuidF	*uuid8*
Nil	Nil
(Any other UUID)	Nil

Mapping the Binding Handle to an Entry-Point Vector

Interface UUID (from client binding handle)	Object UUID (from client binding handle)	Object type (from object registry table)	Manager EPV (from interface registry table)
uuid1	Nil	Nil	*epv1*
uuid1	*uuidA*	*uuid3*	*epv4*
uuid1	*uuidD*	*uuid3*	*epv4*
uuid1	*uuidE*	*uuid3*	*epv4*
uuid2	*uuidB*	*uuid7*	*epv3*
uuid2	*uuidC*	*uuid7*	*epv3*

The following steps describe the actions that the server's run-time library take, as shown in the preceding tables when a client with interface UUID *uuid2* and object UUID *uuidC* calls it.

1. The server calls **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, or **RpcServerRegisterIf2** to associate the interfaces it offers with the different manager EPVs. The entries in the interface registry table reflect four calls of **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, or **RpcServerRegisterIf2** to offer two interfaces, with two implementations (EPVs) for each interface.

2. The server calls **RpcObjectSetType** to establish the type of each object it offers. In addition to the default association of the nil object to a nil type, all other object UUIDs not explicitly found in the object registry table also map to the nil type UUID.

   In this example, the server calls the **RpcObjectSetType** routine six times.
3. The server run-time library receives a remote procedure call containing the interface UUID that the call belongs to and an object UUID from the call's binding handle.
4. Using the interface UUID from the remote procedure call, the server's run-time library locates the interface UUID in the interface registry table.
5. Using the *uuidC* object UUID from the binding handle, the server's run-time library locates the object UUID in the object registry table and finds that it maps to type *uuid7*.
6. To locate the manager type, the server's run-time library combines the interface UUID, *uuid2*, and type *uuid7* in the interface registry table. This resolves to *epv3*, which contains the server manager routine to be executed for the remote procedure call.

The routines in *epv2* will never be executed because the server has not called the **RpcObjectSetType** routine to add any objects with a type UUID of *uuid4* to the object registry table.

A remote procedure call with interface UUID *uuid2* and object UUID *uuidF* returns to the caller with an RPC_S_UNKNOWN_MGR_TYPE status code because the server did not call **RpcServerRegisterIf, RpcServerRegisterIfEx**, or **RpcServerRegisterIf2** to register the interface with a manager type of *uuid8*.

Return Values

This function returns one of the following values.

Value	Meaning
RPC_S_OK	Success
RPC_S_TYPE_ALREADY_REGISTERED	Type UUID already registered

See Also

RpcBindingFromStringBinding, RpcBindingSetObject, RpcNsBindingExport, RpcNsBindingImportBegin, RpcNsBindingLookupBegin, RpcObjectSetType, RpcServerRegisterIf, RpcServerRegisterIfEx, RpcServerRegisterIf2, RpcServerUnregisterIf

Supplying Your Own Object-Inquiry Function

Consider a server that manages thousands of objects of many different types. Whenever the server started, the server application would have to call the function **RpcObjectSetType** for every one of the objects, even though clients might refer to only a few of them (or take a long time to refer to them). These thousands of objects are likely to be on disk, so retrieving their types would be time consuming. Also, the internal table that is mapping the object UUID to the manager type UUID would essentially duplicate the mapping maintained with the objects themselves.

For convenience, the RPC function set includes the function **RpcObjectSetInqFn**. With this function, you provide your own object-inquiry function.

As an example, you can supply your own object-inquiry function when you map objects 100–199 to type number 1, 200–299 to type number 2, and so on. The object inquiry function can also be extended to a distributed file system, where the server application does not have a list of all the files (object UUIDs) available, or when object UUIDs name files in the file system and you do not want to preload all of the mappings between object UUIDs and type UUIDs.

Specifying Protocol Sequences

Server applications must select one or more protocol sequences to use when communicating with the client over the network. The choice of protocol sequences is network-dependent. See *Creating Binding Information* and *Selecting a Protocol Sequence*.

Your server program should typically allow clients to connect using any protocol sequence that the network supports. To do this, invoke **RpcServerUseAllProtseqs** and pass RPC_C_PROTSEQ_MAX_REQS_DEFAULT as the first parameter.

If you want your client to restrict port allocation for dynamic endpoints to a specific port range, call **RpcServerUseAllProtseqsEx** instead. This function is specific to Microsoft RPC, and is extremely useful for remote procedure calls that pass through a firewall. It uses an extra parameter to pass port allocation control flags to the function. See *Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding*.

You can specify protocol sequences and endpoint information in your MIDL file when you develop the server's interfaces. If you do, your server should use **RpcServerUseAllProtseqsIf** to register all the protocol sequences and associated endpoint information provided in the IDL file. In addition, there is a corresponding **RpcServerUseAllProtseqsIfEx** function that also allows the server to pass port allocation-control flags.

If you want to restrict your client and server programs to communicating with a specified protocol sequence, the server application should call **RpcServerUseProtseq.** This may be particularly appropriate during debugging. For instance, you can force your application to use the **ncacn_ip_tcp** protocol to avoid the time-out problems that are introduced with other protocols when your debugger program stops your application at a breakpoint. For a complete list of Microsoft RPC protocol sequences, see ***PROTSEQ***.

Microsoft RPC also provides **RpcServerUseProtseqEx** to enable applications to select specific protocol sequences and control dynamic port allocation.

In addition to connection-oriented protocols, Microsoft RPC supports datagram (connectionless) protocols as well. Some of the features available when using datagram protocols are:

- Datagrams support the UDP and IPX connectionless transport protocols.
- Because it is not necessary to establish and maintain a connection, the datagram RPC protocol requires less resource overhead.
- Datagrams enable faster binding.
- As with connection-oriented RPC, datagram RPC calls are by default *nonidempotent*. This means the call is guaranteed not to be executed more than once. However, a function can be marked as idempotent in the IDL file telling RPC that it is harmless to execute the function more than once in response to a single, client request. This allows the run time to maintain less state on the server. Note that an idempotent call would be re-executed only in rare circumstances on an unstable network.
- Datagram RPC supports the **broadcast** IDL attribute. Broadcast enables a client to issue messages to multiple servers at the same time. This lets the client locate one of several available servers on the network, or control multiple servers simultaneously. Broadcast calls are implicitly idempotent. If the call contains [**out**] parameters, only the first server response is returned. Once a server responds, all future RPCs over that binding handle will be sent to that server only, including calls with the broadcast attribute. To send another broadcast, create a new binding handle or call **RpcBindingReset** on the existing handle.
- Datagram RPC supports the **maybe** IDL attribute. This lets the client send a call to the server without waiting for a response or confirmation. The call cannot contain [**out**] parameters. Calls using the **[maybe]** calls are implicitly idempotent.

Specifying Endpoints

An endpoint is a hardware port or named pipe that the server application listens to for client remote procedure calls. Client/server applications can use either well-known or dynamic applications. This section presents the techniques that server programs use to specify well-known and dynamic endpoints. The information is discussed in the following topics:

- Specifying Well-Known Endpoints
- Specifying Dynamic Endpoints

Specifying Well-Known Endpoints

When your server uses a well-known endpoint, it can include the endpoint data as part of its name-service database entry. If it does, the client's binding handle contains a complete server address that includes the well-known endpoint when the client imports the binding handle from the server entry.

Your server program can also specify well-known endpoints at the same time it specifies protocol sequences. Invoke either the **RpcServerUseProtseqEp** or **RpcServerUseProtseqEpEx** function. The difference between the two is that the latter function has an extra parameter your server can use to control dynamic port allocation.

If your server program specifies its endpoint information in this way, it should also call the **RpcEpRegister** function to register the endpoint in the endpoint map. Even though the endpoint is always the same, the client may use the endpoint map to find the server.

Like protocol sequences, an application can specify endpoint information in its IDL file. When it does, it should register both the protocol sequences and endpoints at the same time by invoking the **RpcServerUseAllProtseqsIf** or **RpcServerUseAllProtseqsIfEx** function. In this case, the client has access to the endpoint information through the interface specification in the IDL file. Therefore, it is not necessary to call the **RpcEpRegister** function.

Specifying Dynamic Endpoints

A dynamic endpoint is an endpoint that the server host computer assigns when the server begins execution. Most server applications use dynamic endpoints to avoid conflict with other programs over the limited number of ports that are available on the server host computer system. However, programs using named pipes or the **ncalrpc** protocol sequence have a very large endpoint name space. They benefit less from dynamic endpoints than programs using other transports.

Server programs register dynamic endpoints in an endpoint map database. If you want the server to use any available endpoint, call **RpcServerUseAllProtSeqs**. This sets the RPC run-time library to use all valid protocol sequences with dynamic endpoints. The server should then call **RpcServerInqBindings** to obtain a set of valid binding handles. The server passes the set of binding handles, or binding vector, to the function **RpcEpRegister** to register all suitable endpoints in the endpoint map. For each call your server makes to **RpcEpRegister**, there should be a corresponding call to **RpcBindingVectorFree** to release the memory that the binding vector uses.

Note that server programs can use the **RpcEpRegisterNoReplace** function rather than **RpcEpRegister**. Programs typically use **RpcEpRegisterNoReplace** when multiple copies of a server program must run on a server host system.

Both the **RpcEpRegister** and **RpcEpRegisterNoReplace** functions add the server's interfaces and binding handles to the endpoint mapper database. When the client makes a remote procedure call using a partially bound handle, the client's run-time library asks the server machine's endpoint mapper for the endpoint of a compatible server instance. The client library supplies the interface UUID, protocol sequence, and, if present, the object UUID in the client binding handle. The endpoint mapper looks for a database entry that matches the client's information. The client/server interface UUID, the interface major version, and protocol sequence must all match exactly. In addition, the server's interface minor version must be greater than or equal to the client's minor version.

If all tests are successful, the endpoint mapper returns the valid endpoint and the client run-time library updates the endpoint in the binding handle.

Dynamic endpoints expire when the server instance stops running. To remove the endpoint from the endpoint mapper database before the server program exits, call **RpcEpUnregister**.

Advertising Server Interfaces

The server side of an application that uses automatic handles must call the function **RpcNsBindingExport** to make binding information about the server available to clients. Automatic binding handles require a name service running on a server that is accessible to the client. The Microsoft implementation of the name service, Microsoft Locator, manages automatic handles. Server applications that use implicit and explicit binding handles can also advertise their presence in the name-service database.

Typically, the server calls the following run-time functions:

```
/* auto handle server application (fragment) */

//interface header file that the MIDL compiler generates
#include "auto.h"

void main(void)
{
    RpcUseProtseqEp(...);
    RpcServerRegisterIf(...);
    RpcServerInqBindings(...);
    RpcNsBindingExport(...);
    ...
}
```

The calls to the first two functions in this code fragment are similar to the Hello, World example. These functions make information about the binding available to the client. The calls to **RpcServerInqBindings** and **RpcNsBindingExport** put the information in the name-service database. The call to **RpcServerInqBindings** fills the binding vector with valid binding handles before the handles are exported to the name service. After the server program exports the handles to the database, the client (or client stubs) can call **RpcNsBindingImportBegin** and **RpcNsBindingImportNext** to obtain this information. For details, see *Finding Server Host Systems*.

The calls to **RpcServerInqBindings** and **RpcNsBindingExport** and their associated data structures look similar to the following:

```
RPC_BINDING_VECTOR * pBindingVector;
RPCSTATUS status;

status = RpcServerInqBindings(&pBindingVector);
```

(continued)

(continued)

```
status = RpcNsBindingExport(
               fNameSyntaxType,       // name syntax type
               pszAutoEntryName,      // nsi entry name
               autoh_ServerIfHandle,  // if server handle
               pBindingVector,        // set in previous call
               NULL);                 // UUID vector
```

Note that the **RpcServerInqBindings** parameter &*pBindingVector* is a pointer to a pointer to **RPC_BINDING_VECTOR**. Also remember that each call to **RpcNsBindingExport** must be followed by a call to **RpcBindingVectorFree**.

To remove the exported interface from the name-service database, the server calls **RpcNsBindingUnexport** as shown:

```
status = RpcNsBindingUnexport(
               fNameSyntaxType,
               pszAutoEntryName,
               auto_ServerIfHandle,
               NULL);               // unexport handles only
```

The **RpcNsBindingUnexport** function should be used only when the service is being permanently removed. It should not be used when the service is temporarily disabled, such as when the server is shut down for maintenance. A server program can register itself with the name-service database, yet be unavailable because the server is temporarily offline. The client application should contain exception-handling code for such a condition.

For more information on the content and format of the name-service database, see *The RPC Name-Service Database.*

Applications can utilize the Active Directory service if both the client and server programs are running under Windows 2000. The computers running the client and server programs must both be members of a Windows 2000 domain.

To advertise its presence using the Active Directory service, the server program should run in the security context of a domain administrator. If it is running in the context of domain users, a domain administrator must modify the Access Control List (ACL) on the RPC Services container. For more details, see the *Active Directory* documentation.

Listening for Remote Procedure Calls

After a server program registers its binding information and advertises its presence in a name-service database, it can begin listening to the endpoint for remote procedure calls. Server programs call the **RpcServerListen** function to monitor endpoints for client invocations of remote procedures.

The DCE specification of **RpcServerListen** indicates that it should not return until a function in the server program calls **RpcMgmtStopServerListening**. The Microsoft RPC implementation of **RpcServerListen** uses two parameters that do not appear in the DCE specification: *DontWait* and *MinimumCallThreads*. Your server program can pass a nonzero value for the *DontWait* parameter. If it does, the **RpcServerListen** function will return immediately. Use the **RpcMgmtWaitServerListen** routine to perform the wait operation usually associated with **RpcServerListen**.

Fully and Partially Bound Handles

When you use dynamic endpoints, the run-time libraries obtain endpoint information as they need it. The run-time libraries make the distinction between a *fully bound handle* (one that includes endpoint information) and a *partially bound handle* (one that does not include endpoint information).

The client run-time library must convert the partially bound handle to a fully bound handle before the client can bind to the server. The client run-time library tries to convert the partially bound handle for the client application by obtaining the endpoint information as shown:

- From the client's interface specification
- From an endpoint-mapping service running on the server

If the client tries to use a partially bound handle when the endpoint information is not available in the interface specification and the server's endpoint-mapper does not have information about the server endpoint, the client will not have enough information to make its remote procedure call and that call will fail. To prevent this, you must register the endpoint in the endpoint mapper when your distributed application uses partially bound handles. For more information about the endpoint mapper, see *Specifying Dynamic Endpoints*.

When a remote procedure call fails, the client application can call **RpcBindingReset** to remove out-of-date endpoint information. When the client tries to call the remote procedure, the client run-time library again tries to convert the fully bound handle to a partially bound handle. This is useful when the server has been stopped and restarted using a different dynamic endpoint.

Interpreting Binding Information

Microsoft RPC enables your client and server programs access to and interpret the information in a binding handle. This does not mean that you can or should try to access the contents of a binding handle directly. Microsoft RPC provides functions that set and retrieve the information in binding handles.

To get the information in a binding handle, pass the handle to **RpcBindingToStringBinding**. It returns the binding information as a string. For every call to **RpcBindingToStringBinding**, you must have a corresponding call to the function **RpcStringFree**.

You can call the function **RpcStringBindingParse** to parse the string you obtain from **RpcBindingToStringBinding**. This function allocates strings to contain the information it parses. If you do not want it to parse a particular piece of binding information, pass a NULL as the value of that parameter. Be sure to call **RpcStringFree** for each of the strings it allocates.

The following code fragment illustrates how an application might call these functions.

```
RPC_STATUS status;
UCHAR *lpzStringBinding;
UCHAR *lpzProtocolSequence;
UCHAR *lpzNetworkAddress;
UCHAR *lpzEndpoint;
UCHAR *NetworkOptions;

// The variable hBindingHandle is a valid binding handle.

status = RpcBindingToStringBinding(hBindingHandle,&lpzStringBinding);
// Code to check the status goes here.

status = RpcStringBindingParse(
    lpzStringBinding,
    NULL,
    &lpzProtocolSequence;
    &lpzNetworkAddress;
    &lpzEndpoint;
    &NetworkOptions);
// Code to check the status goes here.

// Code to analyze and alter the binding information in the strings
// goes here.

status = RpcStringFree(&lpzStringBinding);
// Code to check the status goes here.

status = RpcStringFree(&lpzProtocolSequence);
// Code to check the status goes here.

status = RpcStringFree(&lpzNetworkAddress);
// Code to check the status goes here.

status = RpcStringFree(&NetworkOptions);
// Code to check the status goes here.
```

The preceding sample code calls the functions **RpcBindingToStringBinding** and **RpcStringBindingParse** to get and parse the information in a valid binding handle. Note that the value NULL was passed as the second parameter to **RpcStringBindingParse.**

This causes that function to skip parsing the object UUID. Since it doesn't parse the UUID, **RpcStringBindingParse** will not allocate a string for it. This technique enables your application to only allocate memory for the information you are interested in parsing and analyzing.

Microsoft RPC Binding-Handle Extensions

The Microsoft extensions to the IDL language support multiple handle parameters that appear in positions other than the first, leftmost, parameter.

The following table describes the sequence of steps that the MIDL compiler goes through to resolve the binding-handle parameter in DCE-compatibility mode (/**osf**) and in default (Microsoft-extended) mode.

	DCE-compatibility mode		Default mode
1.	Binding handle that appears in first parameter position.	1.	Leftmost explicit binding handle.
2.	Leftmost **[in, context_handle]** parameter.	2.	Implicit binding handle specified by **[implicit_handle]** or **[auto_handle].**
3.	Implicit binding handle specified by **[implicit_handle]** or **[auto_handle].**	3.	If no ACF present, default to use of **[auto_handle].**
4.	If no ACF present, default to use of **[auto_handle].**		

DCE IDL compilers look for an explicit binding handle as the first parameter. When the first parameter is not a binding handle and one or more context handles are specified, the leftmost context handle is used as the binding handle. When the first parameter is not a handle and there are no context handles, the procedure uses implicit binding using the ACF attribute **[implicit_handle]** or **[auto_handle]**.

The Microsoft extensions to the IDL allow the binding handle to be in a position other than the first parameter. The leftmost **[in]** explicit-handle parameter—whether it is a primitive, programmer-defined, or context handle—is the binding handle. When there are no handle parameters, the procedure uses implicit binding using the ACF attribute **[implicit_handle]** or **[auto_handle]**.

The following rules apply to both DCE-compatibility (/**osf**) mode and default mode:

- Auto-handle binding is used when no ACF is present.
- Explicit **[in]** or **[in**, **out]** handles for an individual function pre-empt any implicit binding specified for the interface.
- Multiple **[in]** or **[in**, **out]** primitive handles are not supported.
- Multiple **[in]** or **[in**, **out]** explicit context handles are allowed.
- All programmer-defined handle parameters except the binding-handle parameter are treated as transmissible data.

The following table contains examples and describes how binding handles are assigned in each compiler mode.

Example | **Description**

```
void proc1(void );
```

No explicit handle is specified. The implicit binding handle, specified by **[implicit_handle]** or **[auto_handle]**, is used. When no ACF is present, an auto handle is used.

```
void proc2([in] handle_t H,
           [in] short s );
```

An explicit handle of type **handle_t** is specified. The parameter *H* is the binding handle for the procedure.

```
void proc3([in] short s,
           [in] handle_t H );
```

The first parameter is not a handle. In default mode, the leftmost handle parameter, *H*, is the binding handle. In /**osf** mode, implicit binding is used. An error is reported because the second parameter is expected to be transmissible, and **handle_t** cannot be transmitted.

```
typedef [handle] short * MY_HDL;

void proc1([in] short s,
           [in] MY_HDL H );
```

The first parameter is not a handle. In default mode, the leftmost handle parameter, *H*, is the binding handle. The stubs call the user-supplied routines MY_HDL_bind and MY_HDL_unbind. In/**osf** mode, implicit binding is used. The programmer-defined handle parameter *H* is treated as transmissible data.

```
Typedef [handle] short * MY_HDL;

void proc1([in] MY_HDL H,
           [in] MY_HDL p );
```

The first parameter is a binding handle. The parameter *H* is the binding-handle parameter. The second programmer-defined handle parameter is treated as transmissible data.

```
Typedef [context_handle]
void * CTXT_HDL;

void proc1([in] short s,
           [in] long l,
           [in] CTXT_HDL H ,
           [in] char c);
```

The binding handle is a context handle. The parameter *H* is the binding handle.

Binding-Handle Functions

The following table contains the list of RPC run-time routines that operate on binding handles and specifies the type of binding handle allowed.

Routine	Input argument	Output argument
RpcBindingCopy	Server	Server
RpcBindingFree	Server	None
RpcBindingFromStringBinding	None	Server
RpcBindingInqAuthClient	Client	None
RpcBindingInqAuthInfo	Server	None
RpcBindingInqObject	Server or client	None
RpcBindingReset	Server	None
RpcBindingSetAuthInfo	Server	None
RpcBindingSetObject	Server	None
RpcBindingToStringBinding	Server or client	None
RpcBindingVectorFree	Server	None
RpcNsBindingExport	Server	None
RpcNsBindingImportNext	None	Server
RpcNsBindingLookupNext	None	Server
RpcNsBindingSelect	Server	Server
RpcServerInqBindings	None	Server

The RPC Name-Service Database

A name service is a service that maps names to objects, and usually maintains the (name, object) pairs in a database. Generally, the name is a logical name that is easy for users to remember and use. For example, a name service would allow a user to use the logical name "laserprinter." The name service maps this name to the network-specific name used by the print server.

To use a simplified explanation, the RPC name service maps a name to a binding handle and maintains the (name, binding handle) mappings in the RPC name-service database. The RPC name service allows client applications to use a logical name instead of a specific protocol sequence and network address. The use of the logical name makes it easier for network administrators to install and configure your distributed application.

An RPC name-service database entry has one of the following attributes: **server**, **group**, or **profile**. In the Microsoft implementation, entries can have only one attribute, so these entries are also known as server entries, group entries, and profile entries.

The server entry consists of interface UUIDs, object UUIDs (needed when the server implements multiple-entry points), network address, protocol sequence, and any endpoint information associated with well-known endpoints. When a dynamic endpoint is used, the endpoint information is kept in the endpoint-map database rather than the name-service database, and the endpoint is resolved like any other dynamic endpoint. Server entries are managed by functions that start with the prefix **RpcNsBinding**.

The group entry can contain server entries or other group entries. Group entries are managed by functions that start with the prefix **RpcNsGroup**.

The profile entry can contain profile, group, or server entries. Profile entries are managed by the functions that start with the prefix **RpcNsProfile**.

This section presents an overview of the name-service database in the following topics:

- Name-Service Application Guidelines
- An Overview of the Name Service Entry
- Criteria for Name Service Entries
- Name Service Entry Cleanup
- What Happens During a Query
- Using Microsoft Locator
- Using the Cell Directory Service (CDS)
- Name Syntax

Name-Service Application Guidelines

When you develop your distributed application, you need to provide the application users with a method for specifying the name under which they can register the application in the name-service database. This method can consist of a data file, command-line input, or dialog box.

Although the RPC name-service architecture supports various methods for organizing an application's server entries, it is optimized for lookups. As a result, frequent updates can hinder the performance of both the name service and the application. To avoid exporting information unnecessarily, choose a design that lets the server determine whether its information is in the name-service database. In addition, each server instance should export to its own entry name. Otherwise, it will be difficult for an instance to change its supported object UUIDs or protocol sequences without disturbing another instance's information.

The following method avoids these pitfalls and provides good performance, no matter what name service your network uses.

To begin with, design your application so that the first time a given server instance starts, it picks a unique server-entry name and saves this name in an .ini file along with the application's other configuration information. Then, have it export its binding handles and object UUIDs, if any, to its name-service entry.

Subsequent invocations of the server instance should check that the name-service entry is present and contains the correct set of object UUIDs and binding handles. A missing entry may mean that an administrator removed it, or that a power outage caused the name-service information to be lost. It is important to verify that the binding handles in the entry are correct; if an administrator adds TCP/IP support to a computer, for example, RPC servers will listen on that protocol sequence when they call **RpcServerUseAllProtseqs**. However, if the server does not update the name-service entry, clients will not be informed that TCP is supported.

When the client imports, it should specify NULL as the entry name. Specifying NULL causes the Microsoft RPC library functions to search for the interface in all name-service entries in the client machine's domain or workgroup, thus finding the information for every instance.

If you use object UUIDs to represent well-known objects such as printers, you can use a variation of this method. Instead of exporting bindings to one entry, design your application so that each instance creates an entry for each supported object, such as "/.:/printers/Laser1" and "/.:/printers/Laser2." Then, have the server export its binding handles to each server entry, along with the object UUID relevant to that entry.

In this case, a client can look up a resource by name by importing from the relevant server entry; it does not require the object UUID of the resource. If it has the resource UUID but not the name, it can import from the null entry.

An Overview of the Name Service Entry

The name-service entry consists of three distinct sections. The first section is for interfaces (UUID + version), the second section contains the object UUIDs, and the third section is for binding handles. You provide a name for the entry that will serve as a way to identify it.

When calling **RpcNsBindingExport**, the server specifies the name of the entry in which to place the exported information. This newly exported interface is then added to the interface section of the name-service entry. Any interfaces that are already present in the name-service entry remain as well. This same process is followed for object UUIDs and binding handles.

The client calls **RpcNsBindingLookupBegin** (or **RpcNsBindingImportBegin**) to search for an appropriate binding handle. The entry name, interface handle, and an object UUID are extracted. These restrict the entries from which binding handles are returned. If an entry matches the search criteria, all the binding handles in that entry are returned from **RpcNsBindingImportNext**.

Criteria for Name Service Entries

The following criteria are used when processing name-service entries:

- If you provide a non-NULL entry name for **RpcNsBindingLookupBegin**, that entry will be the only entry searched for binding handles. If you pass NULL, all entries in your logon domain will be searched. Note that this does not include trusted domains.
- If you provide an interface handle, binding handles are returned from an entry only if the interface section of the entry contains a compatible version of that interface UUID. That is, the major version number must be the same as your interface UUID, while the minor version number must be equal to or greater than yours.
- If you provide an object UUID, binding handles are returned from an entry only if the object UUID section of the entry contains that particular object UUID.

If a name-service entry survives the criteria described above, all the binding handles from those entries are gathered. Handles with a protocol sequence that is unsupported by the client are discarded and the remaining handles are returned to you as the output from **RpcNsBindingLookupNext**.

Name Service Entry Cleanup

A name-service entry should contain information that does not change frequently. For this reason, do not include dynamic endpoints in your exported binding handles because they will change at each invocation of the server and will clutter up your name-service entry. To remove these binding handles, use **RpcBindingReset**. For example, a reasonable sequence of server operations would be:

For more than one transport:

```
RpcServerUseProtseq();
RpcServerUseProtseq();
```

To place bindings in the endpoint mapper:

```
RpcServerInqBindings(&Vector);
RpcEpRegister(Interface, Vector);
```

To remove endpoints from bindings:

```
for (i=0; i < Vector- > Count; + + i)
{
    RpcBindingReset(Vector->BindingH[i];
}
```

To add bindings to the name service:

```
RpcNsBindingExport(RPC_C_NS_SYNTAX_DEFAULT, EntryName, Interface
                   Vector);
RpcServerListen();
```

What Happens During a Query

This section describes how the network handles the query when a 32-bit client searches for a name in its own domain.

When your client application calls **RpcNsBindingImportBegin**, the locator residing on your client computer will try to satisfy this request. If there is nothing in the cache, it will forward the request by RPC to a master locator. If the master locator finds nothing in its cache, it sends the request to all the computers in the domain using a mail-slot broadcast. If there is a match, the locator on each computer will respond by a directed mail slot. (For example, if a process on that computer has exported the interface.) The responses are collated and the RPC is completed from the client's process locator, which will reply to the client process itself.

In a domain, the client locator searches for a master locator in the following places:

- On the primary domain controller
- On each backup domain controller

If a match is not found, the client locator declares itself to be the master locator. As such, it will broadcast queries if they cannot be satisfied locally.

In a workgroup, the client locator maintains a cache of the computers whose locators have broadcast. It uses the one that has been running the longest as the master locator. If that computer is unavailable, the next, longest-broadcasting computer is used, and so on. If the client needs a master locator and the cache is empty, it replenishes the cache by sending a special mail-slot broadcast that requests master locators to respond. If there are no responses, the client locator declares itself to be the master locator and will broadcast queries if they cannot be satisfied locally.

This changes if your client application is a Microsoft® Windows® 3.*x* or MS-DOS® program. In this case, there is no locator running on the client computer, and Rpcns1.dll or Rpcnslm.rpc contains the code to find a master locator. All requests are forwarded directly to the master locator.

These guidelines are valid for names in the client's domain, such as names for "/.:/entryname". If the client requests a name from another domain through the use of "/.../DOMAIN/entryname;" the client locator forwards the request to the specified domain which will broadcast it if it does not have the answer. If the domain is down or is actually a workgroup, the request will fail.

Note Remember the following when working with entries in the name service:

- A client cannot use the "/.../DOMAIN/entryname" syntax to find an entry in its own domain. Use the syntax "/.:/entryname". However, you can use "/.../DOMAIN/entryname" to find an entry in another domain.
- The domain name in "/.../DOMAIN/entryname" must be uppercase. When looking for a match, the locator is case-sensitive.
- Locator entry names are also case-sensitive, but need not be uppercase.
- When the client uses the "/.:/entryname" syntax, the locator will not search for entries in other domains, even if they have a trust relationship with the logon domain.
- Broadcasts do not cross LAN segments (for example, subnets). Thus, the usefulness of the locator is limited in an organization with multiple subnets.

Using Microsoft Locator

Microsoft Locator is the default name service that ships with Microsoft® Windows NT®/Windows 2000. The RPC run-time library uses it to find server programs on server host systems.

Prior to Windows 2000, Microsoft Locator did not provide persistent name-service entries. All entries in the name service were stored in a memory cache on the server program's host computer. The locator used a broadcast mechanism to discover the location of servers as requested by clients. Whenever the host system shut down, all name-service entries were lost.

Beginning with the release of Windows 2000, Microsoft Locator now supports persistent name-service entries. To accomplish this, Windows 2000 employs a distributed directory service to store name-service entries. This mechanism has several advantages:

- **Persistence**. Server programs are no longer required to export their binding information to the name service each time they start up. The name service now stores the information until the server program on the network administrator explicitly removes it.
- **Efficiency**. By eliminating the majority of broadcasting for name-service entries, the locator reduces network traffic. It also finds name-service entries more rapidly.
- **Cross-Domain Interopability**. Clients are now able to make name-service requests across multiple domains.

Current versions of Microsoft Locator are backward compatible. For instance, a server host system running the locator that ships with Windows 2000 can operate correctly on a network that contains server host systems running the locator that ships with Windows NT 4.0.

In addition, the current version of Microsoft Locator supports the use of Access Control Lists in name-service entries. This ability enhances the security of the network.

Plug and Play support is now included in Microsoft Locator. Therefore, it can transparently handle Plug and Play events such as domain changes. For more information, see ***RpcNsBindingExportPnP*** and ***RpcNsBindingUnexportPnP***.

Using the Cell Directory Service (CDS)

If you have CDS, you can use it instead of Microsoft Locator. Change the registry entries as shown:

```
HKEY_LOCAL_MACHINE
    Software
        Microsoft
            Rpc
                Name Service
                    NetworkAddress

HKEY_LOCAL_MACHINE
    Software
        Microsoft
            Rpc
                Name Service
                    Endpoint
```

Changing these entries will point to a gateway computer that is running the NSID. This will be used as the master locator. In the event of a crash, there will be no search for a replacement.

Name Syntax

Microsoft RPC accepts names that conform to the following syntax:

```
/.:/name[[/name...]]
/.../domainname/name[[/name...]]
```

Parameters

name
Specifies an identifier that can contain any character except the delimiting slash (/) character.

domainname
Specifies the name of the Windows NT/Windows 2000 domain.

A parameter that selects the name-syntax type and the string that specifies the name are supplied to many of the name-service interface (NSI) RPC functions.

Only one name-syntax type is supported by Microsoft RPC, as specified by the constant RPC_C_NS_SYNTAX_DCE. This constant is defined in the header file RPCNSI.H.

The name syntax specified by RPC_C_NS_SYNTAX_DCE is an extension of the OSF_DCE Cell Directory Service (CDS) name syntax. The ability to specify a domain name represents an extension to that syntax. There is no absolute limit on the number of names that can be separated by slash characters as long as the overall string is less than 256 characters.

The slashes allow you to specify a logical structure to the name, but they do not correspond to a logical structure in the objects themselves.

Context Handles

It is sometimes the case that distributed applications require the server program to maintain status information between client calls. Server programs that service more than one client at a time must keep the status information for each client. Because the client and the server use different address spaces on different computers, common approaches to data sharing often don't work. For instance, the client and server are unable to maintain status information on their remote session in global variables because they don't share the same global address space. It is difficult to keep the information in a shared file because they run on different computers.

Microsoft® RPC provides a mechanism called context handles for keeping status information on a server. The status information is called the server's context. Clients can obtain a context handle to identify the server's context for their individual RPC sessions.

As an example, each client in a distributed application can have the server program create and update a data file for their RPC session. The server can use its file handle for each client's data file as the context handle. Each time a client requests operations on the data file that the server creates for it, the client passes the context handle to the server. Since the context handle is really a file handle, the context handle only makes sense in the server's address space. However, the client program can use the context handle to tell the server on which file to perform updates.

Other data can also be context handles. For instance, a client and server can use a record number of a database record as a file handle. If the client needed to perform a number of updates on a particular record, it could obtain the record number as a context handle. It would pass the record number to the server each time it invoked a remote procedure to update the database record.

Interface Development Using Context Handles

Typically, you create a context handle by specifying the **[context_handle]** attribute on a type definition in the IDL file. The type definition also implicitly specifies a context rundown routine, which you must provide. If communication between the client and server breaks down, the server run time invokes this routine to perform any needed cleanup. For more information on context rundown routines, see *Server Context Rundown Routine*.

An interface that uses a context handle must have a binding handle for the initial binding, which has to take place before the server can return a context handle. You can use an automatic, implicit, or explicit binding handle to create the binding and establish the context.

A context handle must be of the **void** * type, or a type that resolves to void *. The server program casts it to the required type.

The following fragment of a sample interface definition shows how a distributed application can use a context handle to have a server open and update a data file for each client.

The interface must contain a remote procedure call to initialize the handle and set it to a non-null value. In this example, the **RemoteOpen** function performs this operation. It specifies the context handle with an **[out]** directional attribute. Alternatively, you could return the context handle as the procedure's return value. However in this example, we'll pass the context handle out through the parameter list.

In this example, the context information is a file handle. It keeps track of the current location in the file. The interface packages the file handle as a context handle and passes it as a parameter to remote procedure calls. A structure contains the file name and the file handle.

```
/* file: cxhndl.idl (fragment of interface definition file) */
typedef [context_handle] void * PCONTEXT_HANDLE_TYPE;
typedef [ref] PCONTEXT_HANDLE_TYPE * PPCONTEXT_HANDLE_TYPE;

short RemoteOpen([out] PPCONTEXT_HANDLE_TYPE pphContext,
    [in, string] unsigned char * pszFile);

void RemoteRead(
    [in] PCONTEXT_HANDLE_TYPE phContext,
    [out, size_is(cbBuf)] unsigned char achBuf[],
    [in, out] short *pcbBuf);

short RemoteClose([in, out] PPCONTEXT_HANDLE_TYPE pphContext);
```

The **RemoteOpen** function creates a valid, non-null context handle. It passes the context handle to the client. Subsequent remote procedure calls, such as **RemoteRead**, use the context handle as an **in** pointer:

In addition to the remote procedure that initializes the context handle, the interface must contain a procedure that frees the server context and sets context handle to NULL. In the preceding example, the **RemoteClose** function performs this operation.

Server Development Using Context Handles

From the perspective of server program development, a context handle is an untyped pointer. Server programs initialize context handles by pointing them at data in memory or on some other form of storage (such as files on disks).

For instance, suppose that a client uses a context handle to request a series of updates to a record in a database. The client calls a remote procedure on the server and passes it a search key. The server program searches the database for the search key and obtains the integer record number of the matching record. The server can then point a pointer to void at a memory location containing the record number. When it returns, the remote procedure would need to return the pointer as a context handle through its return value or its parameter list. The client would need to pass the pointer to the server each time it called remote procedures to update the record. During each of these update operations, the server would cast the void pointer to be a pointer to an integer.

After the server program points the context handle at context data, the handle is considered active. Handles containing a NULL value are inactive. The server maintains an active context handle until the client calls a remote procedure that frees it. If the client terminates while the handle is active, the server can free the handle. In addition, the server will free the handle when communication between the client and the server breaks down.

The following code fragment demonstrates how a server might implement a context handle. In this example, the server maintains a data file that the client writes to using remote procedures. The context information is a file handle that keeps track of the current location in the file where the server will write data. The file handle is packaged as a context handle in the parameter list to remote procedure calls. A structure contains the file name and the file handle. The interface definition for this example is shown in Interface Development Using Context Handles.

```
/* cxhndlp.c (fragment of file containing remote procedures) */
typedef struct
{
    FILE * hFile;
    char   achFile[256];
} FILE_CONTEXT_TYPE;
```

The function **RemoteOpen** opens a file on the server:

```
short RemoteOpen(
    PPCONTEXT_HANDLE_TYPE pphContext,
    unsigned char *pszFileName)
{
    FILE               *hFile;
    FILE_CONTEXT_TYPE  *pFileContext;

    if ((hFile = fopen(pszFileName, "r")) == NULL)
    {
        *pphContext = (PCONTEXT_HANDLE_TYPE) NULL;
        return(-1);
    }
    else
    {
        pFileContext = (FILE_CONTEXT_TYPE *)
                       MIDL_user_allocate(sizeof(FILE_CONTEXT_TYPE));
        pFileContext->hFile = hFile;
        strcpy(pFileContext->achFile, pszFileName);
        *pphContext = (PCONTEXT_HANDLE_TYPE) pFileContext;
        return(0);
    }
}
```

The function **RemoteRead** reads a file on the server.

```
short RemoteRead(
    PCONTEXT_HANDLE_TYPE phContext,
    unsigned char *pbBuf,
    short *pcbBuf)
```

```
{
    FILE_CONTEXT_TYPE *pFileContext;
    printf("in RemoteRead\n");
    pFileContext = (FILE_CONTEXT_TYPE *) phContext;
    *pcbBuf = (short) fread(pbBuf, sizeof(char),
                            BUFSIZE,
                            pFileContext->hFile);
    return(*pcbBuf);
}
```

The function **RemoteClose** closes a file on the server. Note that the server application has to assign NULL to the context handle as part of the close function. This communicates to the server stub and the RPC run-time library that the context handle has been deleted. Otherwise, the connection will be kept open and eventually a context rundown will occur.

```
void RemoteClose(PPCONTEXT_HANDLE_TYPE pphContext)
{
    FILE_CONTEXT_TYPE *pFileContext;

    if (*pphContext == NULL)
    {
        //Log error, client tried to close a NULL handle.
        return;
    }
    pFileContext = (FILE_CONTEXT_TYPE *)*pphContext;
    printf("File %s closed.\n", pFileContext->achFile);
    fclose(pFileConext->hFile;
    MIDL_user_free(pFileContext);

    // This tells the run-time, when it is marshalling the out
    // parameters, that the context handle has been closed normally.
    *pphContext = NULL;
}
```

Client Development Using Context Handles

The only use a client program has for a context handle is to pass it to the server each time the client makes a remote procedure call. The client application does not need to access the contents of the handle. It should not try to change the context handle data in any way. The remote procedures that the client invokes perform all necessary operations on the server's context.

Prior to requesting a context handle from a server program, clients must establish a binding with the server. The client may use an automatic, implicit, or explicit binding handle. With a valid binding handle, the client can call a remote procedure on the server that either returns an active (non-NULL) context handle or passes one through an [**out**] parameter in the remote procedure's parameter list.

Clients may use active context handles in any way they require. They should, however, invalidate the handle when they no longer need it. To do this, the client should invoke a remote procedure offered by the server program that frees the context and sets the context handle as inactive (NULL).

The following code fragment presents an example of how a client might use a context handle. To view the definition of the interface that this example uses, see *Interface Development Using Context Handles*. For the server implementation, see *Server Development Using Context Handles*.

In this example, the client calls **RemoteOpen** to obtain a context handle that contains valid data. The client can then use the context handle in remote procedure calls. Because it no longer needs the binding handle, the client can free the explicit handle it used to create the context handle:

```
// cxhndlc.c  (fragment of client side application)
printf("Calling the remote procedure RemoteOpen\n");
if (RemoteOpen(&phContext, pszFileName) < 0)
{
    printf("Unable to open %s\n", pszFileName);
    Shutdown();
    exit(2);
}

// Now the context handle also manages the binding.
// The variable hBindingHandle is a valid binding handle.
status = RpcBindingFree(&hBindingHandle);
printf("RpcBindingFree returned 0x%x\n", status);
if (status)
    exit(status);
```

The client application in this example uses a procedure called **RemoteRead** to read a data file on the server until it encounters an end of file. It then closes the file by calling **RemoteClose**. The context handle appears as a parameter in the **RemoteRead** and **RemoteClose** functions as:

```
printf("Calling the remote procedure RemoteRead\n");
do
{
    cbRead = 1024; // Using a 1K buffer
    RemoteRead(phContext, pbBuf, &cbRead);
    // cbRead contains the number of bytes actually read.
    for (i = 0; i < cbRead; i++)
        putchar(*(pbBuf+i));
} while(cbRead);

printf("Calling the remote procedure RemoteClose\n");
if (RemoteClose(&phContext) < 0 )
{
    printf("Close failed on %s\n", pszFileName);
    exit(2);
}
```

Server Context Rundown Routine

If communication breaks down while the server is maintaining context on behalf of the client, a cleanup routine may be needed to reset the context information. This cleanup routine is called a *context rundown routine*. When a connection breaks, the server stub and the run-time library will call this routine on every context handle opened by the client.

The context rundown routine is required, and is implicitly declared and named, when you apply the **[context_handle]** attribute to a type definition. The server will *not* call the context rundown routine if the **[context_handle]** attribute was applied directly to a parameter.

The context rundown routine syntax is:

```
void __RPC_USER type-id_rundown (type-id);
```

Note that the type name determines the name of the context rundown routine.

The code fragment that follows presents a sample context rundown routine. that calls the **RemoteClose** procedure used in the example in Interface Development Using Context Handles, Server Development Using Context Handles, and Client Development Using Context Handles. This procedure closes the file handle, frees the memory associated with the file, and assigns NULL to the context handle. The NULL value indicates to the run-time library that the context handle is inactive so that the rundown routine will not be called when the connection is removed. Your context rundown routine could perform other tasks, such as logging an event when a connection fails.

```
//file: cxhndp.c (fragment of file containing remote procedures)
//The rundown routine is associated with the context handle type.
void __RPC_USER PCONTEXT_HANDLE_TYPE_rundown(
    PCONTEXT_HANDLE_TYPE phContext)
{
    printf("Client died with an open file, closing it..\n");
    RemoteClose(&phContext);
    assert(phContext == 0);
}
```

Client Context Reset

When the server becomes unavailable, the client application can free its context data by calling the RPC function **RpcSsDestroyClientContext**.

Multi-Threaded Clients and Context Handles

When you have a multi-threaded client where multiple threads are using the same context handle, the calls will be serialized at the server. This saves the server manager from having to guard against another thread from the same client changing the context or from the context running down while a call is dispatched. However, in certain cases serialization may create deadlock. For example, consider the following sequence:

Thread 1: Gets a context handle and makes a call. This call blocks on some synchronization event sitting on the server.

Thread 2: Makes a call to the same server, using the same context handle. This call is intended to trigger the event that thread 1 is blocking on. Because the calls are serialized, the event is never triggered.

If you have a situation like this you can use the **RpcSsDontSerializeContext** function to allow multiple calls to be dispatched on a single context handle. Calling this function does not disable serialization entirely. When a context rundown occurs, your context rundown routine will not run until all outstanding client requests have completed. Be aware that a call to **RpcScDontSerializeContext** affects the entire process and is not revertible.

CHAPTER 16

Memory Management

This chapter discusses how RPC programs allocate and deallocate memory for data passed between client and server programs.

Introduction to RPC Memory Management

In the context of RPC, memory management involves:

- Allocating and deallocating the memory needed to simulate a single conceptual address space between the client and the server in the different address spaces of the client and server's threads.
- Determining which software component is responsible for managing memory—the application or the MIDL-generated stub.
- Selecting MIDL attributes that affect memory management: directional attributes, pointer attributes, array attributes, and the ACF attributes **[byte_count]**, **[allocate]**, and **[enable_allocate]**.

When a program calls a function or procedure in its address space, memory management is more straightforward than in a distributed application. To illustrate, Figure 16-1 depicts a binary tree. To pass this data structure to a procedure in its address space, a program simply passes a pointer to the root of the tree.

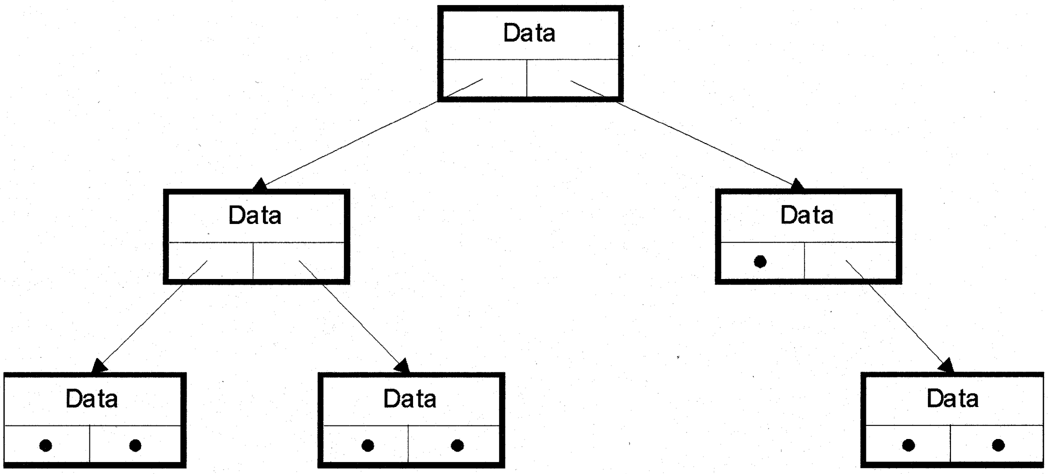


Figure 16-1: Binary Tree.

Client/server RPC applications share data across two different memory spaces. These memory spaces may or may not be on the same computer. Either way, the client and server have no direct access to each other's memory space. RPC depends on the ability to simulate the client program's address space in the server program's address space. It must also return data, including new and changed data, from the server to the client memory.

In cases such as the binary tree depicted in the preceding diagram, it is not sufficient to pass a pointer to the root node to a remote procedure. Either the program or the stubs must pass the entire tree to the server's address space for the remote procedure to operate on it.

How Memory Is Allocated and Deallocated

By default, stub code generated by the MIDL compiler calls user-supplied functions to allocate and free memory. These functions, named **midl_user_allocate** and **midl_user_free**, must be supplied by the developer and linked with the application.

All applications must supply implementations of **midl_user_allocate** and **midl_user_free**, even though the names of these functions may not appear explicitly in the stubs. The only exception is when you are compiling in OSF-compatibility (/**osf**) mode. These user-supplied functions must match a specific, defined function prototype but otherwise, can be implemented in any way that is convenient or useful for the application. Alternatively, applications can use the RpcSs Memory Management Package. The Microsoft® RPC run-time library supplies this group of functions.

The midl_user_allocate Function

The **midl_user_allocate** function is a procedure that must be supplied by developers of RPC applications. It allocates memory for the RPC stubs and library routines. Your **midl_user_allocate** function must match the following prototype:

```
void __RPC_FAR * __RPC_USER midl_user_allocate (size_t cBytes);
```

The *cBytes* parameter specifies the number of bytes to allocate. Both client applications and server applications must implement the **midl_user_allocate** function unless you are compiling in OSF-compatibility (/**osf**) mode. Applications and generated stubs call **midl_user_allocate** directly or indirectly to manage allocated objects. For example:

- The client and server applications call **midl_user_allocate** to allocate memory for the application, such as when creating a new node in a tree or linked list.
- The server stub calls **midl_user_allocate** when unmarshaling data into the server address space.

- The client stub calls **midl_user_allocate** when unmarshaling data from the server that is referenced by an **[out]** pointer. Note that for **[in, out, unique]** pointers, the client stub calls **midl_user_allocate** only if the **[unique]** pointer value was null on input and changes to a non-null value during the call. If the **[unique]** pointer was non-null on input, the client stub writes the associated data into existing memory.

If **midl_user_allocate** fails to allocate memory, it should return a null pointer or raise a user-defined exception.

The **midl_user_allocate** function should return a pointer such that:

- For Microsoft® Windows NT®/Windows® 2000 running on Intel platforms, the pointer is 4 bytes aligned.
- For Windows NT/Windows 2000 running on MIPS and Alpha platforms, the pointer is 8 bytes aligned.
- For Microsoft Windows 95, the pointer is 4 bytes aligned.
- For Windows 3.*x* and MS-DOS® platforms, the pointer is 2 bytes aligned.

For example, the sample programs provided with the Platform SDK implement **midl_user_allocate** in terms of the C function **malloc**:

```
void __RPC_FAR * __RPC_USER midl_user_allocate(size_t cBytes)
{
    return((void __RPC_FAR *) malloc(cBytes));
}
```

Note If the RpcSs package is enabled (for example, as the result of using the **[enable_allocate]** attribute), use **RpcSmAllocate** to allocate memory on the server side. For additional information on **[enable_allocate]**, see *MIDL Reference*.

The midl_user_free Function

The **midl_user_free** function must be supplied by RPC developers. It allocates memory for the RPC stubs and library routines. Your **midl_user_free** function must match the following prototype:

```
void __RPC_USER midl_user_free(void __RPC_FAR * pBuffer);
```

The *pBuffer* parameter specifies a pointer to the memory that is to be freed. Both client application and server application must implement the **midl_user_free** function unless you are compiling in OSF-compatibility (**/osf**) mode. The **midl_user_free** function must be able to free all storage allocated by **midl_user_allocate**.

Applications and stubs call **midl_user_free** when dealing with allocated objects:

- The server application should call **midl_user_free** to free memory allocated by the application, such as when deleting a dynamically allocated node of data.
- The server stub calls **midl_user_free** to release memory on the server after marshaling all **[out]** arguments, **[in,out]** arguments, and the function return value.

For example, the RPC Win32 sample program that displays "Hello, world" implements **midl_user_free** in terms of the C function **free**:

```
void __RPC_USER midl_user_free(void __RPC_FAR * p)
{
    free(p);
}
```

Note If the RpcSs package is enabled (for example, as the result of using the **[enable_allocate]** attribute), your server program should use **RpcSmFree** to free memory. For more information, see *RpcSs Memory Management Package*.

RpcSs Memory Management Package

The default allocator/deallocator pair used by the stubs and run time when allocating memory on behalf of the application is **midl_user_allocate/midl_user_free**. However, you can choose the RpcSs package instead of the default by using the ACF attribute **[enable_allocate]**. The RpcSs package consists of RPC functions that begin with the prefix **RpcSs** or **RpcSm**. It is the recommended memory management model and provides the best overall stub performance for memory management.

In **/osf** mode, the RpcSs package is enabled for MIDL-generated stubs automatically when full pointers are used, when the arguments require memory allocation, or as a result of using the **[enable_allocate]** attribute. In the default (Microsoft extended) mode, the RpcSs package is enabled only when the **[enable_allocate]** attribute is used. The **[enable_allocate]** attribute enables the RpcSs environment by the server-side stubs. The client side becomes alerted to the possibility that the RpcSs package may be enabled. In /**osf** mode, the client side is not affected.

When the RpcSs package is enabled, allocation of memory on the server side is accomplished with the private RpcSs memory management allocator and deallocator pair. You can allocate memory using the same mechanism by calling **RpcSmAllocate** (or **RpcSsAllocate**). Upon return from the server stub, all the memory allocated by the RpcSs package is automatically freed. The following example shows how to enable the RpcSs package:

```
/* ACF file fragment */

[
    implicit_handle(handle_t GlobalHandle),
    enable_allocate
]
interface iface
{
}

/*Server management routine fragment. Replaces p=midl_user_allocate(size); */

    p=RpcSsAllocate(size);                /*raises exception */
    p=RpcSmAllocate(size, &status);       /*returns error code */
```

Your application can explicitly free memory by invoking the **RpcSsFree** or **RpcSmFree** function. Note that these functions do not actually free memory. They mark it for deletion. The RPC library releases the memory when your program calls **RpcSsDisableAllocate** or **RpcSsDisableAllocate**.

You can also enable the memory management environment for your application by calling the **RpcSmEnableAllocate** routine (and you can disable it by calling the **RpcSmDisableAllocate** routine). Once enabled, application code can allocate and deallocate memory by calling functions from the RpcSs package.

Memory-Management Models

As a developer, you have several choices for how memory is allocated and freed. Consider a complex data structure that consists of nodes connected with pointers, such as a linked list or a tree. You can apply attributes that select one of the following models:

- Node-by-node allocation and deallocation.
- A single linear buffer allocated by the stub for the entire tree.
- A single linear buffer allocated by the client application for the entire tree.
- Persistent storage on the server.

Node-by-Node Allocation and Deallocation

Node-by-node allocation and deallocation of data structures by the stubs is the default method of memory management for all parameters on both the client and the server. On the client side, the stub allocates each node with a separate call to **midl_user_allocate**. On the server side, rather than calling **midl_user_allocate**, private memory is used whenever possible. If **midl_user_allocate** is called, the server stubs will call **midl_user_free** to free the data. In most cases, using node-by-node allocation and deallocation instead of using **[allocate (all_nodes)]** will result in increased performance of the server side stubs.

Stub-Allocated Buffers

Rather than forcing a distinct call for each node of the tree or graph, you can direct the stubs to compute the size of the data and to allocate and free memory by making a single call to **midl_user_allocate** or **midl_user_free**. The ACF attribute **[allocate(all_nodes)]** directs the stubs to allocate or free all nodes in a single call to the user supplied—memory management functions.

For example, an RPC application might use the following binary tree data structure:

```
/* IDL file fragment */
typedef struct _TREE_TYPE
{
    short sNumber;
    struct _TREE_TYPE * pLeft;
    struct _TREE_TYPE * pRight;
} TREE_TYPE;

typedef TREE_TYPE * P_TREE_TYPE;
```

The ACF attribute **[allocate(all_nodes)]** applied to this data type appears in the **typedef** declaration in the ACF as:

```
/* ACF file fragment */
typedef [allocate(all_nodes)] P_TREE_TYPE;
```

The **[allocate]** attribute can only be applied to pointer types. The **[allocate]** ACF attribute is a Microsoft extension to DCE IDL and, as such, is not available if you compile with the MIDL /**osf** switch. When **[allocate(all_nodes)]** is applied to a pointer type, the stubs generated by the MIDL compiler traverse the specified data structure to determine the allocation size. The stubs then make a single call to allocate the entire amount of memory needed by the graph or tree. A client application can free memory much more efficiently by making a single call to **midl_user_free**. However, server-stub performance is generally increased when using node-by-node memory allocation because the server stubs can often use private memory that requires no allocations.

For additional information, see *Node-by-Node Allocation and Deallocation.*

Application-Allocated Buffer

The ACF attribute **[byte_count]** directs the stubs to use a preallocated buffer that is not allocated or freed by the client support routines. The **[byte_count]** attribute is applied to a pointer or array parameter that points to the buffer. It requires a parameter that specifies the buffer size in bytes.

The client-allocated memory area can contain complex data structures with multiple pointers. Because the memory area is contiguous, the application does not have to make several calls to free each pointer and structure individually. As when using the

[allocate(all_nodes)] attribute, the memory area can be allocated or freed with one call to the memory-allocation routine or the free routine. However, unlike using the **[allocate(all_nodes)]** attribute, the buffer parameter is not managed by the client stub but by the client application.

The buffer must be an **[out]**-only parameter and the buffer length in bytes must be an **in**-only parameter. The **[byte_count]** attribute can only be applied to pointer types. The **[byte_count]** ACF attribute is a Microsoft extension to DCE IDL and, as such, is not available if you compile using the MIDL /**osf** switch.

In the following example, the parameter *pRoot* uses byte count:

```
/* function prototype in IDL file (fragment) */
void SortNames(
    [in] short cNames,
    [in, size_is(cNames)] STRINGTYPE pszArray[],
    [in] short cBytes,
    [out, ref] P_TREE_TYPE pRoot  /* tree with sorted data */
);
```

The **[byte_count]** attribute appears in the ACF as:

```
/* ACF file (fragment) */
SortNames([byte_count(cBytes)] pRoot);
```

The client stub generated from these IDL and ACF files does not allocate or free the memory for this buffer. The server stub allocates and frees the buffer in a single call using the provided size parameter. If the data is too large for the specified buffer size, an exception is raised.

Persistent Storage on the Server

You can optimize your application so the server stub does not free memory on the server at the conclusion of a remote procedure call. For example, when a context handle will be manipulated by several remote procedures, you can use the ACF attribute **[allocate(dont_free)]** to retain the allocated memory on the server.

The **[allocate(dont_free)]** attribute is added to the ACF **typedef** declaration in the ACF. For example:

```
/* ACF file fragment */
typedef [allocate(all_nodes, dont_free)] P_TREE_TYPE;
```

When the **[allocate(dont_free)]** attribute is specified, the tree data structure is allocated, but not freed, by the server stub. When you make the pointers to such persistent data areas available to other routines—for example, by copying the pointers to global variables—the retained data is accessible to other server functions. The **[allocate(dont_free)]** attribute is particularly useful for maintaining persistent pointer structures as part of the server state information associated with a context-handle type.

Who Manages Memory?

Generally, the stubs are responsible for packaging and unpackaging data, allocating and freeing memory, and transferring the data to and from memory. In some cases, however, the application is responsible for allocating and freeing memory.

Top-Level and Embedded Pointers

To understand how pointers and their associated data elements are allocated in Microsoft RPC, you have to differentiate between *top-level pointers* and *embedded pointers*. It is also useful to refer to the set of all pointers that are not top-level pointers.

Top-level pointers are those that are specified as the names of parameters in function prototypes. Top-level pointers and their referents are always allocated on the server.

Embedded pointers are pointers that are embedded in data structures such as arrays, structures, and unions. When embedded pointers are **[out]**-only and null on input, the server application can change their values to non-null. In this case, the client stubs allocate new memory for this data.

If the embedded pointer is not null on the client before the call, the stubs do not allocate memory on the client on return. Instead, the stubs attempt to write the memory associated with the embedded pointer into the existing memory on the client associated with that pointer, overwriting the data already there.

Embedded **[out]**-only pointers are discussed in *Combining Pointer and Directional Attributes*.

The term *nontop-level pointers* refers to all pointers that are not specified as parameter names in the function prototype, including both embedded pointers and multiple levels of nested pointers.

Directional Attributes Applied to the Parameter

The directional attributes **[in]** and **[out]** determine how the client and server allocate and free memory. The following table summarizes the effect of directional attributes on memory allocation.

Directional attribute	Memory on client	Memory on server
[in]	Client application must allocate before the call.	Server stub allocates.
[out]	Client stub allocates on return.	Server stub allocates top-level pointer only; the server application must allocate all embedded pointers. The server also allocates new data as needed.
[in, out]	Client application must allocate initial data transmitted to server; client stub allocates additional data.	Server stub allocates initial data transmitted from client; the server application allocates new data as needed.

In all of these cases the client stub does not free memory. The client application must free the memory before it terminates. The server stub frees memory when the remote procedure call returns (subject to the **[allocate]** ACF attribute).

Length, Size, and Directional Attributes

In passing arrays between the client and the server, the size-related attributes **[max_is]** and **[size_is]** determine how many array elements the server stub allocates. The length-related attributes **[length_is]**, **[first_is]**, and **[last_is]** determine how many elements are transmitted to both the server and the client.

Different directional attributes can be applied to parameters. However, some combinations of directional attributes can cause errors. As an example, suppose you are writing an interface that specifies a procedure with two parameters, an array, and the transmitted length of the array. The italicized term *dir_attr* refers to the directional attribute applied to the parameter as:

```
Proc1(
    [dir_attr] short *plength;
    [dir_attr, length_is(pLength)] short array[MAX_SIZE]);
```

The MIDL compiler behavior for each combination of directional attributes is described in the following table.

Array	Length parameter	Stub actions during call from client to server	Stub actions on return from server to client
[in]	**[in]**	Transmit the length and the number of elements indicated by the parameter.	No data transmitted.
[in]	**[out]**	Not legal; MIDL compiler error.	Not legal; MIDL compiler error.

(continued)

(continued)

Array	Length parameter	Stub actions during call from client to server	Stub actions on return from server to client
[in]	**[in, out]**	Transmit the length and the number of elements indicated by the length parameter.	Transmit the length only.
[out]	**[in]**	Transmit the length. If array size is fixed, allocate the array size on the server, but transmit no elements. If array size is not bound, not legal: MIDL compiler error.	Transmit the number of elements indicated by the length. Note that the length can be changed and can have a different value from the value on the client. Do not transmit the length.
[out]	**[out]**	Allocate space for the length parameter on the server but do not transmit the parameter. If the array size is fixed, allocate the array size on the server, but transmit no elements. If array size is not fixed, not legal: MIDL compiler error.	Transmit the length and the number of elements indicated by the length as set by the server application.
[out]	**[in, out]**	Transmit the length parameter. If the array size is bound, allocate the array size on the server, but transmit no elements. If array size is not bound, not legal: MIDL compiler error.	Transmit the length. Transmit the number of array elements indicated by the length.
[in, out]	**[in]**	Transmit the length and the number of elements indicated by the parameter.	Do not transmit the length. Transmit the number of elements indicated by the length. Note that the length can be changed and can have a different value from the original value on the client.
[in, out]	**[out]**	Not legal; MIDL compiler error.	Not legal; MIDL compiler error.
[in, out]	**[in, out]**	Transmit the length and the number of elements indicated by the parameter.	Transmit the length and the number of elements indicated by the parameter.

In general, you should not modify the length or size parameters on the server side. If you change the length parameter, you can orphan memory. For more information, see *Memory Orphaning*.

Pointer Attributes Applied to the Parameter

Each pointer attribute (**[ref]**, **[unique]**, and **[ptr]**) has characteristics that affect memory allocation. The following table summarizes these characteristics.

Pointer attribute	Client	Server
Reference (**[ref]**)	Client application must allocate.	Special handling needed for **[out]**-only nontop-level pointers.
Unique (**[unique]**)	If a parameter, the client application must allocate; if embedded, can be null. Changing from null to non-null causes the client stub to allocate; changing from non-null to null can cause orphaning.	
Full (**[ptr]**)	If a parameter, the client application must allocate; if embedded, can be null. Changing from null to non-null causes the client stub to allocate; changing from non-null to null can cause orphaning.	

The **[ref]** attribute indicates that the pointer points to valid memory. By definition, the client application must allocate all the memory that the reference pointers require.

The unique pointer can change from null to non-null. If the unique pointer changes from null to non-null, new memory is allocated on the client. If the unique pointer changes from non-null to null, orphaning can result. For more information, see *Memory Orphaning*.

Combining Pointer and Directional Attributes

A few caveats apply to certain combinations of directional attributes and pointer attributes. These are discussed in the following sections.

Embedded Out-Only Reference Pointers

When you use **[out]**-only reference pointers in Microsoft RPC, the generated server stubs allocate only the first level of pointers accessible from the reference pointer. Pointers at deeper levels are not allocated by the stubs, but must be allocated by the server application layer. For example, suppose an interface specifies an **[out]**-only array of reference pointers:

```
/* IDL file (fragment) */
typedef [ref] short * PREF;

Proc1([out] PREF array[10]);
```

In this example, the server stub allocates memory for 10 pointers and sets the value of each pointer to null. The server application must allocate the memory for the 10 **short** integers referenced by the pointers and then set the 10 pointers to point to the integers.

When the **[out]**-only data structure includes nested reference pointers, the server stubs allocate only the first pointer accessible from the reference pointer. For example:

```
/* IDL file (fragment) */
typedef struct
{
    [ref] small * psValue;
} STRUCT1_TYPE;

typedef struct
{
    [ref] STRUCT1_TYPE * ps1;
} STRUCT_TOP_TYPE;

Proc2([out, ref] STRUCT_TOP_TYPE * psTop);
```

In the preceding example, the server stubs allocate the pointer *psTop* and the structure **STRUCT_TOP_TYPE**. The reference pointer *ps1* in **STRUCT_TOP_TYPE** is set to null. The server stub does not allocate every level of the data structure, nor does it allocate the **STRUCT1_TYPE** or its embedded pointer, *psValue*.

Out-Only Unique or Full Pointer Parameters Not Accepted

Unique or full pointers that are **[out]**-only are not accepted by the MIDL compiler. Such specifications cause the MIDL compiler to generate an error message.

The automatically generated server stub has to allocate memory for the pointer referent so that the server application can store data in that memory area. According to the definition of an **[out]**-only parameter, no information about the parameter is transmitted from client to server. In the case of a unique pointer, which can take the value null, the server stub does not have enough information to correctly duplicate the unique pointer in the server's address space, nor does the stub have any information about whether the pointer should point to a valid address or whether it should be set to null. Therefore, this combination is not allowed.

Rather than **[out**, **unique]** or **[out**, **ptr]** pointers, use **[in**, **out**, **unique]** or **[in**, **out**, **ptr]** pointers, or use another level of indirection such as a reference pointer that points to the valid unique or full pointer.

Function Return Values

Function return values are similar to **[out]**-only parameters because their data is not provided by the client application. However they are managed differently. Unlike **[out]**-only parameters, they are not required to be pointers. The remote procedure can return any valid data type except reference pointers and nonencapsulated unions.

Function return values that are pointer types are allocated by the client stub with a call to **midl_user_allocate**. Accordingly, only the unique or full pointer attribute can be applied to a pointer function-return type.

Memory Orphaning

If your distributed application uses an **[in**, **out**, **unique]** or **[in**, **out**, **ptr]** pointer parameter, the server side of the application can change the value of the pointer parameter to null. When the server subsequently returns the null value to the client, memory referenced by the pointer before the remote procedure call is still present on the client side, but is no longer accessible from that pointer (except in the case of an aliased full pointer). This memory is said to be *orphaned*. This is also termed a *memory leak*. Repeated orphaning of memory on the client causes the client to run out of available memory resources.

Memory can also be orphaned whenever the server changes an embedded pointer to a null value. For example, if the parameter points to a complex data structure such as a tree, the server side of the application can delete nodes of the tree and set pointers inside the tree to null.

Another situation that can lead to a memory leak involves conformant, varying, and open arrays containing pointers. When the server application modifies the parameter that specifies the array size or transmitted range so that it represents a smaller value, the stubs use the smaller value(s) to free memory. The array elements with indices larger than the size parameter are orphaned. Your application must free elements outside the transmitted range.

Summary of Memory Allocation Rules

The following table summarizes key rules regarding memory allocation.

MIDL element	Description
Top-level **[ref]** pointers	Must be non-null pointers.
Function return value	New memory is always allocated for pointer return values.
[unique, **out]** or **[ptr**, **out]** pointer	Not allowed by MIDL.
Non-top-level **[unique**, **in**, **out]** or **[ptr**, **in**, **out]** pointer that changes from null to non-null	Client stubs allocate new memory on client on return.
Non-top-level **[unique**, **in**, **out]** pointer that changes from non-null to null	Memory is orphaned on client; client application is responsible for freeing memory and preventing leaks.
Non-top-level **[ptr**, **in**, **out]** pointer that changes from non-null to null	Memory will be orphaned on client if not aliased; client application is responsible for freeing and preventing memory leaks in this case.

(continued)

(continued)

MIDL element	Description
[ref] pointers	Client-application layer usually allocates.
Non-null **[in**, **out]** pointer	Stubs attempt to write into existing storage on client. If **[string]** and size increases beyond size allocated on the client, it will cause a GP-fault on return.

The following table summarizes the effects of key IDL and ACF attributes on memory management.

MIDL feature	Client issues	Server issues
[allocate(single_node)], **[allocate(all_nodes)]**	Determines whether one or many calls are made to the memory functions.	Same as client, except private memory can often be used for **allocate (single_node) [in]** and **[in,out]** data.
[allocate(free)] or **[allocate(dont_free)]**	(None; affects server.)	Determines whether memory on the server is freed after each remote procedure call.
array attributes **[max_is]** and **[size_is]**	(None; affects server.)	Determines size of memory to be allocated.
[byte_count]	Client must allocate buffer; not allocated or freed by client stubs.	ACF parameter attribute determines size of buffer allocated on server.
[enable_allocate]	Usually, none. However, the client may be using a different memory management environment.	Server uses a different memory management environment. **RpcSmAllocate** should be used for allocations.
[in]attribute	Client application responsible for allocating memory for data.	Allocated on server by stubs.
[out] attribute	Allocated on client by stubs.	**[out]**-only pointer must be **[ref]** pointer; allocated on server by stubs.
[ref] attribute	Memory referenced by pointer must be allocated by client application.	Top-level and first-level reference pointers managed by stubs.
[unique] attribute	Non-null to null can result in orphaned memory; null to non-null causes client stub to call **midl_user_allocate**.	(Affects client.)
[ptr] attribute	(See **[*unique*]**.)	(See **[*unique*]**.)

CHAPTER 17

Serialization Services

Microsoft® RPC supports two methods for encoding and decoding data, collectively called *serializing* data. Serialization means that the data is marshaled to and unmarshaled from buffers that you control. This differs from the traditional usage of RPC, in which the stubs and the RPC run-time library have full control of the marshaling buffers, and the process is transparent. You can use the buffer for storage on permanent media, encryption, and so on. When you encode data, the RPC stubs marshal the data to a buffer and pass the buffer to you. When you decode data, you supply a marshaling buffer with data in it, and the data is unmarshaled from the buffer to memory. You can serialize on a procedure or type basis.

Note The term *pickling* is commonly used among developers to describe serialization. In fact, the Platform SDK samples contains a directory called **pickle** that preserves the RPC serialization sample programs.

Serialization leverages the RPC mechanisms for marshaling and unmarshaling data for other purposes. For example, instead of using several I/O operations to serialize a group of objects to a stream, an application can optimize performance by serializing several objects of different types into a buffer and then writing the entire buffer in a single operation. The functions that manipulate serialization handles are independent of the type of serialization you are using.

As another example, if you need to use a network transport mechanism besides RPC, such as Microsoft® Windows® Sockets (Winsock). With RPC serialization, your program can make calls to functions that marshal your data into buffers and then transmit this data using Winsock. When your application receives data, it can use the RPC serialization mechanism to unmarshal data from buffers filled by the Winsock routines. This provides you with many of the advantages of RPC-style applications, and at the same time, it enables you to use non-RPC transport mechanisms.

You can also use serialization for purposes unrelated to network communications. For example, once you use the RPC encoding functions to marshal data to a buffer, you can store it in a file for use by another application. You can also encrypt it. You can even use it to store a hardware- and operating system-independent representation of data in a database.

Using Serialization Services

MIDL generates a serialization stub for the procedure with the attributes **[encode]** and **[decode]**. When you call this routine, you execute a serialization call instead of a remote call. The procedure arguments are marshaled to or unmarshaled from a buffer in the usual way. You then have complete control of the buffers.

In contrast, when your program performs type serialization (a type is labeled with serialization attributes), MIDL generates routines to size, encode, and decode objects of that type. To serialize data, you must call these routines in the appropriate way. Type serialization is a Microsoft extension and, as such, is not available when you compile in DCE-compatibility (**/osf**) mode. By using the **[encode]** and **[decode]** attributes as interface attributes, RPC applies encoding to all the types and routines defined in the IDL file.

You must supply adequately aligned buffers when using serialization services. The beginning of the buffer must be aligned at an address that is a multiple of 8, or 8-byte aligned. For procedure serialization, each procedure call must marshal into or unmarshal from a buffer position that is 8-byte aligned. For type serialization, sizing, encoding, and decoding must start at a position that is 8-byte aligned.

One way for your application to ensure that its buffers are aligned is to write the **midl_user_allocate** function such that it creates aligned buffers. The following code sample demonstrates how this can be done.

```
#define ALIGN_TO8(p)   (char *)((unsigned long)((char *)p + 7) & ~7)

void __RPC_FAR *__RPC_USER  MIDL_user_allocate(size_t sizeInBytes)
{
    unsigned char *pcAllocated;
    unsigned char *pcUserPtr;

    pcAllocated = (unsigned char *) malloc( sizeInBytes + 15 );
    pcUserPtr =  ALIGN_TO8( pcAllocated );
    if ( pcUserPtr == pcAllocated )
        pcUserPtr = pcAllocated + 8;

    *(pcUserPtr - 1) = pcUserPtr - pcAllocated;

    return( pcUserPtr );
}
```

The following example shows the corresponding **midl_user_free** function.

```
void __RPC_USER  MIDL_user_free(void __RPC_FAR *f)
{
    unsigned char * pcAllocated;
    unsigned char * pcUserPtr;

    pcUserPtr = (unsigned char *) f;
    pcAllocated = pcUserPtr - *(pcUserPtr - 1);

    free( pcAllocated );
}
```

Procedure Serialization

When you use procedure serialization, a procedure is labeled with the **[encode]** or **[decode]** attribute. Instead of generating the usual remote stub, the compiler generates a serialization stub for the routine.

Just as a remote procedure must use a binding handle to make a remote call, a serialization procedure must use a serialization handle to use serialization services. If a serialization handle is not specified, a default implicit handle is used to direct the call. On the other hand, if the serialization handle is specified, either as an explicit **handle_t** argument of the routine or by using the **[explicit_handle]** attribute, you must pass a valid handle as an argument of the call. For additional information on how to create a valid serialization handle, see *Serialization Handles*, *Examples of Fixed Buffer Encoding*, and *Examples of Incremental Encoding*.

Note Microsoft® RPC allows remote and serialization procedures to be mixed in one interface. However, use caution when doing so.

For remote procedures with implicit binding handles, the MIDL compiler generates a global handle variable of type **handle_t**. Procedures and types with implicit serialization handles use this same global handle variable.

For implicit handles, the global implicit handle must be set to a valid binding handle before a remote call. The implicit handle must be set to a valid serialization handle before a serialization call. Therefore, a procedure cannot be both remote and serialized. It must be one or the other.

Type Serialization

The MIDL compiler generates up to three functions for each type to which the **[encode]** or **[decode]** attribute is applied. For example, for a user-defined type named **MyType**, the compiler generates code for the **MyType_Encode**, **MyType_Decode**, and **MyType_AlignSize** functions. For these functions, the compiler writes prototypes to Stub.h and source code to Stub_c.c. Generally, you can encode a *MyType* object with **MyType_Encode** and decode an object from the buffer using **MyType_Decode**. **MyType_AlignSize** is used if you need to know the size of the marshaling buffer before allocating it.

The following encoding function is generated by the MIDL compiler. This function serializes the data for the object pointed to by *pObject*, and the buffer is obtained according to the method specified in the handle. After writing the serialized data to the buffer, you control the buffer. Note that the handle inherits the status from the previous calls, and the buffers must be aligned at 8.

For an implicit handle:

```
void MyType_Encode (MyType __RPC_FAR * pObject);
```

For an explicit handle:

```
void MyType_Encode (handle_t Handle, MyType __RPC_FAR * pObject);
```

The following function deserializes the data from the application's storage into the object pointed to by *pObject*. You supply a marshaled buffer according to the method specified in the handle. Note that the handle can inherit the status from the previous calls and the buffers must be aligned at 8.

For an implicit handle:

```
void MyType_Decode (MyType __RPC_FAR * pObject);
```

For an explicit handle:

```
void MyType_Decode (handle_t Handle, MyType __RPC_FAR * pObject);
```

The following function returns a size, in bytes, that includes the type instance plus any padding bytes needed to align the data. This enables serializing a set of instances of the same or different types into a buffer while ensuring that the data for each object is appropriately aligned. **MyType_AlignSize** assumes that the instance pointed to by *pObject* will be marshaled into a buffer beginning at the offset aligned at 8.

For an implicit handle:

```
size_t MyType_AlignSize (MyType __RPC_FAR * pObject);
```

For an explicit handle:

```
size_t MyType_AlignSize (handle_t Handle, MyType __RPC_FAR * pObject);
```

Note that both remote procedures with implicit binding handles and serialized types with implicit serialization handles use the same global handle variable. Therefore, it is advisable not to mix type serialization and remote procedures in an interface with implicit handles. For details, see *Implicit Versus Explicit Handles*.

Serialization Handles

An application uses the serializing procedures or the serializing support routines generated by the MIDL compiler in conjunction with a set of library functions to manipulate a serialization handle. Together, these functions provide a mechanism for customizing the way an application serializes data.

A serializing handle is required for any serializing operation, and all serializing handles must be managed explicitly by you. To do this, you first create a valid handle by calling one of the following routines:

- **MesDecodeBufferHandleCreate**
- **MesDecodeIncrementalHandleCreate**
- **MesEncodeDynBufferHandleCreate**
- **MesEncodeFixedBufferHandleCreate**
- **MesEncodeIncrementalHandleCreate**

You release the handle with a call to **MesHandleFree**. Once the handle has been created or reinitialized, it represents a valid serialization context and can be used to encode or decode, depending on the type of the handle.

Implicit Versus Explicit Handles

To declare a serialization handle, use the primitive handle type **handle_t**. Serialization handles can be explicit or implicit. Specify implicit handles in your application's ACF by using the **[implicit_handle]** attribute. The MIDL compiler will generate a global serialization handle variable. Serialization procedures with an implicit handle use this global variable in order to access a valid serializing context.

When using type encoding, the generated routines supporting serialization of a particular type use the global implicit handle to access the serialization context. Note that remote routines may need to use the implicit handle as a binding handle. Be sure that the implicit handle is set to a valid serializing handle prior to making a serializing call.

An explicit handle is specified as a parameter of the serialization procedure prototype in the IDL file, or it can also be specified by using the **[explicit_handle]** attribute in the ACF. The explicit handle parameter is used to establish the proper serialization context for the procedure. To establish the correct context in the case of type serialization, the compiler generates the supporting routines that use explicit **handle_t** parameter as the serialization handle. You must supply a valid serializing handle when calling a serialization procedure or serialization type support routine.

Serialization Styles

There are three styles you can use to manipulate serialization handles. These are:

- Fixed Buffer Serialization
- Dynamic Buffer Serialization
- Incremental Serialization

Regardless of the style you use, you must create a serialization handle, serialize the data, and then free the handle. The style is set when your program creates the handle and defines the way a buffer is manipulated. The handle maintains the appropriate context associated with each of the three serialization styles.

Fixed Buffer Serialization

When using the fixed buffer style, specify a buffer that is large enough to accommodate the encoding (marshalling) operations performed with the handle. When unmarshaling, you provide the buffer that contains all of the data to decode.

The fixed buffer style of serialization uses the following routines:

- **MesEncodeFixedBufferHandleCreate**
- **MesDecodeBufferHandleCreate**
- **MesBufferHandleReset**
- **MesHandleFree**

The **MesEncodeFixedBufferHandleCreate** function allocates the memory needed for the encoding handle, and then initializes it. The application can call **MesBufferHandleReset** to reinitialize the handle, or it can call **MesHandleFree** to free the handle's memory. To create a decoding handle corresponding to the fixed style–encoding handle, you must use **MesDecodeBufferHandleCreate**.

The application calls **MesHandleFree** to free the encoding or decoding buffer handle.

Examples of Fixed Buffer Encoding

The following section provides an example of how to use a fixed buffer style–serializing handle for procedure encoding.

```
/*This is a fragment of the IDL file defining MooProc */

...
void __RPC_USER
MyProc( [in] handle_t Handle, [in,out] MyType * pMyObject,
        [in, out] ThisType * pThisObject);
...

/*This is an ACF file. MyProc is defined in the IDL file */

[
    explicit_handle
]
interface regress
{
    [ encode,decode ] MyProc();
}
```

The following excerpt represents a part of an application.

```
if (MesEncodeFixedBufferHandleCreate (
        Buffer, BufferSize,
        pEncodedSize, &Handle) == RPC_S_OK)
{
    ...
    /* Manufacture a MyObject and a ThisObject */
    ...
    /* The serialization works from the beginning of the buffer because
   the handle is in the initial state. The function MyProc does the
   following when called with an encoding handle:
      - sizes all the parameters for marshalling,
      - marshalls into the buffer (and sets the internal state
    appropriately)
    */

    MyProc ( Handle, pMyObject, pThisObject );
    ...
    MesHandleFree ();
}
if (MesDecodeBufferHandleCreate (Buffer, BufferSize, &Handle) ==
    RPC_S_OK)
{

/*  The MooProc does the following for you when called with a decoding
    handle:
      - unmarshalls the objects from the buffer into *pMooObject and
        *pBarObject
*/

MyProc ( Handle, pMyObject, pThisObject);
...
MesHandleFree ( Handle );
}
```

The following section provides an example of how to use a fixed buffer style–serializing handle for type encoding.

```
/* This is an ACF file. MyType is defined in the IDL file */

[
    explicit_handle
]
interface regress
{
```

(continued)

(continued)

```
    typedef [ encode,decode ] MyType;
}
```

The following excerpt represents the relevant application fragments.

```
if (MesEncodeFixedBufferHandleCreate (Buffer, BufferSize,
    pEncodedSize, &Handle) == RPC_S_OK)
{
    ...
    /* Manufacture a MyObject and a pMyObject */
    ...
    MyType_Encode ( Handle, pMyObject );
    ...
    MesHandleFree ();
}
if (MesDecodeBufferHandleCreate (Buffer, BufferSize, &Handle) ==
    RPC_S_OK )
{
    MooType_Decode (Handle, pMooObject);
    ...
    MesHandleFree ( Handle );
}
```

Dynamic Buffer Serialization

When using the dynamic buffer style of serialization, the marshalling buffer is allocated by the stub, and the data is encoded into this buffer and passed back to you. When unmarshaling, you supply the buffer that contains the data.

The dynamic buffer style of serialization uses the following routines:

- **MesEncodeDynBufferHandleCreate**
- **MesDecodeBufferHandleCreate**
- **MesBufferHandleReset**
- **MesHandleFree**

The **MesEncodeDynBufferHandleCreate** function allocates the memory needed for the encoding handle and then initializes it. The application can call **MesBufferHandleReset** to reinitialize the handle. It calls **MesHandleFree** to free the handle's memory. To create a decoding handle corresponding to the dynamic buffer encoding handle, use **MesDecodeBufferHandleCreate**.

Incremental Serialization

When using the incremental style serialization, you supply three routines to manipulate the buffer. These routines are: **Alloc**, **Read**, and **Write**. The **Alloc** routine allocates a buffer of the required size. The **Write** routine writes the data into the buffer, and the **Read** routine retrieves a buffer that contains marshaled data. A single serialization call can make several calls to these routines.

The incremental style of serialization uses the following routines:

- **MesEncodeIncrementalHandleCreate**
- **MesDecodeIncrementalHandleCreate**
- **MesIncrementalHandleReset**
- **MesHandleFree**

The prototypes for the **Alloc**, **Read**, and **Write** functions that you must provide are shown below:

```
void __RPC_USER Alloc (
   void *State,           /* application-defined pointer */
   char **pBuffer,        /* returns pointer to allocated buffer */
   unsigned int *pSize); /* inputs requested bytes; outputs
                          /* pBuffer size */

void __RPC_USER Write (
   void *State,           /* application-defined pointer */
   char *Buffer,          /* buffer with serialized data */
   unsigned int Size);    /* number of bytes to write from Buffer */

void __RPC_USER Read (
   void *State,           /* application-defined pointer */
   char **pBuffer,        /* returned pointer to buffer with data */
   unsigned int *pSize); /* number of bytes to read into pBuffer */
```

The *State* input argument for all three functions is the application-defined pointer that was associated with the encoding services handle. The application can use this pointer to access the structure containing application-specific information, such as a file handle or stream pointer. Note that the stubs do not modify the *State* pointer other than to pass it to the **Alloc**, **Read**, and **Write** functions. During encoding, **Alloc** is called to obtain a buffer into which the data is serialized. Then, **Write** is called to enable the application to control when and where the serialized data is stored. During decoding, **Read** is called to return the requested number of bytes of serialized data from where the application stored it.

An important feature of the incremental style is that the handle keeps the state pointer for you. This pointer maintains the state and is never touched by the RPC functions, except when passing the pointer to **Alloc**, **Write**, or **Read** function. The handle also maintains an internal state that makes it possible to encode and decode several type instances to the same buffer by adding padding as needed for alignment. The **MesIncrementalHandleReset** function resets a handle to its initial state to enable reading or writing from the beginning of the buffer.

The **Alloc** and **Write** functions, along with an application-defined pointer, are associated with an encoding-services handle by a call to the **MesEncodeIncrementalHandleCreate** function. **MesEncodeIncrementalHandleCreate** allocates the memory needed for the handle and then initializes it.

The application can call **MesDecodeIncrementalHandleCreate** to create a decoding handle, **MesIncrementalHandleReset** to reinitialize the handle, or **MesHandleFree** to free the handle's memory. The **Read** function, along with an application-defined parameter, is associated with a decoding handle by a call to the **MesDecodeIncrementalHandleCreate** routine. The function creates the handle and initializes it.

The *UserState, Alloc, Write,* and *Read* parameters of **MesIncrementalHandleReset** can be NULL to indicate no change.

Examples of Incremental Encoding

The following section provides an example of how to use the incremental style serializing handle for type encoding.

```
/* This is an acf file. MooType is defined in the idl file */

[   explicit_handle
]
interface regress
{
typedef [ encode,decode ] MooType;
}
```

The following excerpt represents the relevant application fragments.

```
if (MesEncodeIncrementalHandleCreate (State, AllocFn, WriteFn,
    &Handle) == RPC_S_OK)
{
...
/* The serialize works from the beginning of the buffer because
    the handle is in the initial state. The Moo_Encode does the
    following:
    - sizes *pMooObject for marshalling,
    - calls AllocFn with the size obtained,
    - marshalls into the buffer returned by Alloc, and
    - calls WriteFn with the filled buffer
*/

Moo_Encode ( Handle, pMooObject );
...
}
```

```
if (MesIncrementalHandleReset (Handle, NULL, NULL, NULL, ReadFn,
    MES_DECODE ) == RPC_OK)
{
/*The ReadFn is needed to reset the handle. The arguments
    that are NULL do not change. You can also call
    MesDecodeIncrementalHandleCreate (State, ReadFn, &Handle);
    The Moo_Decode does the following:
    - calls Read with the appropriate size of data to read and
        receives a buffer with the data
    - unmarshalls the object from the buffer into *pMooObject
*/

Moo_Decode ( Handle, pMooObject );
...
MesHandleFree ( Handle );
}
```

Obtaining an Encoding Identity

An application that is decoding encoded data can obtain the identity of the routine used to encode the data, prior to calling a routine to decode it. The **MesInqProcEncodingId** routine provides this identity.

Each procedure in an interface is assigned an integer identification number, called a *procedure ID* or a *proc ID*, by the MIDL compiler. Numbering begins with zero. The RPC run-time libraries are not involved in translating the procedure ID into an actual procedure call. Given a proc ID, your application must provide a means of calling the correct procedure. Typically, application developers use a series of **if** statements, or (when using C/C++) a **switch** statement for this purpose.

CHAPTER 18

Security

With the increased use of distributed applications, the need for secure communications between the client and server portions of applications is of paramount importance. The Remote Procedure Call (RPC) run-time library provides a standardized interface to authentication services for both clients and servers. The authentication services on the server host system provide RPC authentication. Applications use authenticated remote procedure calls to ensure that all calls come from authorized clients. They can also help ensure that all server replies come from authenticated servers.

RPC Security Essentials

To complete any remote procedure call, all distributed applications must create a binding between the client and the server. For more information on bindings, see *Binding and Handles*. After the client obtains a binding to the server, both the client and server portions of the distributed application use the RPC authorization functions to create an authenticated binding. The binding is predicated upon the rights contained in the user's security credentials.

This section explains the essential concepts and information required to use the RPC functions to create a client and server for an authenticated distributed application.

Principal Names

In order for a client to create an authenticated session with a server program, it must provide the server's expected principal name. A principal is an entity that the security system recognizes. This includes human users as well as autonomous processes. All principal names take the same form for a given security support provider (SSP). An SSP is a software module that performs security validation. For more information, see *SSPI Architectural Overview*. Strictly speaking, the SSP cannot tell the difference between users who are logged on and processes running on the computer. It sees both as principals with principal names. Therefore, principal names take the same form as user names.

The server registers its principal name for the security provider. The SSP dictates the format of the principal name. For example, the Kerberos protocol SSP requires that the principal name be in the form **servername** or **domain\servername**, where **servername** is the server program's account name.

The SCHANNEL SSP takes principal names in either of two forms. The first is the *msstd* form. Names in msstd form generally follow the pattern **msstd:servername@serverdomain.com**. This is referred to as an email name property. If the certificate contains an email name property, and it contains an at sign (@), the principal name is msstd:email name. Otherwise it must contain the common name property. If neither exists, SCHANNEL SSP returns the message ERROR_INVALID_PARAMETER. Internal backslashes are doubled, just as in string bindings.

The second SCHANNEL principal name form is *fullsic* form. This is a series of RFC1779-compliant names bounded by angle brackets and separated by backslashes. It typically follows the pattern **fullsic:\<\Authority\SubAuthority\.....\Person>** or **fullsic:\<\Authority\SubAuthority\.....\ServerProgram>**.

Server programs invoke the **RpcServerRegisterAuthInfo** function to register their principal names. Pass the server's principal name as the first parameter.

To query for the server's principal name, applications can call **RpcMgmtInqServerPrincName**. This allocates a null-terminated string to hold the principal name. Before it terminates, your application must invoke **RpcStringFree** to release the memory this string occupies

Querying for the server name in this manner is not the most secure method of connecting to a server. For server authentication, the client program should "know" the name of the secure server by some method that does not involve transmitting the server's name over the network.

Authentication Levels

Microsoft® RPC provides multiple levels of authentication. Authentication can take place each time the client establishes a connection with the server, each time the client executes a remote procedure call, or each time the client and the server exchange a packet of data.

In addition, the RPC run-time library can validate that the packet came from the client program. This does not mean, however, that the packet was not modified or corrupted en route, only that it came from the authenticated client. For greater security, distributed applications can set the RPC run-time library to verify that none of the data exchanged between the client and server is modified. The RPC library can also encrypt the contents of every packet before sending it.

Be aware that higher levels of authentication require higher computational overhead. You, as the developer, must decide which is more important for your application speed or security. Most developers find that with some performance testing, they can achieve acceptable performance levels while maintaining adequate security.

The client and the server portions of the distributed application must use the same authentication level. For a list of RPC authentication levels, see *Authentication-Level Constants*.

Authentication Services

The Security Support Provider Interface (SSPI) provides the underlying authentication services for RPC. Therefore, when your application specifies an authentication service, it selects an SSP. For a list of the SSPs that SPPI currently supports, see *Authentication-Service Constants*. For more information on the SSPI, see *Security Support Provider Interface (SSPI).*

Client Authentication Credentials

Every authenticated client must provide authentication credentials to the server. Under RPC, the client stores its authentication credentials in the binding between the client and the server. To do this, it calls **RpcBindingSetAuthInfo** or **RpcBindingSetAuthInfoEx**. The fifth parameter of these two functions is of type **RPC_AUTH_IDENTITY_HANDLE**. This is a flexible type that is a pointer to a data structure. What the data structure actually contains can differ with each authentication service. Currently, the SSPs that RPC supports require that your application set **RPC_AUTH_IDENTITY_HANDLE** to point to a **SEC_WINNT_AUTH_IDENTITY** structure. The **SEC_WINNT_AUTH_IDENTITY** structure contains fields for a user name, domain, and password.

Authorization Services

An authorization service is the method that the SSP uses to authorize access to a remote procedure. SSPs can provide more than one authorization service. However, they usually select one as a default.

Your application can use the default authorization method for the current SSP, or it can specify one. At present, Microsoft RPC supports two methods of authorization. One is for the server to provide authorization based on the name of the client program. The other is for the server to compare the client's authentication credentials against the server's access control list (ACL).

For a list of authorization services, see *Authorization-Service Constants*.

Quality of Service

Client programs can use the **RpcBindingSetAuthInfoEx** function rather than the **RpcBindingSetAuthInfo** function to create an authenticated binding. If they do, they pass a pointer to an **RPC_SECURITY_QOS** structure as the final parameter of **RpcBindingSetAuthInfoEx**. This structure contains information about the quality of service. Specifically, the information in this structure allows client programs to set the security services provided to the distributed application. Client programs can also specify the identity tracking and select the impersonation type. In addition, client programs can use it to validate the RPC version number.

Use the **Capabilities** member of the **RPC_SECURITY_QOS** structure to set which portions of your client/server application are authenticated. If you select **RPC_C_QOS_CAPABILITIES_DEFAULT**, the RPC run-time library will authenticate the client or server according to the default for the SSP. By default, the Kerberos protocol SSP authenticates both the client and the server. The default for all other SSPs that Microsoft provides is to authenticate the client to the server, but not to authenticate the server to the client.

If you always want the client and the server to authenticate themselves to each other, set the **Capabilities** member of the **RPC_SECURITY_QOS** structure to RPC_C_QOS_CAPABILITIES_MUTUAL_AUTH. If you are using the SCHANNEL SSP, you can also set the **Capabilities** member to RPC_C_QOS_CAPABILITIES_ANY_AUTHORITY. This constant specifies that the SSP will validate the remote procedure call even if the certificate authority that issued the client's authentication certificate is not in the SSP's root certificate store. The default is to reject the certificate if the SSP does not recognize the certificate authority. The certificate authority is an independent company or organization, such as Verisign, that issues authentication certificates.

Your application can also set the identity tracking that the RPC run-time library uses. Typically, programs use *static identity tracking*, which is the fastest. With static tracking, the client's credentials are set when it calls **RpcBindingSetAuthInfo**. The RPC run-time library then uses those credentials for all RPC calls on the binding. In addition, applications can select *dynamic identity tracking*. Dynamic identity tracking means that the RPC run-time library will use the credentials of the calling thread, rather than the binding handle, for authentication each time the client calls a remote procedure. It is typically only used if the client impersonates different users, or if the server calls the **RpcImpersonateClient** function. Static identity tracking is faster.

As part of the QOS specification, the client program can also set the type of impersonation that a server program can perform on its behalf. For more information on impersonation, see *Client Impersonation*.

The version number field of the **RPC_SECURITY_QOS** structure should always be set to RPC_C_SECURITY_QOS_VERSION.

Authorization Functions

Each time a server program receives a client request for access to a remote procedure, the RPC run-time library invokes a default authorization function. This function uses the SSP to check the client's credentials and authorize or reject the request.

Your server program can override the authorization function that the SSP provides. Invoke the function **RpcMgmtSetAuthorizationFn** and pass it the address of your authorization function. Once the server program sets the authorization function, the RPC run-time library will call it every time the server program receives a client request. For related information, see ***RpcMgmtIsServerListening***, ***RpcMgmtStopServerListening***, ***RpcMgmtInqIfIds***, ***RpcMgmtInqServerPrincName***, and ***RpcMgmtInqStats***.

Key Acquisition Functions

By default, the SSP supplies encryption keys to the server programs that request them. Each SSP implements its own system of generating keys. The format of the keys the SSP generates are specific to the SSP.

RPC provides you with the ability to override the default method of generating encryption keys. Your application can call the **RpcServerRegisterAuthInfo** function and pass it a pointer to a key acquisition function. You can write the key acquisition function so that it generates keys using any method you choose. However, the key it passes to the server program must match the format that the SSP requires.

Client Impersonation

Impersonation is useful in a distributed computing environment when servers must pass client requests to other server processes or to the operating system. In Figure 18-1, a server impersonates the client's security context. Other server processes can then handle the request as if the original client made it.

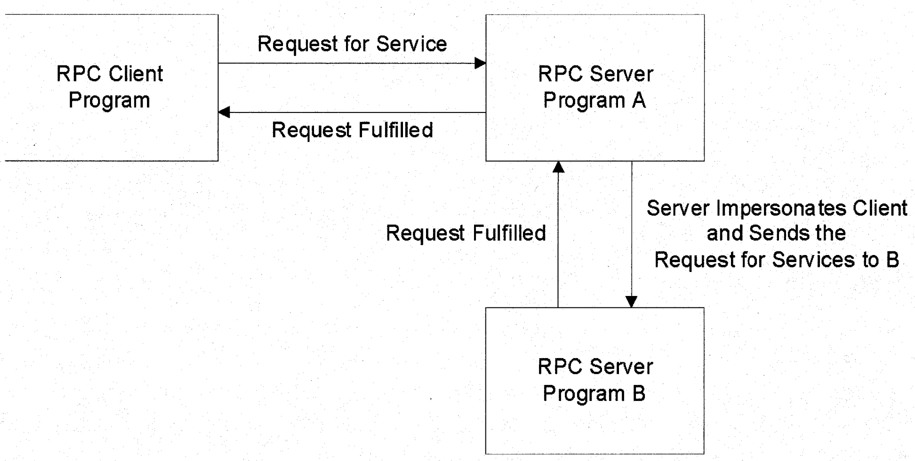


Figure 18-1: Server Impersonating the Client's Security Context.

For example, a client makes a request to Server A. If Server A must query Server B to complete the request, Server A impersonates the client security context and makes the request to Server B on behalf of the client. Server B uses the original client's security context, instead of the security identity for Server A, to determine whether to complete the task.

The server calls **RpcImpersonateClient** to overwrite the security for the server thread with the client security context. After the task is completed, the server calls **RpcRevertToSelf** or **RpcRevertToSelfEx** to restore the security context defined for the server thread.

When binding, the client can specify quality-of-service information about security which specifies how the server can impersonate the client. For example, one of the settings lets the client specify that the server is not allowed to impersonate it. For more information, see *Quality of Service*.

Security Methods

Microsoft® RPC supports two different methods for adding security to your distributed application. The first method is to use the Security Support Provider Interface (SSPI), which can be accessed using the RPC functions. In general, it is best to use this method. The SSPI provides the most flexible and network-independent authentication features.

The second method is to use the security features built into the Microsoft® Windows NT® and Windows® 2000 operating system transport protocols. The transport-level security method is not the preferred method. Using the SSPI is recommended because it works on all transports, across platforms, and provides high levels of security, including privacy. The following sections provide overviews of both SSPI and transport-level security.

Security Support Provider Interface (SSPI)

In conjunction with its operating systems, Microsoft offers the Security Support Provider Interface (SSPI). The SSPI provides a universal, industry-standard interface for secure distributed applications. SSPI is supported by Microsoft RPC for Windows NT, Windows 2000, Windows 95/98, MS-DOS®, Windows 3.1, and Macintosh.

SSPI Architectural Overview

SSPI is a software interface. Distributed programming libraries such as RPC can use it for authenticated communications. One or more software modules provide the actual authentication capabilities. Each module, called a security support provider (SSP), is implemented as a dynamic link library (DLL). An SSP provides one or more security packages.

A variety of SSPs and packages are available. For instance, Windows NT and Windows 2000 ship with the NTLM security package. Beginning with Windows 2000, Microsoft also provides the Microsoft Kerberos protocol security package. In addition, you may choose to install the Secure Socket Layer (SSL) security package. These security packages are implemented by the Microsoft® Win32® SSP, which is implemented in Secur32.dll. You may also choose to install any other SSPI-compatible SSP.

Using SSPI ensures that no matter which SSP you select, your application accesses the authentication features in a uniform manner. This capability provides your application greater independence from the implementation of the network than was available in the past.

The architecture of SSPI is illustrated in Figure 18-2.

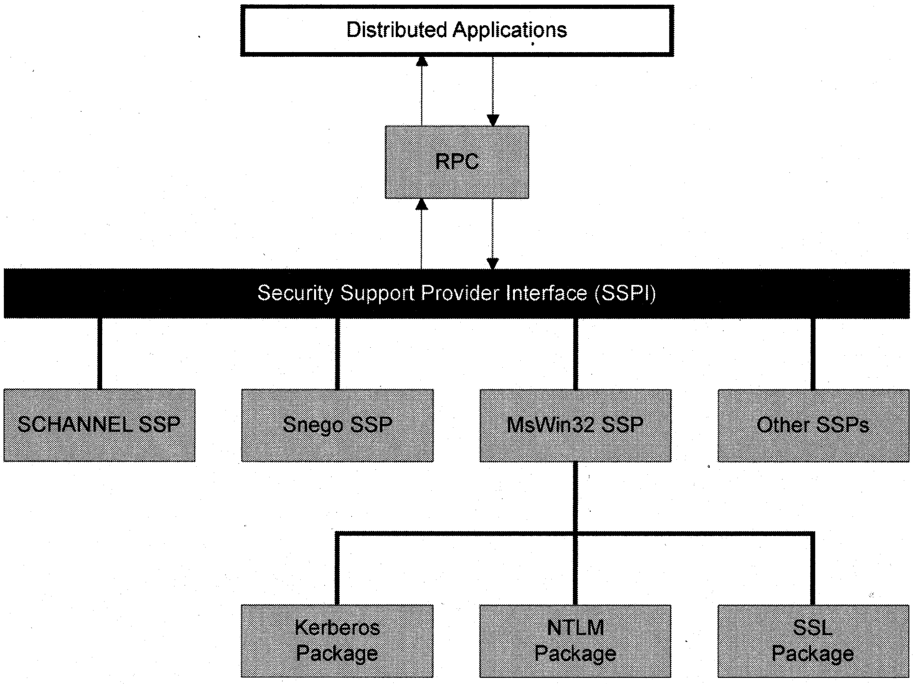


Figure 18-2: SSPI Architecture.

The preceding illustration shows that distributed applications communicate through the RPC interface. The RPC software in turn, accesses the authentication features of an SSP through the SSPI.

For more information, see *Security Support Provider Interface.*

Security Support Providers (SSPs)

Beginning with Windows 2000, RPC supports a variety of security providers and packages. These include:

- **Kerberos Protocol Security Package**. Kerberos v5 protocol is an industry-standard security package. It uses fullsic principal names.
- **SCHANNEL SSP**. This SSP implements the Microsoft Unified Protocol Provider security package, which unifies SSL, Private Communication Technology (PCT), and Transport Level Security (TLS) into one security package. It recognizes msstd and fullsic principal names.
- **Distributed Password Authentication (DPA) Security Package**. Written primarily for authentication over the Internet, the DPA security package provides the capabilities for a user database that scales to millions of users. It uses the same principal name form as the NTLM security package.

- **MSN Security Package**. MSN™, the network of Internet services, currently uses this security package. It uses the same principal name form as the NTLM security package.
- **NTLM Security Package**. This was the primary security package for NTLM networks prior to Windows 2000.
- **Distributed Computing Environment (DCE) Security Package**. The DCE security package is an industry-standard security protocol that provides private and key authentication.

In addition, Microsoft RPC provides a *pseudo-SSP* that enables applications to negotiate between the use of real SSPs. This pseudo-SSP, called the Simple GSS-API Negotiation Mechanism (Snego) SSP, does not provide any actual authentication features. Its only use is to help applications select a real SSP. Currently, client and server programs can use the Snego SSP to negotiate between the use of the NTLM security package and Kerberos protocol security package.

For more information on selecting SSPs, see *Authentication-Service Constants*.

All of the SSPs that Microsoft Corporation provides recognize authentication credentials in the form provided by the **SEC_WINNT_AUTH_IDENTITY** structure. For details, see *Client Authentication Credentials*. For information on how to use specific SSPs, see *SSPI Functions* and *Using the Schannel Security Provider* in the Security documentation of the Platforms SDK.

Writing an Authenticated SSPI Client

All RPC client/server sessions require a binding between the client and the server. To add security to client/server applications, the programs must use an authenticated binding. This section describes the process of creating an authenticated binding between the client and the server. It also presents special platform-specific considerations that are imposed on developers of client programs.

- Creating Client-side Binding Handles
- MS-DOS Considerations
- Windows 95/98 Considerations
- Providing Client Credentials to the Server

For related information, see *Procedures Used with Most Security Packages and Protocols* in the Platform SDK.

Creating Client-side Binding Handles

To create an authenticated session with a server program, client applications must provide authentication information with their binding handle. To set up an authenticated binding handle, clients invoke the **RpcBindingSetAuthInfo** or **RpcBindingSetAuthInfoEx** function. These two functions are nearly identical. The only difference between them is that the client can specify the quality of service with the **RpcBindingSetAuthInfoEx** function. The following code fragment shows how a call to **RpcBindingSetAuthInfo** might look.

```
// This code fragment assumes that rpcBinding is a valid binding
// handle between the client and the server. It also assumes that
// pAuthCredentials is a valid pointer to a data structure which
// contains the user's authentication credentials.

RPC_STATUS RPC_ENTRY rpcStatus;

rpcStatus = RpcBindingSetAuthInfo(
    rpcBinding,                       // Valid binding handle
    "ServerName",                     // Principal name
    RPC_C_AUTHN_LEVEL_PKT_INTEGRITY,  // Authentication level
    RPC_C_AUTHN_GSS_KERBEROS,         // Use KERBEROS SSP
    pAuthCredentials,                 // Authentication credentials
    RPC_C_AUTHZ_NAME);                // Authorization service
```

After the client successfully calls the **RpcBindingSetAuthInfo** or **RpcBindingSetAuthInfoEx** functions, the RPC run-time library automatically authenticates all RPC calls on the binding. The level of security and authentication that the client selects applies only to that binding handle. Context handles derived from the binding handle will use the same security information, but subsequent modifications to the binding handle will not be reflected in the context handles. For more information on context handles, see *Context Handles*.

The authentication level stays in effect until the client chooses another level, or until the process terminates. Most applications will not require a change in the security level.

The client can query any binding handle to obtain its authorization information. Invoke the function **RpcBindingInqAuthClient** and pass it the binding handle.

MS-DOS Considerations

When developing applications for MS-DOS, you must manually feed the password and credential information into **RpcBindingSetAuthInfo**. This is optional for a 16-bit or 32-bit Windows platform because the default is to use the credentials of the current user. If a computer running Windows 95/98, Windows for Workgroups, or Windows 3.*x* is not part of a domain, the user will be prompted for the password.

To manually pass credentials to **RpcBindingSetAuthInfo**, create a pointer to the **SEC_WINNT_AUTH_IDENTITY** structure. Pass in the credential information using the *AuthIdentity* parameter. Note that this structure must remain valid for the lifetime of the binding handle.

Windows 95/98 Considerations

For systems configured for Novell NetWare clients on a Windows 95/98 network, the server side of the application must obtain the server principal name, and then pass this value to **RpcServerRegisterAuthInfo**. Use the **RpcServerInqDefaultPrincName** routine to obtain the server principal name. In this situation, the client calls **RpcBindingSetAuthInfo** in the usual manner, but specifies a value of NULL for *PrincipalName*. Behind the scenes, the Windows 95/98 run-time library queries the server to obtain the value of the *PrincipalName* parameter specified to **RpcServerRegisterAuthInfo**. This is the name that is actually used. The binding handle will be authenticated on the NetWare server.

For Windows 95/98, if **RpcBindingSetAuthInfo** is called with a NULL server principal name (as described above), the binding handle must be fully bound. For more information on fully-bound handles, see *Binding and Handles*. If it is a dynamic endpoint in which the server registers the endpoint with the endpoint mapper and, therefore, is not known by the client, you must use **RpcEpResolveBinding** to bind the handle. This is because in order to obtain the principal name from the server, the Windows 95/98 run-time library implicitly calls **RpcMgmtInqServerPrincName**. Calls to management interfaces cannot be made to unbound handles. All RPC server processes have the same management interface. Registering the handle with the endpoint mapper is not sufficient to uniquely identify a server. For more information on endpoints and the endpoint mapper, see *endpoint*.

Note The Windows 95/98 run-time library ignores the **ncacn_np** and **ncalrpc** security descriptors, because Windows 95/98 does not support the Windows NT/Windows 2000 security model.

Providing Client Credentials to the Server

Servers use the client's binding information to enforce security. Clients always pass a binding handle as the first parameter of a remote procedure call. However, servers cannot use the handle unless it is declared as the first parameter to remote procedures in either the IDL file or in the server's Application Configuration File (ACF). You can choose to list the binding handle in the IDL file, but this forces all clients to declare and manipulate the binding handle rather than using automatic or implicit binding. For further information, see *The IDL and ACF Files*.

Another method is to leave the binding handles out of the IDL file and to place the **explicit_handle** attribute into the server's ACF. In this way, the client can use whatever type of binding is best suited to the application, while the server uses the binding handle as though it were declared explicitly.

The process of extracting the client credentials from the binding handle is as follows:

- RPC clients call **RpcBindingSetAuthInfo** and include their authentication information as part of the binding information passed to the server.

- Usually, the server calls **RpcImpersonateClient** in order to behave as though it were the client. If the binding handle is not authenticated, the call fails with RPC_S_NO_CONTEXT_AVAILABLE. To obtain the client's user name, call **GetUserName** while impersonating.
- The server will normally use the Windows NT or Windows 2000 call **CreatePrivateObjectSecurity** to create objects with ACLs. After this is accomplished, later security checks become automatic.

Writing an Authenticated SSPI Server

Before authenticated communication can take place between the client and server programs, the server must register its authentication information. In particular, the server must register its principal name and specify the authentication service it uses. For more information on principal names, see *Principal Names*. For details about authentication services, see *Authentication Services*.

To register its authentication information, servers call the **RpcServerRegisterAuthInfo** function. Pass a pointer to the principal name as the value of the first parameter. Set the second parameter to a constant indicating the authentication service that the application will use. For a description of authentication services, see ***Authentication-Service Constants***.

The server may also pass the address of a key acquisition function as the value of the third parameter. See *Key Acquisition Functions*. To use the default key acquisition function for the selected authentication service, set the third parameter to NULL. The last parameter to the **RpcServerRegisterAuthInfo** function is a pointer data to pass to the key acquisition function, if you provide a key acquisition function. A call to **RpcServerRegisterAuthInfo** is shown in the following code fragment.

```
RPC_STATUS RPC_ENTRY rpcStatus;

rpcStatus = RpcServerRegisterAuthInfo (
    "ServerName",                     // Server principal name
    RPC_C_AUTHN_GSS_KERBEROS,         // Authentication service
    NULL,                             // Use default key function
    NULL);                            // No arg for key function
```

In addition, the server may provide the RPC run-time library with an authorization function. This callback function allows server programs to implement custom authentication methods. For details, see *Authorization Functions*. Every time a client request arrives, the RPC run-time library calls the authorization function. Your authorization function can then examine the query and authorize or deny the completion of the remote procedure call. To set an authorization function, call the **RpcMgmtSetAuthorizationFn** function.

The server portion of a distributed application can call the function **RpcBindingInqAuthInfo** to query a binding handle for authentication information.

If your server registers with a security support provider, client calls with invalid credentials will not be dispatched. However, calls with no credentials will be dispatched. There are three ways to keep this from happening:

- Register the interface using **RpcServerRegisterIfEx**, with a security callback function; this will cause the RPC run-time library to automatically reject unauthenticated calls to that interface.
- Call **RpcBindingInqAuthClient** to determine the security level that the client is using. Your stub can then return an error if the client is unauthenticated.
- Only allow calls using the RPC_C_AUTHN_PACKET_PRIVACY level. Then, all server replies will be encrypted during transmission.

Note If you are using the NTLM security package (with the authentication-service constant RPC_C_AUTHN_WINNT), you should be aware that a client whose credentials specify an unknown user name will be given *guest* access permission. If you do not want this behavior, remove the guest account from your server.

The NTLM security package also lets your server impersonate the client. To do this, call the **RpcImpersonateClient** function. For more information on the Windows NT security model, see *Access Control Model*.

If you need additional information on how to write a secure server, check with the manufacturer of your security support provider.

Windows NT and Windows 2000 Transport Security

Although this is not the preferred method, you can use the security settings that the Windows NT and Windows 2000 named pipe transport offers to add security features to your distributed application. These security settings are used with the Microsoft RPC functions that start with the prefixes **RpcServerUseProtseq** and **RpcServerUseAllProtseqs**, and the functions **RpcImpersonateClient** and **RpcRevertToSelf**. Microsoft Windows 95/98 does not support transport security in named pipes or local procedure calls.

Note If you are running an application that is a service and you are using NTLM for security, you must add an explicit service dependency for your application. The Secur32.dll will call the Service Controller (SC) to begin the NTLM security package service. However, an RPC application that is a service and is running as a system, must also contact the SC unless it is connecting to another service on the same computer.

Using Transport-Level Security on the Server

This section presents discussions of transport-level security, divided into the following topics:

- Using Transport-Level Security on the Server
- Using Transport-Level Security on the Client

When you use **ncacn_np** or **ncalrpc** as the protocol sequence, the server specifies a security descriptor for the endpoint at the time it selects the protocol sequence. For more information on protocol sequences, see *Specifying Protocol Sequences*. Your application provides the security descriptor as an additional parameter (an extension to the standard OSF-DCE parameters) on all functions that start with the prefixes "**RpcServerUseProtseq**" and "**RpcServerUseAllProtseqs**". The security descriptor controls whether a client can connect to the endpoint.

Each Windows NT and Windows 2000 process and thread is associated with a security token. This token includes a default security descriptor that is used for any objects that the process creates, such as the endpoint. If your application does not specify a security descriptor when calling a function with the prefixes "**RpcServerUseProtseq**" and "**RpcServerUseAllProtseqs**", the RPC run-time library applies the default security descriptor from the process security token to the endpoint.

To guarantee that the server application is accessible to all clients, the administrator should start the server application on a process that has a default security descriptor that all clients can use. On Windows NT and Windows 2000, generally only system processes have a default security descriptor.

For more information about these functions and the functions **RpcImpersonateClient** and **RpcRevertToSelf**.

Using Transport-Level Security on the Client

The client specifies how the server impersonates the client when the client establishes the string binding. This quality-of-service information is provided as an endpoint option in the string binding. The client can specify the level of impersonation, dynamic or static tracking, and the effective-only flag.

▶ To supply quality-of-service information for the server

1. The client imports a handle from the name-service database.

   The client specifies the name of the name-service database entry and obtains a binding handle.
2. The client calls **RpcBindingToStringBinding** to obtain the protocol sequence, network address, and endpoint.
3. The client calls **RpcStringBindingParse** to split the string binding into its component substrings.

4. The client verifies that the protocol sequence is equal to **ncacn_np** or **ncalrpc**.

   Client quality-of-service information is supported only on named pipes and LRPC in Microsoft RPC.
5. The client adds the security information to the endpoint string as an option.

   For more information about the syntax, see ***String Binding***.
6. The client calls **RpcStringBindingCompose** to reassemble the component strings, including the new endpoint options, in the correct string-binding syntax.
7. The client calls **RpcBindingFromStringBinding** to obtain a new binding handle and to apply the quality-of-service information for the client.
8. The client makes remote procedure calls using the handle.

Microsoft RPC supports Windows NT and Windows 2000 security features only on **ncacn_np** and **ncalrpc**. Windows NT and Windows 2000 security options for other transports are ignored.

Note Because it does not support the Windows NT and Windows 2000 security models, the Windows 95 run-time library ignores the security descriptors **ncalrpc** and **ncacn_np**.

The client can associate the following security parameters to the binding for the named-pipe transport **ncacn_np** or **ncalrpc**:

- **Identification**, **Impersonation**, or **Anonymous**. Specifies the type of security used.
- **Dynamic** or **Static**. Determines whether security information associated with a thread is a copy of the security information created at the time the remote procedure call is made (static) or a pointer to the security information (dynamic).

  Static security information does not change. The dynamic setting reflects the current security settings, including changes made after the remote procedure call is made.
- **TRUE** or **FALSE**. Specifies the value of the effective-only flag. A value of TRUE indicates that only security settings set to on at the time of the call are effective. A value of FALSE indicates that all security settings are available and that the application can modify them.

Any combination of these settings can be assigned to the binding, as shown in the following example:

```
"Security=Identification Dynamic True"
"Security=Anonymous Static True"
"Security=Impersonation Static False"
```

Default security-parameter settings vary according to the transport protocol.

For more information about the security features of Windows NT and Windows 2000, see your Windows NT and Windows 2000 documentation. For additional information about the syntax of the endpoint options, see *endpoint*.

CHAPTER 19

Installing and Configuring RPC Applications

When the Microsoft® Windows NT®, Microsoft Windows® 2000, or Windows 95 operating system is installed on a server or client, setup automatically installs the RPC run-time files. No further RPC installation is required. You must ensure, however, that the version of Windows you install supports all the features used in your distributed application.

When you use an RPC application on a Windows 3.*x* or Microsoft® MS-DOS® operating system, you must copy the RPC run-time executable files to the Windows 3.*x* or MS-DOS computers that will be using the application. The directory \mstools\rpc_rt16 on the Platform SDK CD contains a disk image of these files along with a setup program to install the files. Use this disk image to create an install disk for distribution with your RPC application. You can also use 16-bit client applications targeted toward MS-DOS or Windows on a 32-bit Windows operating system. However, your application's installation program must install the executable files contained in this disk image.

When you build an RPC application for a Macintosh client, you must link the necessary files to the application at build time. No additional RPC installation is needed.

For more details about redistributable files and licensing agreements, see \LICENSE\Redist.txt and \LICENSE\License.txt on the Platform SDK CD.

This chapter contains information about setting up RPC applications on a variety of hardware and software platforms.

Configuring the Name Service Provider

If your distributed application registers its interface with a name service, both the client and server must be using the same name service. Microsoft® RPC interoperates with Microsoft Locator and any name service provider (NSP) that adheres to the Microsoft RPC name service interface (NSI)—for example, the DCE Cell Directory Service accessed through Digital Equipment Corporation's name service daemon (NSID). Microsoft Locator, which is designed for use in Microsoft Windows® environments, is the default name service provider.

Configuring the Name Service for Windows 95

Microsoft Windows 95 does not use Microsoft Locator. In order to use a name service in a Windows 95 application, the computer with Windows 95 must either:

- Be part of a workgroup or domain that includes a computer running Microsoft® Windows NT® or Windows 2000 to serve as a proxy name service provider.
- Be connected to a host computer running the NSI daemon (nsid), which serves as a gateway to the Digital Equipment Corporation DCE Cell Directory Service.

Editing the Windows 95 Registry

You can use REGEDIT to edit the Windows 95 registry to designate an NSP.

▶ To designate a Microsoft Locator name service provider for Windows 95

1. Select **HKEY_LOCALMACHINE\SOFTWARE\Microsoft\Rpc**.
2. Create a new key called **NameService**.
3. With the **NameService** key selected, create the new **StringValue** names and modify them to contain data as shown in the following table.

Name	Data
4DefaultSyntax	3
Protocol	**ncacn_np**
Endpoint	\pipe\locator
NetworkAddress	"myserver" (where myserver is the name of the Windows NT computer)
ServerNetworkAddress	myserver

You can use REGEDIT to edit the Windows 95 registry to designate a DCE CDS NSP.

▶ To designate a DCE CDS name service provider for Windows 95

- Edit the Windows 95 registry as described in the preceding, using the data shown in the following table.

Name	Data
DefaultSyntax	3
Protocol	**ncacn_ip_tcp**
Endpoint	" " (an empty string)
NetworkAddress	myserver (the name of the host computer running nsid)
ServerNetworkAddress	myserver (the name of the host computer running nsid)

Note You must have the Digital Equipment Corporation DCE Cell Directory Service product to configure the DCE CDS as your name service provider. See the documentation provided by Digital Equipment Corporation for information about DCE CDS.

Configuring the Name Service for Windows NT or Windows 2000

When you install the Platform SDK on Windows NT or Windows 2000, the setup program automatically selects Microsoft Locator as the name service provider. You can change the name service provider through Control Panel.

▶ To reconfigure the name service provider for Windows NT and Windows 2000

1. In Control Panel, click the **Network** icon.

   The **Network** dialog box appears.
2. In the **Network** dialog box, click **Configure**.
3. In the **NetworkSoftware** list, select **RPC Configuration**.

   The **RPC Name Service Provider Configuration** dialog box appears.
4. In the **RPC Name Service Provider Configuration** dialog box, select a name service provider from the list.
   a. When you select Microsoft Locator, click **OK**. The configuration process is then complete.
   b. When you select the DCE Cell Directory Service, in the **Network Address** box, type the name of the host computer that runs the NSI daemon (nsid), and then click **OK**.

The host computer that runs the nsid acts as a gateway to the DCE Cell Directory Service, passing NSI function calls between computers that run Microsoft operating systems and DCE computers. A network address can be up to 80 characters—for example, 11.1.9.169 is a valid address.

Note You must have the Digital Equipment Corporation DCE CDS product to configure the DCE CDS as your name service provider. See the documentation provided by Digital Equipment Corporation for information about DCE CDS.

Configuring the Name Service for Windows 3.*x* or MS-DOS

When you run Setup.exe to install the 16-bit RPC run-time library, it prompts you to select a name service provider:

- If you choose **Install Default Name Service Provider**, the default NSP, Microsoft Locator, is installed. Microsoft Locator works in Windows NT and Windows 2000 domains.
- If you choose **Install Custom Name Service Provider**, complete the **Define Network Address** dialog box to install the DCE Cell Directory Service as your NSP. The DCE Cell Directory Service is the NSP used with DCE servers.

The network address is the name of the host computer that runs the NSI daemon (nsid). This computer acts as a gateway to the DCE Cell Directory Service, passing NSI function calls between computers that run Microsoft operating systems and DCE computers. The network address can be 80 characters or less—for example, 11.1.9.169 is a valid address.

Configuring the Microsoft Locator Name Service Provider

You can change the name service provider you specified by editing the Rpcreg.dat configuration file, which contains the NSP parameters and RPC protocol settings. Use a text editor to change NSP entries.

▶ To reconfigure the Microsoft Locator name service provider

1. Open the Rpcreg.dat file using a text editor.

   Rpcreg.dat is in the root directory unless you specified a different path during setup.
2. Set the following values for the registry entries.

Registry entry	Value
Software\Microsoft\RPC \NameService\Protocol	The protocol sequence for the protocol you are using. The default is **ncacn_np**.
Software\Microsoft\RPC\ NameService\NetworkAddress	The name of the computer running Locator that is used by clients during name service lookup operations. The default is the primary domain controller.
Software\Microsoft\RPC\ NameService\Endpoint	The name of the endpoint used by the name service. The default is \pipe\locator.

3. Save and close the file.

Configuring the DCE CDS Name-Service Provider

You must have the Digital Equipment Corporation DCE Cell Directory Service product to configure the DCE CDS as your name service provider. See the documentation provided by Digital Equipment Corporation for information about DCE CDS.

Starting and Stopping Microsoft Locator

On Windows NT and Windows 2000 platforms, the RPC run-time libraries automatically start Microsoft Locator when necessary. You can manually stop and start the Locator if, for example, you need to clear the database while debugging a distributed application.

▶ To stop and start Microsoft Locator

1. From Control Panel, click the **Services** icon.
   The **Services** dialog box appears.
2. In the **Service** dialog box, select **Microsoft Locator** and then click **Start** or **Stop**.

You can also start and stop Microsoft Locator from the command line by typing:

```
C:\> net [start/stop] rpclocator
```

Note Only an administrator can start Microsoft Locator once it is stopped. If stopped, it will be restarted as necessary by the RPC run-time libraries.

Registry Information

This section contains a discussion of the registry entries that are relevant to RPC.

Using RPC Registry Entries

The setup programs for Microsoft® Windows NT® and Microsoft Windows® 2000, Windows 95/98, Windows 3.*x*, and Microsoft MS-DOS® store the RPC protocol information you specify in the registry file. For 32-bit versions of Windows, setup automatically configures the registry entries. No further configuration is necessary. With MS-DOS and Windows 3.*x*, however, use a text editor to change entries in the Rpcreg.dat file as shown in the following table.

Registry entry	Description
SOFTWARE\Microsoft\Rpc\NameService\DefaultSyntax	Specifies the default syntax that the RPC functions **RpcNsBindingImportBegin** and **RpcNsBindingExport** use. This registry entry corresponds to the DCE environment variable RPC_DEFAULT_ENTRY_SYNTAX.
SOFTWARE\Microsoft\Rpc\NameService\NetworkAddress	Specifies the address of the computer running the Locator that clients use during name service lookup operations. The default setting is the *primary domain controller*.

(continued)

(continued)

Registry entry	Description
SOFTWARE\Microsoft\Rpc\NameService\ServerNetworkAddress	Specifies the address of the computer running the Locator that servers use during name service export operations. The default is PDC (Windows NT/Windows 2000 only).
SOFTWARE\Microsoft\Rpc\NameService\Endpoint	Specifies the endpoint that the name service uses.
SOFTWARE\Microsoft\Rpc\NameService\Protocol	Specifies the protocol that the name service uses.
SOFTWARE\Microsoft\Rpc\ClientProtocols\ncacn_np	Specifies the name of the RPC client transport DLL for named pipes.
SOFTWARE\Microsoft\Rpc\ClientProtocols\ncacn_ip_tcp	Specifies the name of the RPC client transport DLL for TCP/IP.
SOFTWARE\Microsoft\Rpc\ClientProtocols\ncacn_nb_nb	Specifies the name of the RPC client transport DLL for NetBEUI over NetBIOS.
SOFTWARE\Microsoft\Rpc\ClientProtocols\ncalrpc	Specifies the name of the RPC client transport DLL for local RPC (Windows NT/Windows 2000 only).
SOFTWARE\Microsoft\Rpc\ServerProtocols\ncacn_np	Specifies the name of the RPC server transport DLL for named pipes.
SOFTWARE\Microsoft\Rpc\ServerProtocols\ncacn_ip_tcp	Specifies the name of the RPC server transport DLL for TCP/IP.
SOFTWARE\Microsoft\Rpc\ServerProtocols\ncacn_nb_nb	Specifies the name of the RPC server transport DLL for NetBEUI.
SOFTWARE\Microsoft\Rpc\ServerProtocols\ncalrpc	Specifies the name of the RPC server transport DLL for local RPC (Windows NT/Windows 2000 only).
SOFTWARE\Microsoft\Rpc\NetBios	Consists of mapping strings that map protocols to NetBIOS lana numbers.

Microsoft RPC setup automatically maps protocol strings to NetBIOS lana numbers and writes these settings in the registry. These mappings work as long as you have only one network card and one network protocol. If you have more than one network card and network protocol, or if you change your network configuration after installing Microsoft RPC, you must update the registry to indicate the new correspondences between protocol strings and NetBIOS lana numbers.

For 32-bit Windows platforms, the mapping string appears in the registry tree under

\SOFTWARE\Microsoft\Rpc\NetBios.

For MS-DOS and Windows 3.*x*, the mapping string appears in the registry file Rpcreg.dat.

The mapping string uses the following syntax:

```
ncacn_nb_protocol digit=lana_number
```

Parameters

protocol

Specifies the protocol type. Valid *protocol* values are as shown in the following table.

Protocol	Protocol type
nb	NetBEUI
tcp	TCP/IP

digit

Specifies a unique number associated with each instance of a protocol. Begin numbering with 0 for the first instance of a protocol, and use the next consecutive number for each additional instance of that protocol. For example, assign the value **ncacn_nb_nb0** to the first NetBEUI entry; assign the value **ncacn_nb_nb1** to the second NetBEUI entry.

lana_number

Specifies the NetBIOS lana number.

A unique lana number is associated with each network adapter present in the computer. For LAN Manager networks, the lana numbers for each network card are available in the configuration files Lanman.ini and Protocol.ini. For more information about the lana number, see your network documentation.

For example, the following mapping string describes a configuration that uses the NetBEUI protocol over an adapter card that is assigned lana number 0:

```
ncacn_nb_nb0=0
```

When you install a second card that supports both XNS and NetBEUI protocols, the mapping strings appear as follows:

```
ncacn_nb_tcp0=0
ncacn_nb_nb1=1
```

Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding

The following registry keys specify the system defaults for dynamic port allocation and for binding to NICs on multihomed computers. You must first create these keys and then specify the appropriate settings. If a key is missing or if it contains an invalid value, the entire configuration is marked as invalid, and all **RpcServerUseProtseq*** calls over **ncacn_ip_tcp** or **ncadg_ip_udp** will fail.

Note Ports allocated to a process remain allocated until that process dies. If all available ports are in use, then the API will return RPC_S_OUT_OF_RESOURCES.

Port key	Data type	Description
HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\Internet\Ports	**REG_MULTI_SZ**	Specifies a set of IP port ranges consisting of either all the ports available from the Internet or all the ports not available from the Internet. Each string represents a single port or an inclusive set of ports (for example,1000-1050, 1984). If any entries are outside the range 0 to 65535, or if any string cannot be interpreted, the RPC run time will treat the entire configuration as invalid.
HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\Internet\PortsInternetAvailable	**REG_SZ**	Y or N (not case-sensitive). If Y, the ports listed in the Ports key are all the Internet-available ports on that computer. If N, the ports listed in the Ports key are all those ports that are not Internet-available.
HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\Internet\UseInternetPorts	**REG_SZ**	Y or N (not case-sensitive). Specifies the system default policy. If Y, the processes using the default will be assigned ports from the set of Internet-available ports, as defined above. If N, processes using the default will be assigned ports from the set of intranet-only ports.
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Rpc\Linkage\Bind	**REG_MULTI_SZ**	Lists the device names of all the NICs on which to bind by default (for example, \Device\AMDPCN1). If the key does not exist, the server will bind to all NICs. If the key does exist, the server will bind to the NICs specified in the key, unless the NICFlags field is set to RPC_C_BIND_TO_ALL_NICS. If the key has a null ("") value, the configuration will be marked as invalid and all calls to **RpcServerUseProtseq*** over **ncacn_ip_tcp** or **ncadg_ip_udp** will fail.

See Also

RPC_POLICY, **RpcServerUseAllProtseqsEx**, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqEpEx**, **RpcServerUseProtseqIfEx**, **ncacn_ip_tcp**, **ncadg_ip_udp**

Using RPC with Winsock Proxy

Note The information in this topic is specific to Microsoft® Windows NT®, version 4.0 Service Pack 3 or later.

The release of Microsoft® Internet Access Server included Winsock Proxy, an enhanced version of Windows Sockets API version 1.1. Winsock Proxy lets a Windows Sockets application, running on a private network client, behave as if it were directly connected to a remote Internet server application. The Microsoft Proxy Server acts as the host for this connection. This means that all application-level communications are channeled through a single secured computer—the gateway computer running Microsoft Proxy Server.

Ordinarily, for datagram-packet transfers, the RPC transport DLL bypasses the **sendto( )** and **recvfrom( )** functions provided in Wsock32.dll, and communicates directly with the underlying device driver. This improves the speed of packet transfers but makes Winsock Proxy features unavailable to the application.

On Microsoft® Windows NT® version 4.0 with SP2, the RPC transport checks the registry to see whether to use the function calls provided in Wsock32.dll, or to interact directly with the device driver. To use RPC with Winsock Proxy, edit the system registry on each computer to add the following entry:

```
HKEY_LOCAL_MACHINE
  Software
    Microsoft
      Rpc
        UseWinsockForIP : REG_DWORD "1"
```

Microsoft Windows® 2000 enables each network protocol provider to have an associated GUID. The RPC run-time library compares the UDP and IPX GUIDs to the well known Microsoft identifiers. If they don't match, RPC automatically uses Winsock.

Another feature of Winsock Proxy is its ability to emulate the TCP transport protocol over the Novell SPX transport when the SPX client computer does not have TCP installed. To use this feature with RPC applications, edit the system registry on each client computer to add this entry.

```
HKEY_LOCAL_MACHINE
   Software
      Microsoft
         Rpc
            ClientProtocols
               ncacn_ip_tcp: REG_SZ "rpcltccm.dll"
               ncadg_ip_udp: REG_SZ "rpcltccm.dll"
```

Edit the registry on each server computer to add this entry:

```
HKEY_LOCAL_MACHINE
   Software
      Microsoft
         Rpc
            ServerProtocols
               ncacn_ip_tcp: REG_SZ "rpcltscm.dll"
               ncadg_ip_udp: REG_SZ "rpcltscm.dll"
```

For more information about RPC transport protocols see Specifying Protocol Sequences. For more information about Winsock Proxy, see the product documentation for Microsoft Internet Access Server.

Windows 2000 does not implement the **ClientProtocols** and **ServerProtocols** registry entries. Microsoft provides all well known transports as part of the run-time library. Therefore, these entries are not necessary.

SPX/IPX Installation

Applications can utilize Microsoft® RPC on networks that use the **ncacn_spx** and **ncadg_ipx** transports.

Configuring RPC for SPX/IPX

When using the **ncacn_spx** and **ncadg_ipx** transports, the server name is exactly the same as the Microsoft® Windows NT®, Windows® 2000, or Windows 95 server name. However, since the names are distributed using Novell protocols, they must conform to the Novell naming conventions. If a server name is not a valid Novell name, servers will not be able to create endpoints with the **ncacn_spx** or **ncadg_ipx** transports.

A valid Novell server name contains only the characters between 0x20 and 0x7f. Lowercase characters are changed to uppercase. The following characters cannot be used:

```
" * , . / : ; < = > ? [ ] \ | ]
```

To maintain compatibility with the first version of Windows NT, **ncacn_spx** and **ncadg_ipx** also enable you to use a special format of the server name known as the tilde name. The tilde name consists of a tilde (~), followed by the server's eight-digit network number, and then followed by its 12-digit Ethernet address. Tilde names have an advantage in that they do not require any name service capabilities. Thus, if you are connected to a server, the tilde name will work.

The following tables contain two sample configurations that illustrate the points previously described.

Component	**Configured as**
Windows NT/Windows 2000 or Windows 95 Server	NWCS
Windows NT/Windows 2000 or Windows 95 Client	NWCS
Windows 3.*x*/MS-DOS Client	NetWare Redirector

The configuration in the previous table requires that you have NetWare file servers or routers on your network. It will produce the best performance because the server names are stored in the NetWare Bindery.

Component	**Configured as**
Windows NT/Windows 2000 or Windows 95 Server	SAP Agent
Windows NT/Windows 2000 or Windows 95 Client	IPX/SPX
Windows 3.*x*/MS-DOS Client	IPX/SPX

The second configuration works in an environment that does not contain NetWare file servers or routers (for example, a network of two computers: a Windows 2000 server and an MS-DOS® client). Name resolution, which is accomplished during the first call over a binding handle, will be slightly slower than in the first configuration. In addition, the second configuration results in more traffic generated over the network.

To implement name resolution, when an RPC server uses an SPX or IPX endpoint, the server name and endpoint are registered as a Service Advertising Protocol (SAP) server of type 640 (hexadecimal). To resolve a server name, the RPC client sends a SAP request for all services of the same type, and then scans the list of responses for the name of the server. This process occurs during the first RPC call over each binding handle. For additional information on the SAP protocol for Novell, see your NetWare documentation.

Note The 16-bit Windows client applications that use the **ncacn_spx** or **ncadg_ipx** transports require that the file Nwipxspx.dll be installed in order to run under the Windows NT and Windows 2000 Win16-on-Win32 (WOW) subsystem. Contact Novell to obtain this file.

Configuring SAP and RPC

Novell Netware network servers use the Service Advertising Protocol (SAP) to broadcast information about available services on the network to other networked devices. A server may send out an SAP broadcast every 60 seconds to inform other network devices about the services it offers. Workstations use SAPs to find services they need on the network.

Microsoft Windows NT and Windows 2000 include a SAP Agent service to enable Windows-based servers to interact with Netware servers. The SAP Agent service will listen for network clients' SAP requests for IPX-based services that are installed and running on the server.

Software that is designed to be advertised as a service over the network by means of a SAP broadcast will issue the SAP announcements every 60 seconds, without having the SAP Agent installed. However, in order for network clients to quickly locate an IPX network service, a server that maintains a service database must be available on the network, to respond to the service location request. This service database is usually maintained by a Novell NetWare or by a NetWare compatible server. Microsoft File and Print Services for NetWare will also maintain an IPX network service database.

On a computer running Windows NT Server, if Gateway Services for NetWare (GSNW) is installed, a SAP Type 640 will broadcast every 60 seconds by the Remote Procedure Call (RPC) service. This SAP broadcast will continue even if the user disables the GSNW and the SAP Agent Service.

On a computer running Windows NT Workstation or Windows 2000 Professional, the RPC service will do the SAP broadcast if the Client Services for NetWare (CSNW) and the SAP Agent service are installed. This SAP broadcast will continue even if the user disables the SAP agent.

By default, the RPC service will check for the presence of Gateway Services for NetWare and the SAP Agent service on the computer running Windows NT Server. Installing File and Print services for Netware installs a SAP Agent.

On the computer running Windows NT Workstation with CSNW, the RPC service checks for the SAP Agent service. If the services are present, RPC will start its own thread that will do the SAP broadcast Type 640 every minute.

Note If you do not want SAP broadcasts on the network every 60 seconds, you must have or obtain Windows 2000 or Windows NT Server Service Pack 4 or later. If one of these is installed, you can disable SAP broadcasts using the Registry Editor. Be warned that using Registry Editor incorrectly can cause serious problems that may require you to reinstall your operating system. Microsoft cannot guarantee that problems resulting from the incorrect use of Registry Editor can be solved. Use Registry Editor at your own risk. You should back up the registry before you edit it. If you are running Windows NT, you should also update your Emergency Repair Disk (ERD).

▶ To configure for SAP

1. Run Registry Editor (Regedt32.exe) and go to the following key in the registry:

   HKEY_LOCAL_MACHINE\Software\Microsoft\RPC

2. On the **Edit** menu, click **Add Value**, and use the following entry:
   a. Value Name: AdvertiseRpcService
   b. Data Type: REG_SZ
   c. String: No
3. Using No for the string turns RPC SAP broadcasting off. Using Yes for the string turns RPC SAP broadcasting on.
4. Restart the computer for the registry change to take effect.

Note If the SAP broadcasts continue after following these steps, you may want to try a troubleshooting step. Delete the **Ncacn_spx** string value in following registry key:

HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\ServerProtocols\

This should be used only as a troubleshooting step. Deleting this string value completely disables SAP broadcasts which some programs may need in order to function properly.

Configuring the Security Server

The following procedure details configuring the security server for Microsoft® RPC.

▶ To configure the security server for RPC

1. Start Microsoft® Windows NT® or Windows® 2000 and click the Control Panel icon.
2. In Control Panel, click the **Networks** icon.

   The **Network Settings** dialog box appears.
3. In the **Installed Network Software** list, select **RPC Configuration**.

   The **RPC Configuration** dialog box appears.
4. In the **Security Service Provider** list, select from one or more security providers.
5. Click **OK**.

CHAPTER 20

Asynchronous RPC

Asynchronous Remote Procedure Call (RPC) is a Microsoft® extension that addresses several limitations of the traditional RPC model as defined by the Open Software Foundation–Distributed Computing Environment (OSF-DCE). Asynchronous RPC separates a remote procedure call from its return value, which resolves the following limitations of traditional, synchronous RPC:

- **Multiple outstanding calls from a single-threaded client.** In the traditional RPC model, a client is blocked in a remote procedure call until the call returns. This prevents a client from having multiple outstanding calls, while still having its thread available to do other work.
- **Slow or delayed clients.** A client that is slow to produce data may want to make a remote procedure call with initial data and then supply additional data as it is produced. This is not possible with conventional (synchronous) RPC.
- **Slow or delayed servers.** A remote procedure call that takes a long time to complete will tie up the dispatch thread for the duration of the task. With asynchronous RPC, the server can start a separate (asynchronous) operation to process the request and send back the reply when it is available. The server can also send the reply incrementally as results become available without having to tie up a dispatch thread for the duration of the remote call. By making the client application asynchronous, you can prevent a slow server from unnecessarily tying up a client application.
- **Transfer of large amounts of data.** Transferring large amounts of data between the client and server, especially over slow links, ties up both the client thread and the server manager thread for the duration of the transfer. With asynchronous RPC and pipes, data transfer can take place incrementally, and without blocking the client or server from performing other tasks.

You take advantage of asynchronous RPC mechanisms by declaring functions with the **[async]** attribute. Because you make this declaration in an attribute configuration file (ACF), you do not have to make any changes to the Interface Definition Language (IDL) file; asynchronous RPC has no effect on the wire protocol (how the data is transmitted between client and server). This means that both synchronous and asynchronous clients can communicate with an asynchronous server application.

Asynchronous RPC is supported on the Microsoft® Windows NT®/Windows® 2000 operating system.

This chapter presents an overview of how to develop distributed applications using asynchronous RPC.

Declaring Asynchronous Functions

To declare an RPC function as asynchronous, first declare the function as part of an interface definition in an Interface Definition Language (IDL) file. The use of asynchronous RPC functions does not require you to make any special alterations to your IDL file. The following example shows an IDL file for an application that uses one asynchronous function.

```
[
    uuid (7f6c4340-eb67-11d1-b9d7-00c04fad9a3b),
    version(1.0),
    pointer_default(unique)
]
interface AsyncRPC
{
    const long DEFAULT_ASYNC_DELAY        = 10000;
    const short APP_ERROR                 = -1;
    const char* DEFAULT_PROTOCOL_SEQUENCE = "ncacn_ip_tcp";
    const char* DEFAULT_ENDPOINT          = "8765";

    void NonAsyncFunc(handle_t hBinding,
                      [in, string] unsigned char * pszMessage);

    void AsyncFunc(handle_t hBinding,
                   [in] unsigned long nAsychDelay);

    void Shutdown(handle_t hBinding);
}
```

For all asynchronous RPC functions that your application uses, you will need to modify the declaration of the asynchronous functions within your application's ACF file. Apply the **[async]** attribute to each asynchronous function name, as shown in the following example:

```
interface AsyncRPC
{
    [async] AsyncFunc();
}
```

When you apply the **[async]** attribute in the ACF file, the MIDL compiler automatically generates an additional asynchronous handle parameter in the stub code.

Client-Side Asynchronous RPC

Client-side asynchronous function handling consists of three primary tasks discussed in the following sections.

Making the Asynchronous Call

Before it can make an asynchronous remote call, the client must initialize the asynchronous handle. Client and server programs use pointers to the **RPC_ASYNC_STATE** structure for asynchronous handles.

Every outstanding call must have its own unique asynchronous handle. The client creates the handle and passes it to the **RpcAsyncInitializeHandle** function. For the call to complete correctly, the client must ensure that the memory for the handle is not released until it receives the server's asynchronous reply. Also, before making another call on an existing asynchronous handle, the client *must* reinitialize the handle. Failure to do this can cause the client stub to raise an exception during the call. The client must also ensure that the buffers it supplies for **[out]** parameters and **[in, out]** parameters to an asynchronous remote procedure remain allocated until it has received the reply from the server.

When it invokes an asynchronous remote procedure, the client must select the method that the RPC run-time library will use to notify it of the call's completion. The client can receive this notification in any one of the following ways:

- **Event.** The client can specify an event to be fired when the call has completed. For details, see Event Objects.
- **Polling.** The client can repeatedly call **RpcAsyncGetCallStatus**. If the return value is anything other than RPC_S_ASYNC_CALL_PENDING, the call is complete. This method uses more CPU time than the other methods described here.
- **APC.** The client can specify an asynchronous procedure call (APC) that gets called when the call completes. For the prototype of the APC function, see RPCNOTIFICATION_ROUTINE. The APC is called with its *Event* parameter set to RpcCallComplete. For APCs to get dispatched, the client thread must be in an alertable wait state.

  If the *hThread* field in the asynchronous handle is set to 0, the APCs are queued on the thread that made the asynchronous call. If it is nonzero, the APCs are queued on the thread specified by *hThread*.
- **IOC.** The I/O completion port is notified with the parameters specified in the asynchronous handle. For more information, see **CreateIoCompletionPort**.
- **Windows handle**. A message is posted to the specified window handle (HWND).

The following code fragment shows the essential steps required for initializing an asynchronous handle and using it to make an asynchronous remote procedure call.

```
RPC_ASYNC_STATE Async;
RPC_STATUS status;

// Initialize the handle.
status = RpcAsyncInitializeHandle(&Async, sizeof(RPC_ASYNC_STATE));
if (status)
```

(continued)

(continued)

```
{
    // Code to handle the error goes here.
}

Async.UserInfo = NULL;
Async.NotificationType = RpcNotificationTypeEvent;

Async.u.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
if (Async.u.hEvent == 0)
{
    // Code to handle the error goes here.
}
// Call an asynchronous RPC routine here
RpcTryExcept
{
    printf("\nCalling the remote procedure 'AsyncFunc'\n");
    AsyncFunc(&Async, AsyncRPC_ClientIfHandle, nAsychDelay);
}
RpcExcept(1)
{
    ulCode = RpcExceptionCode();
    printf("AsyncFunc: Run time reported exception 0x%lx = %ld\n",
            ulCode, ulCode);
}
RpcEndExcept

// Call a synchronous routine while
// the asynchronous procedure is still running
RpcTryExcept
{
    printf("\nCalling the remote procedure 'NonAsyncFunc'\n");
    NonAsyncFunc(AsyncRPC_ClientIfHandle, pszMessage);
    fprintf(stderr,
            "While 'AsyncFunc' is running asyncronously,\n"
            "we still can send message to the server in the mean "
            "time.\n\n");
}
RpcExcept(1)
{
    ulCode = RpcExceptionCode();
    printf("NonAsyncFunc: Run time reported exception 0x%lx = %ld\n",
            ulCode, ulCode);
}
RpcEndExcept
```

As this example demonstrates, your client program can execute synchronous remote procedure calls while an asynchronous procedure call is still pending. This client creates an event object for the RPC run-time library to use to notify it when the asynchronous call completes.

Waiting for the Asynchronous Reply

What the client does while it waits to be notified of a reply from the server depends on the notification mechanism it selects.

If the client uses an event for notification, it will typically call the **WaitForSingleObject** function or the **WaitForSingleObjectEx** function. The client enters a blocked state when it calls either of these functions. This is efficient because the client does not consume CPU run cycles while it is blocked.

When it uses polling to wait for its results, the client program enters a loop that repeatedly calls the function **RpcAsyncGetCallStatus**. This is an efficient method of waiting if your client program does other processing in the polling loop. For instance, it can prepare data in small chunks for a subsequent asynchronous remote procedure call. After each chunk is finished, your client can poll the outstanding asynchronous remote procedure call to see if it is complete.

Your client program can provide an asynchronous procedure call (APC), which is a type of callback function that the RPC run-time library will invoke when the asynchronous remote procedure call completes. Your client program must be in an alertable wait state. This typically means that the client calls a Microsoft® Win32® API function to put itself in a blocked state. For more information, see Asynchronous Procedure Calls

If your client program uses an I/O completion port to receive completion notification, it must call the **GetQueuedCompletionStatus** function. When it does, it can either wait indefinitely for a response or continue to do other processing. If it does other processing while it waits for a reply, it must poll the completion port with the **GetQueuedCompletionStatus** function. In this case, it typically needs to set the *dwMilliseconds* to zero. This causes **GetQueuedCompletionStatus** to return immediately, even if the asynchronous call has not completed.

Client programs can also receive completion notification through their message queues. In this situation, they simply process the completion message as they would any Microsoft® Windows® message.

The client can, at any time, call **RpcAsyncCancelCall** to request cancellation of an outstanding call. Note that in a multithreaded application, the thread that originated the call is the only thread that can cancel it.

The following code fragment illustrates how a client program can use an event to wait for an asynchronous reply.

```
// This code fragment assumes that Async is a valid asynchronous
// RPC handle.
if (WaitForSingleObject(Async.u.hEvent, INFINITE) == WAIT_FAILED)
{
    RpcRaiseException(APP_ERROR);
}
```

Client programs that use an APC to receive notification of an asynchronous reply typically put themselves into a blocked state. The following code fragment shows this.

```
if (SleepEx(INFINITE, TRUE) != WAIT_IO_COMPLETION)
{
    RpcRaiseException(APP_ERROR);
}
```

In this case, the client program goes to sleep, consuming no CPU cycles, until the RPC run-time library calls the APC (not shown).

The next example demonstrates a client that uses an I/O completion port to wait for an asynchronous reply.

```
// This code fragment assumes that Async is a valid asynchronous
// RPC handle.
if (!GetQueuedCompletionStatus(
         Async.u.IOC.hIOPort,
         &Async.u.IOC.dwNumberOfBytesTransferred,
         &Async.u.IOC.dwCompletionKey,
         &Async.u.IOC.lpOverlapped,
         INFINITE))
{
    RpcRaiseException(APP_ERROR);
}
```

In the preceding example, the call to **GetQueuedCompletionStatus** waits indefinitely until the asynchronous remote procedure call completes.

One potential pitfall occurs when writing multithreaded applications. If a thread invokes a remote procedure call and then terminates before it receives notification that the send completed, the remote procedure call will be canceled. The client stub will also close the connection to the server. Therefore, all threads that call a remote procedure should wait for completion notification or cancel the call before they terminate.

Receiving the Asynchronous Reply

After it is notified that the server has sent a reply, the client calls **RpcAsyncCompleteCall** with the asynchronous handle so that it can receive the reply. When **RpcAsyncCompleteCall** has completed successfully, the *Reply* parameter points to a buffer that contains the return value of the remote function. Any buffers supplied by the client program as **[out]** or **[in, out]** parameters to the asynchronous remote function contain valid data. If the client calls **RpcAsyncCompleteCall** before the server has sent the reply, that call will fail and return a value of RPC_S_ASYNC_CALL_PENDING.

If your client program uses I/O completion ports or events for notification, it must call **CloseHandle** to release the port or handle when it no longer needs them.

Server-Side Asynchronous RPC

To receive and dispatch asynchronous remote procedure calls, the server application essential tasks are:

- Handling Asynchronous Calls
- Receiving Cancellations
- Sending the Asynchronous Reply
- Asynchronous I/O and Asynchronous RPC

Handling Asynchronous Calls

The manager routine of an asynchronous function always receives the asynchronous handle as the first parameter. The server must keep track of this handle and use it to send the reply when the asynchronous remote procedure call finishes.

If the server needs to abort an asynchronous RPC, it calls **RpcAsyncAbortCall**. This function performs the same server-side cleanup as **RpcAsyncCompleteCall** and propagates an exception code (provided by the server application) back to the client.

For an example of an asynchronous procedure, see *Sending the Asynchronous Reply*.

Receiving Cancellations

The server application can call **RpcServerTestCancel** with the binding handle of the thread in question to find out if the client has requested cancellation of an outstanding asynchronous call. It will return RPC_S_OK if the client canceled the call.

Sending the Asynchronous Reply

When the asynchronous call is complete, the server sends a reply to the client by calling the **RpcAsyncCompleteCall** function and passing it the asynchronous handle. This call is necessary even if the asynchronous call has a void return value and no **[out]** parameters. If the function has a return value, it is passed by reference to **RpcAsyncCompleteCall.**

When the server calls **RpcAsyncCompleteCall** or **RpcAsyncAbortCall**, or a call completes because an exception was raised in the server-manager routine, the RPC run-time library automatically destroys the server's asynchronous handle.

Note The server must finish updating the **[in, out]** and **[out]** parameters before calling **RpcAsyncCompleteCall**. No changes can be made to those parameters or to the asynchronous handle after calling **RpcAsyncCompleteCall**.

The following example demonstrates a simple asynchronous procedure call.

```
void AsyncFunc(IN PRPC_ASYNC_STATE pAsync,
               IN RPC_BINDING_HANDLE hBinding,
               IN OUT unsigned long nAsychDelay)
{
    int nReply = 1;
    RPC_STATUS status;
    unsigned long nTmpAsychDelay;
    int i;

    if (nAsychDelay < 0)
        nAsychDelay = DEFAULT_ASYNC_DELAY;
    else if (nAsychDelay < 100)
        nAsychDelay = 100;

    // We only call RpcServerTestCancel if the call
    // takes longer than ASYNC_CANCEL_CHECK ms
    if (nAsychDelay > ASYNC_CANCEL_CHECK)
    {
        nTmpAsychDelay = nAsychDelay/100;

        for (i = 0; i < 100; i++)
        {
            Sleep(nTmpAsychDelay);

            if (i%5 == 0)
            {
                fprintf(stderr,
                        "\rRunning AsyncFunc (%lu ms) (%d%c) ... ",
                        nAsychDelay, i+5, PERCENT);

                status =
                    RpcServerTestCancel(
                        RpcAsyncGetCallHandle(pAsync));
                if (status == RPC_S_OK)
                {
                    fprintf(stderr,
                            "\nAsyncFunc has been cancelled!!!\n");
                    break;
                }
                else if (status != RPC_S_CALL_IN_PROGRESS)
                {
                    printf(
                        "RpcAsyncInitializeHandle returned 0x%x\n",
                        status);
                    exit(status);
                }
            }
        }
    }
    else
        Sleep(nAsychDelay);

    printf("\nCalling RpcAsyncCompleteCall\n");
    status = RpcAsyncCompleteCall(pAsync, &nReply);
    printf("RpcAsyncCompleteCall returned 0x%x\n", status);
    if (status)
        exit(status);
}
```

For the sake of simplicity, this asynchronous server routine does not process actual data. It simply puts itself to sleep for awhile.

See Also

RPC_ASYNC_STATE, **RpcAsyncCompleteCall**, **RpcAsyncAbortCall**, **RpcServerTestCancel**

Asynchronous I/O and Asynchronous RPC

Asynchronous I/O is an efficient means for a single thread to manage multiple I/O requests simultaneously. Asynchronous RPC on the server accomplishes a similar purpose for RPC requests. Designers using asynchronous RPC may be tempted to post asynchronous I/O requests from the server procedures. However, this technique should be avoided.

An asynchronous remote procedure call may complete before the asynchronous I/O request completes. When the asynchronous call completes, its thread terminates. Microsoft® Windows® NT and Windows 2000 bind all I/O requests to the thread that initiates them. If the thread terminates, any I/O requests pending on that thread are aborted. Pending I/O requests cannot be moved to another thread.

Therefore application designers can either use synchronous I/O in server procedures, or they can forward all requests that involve asynchronous I/O to procedures executing on a thread pool that the application manages. The Windows NT/Windows 2000 API provides functions for thread-pool management. See *Process and Thread Functions*.

Causal Ordering of Asynchronous Calls

In an asynchronous RPC application, it is possible for a client thread to make a second asynchronous call on a binding handle before an earlier call made on that handle has completed. The RPC run-time library handles this situation as follows:

- The asynchronous RPC mechanism in Microsoft® Windows® 2000 guarantees that asynchronous calls made on the same binding handle, at the same security level, are dispatched in the order they were made. Actual execution of the calls may occur out of order.
- Calls made to a server application running on Microsoft® Windows NT® 3.51, Windows NT 4.0, or on Windows 95 are dispatched and executed in the order in which they were made. The run-time automatically detects these platforms and serializes the calls on the client.
- As with synchronous calls, asynchronous remote procedure calls from different client threads execute simultaneously.
- If an asynchronous call from a client application is followed by one or more synchronous calls, the asynchronous call can execute while the synchronous calls are executing. Regardless of the status of the asynchronous call, the synchronous calls are executed in the order in which they are received by the server.
- If a client application selects noncausal ordering for a particular binding handle, it disables serialization for that handle. Applications enable noncausal ordering by calling **RpcBindingSetOption** with the *Option* parameter set to RPC_C_OPT_BINDING_NONCAUSAL and the *OptionValue* parameter set to TRUE. For details, see *Binding Option Constants*.

Error Handling

In synchronous RPC, a client makes a remote call that returns with either a success or failure code. Asynchronous RPC provides more opportunities for a call to fail, and these failures are handled differently, depending on where and when they occur. The following table describes the ways in which a call can fail, and how it is handled.

Source of failure	How handled
Client call to remote function fails.	The call raises an exception. Neither the client nor the server needs to call **RpcAsyncCompleteCall**.
Client call to remote function fails because of a failure in the asynchronous mechanism.	Client receives a "call complete" notification. The **RpcAsyncCompleteCall** function return value is the failure code.
Client issues cancel (either abortive or nonabortive).	Server learns of the cancel when it calls **RpcServerTestCancel**. Server must still call **RpcAsyncCompleteCall** or **RpcAsyncAbortCall** to complete the call.

Source of failure	How handled
Client issues an abortive cancel.	Client receives "call complete" notification. When the client calls **RpcAsyncCompleteCall**, it returns RPC_S_CALLED_CANCELLED, *unless* the call failed or completed before it was canceled.
Call fails before server dispatches it.	Run time, client, and server stubs handle the failure.
The synchronous portion of the manager routine raises an exception.	Run time catches the exception and propagates it to the client.
Call fails after server calls **RpcAsyncCompleteCall**.	Run time handles it; the server is not notified.

Special Error Handling Cases for Pipes

Source of failure	How handled
Client calls **push** and the call fails.	The **push** function returns a failure code. Client must call **RpcAsyncCompleteCall**.
Client calls **RpcAsyncCompleteCall** before the **in** pipes are drained.	Call fails with the appropriate pipe-filling error code.
Client calls **pull** and the call fails.	**Pull** returns a failure code. Client must call **RpcAsyncCompleteCall**.
Either client or server calls **push** or **pull** in the wrong order.	Run-time returns pipe-filling error status.
Server calls **push** and the call fails.	**Push** returns a failure code. Server must call **RpcAsyncCompleteCall**.
Server calls **RpcAsyncCompleteCall** before the pipes have been drained.	The pipe call returns a pipe filling error status.
After the dispatch, a **receive** operation fails.	The next time the server calls **pull** to receive pipe data, an error is returned.

Asynchronous RPC Over the Named-Pipe Protocol

If you use named pipes (**ncacn_np**) as your transport protocol, you should avoid allowing a large number of idle pending calls on the server. With named pipes, each client waiting for a reply will have a pending named pipe read on the server, each of which requires a certain amount of kernel memory.

For example, you would not want to use a notification call for new e-mail with the named-pipe transport, because such a call would remain pending even when clients are idle, and kernel memory could be depleted. Note that this is not a problem with the other connection-oriented protocols, such as **ncacn_ip_tcp**.

Since named pipes are a transport protocol, your application can use them by specifying **ncacn_np** as the protocol in a string binding. For more information on named pipes, see Named Pipes. For details on creating string bindings, see *Using String Bindings*.

Using Asynchronous RPC with DCE Pipes

Microsoft RPC supports the use of DCE pipes. Although they share a similar name, DCE pipes are unrelated to named pipes. Named pipes are a transport protocol. DCE pipes are a protocol-independent method of client/server communication.

Asynchronous Pipes

Using **pipe** parameters with asynchronous RPC allows you to transmit data incrementally, as it becomes available, without tying up the client and server. This is particularly useful when you have a large amount of data to transfer, combined with a slow client, a slow server, or a slow network. If you use a pipe in an asynchronous function call, it is, by definition, an asynchronous pipe. Synchronous pipes are not supported in conjunction with asynchronous functions.

Unlike conventional (synchronous) pipes where the server handles all the details of sending and receiving pipe data, asynchronous pipes are symmetrical. That is, both the client and the server can push and pull data through the pipe.

Note Pipe parameters can only be passed by reference. Even if the IDL file shows pipe parameters being passed by value, the generated stubs will accept pipe parameters by reference only.

In the following discussion of asynchronous pipes, familiarity with the **pipe** type constructor is assumed. For more information on the pipe procedures described in these topics, see *Pipes*.

Declaring Asynchronous Pipes

The following example IDL file defines a typical pipe structure, and an asynchronous RPC function with pipes.

Example

```
//file: Xasyncpipe.idl:
//
interface IMyAsyncPipe
{
    //define the pipe type
    typedef pipe int aysnc_intpipe ;
    //then use it as a parameter
    int MyAsyncPipe(
        handle_t hBinding,
        [in] a,
        [in] ASYNC_INTPIPE *inpipe,
        [out] ASYNC_INTPIPE *outpipe,
        [out] int *b) ;
};
//other function declarations

//other interface definitions
//end Xasyncpipe.idl

//file:Xasyncpipe.acf:
//file: Xasyncpipe.acf:
interface IMyAsyncPipe
{
    [async] MyAsyncPipe () ;
} ;
//
//end Xasyncpipe.acf
```

The following code fragment shows a typical pipe structure definition. It contains pointers to **push** and **pull** procedures, a buffer to hold the pipe data, and a state variable to coordinate the procedures:

```
//
typedef struct ASYNC_MYPIPE
{
    RPC_STATUS (__RPC_FAR * pull) (
        char __RPC_FAR * state,
        small __RPC_FAR * buf,
        unsigned long esize,
        unsigned long __RPC_FAR * ecount );
    RPC_STATUS (__RPC_FAR * push) (
        char __RPC_FAR * state,
        small __RPC_FAR * buf,
        unsigned long ecount );
    void *Reserved;
    char __RPC_FAR * state;
}ASYNC_INTPIPE;
```

Client-Side Asynchronous Pipe Handling

Before making an asynchronous remote call, the client must first initialize the asynchronous handle. As with nonpipe procedures, the client calls an asynchronous function with the asynchronous handle as the first parameter and uses the asynchronous handle to send and receive pipe data, query the status of the call, and receive the reply.

The client makes the asynchronous remote procedure call with the asynchronous handle as the first parameter. The client can use this handle to query the status of the call and to receive the reply. The asynchronous pipe model is symmetric. Both client and server applications send and receive pipe data actively (as opposed to synchronous RPC, where the pipe data is sent and received passively).

The client sends asynchronous pipe data by calling the **push** function on the appropriate asynchronous pipe, with the pipe's state variable as the first parameter. When the **push** function returns, the client can modify or free the send buffer.

If the RPC_ASYNC_NOTIFY_ON_SEND_COMPLETE flag is set in the asynchronous handle, and if APCs are used as the notification mechanism, an APC is queued when the pipe send is actually complete. You can take advantage of this mechanism to implement flow control. Note, however, that if the client pushes another buffer before the previous push is complete, the client may, depending on the speed of the transfer operation, receive only one send-complete notification, rather than one notification for each buffer or each **push** operation. When the client has sent all of the pipe data, it makes one final **push** call with the number of elements set to 0.

The client program receives asynchronous pipe data by calling the **pull** function on the appropriate asynchronous pipe, with the pipe's state variable as the first parameter. If no pipe data is available, the **pull** function returns RPC_S_ASYNC_CALL_PENDING.

If the notification mechanism is APC, and the server returned RPC_S_ASYNC_CALL_PENDING, the client must wait until it receives the **RpcReceiveComplete** APC from run-time before calling **pull** again.

Server-Side Asynchronous Pipe Handling

The manager routine of an asynchronous function always receives the asynchronous handle as the first parameter. The server uses this handle to send the reply and to send the **out** pipe data as it becomes available. The handle remains valid until **RpcAsyncCompleteCall** is called on it, the call is aborted by **RpcAsyncAbortCall**, or an exception occurs in the manager routine. The application must keep track of all top-level pointers for the **[out]** and **[in, out]** parameters, in order to update them before completing the call. The application must also keep track of the **[in]** and **[out]** pipes.

The server sends asynchronous pipe data in the same manner as the client. See *Client-Side Asynchronous Pipe Handling.*

The server receives asynchronous pipe data in the same manner as the client. If the receive mechanism is asynchronous procedure calls (APCs), the server must specify a thread handle (in pAsync->u.APC.hThread) and register the asynchronous handle with the run-time library.

Example

In this example, the server manager routine, MyAsyncPipeFunc, handles the remote procedure call from the client.

```
typedef struct
{
    PRPC_ASYNC_STATE pAsync;
    ASYNC_INTPIPE *inpipe;
    ASYNC_INTPIPE *outpipe;
    int i;
    int *b;
    int PipeBuffer[ASYNC_CHUNK_SIZE];
} PIPE_CALL_COOKIE;

void MyAsyncPipeFunc(
    IN PRPC_ASYNC_STATE pAsync,
    IN RPC_BINDING_HANDLE hBinding,
    IN int a,
    IN ASYNC_INTPIPE *inpipe,
    OUT ASYNC_INTPIPE *outpipe,
    OUT int *b)
{
    unsigned long ThreadIdentifier;
    HANDLE HandleToThread;
    PIPE_CALL_COOKIE *PipeCallCookie;

    PipeCallCookie = new PIPE_CALL_COOKIE;
    PipeCallCookie->pAsync = pAsync;
    PipeCallCookie->inpipe = inpipe;
    PipeCallCookie->outpipe = outpipe;
    PipeCallCookie->b = b;

    pAsync->u.APC.hThread = 0;
    pAsync->u.APC.hThread = CreateThread(
                                0, DefaultThreadStackSize,
                                (LPTHREAD_START_ROUTINE)
                                ThreadProcPipes,
                                PipeCallCookie, 0,
                                &ThreadIdentifier);
}// endMyAsyncPipeFunc

//Sending pipe data
//This APC routine is called when a pipe send completes,
//or when an asynchronous call completes.

//This thread routine receives pipe data, processes the call,
//sends the reply back to the client, and
//completes the asynchronous call.
```

(continued)

(continued)

```
void ThreadProcPipes(IN PIPE_CALL_COOKIE  *Cookie)
{
    int *ptr ;
    int  n ;
    int retval ;

    while (pAsync->u.APC.hThread == 0)
    {
        Sleep(10);
    }

    pAsync->Flags = RPC_C_NOTIFY_ON_SEND_COMPLETE;
    pAsync->UserInfo = (void *) PipeCallCookie;
    pAsync->NotificationType = RpcNotificationTypeApc;
    pAsync->u.APC.NotificationRoutine = MyAsyncPipeAPCRoutine;
    pAsync->u.APC.hThread = HandleToThread;

    RpcAsyncRegisterHandle(pAsync);

    while (!fDone)
    {
        Cookie->inpipe->pull(
            Cookie->inpipe.state,
            (int *) Cookie->PipeBuffer,
            ASYNC_CHUNK_SIZE,
            &num_elements);
        switch (Status)
        {
            case RPC_S_ASYNC_CALL_PENDING:
                if (SleepEx(INFINITE, TRUE) != WAIT_IO_COMPLETION)
                {
                    RpcRaiseException(APP_ERROR) ;
                }
            break;

            case RPC_S_OK:
                if (num_elements == 0)
                {
                    fDone = 1;
                }
                else
                {
                    // process the received data
                }
            break;
            default:
                fDone = 1;
            break;
        }
    }

    Cookie->i = 1;
    Cookie->outpipe->push(
        Cookie->outpipe.state,
        0,
        Cookie->PipeBuffer,
        ASYNC_CHUNK_SIZE) ;

    while (Cookie->i < ASYNC_NUM_CHUNKS+1)
    {
        if (SleepEx(INFINITE, TRUE) != WAIT_IO_COMPLETION)
        {
            RpcRaiseException (APP_ERROR);
        }
    }
    // sending non pipe reply
    *(Cookie->b) = 10;
    Status = RpcAsyncCompleteCall (Cookie->pAsync, &retval);
}

void MyAsyncPipeAPCRoutine (
    IN PRPC_ASYNC_STATE pAsync,
    IN void *Context,
    IN unsigned int Flags)
{
    PIPE_CALL_COOKIE *Cookie = (PIPE_CALL_COOKIE *) pAsync->UserInfo;

    if (Flags & RPC_ASYNC_PIPE_SEND_COMPLETE)
    {
        if (Cookie->i <ASYNC_NUM_CHUNKS)
        {
            Cookie->outpipe->push(
                Cookie->outpipe.state,
                0,
                (int *) Cookie->PipeBuffer,
                ASYNC_CHUNK_SIZE);
            Cookie->i++ ;
        }
        else
```

(continued)

(continued)

```
        {
            pAsync->Flags = 0;
            Cookie->outpipe->push(Cookie->outpipe.state, 0, 0, 0);
            Cookie->i++;
        }
    }
} // end MyAsyncPipeAPCRoutine
```

See Also

Pipes, **async**, *Server-Side Asynchronous RPC*

Asynchronous DCOM

To take advantage of asynchronous RPC transparently, COM programmers can use the **IAsync*** interfaces described in Asynchronous DCOM in the COM Programmer's Guide and the MIDL attribute **async_uuid**

CHAPTER 21

RPC Message Queuing

Message Queuing Services (MSMQ) lets users communicate across networks and systems regardless of the current state of the communicating applications and systems. Applications send and receive messages through message queues that MSMQ maintains. The message queues continue to function even when the client or server application isn't running. Message queuing provides:

- **Asynchronous messaging**. With MSMQ asynchronous messaging, a client application can send a message to a server and return immediately, even if the target computer or server program is not responding.
- **Guaranteed message delivery**. When an application sends a message through MSMQ, the message will reach its destination even if the destination application is not running at the same time or the networks and systems are offline.
- **Routing and dynamic configuration**. MSMQ provides flexible routing over heterogeneous networks. The configuration of such networks can be changed dynamically without any major changes to systems and networks themselves.
- **Connectionless messaging**. Applications using MSMQ do not need to set up direct sessions with target applications.
- **Security**. MSMQ provides secure communication based on Microsoft® Windows NT®/Windows® 2000 security and the Cryptographic API (CryptoAPI) for encryption and digital signatures.
- **Prioritized Messaging**. MSMQ transfers messages across networks based on priority, allowing faster communication for critical applications.

Microsoft RPC extends the Open Software Foundation—Data Communications Equipment (OSF-DCE) model for remote procedure calls by allowing distributed applications to use MSMQ as a transport and to control many of its features. This functionality is available both to conventional RPC applications and, through the **IRPCOptions** interface, to DCOM applications.

Overview of Message Queuing Services Architecture

Message Queuing Services (MSMQ) uses a site/enterprise model. Typically, a site is a physical location, such as a building. An enterprise consists of one or more sites and represents an organization.

Figure 21-1 illustrates the architecture of the MSMQ Service.

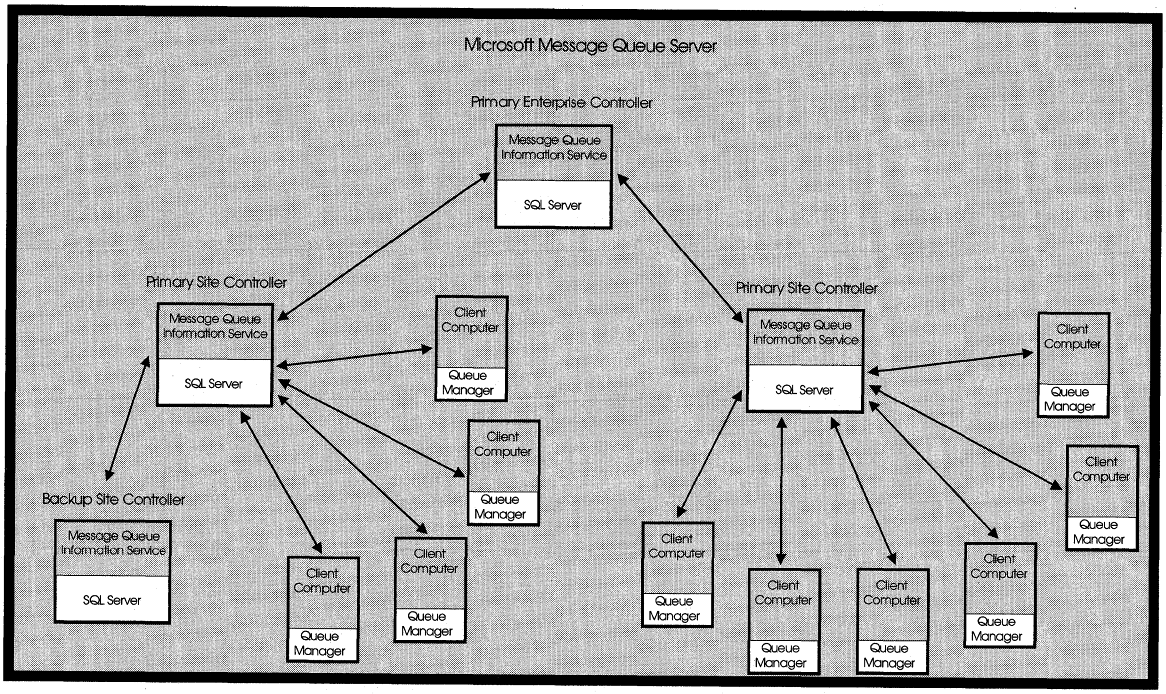

Microsoft Message Queue Server
Primary Enterprise Controller
Message Queue Information Service
SQL Server
Primary Site Controller
Message Queue Information Service
SQL Server
Primary Site Controller
Message Queue Information Service
SQL Server
Backup Site Controller
Message Queue Information Service
SQL Server
Client Computer
Queue Manager

Figure 21-1: MSMQ Service Architecture.

At the heart of MSMQ is the Message Queue Information Service (MQIS) database, which runs on top of SQL Server. An enterprise has a single master MQIS, called the Primary Enterprise Controller. Each site has its own MQIS, called a *primary site controller* and zero or more *backup site controllers*. Finally, there are the individual client computers, each of which has its own queue manager, implemented as a service. The Primary Enterprise Controller can also be a Primary Site Controller, and any controller can also be a client.

Message queues can be either public or private. Public queues are registered in Active Directory and are accessible across the network. Messages in a public queue are routed throughout the enterprise, under the control of MSMQ. Client application messages move from the client's queue manager to the destination queue by traveling between the queue managers of the site controllers.

Private queues are maintained by the local queue manager and are not registered in Active Directory. The scope of private queue messages is limited to the computer on which they reside.

Message and Message Queue Properties

A message has properties, which specify a label, a message body, and a number of options. Message property options can include quality of service, priority, journaling, privacy and authentication levels, and the lifetime of the message. In conventional (non-RPC) message-queuing applications, you specify these properties by calling the MSMQ API functions or COM object methods, which are described in the MSMQ SDK documentation. RPC client applications can set certain properties for remote procedure calls by calling **RpcBindingSetOption** and **RpcBindingSetAuthInfo**. Once set, the properties remain in effect for each message until the client application resets them.

A message queue has its own set of properties, apart from those of the messages. These properties uniquely identify a queue and define the class of service that the queue provides, the privacy and authentication levels required for messages in this queue, and whether the messages are to be part of a distributed transaction. As with message properties, you specify the properties of a message queue by calling the MSMQ API functions or COM object methods, which are described in the MSMQ documentation. However, an RPC server application can specify properties of its own receive queue when it calls **RpcServerUseProtseqEpEx** to establish the binding.

Using MSMQ as an RPC Transport

The RPC subsystem supports using MSMQ as a transport in synchronous and asynchronous modes.

Synchronous mode uses conventional remote procedure calls. These calls use well-known endpoints and the message queue transport, **ncadg_mq**, as the transport protocol. In synchronous mode, your remote procedures can have **[in]** and **[out]** parameters and can use the standard RPC security services. The RPC subsystem creates a reply queue for remote calls containing **[out]** parameters. The synchronous mode is useful for applications where the client needs to receive data from the server. The main limitation of this mode is that, as with conventional remote procedure calls, both the client and server must be running and remain running for the duration of the call.

Asynchronous mode lets client applications make calls to the server and return immediately, regardless of the state of the server application or the server computer. It also makes a subset of MSMQ features available for managing message queues and information flow. The **RpcBindingSetOption** function lets you control quality of service, call priority, journaling, security, and the lifetime of the server process queue. The **RpcServerUseProtseqEpEx** function lets you specify attributes of the server process queue, such as queue persistence, authentication, and encryption.

You implement asynchronous MSMQ as you would synchronous MSMQ. You must use well-known endpoints, and define the transport protocol to be **ncadg_mq**. In your IDL file, apply the **message** attribute to the functions that use asynchronous message queuing. Note that message functions can have **[in]** parameters only.

System Requirements for RPC-MQ Applications

To use the message-queuing transport in an RPC client/server application, the server and client computers must have the appropriate operating system platform and Message Queuing software installed.

Requirements for server computers are:

- Microsoft® Windows 95 with the second release of DCOM 95, Windows 98, Windows NT Server version 4.0 with Service Pack 3 or later, including Windows 2000 Professional.
- SQL Server version 6.5 or later.
- Message Queuing Primary Enterprise Controller or Primary Site Controller.
- RPC server-side transport DLL (RpcMqSvr.dll).

Requirements for client computers are:

- Windows 98 or Windows NT/Windows 2000. Support for Windows 95 is available with the second release of DCOM 95.
- Microsoft Message Queuing Client.
- RPC client-side transport DLL (RpcMqCl.dll).

When the MSMQ components are installed on the client and server computers, the system registries are automatically configured to include the **ncadg_mq** message-queuing transport protocol for remote procedure calls. For detailed information on installing the MSMQ components see the Windows NT or Windows 2000 online help.

To build your RPC-MQ application you need the following:

- Windows NT/Windows 2000. Windows NT 4.0 Service Pack 3 contains new RPC runtime DLLs, new RPC header files, and a new Rpcss.exe.
- MIDL version 3.1.76 or later.

Developing RPC-MQ Applications

Very little effort is necessary to take advantage of the MSMQ transport in your RPC application. For synchronous messaging you need only specify the message queue transport (**ncadg_mq**) as the protocol sequence. The **ncadg_mq** protocol supports all of the standard datagram features except broadcasting calls. Also, note that currently the message-queue transport does not support dynamic endpoints.

By applying the **[message]** attribute to remote procedure declarations in the IDL file, you automatically implement asynchronous-mode message queuing for those calls. This makes it possible for the client and server applications to control many of the properties associated with messages and message queues, including:

- Quality of service
- Acknowledgment of receipt
- Journaling
- Call priority
- Persistence of Server Process Queue

Quality of service is the effort that the transport will make to deliver the call to the server process. An express delivery will be queued in memory, so it is fairly fast, but the call will be lost if a computer or network connection goes down at the wrong time. A recoverable delivery will be posted to a disk file until it is delivered, so the call will not be lost, even in the face of a computer crash. This gives guaranteed delivery, but at a cost in performance as each call is written to disk.

You can also tell the MSMQ transport to wait for acknowledgment that the call reached the destination (server) queue before returning. Choosing this option blocks the client until the server acknowledges the call, otherwise control returns to the client immediately upon making the call.

By using journaling, calls can be logged to disk. If journaling is turned on, each call is logged to disk as it is transmitted to the next hop on its way to the server process.

Call priority can be used in conjunction with the RPC **[message]** function attribute to allow calls with higher priority to take precedence over calls with lower priority, even if the high priority calls arrive later. Call priority will also work in a limited fashion with synchronous RPC, but synchronous RPC calls cannot stack up in the same manner as asynchronous calls.

The client process controls all of the above properties by calling **RpcBindingSetOption**. Once set, these properties remain in effect until they are changed in another call to **RpcBindingSetOption**.

The RPC server process can control the lifetime of its receive queue. By default the queue is deleted when the server process exits. However, the server process can use **RpcServerUseProtseqEpEx** when setting up its endpoint to tell the transport to allow the queue to continue to exist and to accept call requests even when the server process is not running. In this case, the calls are queued up and executed later, when the server process comes back online.

Note If you are using asynchronous **[message]** calls in an interface, you must register the interface by calling **RpcServerRegisterIf** or **RpcServerRegisterIfEx** *before* calling **RpcServerUseProtseqEpEx(ncadg_mq)**. Once you turn on the protocol sequence, any calls already waiting on the queue for the server will begin to be read off the queue. If the corresponding RPC interface has not been registered, the calls will fail. This situation can happen if you have a setup a permanent endpoint for your remote procedure calls, the server has been shutdown, and clients have continued to send calls to the server. These calls will be stacked up in the queue, waiting to be read once the server comes back online.

See Also

RpcBindingSetOption, **RpcServerUseProtseqEpEx**, **message**, **ncadg_mq**

MSMQ Security Services

Synchronous RPC messages can use any of the security features available from the RPC run time. See Security for more details.

Asynchronous **[message]** calls cannot use RPC security because there is no handshake between client and server. In fact, the server may not even be running at the time of the call. To access the security services provided by Message Queuing Services (MSMQ), the client application should call **RpcBindingSetAuthInfo** to control the level of authentication and privacy for its calls to the server.

The server application can call **RpcBindingInqAuthClient** from within a remote procedure call to determine the security level for that call. The mapping between RPC security constants and MSMQ security is shown in the following table.

RPC security level	**Description**
RPC_AUTHN_LEVEL_NONE	The call is not authenticated or encrypted.
RPC_AUTHN_LEVEL_PKT_INTEGRITY	The call is authenticated using MSMQ security.
RPC_AUTHN_LEVEL_PKT_PRIVACY	The call is authenticated and encrypted as it travels between the client and server queue.

The server can also force call authentication and encryption by calling **RpcServerUseProtseqEpEx** and setting the RPC_C_MQ_AUTHN_LEVEL_NONE, RPC_C_MQ_AUTHN_LEVEL_PKT_INTEGRITY and RPC_C_MQ_AUTHN_LEVEL_PKT_PRIVACY flags in the RPC_POLICY structure.

CHAPTER 22

Remote Procedure Calls Using HTTP

Internet browser programs commonly employ the Hypertext Transport Protocol (HTTP) as the primary means of browsing the World Wide Web. HTTP, therefore, sees extensive usage on most computers today. Microsoft® Corporation has extended the capabilities of its Internet Information Server (IIS) to provide remote procedure call services using HTTP.

Microsoft HTTP RPC provides RPC clients with the ability to securely connect across the Internet to RPC server programs and execute remote procedure calls. If the client can make an HTTP connection to a computer on a remote network running an IIS, it can (subject to access restrictions) connect to any available server on the remote network and execute remote procedure calls. The RPC client and server programs can connect across the Internet—even if both are behind firewalls on different networks.

Using HTTP as an RPC Transport

RPC over HTTP supports synchronous remote procedure calls. It enables client programs to use the Internet to execute procedures provided by server programs on distant networks. HTTP RPC *tunnels* its calls through an established HTTP port. Thus, its calls can cross network firewalls on both the client and server networks.

HTTP RPC routes its calls to an Internet Information Server (IIS) located on the RPC server's network. The IIS server establishes and maintains a connection to the RPC server. It serves as a proxy, dispatching remote procedure calls to the RPC server and sending the server's replies back across the Internet to the client application. This process is illustrated in Figure 22-1.

The diagram shows a firewall on the client application's network. This is not required for HTTP RPC to operate. However, if the client network does have a firewall, it will also need a proxy server program such as Microsoft Proxy Server.

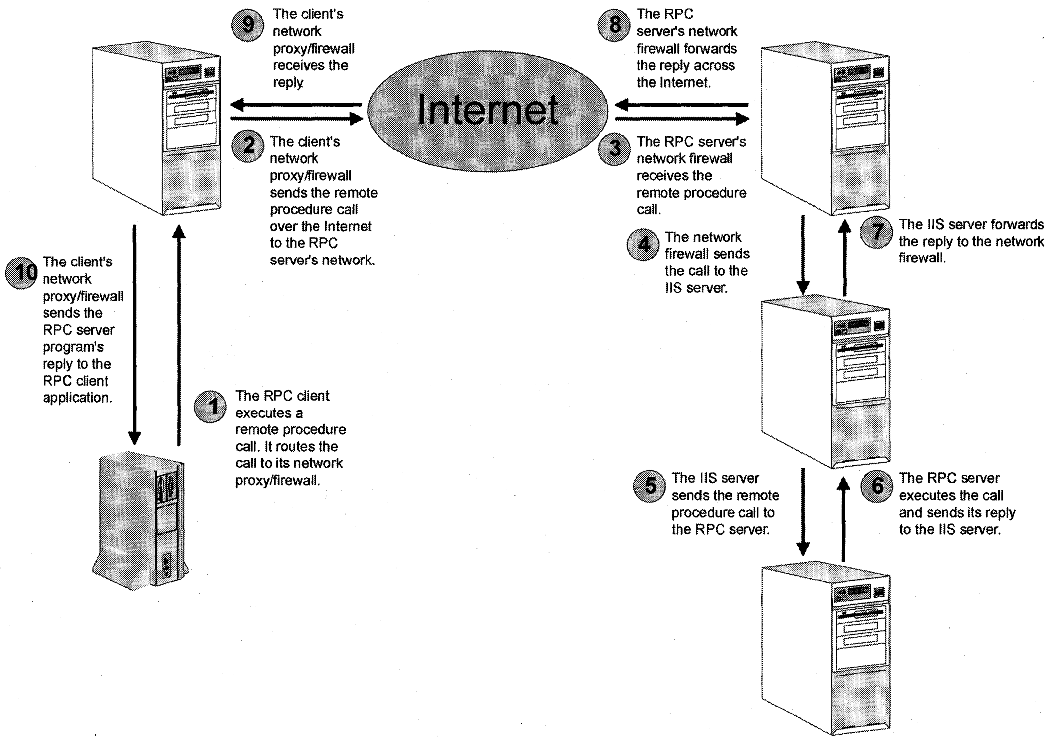

9 The client's network proxy/firewall receives the reply.
8 The RPC server's network firewall forwards the reply across the Internet.
Internet
2 The client's network proxy/firewall sends the remote procedure call over the Internet to the RPC server's network.
3 The RPC server's network firewall receives the remote procedure call.
4 The network firewall sends the call to the IIS server.
7 The IIS server forwards the reply to the network firewall.
10 The client's network proxy/firewall sends the RPC server program's reply to the RPC client application.
1 The RPC client executes a remote procedure call. It routes the call to its network proxy/firewall.
5 The IIS server sends the remote procedure call to the RPC server.
6 The RPC server executes the call and sends its reply to the IIS server.

Figure 22-1: A Firewall on the Client Applications' Network.

When the client program issues a remote procedure call using HTTP as the transport, the RPC run-time library on the client contacts the IIS. It typically uses port 80. The client stub must negotiate a TCP/IP connection to the RPC server program on the remote network. To facilitate this, the IIS acts as a proxy. It contacts the RPC server program and establishes a TCP/IP connection. The client and the IIS maintain their HTTP connection across the Internet and use it as a pure TCP/IP connection. The client's HTTP connection to the IIS can pass through a firewall (subject to appropriate access permissions) if one is present. The server can then execute the remote procedure call and use the connection through the IIS to reply to the client.

If either the client or the server disconnects for any reason, the IIS will detect it and end the RPC session. As long as the session continues, the IIS will maintain its connections to the client and the server. It will forward remote procedure calls from the client to the server, and send replies from the server to the client.

Your RPC client program can tunnel its RPC calls through the Internet by creating a string binding of the form:

```
object_uuid@ncacn_http:rpc_server[endpoint,HttpProxy=proxy_server:http_port,RpcPr
oxy=rpc_proxy:rpc_port]
```

Where:

- *object_uuid* specifies an RPC object UUID. For details see *Generating Interface UUIDs* and *String UUID*.
- **ncacn_http** selects the protocol sequence specification for HTTP RPC. See *PROTSEQ* and *String Binding*.
- *rpc_server* is the network address of the computer that is executing the RPC server process.
- *endpoint* specifies the TCP/IP port that the RPC server process listens to for remote procedure calls. See *Finding Endpoints*.
- **HttpProxy** optionally specifies an HTTP proxy server on the RPC client's network, such as Microsoft Proxy Server. If a proxy server is selected, no port number is specified, the RPC stub uses port 80 by default.
- **RpcProxy** specifies the address and port number of the IIS computer that acts as a proxy to the RPC server. You only need to specify this if the RPC server process resides on a different computer than the IIS RPC proxy. If you do not specify a port number, the RPC client stub uses port 80 by default.

For more information on creating string bindings, see *Binding and Handles*.

Note If Microsoft® Internet Explorer is installed on the client program's computer and your client does not specify an **HttpProxy** in its string binding, the RPC client stub will search the registry on the client computer for an **HttpProxy** entry. If it finds one, it will use the proxy specified in the registry entry.

Suppose, for instance, your client program needs to connect across the Internet to an RPC server on a computer called Major7.somewhere.com. Further, suppose that the RPC server program and the IIS both run on Major7.somewhere.com. The RPC Server program listens to port 2225. Your client would use the string binding:

```
ncacn_http:major7.somewhere.com[2225]
```

If, however, the IIS runs on a computer called WebSvr1.somewhere.com and the RPC server program uses dynamic endpoints, your client would use the following string binding:

```
ncacn_http:major7.somewhere.com[RpcProxy=websvr1.somewhere.com]
```

The RPC client stub will contact the IIS on WebSvr1.somewhere.com and use it as a proxy to connect to the RPC server program on Major7.somewhere.com.

If the client network uses a firewall and an Internet proxy server program called myproxy, you would need to modify the client's string binding to:

```
ncacn_http:major7.somewhere.com[,HttpProxy=myproxy:80,RpcProxy=websvr1.somewhere.
com:80]
```

This directs the client to connect to the RPC server program on Major7.somewhere.com. To do this, the client will first use port 80 to connect the program myproxy. This will give the client program access to the Internet. Using the Internet, the client program next connects to the IIS on WebSvr1.somewhere.com. The IIS RPC proxy will establish a connection to the RPC server program running on Major7.somewhere.com.

HTTP RPC Security

HTTP RPC utilizes two types of security:

- It restricts access to remote server procedures based on the normal RPC security mechanisms. For a detailed presentation, see *Security*.
- It provides security through the IIS.

Note Since the IIS serves as a proxy to connect the RPC client and server programs across the Internet, it also can restrict unauthorized access to servers.

The IIS security configuration is based on allowed computer and port ranges. The ability to use HTTP RPC is controlled through two registry entries on the computer running the IIS. The first entry is a flag that turns the RPC Proxy on or off. The second is an optional list of computers to which the proxy can forward RPC calls.

By default, a client that contacts an IIS to tunnel RPC calls over HTTP cannot access any RPC server processes anywhere. If the ENABLED flag is not defined and set to a nonzero value, the IIS will disable the HTTP RPC proxy. If the ENABLED flag is defined and set to a nonzero value, a client can connect to RPC servers on the computer running IIS. To enable the client to tunnel to an RPC server process on another computer, you must add a registry entry to the IIS computer's list of RPC servers.

The following example demonstrates how to configure the registry to allow clients to connect to servers across the Internet:

```
\HKEY_LOCAL_MACHINE
    \Software\Microsoft\Rpc\RpcProxy\Enabled:REG_DWORD
    \Software\Microsoft\Rpc\RpcProxy\ValidPorts:REG_SZ
```

The **ValidPorts** entry is a REG_SZ entry containing a list of computers to which the IIS RPC proxy is allowed to forward RPC calls, and the ports it should use to connect to the RPC servers. The **REG_SZ** entry takes the form shown in the following example:

```
Rosco:593;Rosco:2000-8000;Data*:4000-8000
```

In this example, the IIS can forward HTTP RPC calls to the server Rosco on ports 593 and 2000 through 8000. It can also send them to any server whose name begins with "Data". It will connect on ports 4000 through 8000.

The IIS reads the **Enabled** and **ValidPorts** registry entries when it starts up. If you change one or both of these entries, you will need to stop and restart the WEB service using the Internet Service Manager for the new values to take effect.

System Requirements for HTTP RPC

Microsoft® RPC supports HTTP RPC as shown in the following table.

Platform	**Supports**	**Comments**
Windows® 2000	Clients and servers	HTTP RPC server program and the IIS can be running on different computers.
Windows NT® 4.0 with SP4	Clients and servers	HTTP RPC server program must be running on the same computer as the IIS.
Windows 95/98	Clients	Does not support HTTP RPC servers. Windows 95 clients must have DCOM 95 v1.2 or later installed.
MS-DOS®	N/A	Does not support HTTP RPC clients or servers.
Macintosh	N/A	Does not support HTTP RPC clients or servers.

In addition, the following requirements apply:

- Windows NT 4.0 with Service Pack 4 requires the use of IIS 4.0 or later.
- The IIS HTTP RPC proxy runs on Windows NT Server or Windows 2000 Server. It does not run on Windows NT Workstation or Windows 2000 Professional. Nor does it run on Personal Web Server for Windows 95/Windows 98.
- If the IIS is running on Windows 2000 Server, the HTTP RPC server program can run on any computer that the IIS has permissions to access. Therefore, it can run on the same computer as the IIS, or a different computer.

Configuring Computers for HTTP RPC

To use HTTP as a transport protocol for RPC, you or your network administrator must configure the Internet Information Server (IIS) on the server program's network. The IIS has to be set so that it accepts HTTP commands to establish a connection to an RPC server program.

Also, you may need to enable COM Internet Services (CIS) on your Microsoft® Windows® 2000 IIS server system.

▶ To enable CIS

1. Click **Settings** from the Windows 2000 **Start** menu in the Taskbar.
2. Select **Control Panel**, and click **Add/Remove Programs**.
3. From the **Add/Remove Programs** dialog box, select **Add/Remove Windows Components**.

   This will start the Windows Component Wizard.
4. Click **Next.**
5. Select **Networking Services** and click **Details**.
6. Select **COM Internet Services Proxy**, then click **OK**.

   This will install CIS.
7. After installation is complete, either reboot the system, or restart IIS.

In addition, you (or the network administrator of the client's network) may also need to configure the proxy server program on the network on which the RPC client program runs. This allows the RPC client program to send HTTP commands through the client network's firewall. For more information on configuring the client network's proxy server, see the documentation provided by the manufacturer of the proxy server program.

CHAPTER 23

RPC Samples

The Platform SDK includes sample programs that demonstrate a variety of Remote Procedure Call (RPC) concepts, as follows:

- ASYNCRPC illustrates the structure of an RPC application that uses asynchronous remote procedure calls. It also demonstrates various methods of notification of the call's completion.
- CALLBACK demonstrates use of the **[callback]** attribute.
- CLUUID demonstrates use of the client-object UUID to enable a client to select from multiple implementations of a remote procedure.
- DATA directory contains four programs: DUNION illustrates discriminated (nonencapsulated) unions; INOUT demonstrates **[in**, **out]** parameters; REPAS demonstrates the **[represent_as]** attribute; XMIT demonstrates the **[transmit_as]** attribute.
- DICT is a remote splay tree-based dictionary program that uses the **[implicit_handle]**, **[context_handle]**, **[in]** attribute, and **[out]** attribute.
- DOCTOR is an RPC psychotherapy application that demonstrates arrays, strings, and the **[size_is]** attribute.
- DYNEPT demonstrates a client application managing its connection to the server through dynamic endpoints.
- DYNOUT demonstrates how to allocate memory at a server for an n-byte object and pass it back to the client as an **[out]**-only parameter. The client then frees the memory. This technique allows the stub to call the server without knowing in advance how much data will be returned.
- HANDLES directory contains three programs, AUTO, CXHNDL, USRDEF, which demonstrate **[auto_handle]**, **[context_handle]**, and generic (user-defined) handles, respectively.
- HELLO is a client/server implementation of "Hello, world."
- INTEROP demonstrates portability between Open Software Foundation–Distributed Computing Environment (OSF-DCE) and Microsoft® RPC.
- MANDEL is a distributed fractal drawing program. It uses **[ref]** pointers, the **[implicit_handle]** attribute, and **handle_t** primitive types.
- NS directory contains the NHELLO program, which demonstrates name service usage. The CDS directory contains the files that describe the gateway protocol to the DCE Cell Directory Service (CDS).

- OBJECT directory contains two programs that demonstrate OLE custom interfaces. CALLAS uses the **[call_as]** attribute to transmit a nonremotable interface. OHELLO demonstrates correct reference counting and shutdown behavior for a multiple-use local server. OHELLO also demonstrates how to use the Win32 registry functions to install a local server and a proxy DLL in the registry.
- PICKLE directory contains two programs: PICKLP demonstrates data procedure serialization; PICKLT demonstrates data type serialization; both programs use the **[encode]** and **[decode]** attributes.
- PIPES demonstrates the use of the **pipe** type constructor.
- RPCSSM demonstrates the RPCSS memory management model.
- RPCSVC demonstrates the implementation of a Windows NT®/Windows® 2000™ service with RPC.
- STROUT demonstrates how to allocate memory at a server for a two-dimensional object (an array of pointers) and pass it back to the client as an **[out]**-only parameter. The client then frees the memory. This technique allows the stub to call the server without knowing in advance how much data will be returned.

  This program also allows the user to compile either for UNICODE or ANSI.

You can browse through most of these files in *Reference/Code Samples/NETDS/RPC* in the Platform SDK. All of the source files and makefiles for these programs are located in the SDK at *mstools\samples\rpc*.

CHAPTER 24

RPC Data Types, Structures, and Constants

This chapter describes the structures, data types, and constants that Microsoft RPC uses.

RPC Structures

This section explains the structures defined and used by Microsoft RPC.

- GUID
- NDR_USER_MARSHAL_INFO
- RPC_ASYNC_STATE
- RPC_BINDING_VECTOR
- RPC_CLIENT_INTERFACE
- RPC_DISPATCH_TABLE
- RPC_IF_ID
- RPC_IF_ID_VECTOR
- RPC_POLICY
- RPC_PROTSEQ_VECTOR
- RPC_SECURITY_QOS
- RPC_STATS_VECTOR
- SEC_WINNT_AUTH_IDENTITY
- UUID
- UUID_VECTOR

GUID

GUIDs identify objects such as interfaces, manager entry-point vectors (EPVs), and class objects. A GUID is a 128-bit value consisting of one group of 8 hexadecimal digits, followed by three groups of 4 hexadecimal digits each, followed by one group of 12 hexadecimal digits. The following example shows the groupings of hexadecimal digits in a GUID.

```
    6B29FC40-CA47-1067-B31D-00DD010662DA
```

The **GUID** structure stores a GUID.

```
typedef struct _GUID
{
  DWORD  Data1;
  WORD   Data2;
  WORD   Data3;
  BYTE   Data4[8];
} GUID;
```

Members

Data1
Specifies the first 8 hexadecimal digits of the GUID.

Data2
Specifies the first group of 4 hexadecimal digits.

Data3
Specifies the second group of 4 hexadecimal digits.

Data4
Specifies an array of 8 bytes. The first 2 bytes contain the third group of 4 hexadecimal digits. The remaining 6 bytes contain the final 12 hexadecimal digits.

Remarks

GUIDs are the Microsoft implementation of the distributed computing environment (DCE) universally unique identifier (UUID).

The RPC run-time libraries use UUIDs to check for compatibility between clients and servers and to select among multiple implementations of an interface.

The Win32 access-control functions use GUIDs to identify the type of object that an object-specific ACE in an access-control list (ACL) protects.

Requirements

Version: Requires MAPI 1.0 or later.
Header: Declared in Mapiguid.h.

See Also

ACCESS_ALLOWED_OBJECT_ACE, **ACE**, **ACL**, **UUID**, **UUID_VECTOR**

NDR_USER_MARSHAL_INFO

The **NDR_USER_MARSHAL_INFO** structure holds information about the state of an RPC call that can be passed to **wire_marshal** and **user_marshal** helper functions.

```
typedef struct _NDR_USER_MARSHAL_INFO_LEVEL1
{
  void *Buffer;
  unsigned long BufferSize;
  void *(_RPC_API *pfnAllocate)(size_t);
  void (_RPC_API *pfnFree)(void *);
  struct IRpcChannelBuffer *pRpcChannelBuffer;
  ULONG_PTR Reserved[5];
} NDR_USER_MARSHAL_INFO_LEVLE1;

typedef struct _NDR_USER_MARSHAL_INFO
{
  unsigned long InformationLevel;
  union{
  NDR_USER_MARSHAL_INFO_LEVEL1 Level1;
  }
} NDR_USER_MARSHAL_INFO;
```

Members

InformationLevel
The information level of the returned data. Currently only a value of 1 is defined.

Level1 Members

Buffer
Points to the beginning of the marshaling buffer tht is available for use by the helper function. If no buffer is available this field is NULL.

BufferSize
The size, in bytes, of the marshaling buffer that is available for use by the helper function. If no buffer is available, **BufferSize** is zero.

pfnAllocate
The function that RPC uses to allocate memory for the application. An example of the use of this function is to create a node.

pfnFree
The function that RPC uses to free memory for the application. An example of the use of this function is to free a node.

pRpcChannelBuffer
If the current call is for a DCOM interface, this member is a pointer to the channel buffer that RPC uses for the call. Otherwise, this member is NULL.

Reserved
Reserved for future use.

Remarks

The function **NdrGetUserMarshalInfo** fills this structure with supplemental information for the **user_marshal** and **wire_marshal** helper functions <*type*>**_UserSize,** <*type*>**_UserMarshal,** <*type*>**_UserUnmarshal**, and <*type*>**_UserFree**. This information supplements the *pFlags* parameter that is passed to these helper functions. Not all of these fields will contain valid information in all contexts. **Level1.pRpcChannelBuffer** is only valid for DCOM interfaces, and the buffer fields are only valid when **NdrGetUserMarshalInfo** is called from <*type*>**_UserMarshal** or <*type*>**_UserUnmarshal**.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Rpcndr.h.
Library: Use Rpct4.lib.

RPC_ASYNC_STATE

The **RPC_ASYNC_STATE** structure holds the state of an asynchronous remote procedure call. **PRPC_ASYNC_STATE** is a handle to this structure, used to wait for, query, reply to, or cancel asynchronous calls.

```
typedef struct _RPC_ASYNC_STATE
{
  unsigned int Size;
  unsigned long Signature;
  unsigned long Flags;
  long Lock;
  void *StubInfo
  void *UserInfo
  void *RuntimeInfo
  RPC_ASYNC_EVENT Event
  RPC_NOTIFICATION_TYPES NotificationType
  union
  {
     struct
     {
       PFN_RPCNOTIFICATION_ROUTINE NotificationRoutine;
       HANDLE hThread;
     } APC;
     struct
     {
       HANDLE hIOPort;
       DWORD dwNumberOfBytesTransferred;
       DWORD dwCompletionKey;
       LPOVERLAPPED lpOverlapped;
     } IOC;
     struct
     {
       HWND   hWnd;
       UINT   Msg
     } HWND;  //not implemented in Beta1
     HANDLE hEvent;
```

```
        PFN_RPCNOTIFICATION_ROUTINE NotificationRoutine;
    } u;
    long Reserved[4];
} RPC_ASYNC_STATE, *PRPC_ASYNC_STATE ;
```

Members

Size

The size, in bytes, of this structure. The environment sets this member when **RpcAsyncInitializeHandle** is called. Do not modify this member.

Signature

The run-time environment sets this member when **RpcAsyncInitializeHandle** is called. Do not modify this member.

Flags

The **flags** member can be set to the following values.

Constant	Description
RPC_C_NOTIFY_ON_SEND_COMPLETE	Posts a notification message when the asynchronous operation is complete.
RPC_C_INFINITE_TIMEOUT	Does not return from the asynchronous operation until it is complete.

These flags are used with DCE pipes, which allow applications to send or receive data in multiple blocks. Programs can either send a continuous stream of data or wait for each block to be transmitted before it sends the next block. If it does not wait, the RPC run-time library will buffer the output until it can be sent. When the data transmission is complete, the RPC library sends the application a notification. If an application specifies the RPC_C_NOTIFY_ON_SEND_COMPLETE flag, the RPC library sends it a member of the **RPC_NOTIFICATION_TYPES** enumeration after it completes each send operation.

Lock

The run-time environment sets this member when **RpcAsyncInitializeHandle** is called. Do not modify this member.

StubInfo

Reserved for use by the stubs. Do not use this member.

UserInfo

Use this member for any application-specific information that you want to keep track of in this structure.

RuntimeInfo

Reserved for use by the RPC run-time environment. Do not use this member.

Event

Specifies the type of event that occurred. The RPC run-time environment sets this field to a member of the **RPC_ASYNC_EVENT** enumeration.

NotificationType
Specifies the kind of notification in use. The notification type may be any of the values defined in **RPC_NOTIFICATION_TYPES**.

APC
Structure used for Windows asynchronous procedure call (APC) notifications.

NotificationRoutine
Calls the user-defined APC notification routine.

hThread
The handle to the thread making the asynchronous call. A value of zero indicates the current thread.

IOC
Structure used for notification by the I/O completion port.

hIOPort
Handle to the I/O completion port.

dwNumberOfBytesTransferred
Set by the calling application (either client or server).

dwCompletionKey
Set by the calling application (either client or server).

lpOverlapped
The address of the **OVERLAPPED** structure containing state information needed for I/O completion.

HWND
Structure used for notification by a Windows message. The *lParam* parameter points to the asynchronous handle for the call. Not implemented in Windows 2000.

hWnd
Identifies the window to which the message should be posted.

Msg
The message to be posted.

hEvent
Handle used for notification by an event.

NotificationRoutine
DCOM uses this internally for direct callbacks. Do not use this member.

Reserved[4]
Reserved for compatibility with future versions, if any. Do not use this member.

Remarks

The client allocates space for the **RPC_ASYNC_STATE** structure and an associated handle, and calls **RpcAsyncInitializeHandle** to initialize the structure. After the run-time environment has successfully initialized the structure, the client initializes the **NotificationType**, **NotificationRoutine**, **hThread**, and **UserInfo** fields.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Rpcasync.h.

+ See Also

Asynchronous RPC, **RpcAsyncAbortCall**, **RpcAsyncCancelCall**, **RpcAsyncCompleteCall**, **RpcAsyncGetCallHandle**, **RpcAsyncGetCallStatus**, **RpcAsyncInitializeHandle**, **RpcAsyncRegisterInfo**, **RpcServerTestCancel**

RPC_BINDING_VECTOR

The **RPC_BINDING_VECTOR** structure contains a list of binding handles over which a server application can receive remote procedure calls.

```
#define rpc_binding_vector_t RPC_BINDING_VECTOR
typedef struct _RPC_BINDING_VECTOR
{
  unsigned long Count;
  RPC_BINDING_HANDLE BindingH[1];
} RPC_BINDING_VECTOR;
```

Members

Count
Specifies the number of binding handles present in the binding-handle array **BindingH**.

BindingH
Specifies an array of binding handles that contains **Count** elements.

Remarks

The binding vector contains a count member (**Count**), followed by an array of binding-handle (**BindingH**) elements.

The RPC run-time library creates binding handles when a server application registers protocol sequences. To obtain a binding vector, a server application calls **RpcServerInqBindings**.

A client application obtains a binding vector of compatible servers from the name-service database by calling **RpcNsBindingLookupNext**.

In both routines, the RPC run-time library allocates memory for the binding vector. An application calls **RpcBindingVectorFree** to free the binding vector.

To remove an individual binding handle from the vector, the application must set the value in the vector to NULL. When setting a vector element to NULL, the application must:

- Free the individual binding.
- Not change the value of **Count**.

Calling **RpcBindingFree** allows an application to both free the unwanted binding handle and set the vector entry to a NULL value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RpcBindingVectorFree, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcEpUnregister**, **RpcNsBindingExport**, **RpcNsBindingLookupNext**, **RpcNsBindingSelect**, **RpcServerInqBindings**

RPC_CLIENT_INTERFACE

Remarks

The **RPC_CLIENT_INTERFACE** structure is part of the private interface between the run-time libraries and the stubs. Most distributed applications that use Microsoft RPC do not need this structure.

The data structure is defined in the header file Rpcdcep.h.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdcep.h.

RPC_DISPATCH_TABLE

Remarks

The **RPC_DISPATCH_TABLE** structure is part of the private interface between the run-time libraries and the stubs. Most distributed applications that use Microsoft RPC do not need this structure.

The structure is defined in the header file Rpcdcep.h.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdcep.h.

RPC_IF_ID

The **RPC_IF_ID** structure contains the interface UUID and major and minor version numbers of an interface.

```
typedef struct _RPC_IF_ID
{
  UUID Uuid;
  unsigned short VersMajor;
  unsigned short VersMinor;
} RPC_IF_ID;
```

Members

Uuid
Specifies the interface UUID.

VersMajor
Specifies the major version number, an integer from 0 to 65535, inclusive.

VersMinor
Specifies the minor version number, an integer from 0 to 65535, inclusive.

Remarks

An interface identification is a subset of the data contained in the interface-specification structure. Routines that require an interface identification structure show a data type of **RPC_IF_ID**. In those routines, the application is responsible for providing memory for the structure.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

+ See Also

RpcIfInqId

RPC_IF_ID_VECTOR

The **RPC_IF_ID_VECTOR** structure contains a list of interfaces offered by a server.

```
typedef struct _RPC_IF_ID_VECTOR
{
  unsigned long Count;
  RPC_IF_ID *IfHandl[1];
} RPC_IF_ID_VECTOR;
```

Members

Count

Specifies the number of interface-identification structures present in the array **IfHandl**.

IfHandl

Specifies an array of pointers to interface-identification structures that contains **Count** elements.

Remarks

The interface identification vector contains a count member (**Count**), followed by an array of pointers to interface identifiers (**RPC_IF_ID**).

The interface identification vector is a read-only vector. To obtain a vector of the interface identifiers registered by a server with the run-time library, an application calls **RpcMgmtInqIfIds**. To obtain a vector of the interface identifiers exported by a server, an application calls **RpcNsMgmtEntryInqIfIds**.

The RPC run-time library allocates memory for the interface identification vector. The application calls **RpcIfIdVectorFree** to free the interface identification vector.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RpcIfIdVectorFree, **RpcMgmtInqIfIds**, **RpcNsMgmtEntryInqIfIds**

RPC_POLICY

The **RPC_POLICY** structure contains flags that determine port allocation and binding to multihomed computers, using the **ncacn_ip_tcp** and **ncadg_ip_udp** protocols, and message queue properties, over the **ncadg_mq** protocol.

```
typedef struct _RPC_POLICY
{
  unsigned int Length;
  unsigned long EndpointFlags;
  unsigned long NICFlags;
} RPC_POLICY, __RPC_FAR *PRPC_POLICY;
```

Members

Length

The size, in bytes, of the **RPC_POLICY** structure. The **Length** member allows compatibility with future versions of this structure, which may contain additional fields. Always set the **Length** equal to **sizeof**(RPC_POLICY) when you initialize the **RPC_POLICY** structure in your code.

EndpointFlags

A set of flags that determine the attributes of the port or ports where the server receives remote procedure calls. You can specify more than one flag (by using the bitwise OR operator) from the set of values for a given protocol sequence. The following table lists the possible values for the **EndpointFlags** member.

Value	Description
0	Specifies the system default.
RPC_C_USE_INTERNET_PORT	Allocates the endpoint from one of the ports defined in the registry as "Internet Available." Valid only with **ncacn_ip_tcp** and **ncadg_ip_udp** protocol sequences.
RPC_C_USE_INTRANET_PORT	Allocates the endpoint from one of the ports defined in the registry as "Intranet Available." Valid only with **ncacn_ip_tcp** and **ncadg_ip_udp** protocol sequences.
RPC_C_MQ_TEMPORARY	The server process–receive queue will be deleted automatically when the RPC server exits. Any outstanding calls still in the queue will be lost. This is the default. Valid only with the **ncadg_mq** protocol sequence.
RPC_C_MQ_PERMANENT	Specifies that the server process–receive queue persists after the server process exits. The default is that the queue is deleted when the server process terminates. Valid only with **ncadg_mq** protocol sequence.
RPC_C_MQ_CLEAR_ON_OPEN	If the receive queue already exists because it was opened previously as a permanent queue, then clear any outstanding calls waiting in the queue. Valid only with the **ncadg_mq** protocol sequence only.

(continued)

(continued)

Value	Description
RPC_C_MQ_USE_EXISTING_ SECURITY	If the receive queue already exists, then don't modify its existing settings for authentication or encryption. Valid only with the **ncadg_mq** protocol sequence.
RPC_C_MQ_AUTHENTICATE	The server process–receive queue accepts only authenticated calls from clients. The default is that both authenticated and unauthenticated calls are accepted. Valid only with **ncadg_mq** protocol sequence.
RPC_C_MQ_ENCRYPT	Calls to server are encrypted. The default is that both encrypted and unencrypted calls are accepted. Valid only with **ncadg_mq** protocol sequence.
RPC_C_MQ_AUTHN_LEVEL_ NONE	The server's receive queue accepts all calls from clients. This is the default authentication level. Valid only with the **ncadg_mq** protocol.
RPC_C_MQ_AUTHN_LEVEL_ PKT_INTEGRITY	Sets the server's receive queue to only accept client calls that have authentication level RPC_C_AUTHN_LEVEL_PKT_INTEGRITY or RPC_C_AUTHN_LEVEL_PKT_PRIVACY. Valid only with the **ncadg_mq** protocol sequence.
RPC_C_MQ_AUTHN_LEVEL_ PKT_PRIVACY	Sets the server's receive queue to only accept client calls that have authentication level RPC_C_AUTHN_LEVEL_PKT_PRIVACY. Calls with a lower authentication level are ignored. Valid only with the **ncadg_mq** protocol sequence.

Note If the registry does not contain any of the keys that specify the default policies, then the **EndpointFlags** member will have no effect at run time. If a key is missing or contains an invalid value, then the entire configuration for that protocol (**ncacn_ip_tcp**, **ncadg_ip_udp** or **ncadg_mq**) is marked as invalid and all calls to **RpcServerUseProtseq*** functions over that protocol will fail.

NICFlags

The **NICFlags** member specifies the policy for binding to Network Interface Cards (NICs). The following table lists the possible values for the **NICFlags** member.

Value	Description
0	Binds to NICs on the basis of the registry settings. Always use this value when you are using the **RPC_POLICY** structure to define message-queue properties.
RPC_C_BIND_TO_ALL_NICS	Overrides the registry settings and binds to all NICs. If the Bind key is missing from the registry, then the **NICFlags** member will have no effect at run time. If the key contains an invalid value, then the entire configuration is marked as invalid and all calls to **RpcServerUseProtseq*** will fail.

Remarks

You can use the **RPC_Policy** structure to set policies for remote procedure calls at run time. These policies include:

- Message queuing: Allows the server to specify message-queuing properties, such as security, quality of delivery, and the lifetime of the server-process queue. This policy is only effective for remote calls over the message-queuing transport (**ncadg_mq**).
- Port allocation for dynamic ports: You can select the endpoint from the ports or sets of ports that are defined in the system registry as being Internet-available or intranet-available.
- Selective binding: Allows multihomed machines to bind selectively to NICs.

Note Port allocation and selective binding policies are effective only for remote calls over TCP (**ncacn_ip_tcp**) and UDP (**ncadg_ip_udp**) connections. For more information, see *Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RPC Message Queuing, Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding, **RpcServerUseAllProtseqsEx**, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqEpEx**, **RpcServerUseProtseqIfEx**

RPC_PROTSEQ_VECTOR

The **RPC_PROTSEQ_VECTOR** structure contains a list of protocol sequences the RPC run-time library uses to send and receive remote procedure calls.

```
typedef struct _RPC_PROTSEQ_VECTOR
{
  unsigned long Count;
  unsigned char *Protseq[1];
} RPC_PROTSEQ_VECTOR;
```

Members

Count
Specifies the number of protocol-sequence strings present in the array **Protseq**.

Protseq
Specifies an array of pointers to protocol-sequence strings. The number of pointers present is specified by the **Count** member.

Remarks

The protocol-sequence vector contains a count member (**Count**), followed by an array of pointers to protocol-sequence strings (**Protseq**).

The protocol-sequence vector is a read-only vector. To obtain a protocol-sequence vector, a server application calls **RpcNetworkInqProtseqs**. The RPC run-time library allocates memory for the protocol-sequence vector. The server application calls **RpcProtseqVectorFree** to free the protocol-sequence vector.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RpcNetworkInqProtseqs, **RpcProtseqVectorFree**

RPC_SECURITY_QOS

The **RPC_SECURITY_QOS** structure defines security quality-of-service settings on a binding handle.

Windows NT® 4.0: Limited functionality.
Windows 2000: Full functionality.
Windows® 95/98: Clients.

```
typedef struct _RPC_SECURITY_QOS
{
  unsigned long Version;
  unsigned long Capabilities;
  unsigned long IdentityTracking;
  unsigned long ImpersonationType;
} RPC_SECURITY_QOS, *PRPC_SECURITY_QOS;
```

Members

Version

Ensures compatibility with future extensions to the RPC security functions that reference this structure. Always assign the constant RPC_C_SECURITY_QOS_VERSION to this member.

Capabilities

The security services being provided to the application.

Value	Description
RPC_C_QOS_CAPABILITIES_ DEFAULT	Uses when no provider-specific capabilities are needed.
RPC_C_QOS_CAPABILITIES_ MUTUAL_AUTH	Not supported on Windows NT 4.0.
RPC_C_QOS_CAPABILITIES_ MAKE_FULLSIC	Not currently implemented.
RPC_C_QOS_CAPABILITIES_ ANY_AUTHORITY	Accepts the client's credentials even if the certificate authority (CA) is not in the server's list of trusted CAs. This constant is used only by the SCHANNEL SSP.

IdentityTracking

Sets the context tracking mode. Should be set to one of the following values:

Value	Description
RPC_C_QOS_IDENTITY_STATIC	Security context is created only once and is never revised during the entire communication, even if the client side changes it.
RPC_C_QOS_IDENTITY_DYNAMIC	Context is revised whenever the *LogonId* in the client's token is changed

ImpersonationType

The level at which the server process can impersonate the client.

Value	Description
RPC_C_IMP_LEVEL_DEFAULT	Uses the default impersonation level.
RPC_C_IMP_LEVEL_ANONYMOUS	Client does not provide identification information to the server.
RPC_C_IMP_LEVEL_IDENTIFY	Server can obtain information about client security identifiers and privileges, but cannot impersonate the client.
RPC_C_IMP_LEVEL_IMPERSONATE	Server can impersonate the client's security context on its local system, but not on remote systems.
RPC_C_IMP_LEVEL_DELEGATE	Not supported on Windows NT 4.0. See *Remarks*.

Remarks

The client-side security functions **RpcBindingInqAuthInfoEx** and **RpcBindingSetAuthInfo** use the **RPC_SECURITY_QOS** structure to inquire about, or to set, the security quality of service for a binding handle.

Note Windows NT and Windows 2000 do not support the delegation-impersonation level natively, but the security package may have the ability to provide it. In the current implementation, RPC always requests delegation level–security context from the security package for unauthenticated transports. It maps the requested level to the native level for authenticated transports (named pipes and LRPC).

The Windows NT LAN Manager (NTLM) security package, the default security package in Windows NT 4.0, does not currently support delegation and ignores the caller's request.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RpcBindingInqAuthInfoEx, **RpcBindingSetAuthInfoEx**

RPC_STATS_VECTOR

The **RPC_STATS_VECTOR** structure contains statistics from the RPC run-time library on a per-server basis.

```
typedef struct
{
  unsigned int Count;
  unsigned long Stats[1];
} RPC_STATS_VECTOR;
```

Members

Count

Specifies the number of statistics values present in the array **Stats**.

Stats

Specifies an array of unsigned long integers representing server statistics that contains **Count** elements. Each array element contains an unsigned long value from the following list.

Constant	Statistics
RPC_C_STATS_CALLS_IN	The number of remote procedure calls received by the server.
RPC_C_STATS_CALLS_OUT	The number of remote procedure calls initiated by the server.
RPC_C_STATS_PKTS_IN	The number of network packets received by the server.
RPC_C_STATS_PKTS_OUT	The number of network packets sent by the server.

Remarks

The statistics vector contains a count member (**Count**), followed by an array of statistics.

To obtain run-time statistics, an application calls **RpcMgmtInqStats**. The RPC run-time library allocates memory for the statistics vector. The application calls **RpcMgmtStatsVectorFree** to free the statistics vector.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RpcMgmtInqStats, **RpcMgmtStatsVectorFree**

SEC_WINNT_AUTH_IDENTITY

The **SEC_WINNT_AUTH_IDENTITY** structure allows you to pass a particular user name and password to the run-time library for the purpose of authentication.

For Windows 3.*x* and MS-DOS:

```
typedef struct _SEC_WINNT_AUTH_IDENTITY
{
  char __RPC_FAR *User;
  char __RPC_FAR *Domain;
  char __RPC_FAR *Password;
} SEC_WINNT_AUTH_IDENTITY;
```

For Windows NT, Windows 2000, Windows 95, and Apple Macintosh:

```
typedef struct _SEC_WINNT_AUTH_IDENTITY
{
  unsigned short __RPC_FAR *User;
  unsigned long UserLength;
  unsigned short __RPC_FAR *Domain;
  unsigned long DomainLength;
  unsigned short __RPC_FAR *Password;
  unsigned long PasswordLength;
  unsigned long Flags;
} SEC_WINNT_AUTH_IDENTITY, *PSEC_WINNT_AUTH_IDENTITY;
```

Members

User
String containing the user name.

Domain
String containing the domain name or the workgroup name.

Password
String containing the user's password in the domain or workgroup.

Flags
The following values are valid flags for this member of the **SEC_WINNT_AUTH_IDENTITY** structure.

Value	Description
SEC_WINNT_AUTH_IDENTITY_ANSI	The strings in this structure are in ANSI format.
SEC_WINNT_AUTH_IDENTITY_UNICODE	The strings in this structure are in Unicode format.

Remarks

Note that this structure must remain valid for the lifetime of the binding handle.

For Windows 95, Windows 3.*x,* and MS-DOS, the strings are ANSI; for Windows NT/Windows 2000, the strings may be ANSI or Unicode, depending on the value you assign to the **Flags** member. For Windows NT, Windows 2000, Windows 95, and Macintosh, the values for **UserLength**; **DomainLength**, and **PasswordLength** are the length of the corresponding string in characters, without the terminating null character.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

UUID

The **UUID** structure defines Universally Unique Identifiers (UUIDs). UUIDs provide unique designations of objects such as interfaces, manager entry-point vectors, and client objects.

```
typedef struct _GUID
{
  unsigned long  Data1;
  unsigned short Data2;
  unsigned short Data3;
  unsigned char  Data4[8];
} GUID;
typedef GUID UUID;
```

Members

Data1
Specifies the first 8 hexadecimal digits of the UUID.

Data2
Specifies the first group of 4 hexadecimal digits of the UUID.

Data3
Specifies the second group of 4 hexadecimal digits of the UUID.

Data4
Specifies an array of eight elements. The first two elements contain the third group of 4 hexadecimal digits of the UUID. The remaining six elements contain the final 12 hexadecimal digits of the UUID.

Remarks

The RPC run-time libraries use UUIDs to check for compatibility between clients and servers and to select among multiple implementations of an interface.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

+ See Also

GUID, **UUID_VECTOR**

UUID_VECTOR

The **UUID_VECTOR** structure contains a list of UUIDs.

```
typedef struct _UUID_VECTOR
{
  unsigned long Count;
  UUID *Uuid[1];
} UUID_VECTOR;
```

Members

Count
Specifies the number of UUIDs present in the array **Uuid**.

Uuid
Specifies an array of pointers to UUIDs that contains **Count** elements.

Remarks

The UUID vector contains a count member containing the total number of UUIDs in the vector, followed by an array of pointers to UUIDs.

An application constructs a UUID vector to contain object UUIDs to be exported or unexported from the name service.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

+ See Also

RpcEpRegister, **RpcEpRegisterNoReplace**, **RpcEpUnregister**, **RpcNsBindingExport**, **RpcNsBindingUnexport**

RPC Enumerated Types

This section details the enumerated types that are part of the RPC run-time library:

- RPC_ASYNC_EVENT
- RPC_NOTIFICATION_TYPES

RPC_ASYNC_EVENT

The **RPC_ASYNC_EVENT** enumerated type describes the asynchronous notification events that an RPC application can receive.

```
typedef enum _RPC_ASYNC_EVENT
{
  RpcCallComplete,
  RpcSendComplete,
  RpcReceiveComplete
} RPC_ASYNC_EVENT;
```

Members

RpcCallComplete
The remote procedure call has completely executed.

RpcSendComplete
The RPC run-time library finished transmitting data. Only applications using DCE pipes will receive this notification.

RpcReceiveComplete
The RPC run-time library finished receiving data. Only applications using DCE pipes will receive this notification.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcasync.h.

RPC_NOTIFICATION_TYPES

The **RPC_NOTIFICATION_TYPES** enumerated type contains values that specify the method of asynchronous notification that a client program will use.

```
typedef enum _RPC_NOTIFICATION_TYPES
{
  RpcNotificationTypeNone,
```

(continued)

(continued)

```
  RpcNotificationTypeEvent,
  RpcNotificationTypeApc,
  RpcNotificationTypeIoc,
  RpcNotificationTypeHwnd,
  RpcNotificationTypeCallback
} RPC_NOTIFICATION_TYPES;
```

Members

RpcNotificationTypeNone
The client does not require notification of the completion of an asynchronous remote procedure call.

RpcNotificationTypeEvent
Notify the client program by triggering an event object. See *Event Objects*.

RpcNotificationTypeApc
Use an asynchronous procedure call to notify the client that the remote procedure call is complete.

RpcNotificationTypeIoc
Send the asynchronous RPC notification to the client through an I/O completion port.

RpcNotificationTypeHwnd
Post a notification message to the specified window handle.

RpcNotificationTypeCallback
Invoke a callback function provided by the client program.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcasync.h.

+ See Also

Making the Asynchronous Call

Other RPC Types

This section contains information about the following RPC types that are neither structures nor enumerated types.

- PROTSEQ
- RPC_AUTH_IDENTITY_HANDLE
- RPC_AUTHZ_HANDLE
- RPC_BINDING_HANDLE
- RPC_EP_INQ_HANDLE
- RPC_IF_HANDLE
- RPC_MGR_EPV
- RPC_NS_HANDLE
- RPC_STATUS
- String Binding
- String UUID

PROTSEQ

The PROTSEQ protocol sequence string is a character string that represents a valid combination of an RPC protocol (such as **ncacn**), a transport protocol (such as TCP), and a network protocol (such as IP).

```
unsigned char * Protseq[1];
```

Members

Protseq

Points to a character string identifying the network protocol used to communicate between client and server. For a list of protocols that Microsoft RPC supports, see *Protocol Sequence Constants*.

Remarks

A server application can use a particular protocol sequence only when the RPC run-time library and operating system support that protocol. A server chooses to accept remote procedure calls over some or all of the supported protocol sequences.

Several server routines allow server applications to register protocol sequences with the run-time library. Microsoft RPC functions that require a protocol-sequence parameter use the data type **UNSIGNED CHAR**.

A client can use the protocol-sequence strings to construct a string binding by calling **RpcStringBindingCompose**.

Note The **ncalrpc** protocol sequence is supported only for 32-bit Windows-based applications.

The **ncacn_dnet_nsp** protocol sequence is supported only for MS-DOS, and 16-bit Windows-based client applications. This release of Microsoft RPC does not include support for the **ncacn_dnet_nsp** protocol sequence with 32-bit client or server applications.

16-bit Windows client applications that use the **ncacn_spx** or **ncadg_ipx** protocol sequences require that the file Nwipxspx.dll be installed in order to run under the Windows NT/Windows 2000 Windows16 on Windows32 (WOW) subsystem. Contact Novell to obtain this file.

The **ncadg_mq** protocol sequence requires that the Microsoft Message Queue Server be installed on a computer visible to both client and server. For more information see the documentation *Procedures Used with Most Security Packages and Protocols* and *Procedures Used with Schannel.*

The **ncacn_http** protocol sequence requires that the Microsoft Internet Information Server be installed on a computer visible to both client and server. See the documentation for Internet Information Server in the Platform SDK for more information.

The **ncacn_vns_spp** protocol sequence requires that Banyan's Enterprise Client For Windows NT be installed. Contact Banyan for more information.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcasync.h.

See Also

RpcServerUseAllProtseqs, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**, **RpcStringBindingCompose**

RPC_AUTH_IDENTITY_HANDLE

The **RPC_AUTH_IDENTITY_HANDLE** data type declares a handle that points to a structure containing the client's authentication and authorization credentials specified for remote procedure calls.

```
typedef void * RPC_AUTH_IDENTITY_HANDLE;
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RpcBindingInqAuthInfo, **RpcBindingSetAuthInfo**

RPC_AUTHZ_HANDLE

The **RPC_AUTHZ_HANDLE** data type declares an authorization handle. This handle points to the privileges information for the client application that made the remote procedure call.

```
typedef void * RPC_AUTHZ_HANDLE;
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RpcBindingInqAuthClient

RPC_BINDING_HANDLE

The **RPC_BINDING HANDLE** data type declares a binding handle containing information that the RPC run-time library uses to access binding information.

```
typedef I_RPC_HANDLE RPC_BINDING_HANDLE;
```

Remarks

The run-time library uses binding information to establish a client-server relationship that allows the execution of remote procedure calls. Based on the context in which a binding handle is created, it is considered a server-binding handle or a client-binding handle.

A server-binding handle contains the information necessary for a client to establish a relationship with a specific server. Any number of RPC API run-time routines return a server-binding handle that can be used for making a remote procedure call.

A client-binding handle cannot be used to make a remote procedure call. The RPC run-time library creates and provides a client-binding handle to a called-server procedure (also called a server-manager routine) as the RPC_BINDING_HANDLE parameter. The client-binding handle contains information about the calling client.

The **RpcBinding*** and **RpcNsBinding*** functions return the status code RPC_S_WRONG_KIND_OF_BINDING when an application provides the incorrect binding-handle type.

An application can share a single binding handle across multiple threads of execution. The RPC run-time library manages concurrent remote procedure calls that use a single binding handle. However, the application is responsible for binding handle–concurrency control for operations that modify a binding handle. These operations include the following routines.

- **RpcBindingFree**
- **RpcBindingReset**
- **RpcBindingSetAuthInfo**
- **RpcBindingSetObject**

For example, if an application shares a binding handle across two threads of execution and resets the binding-handle endpoint in one of the threads by calling **RpcBindingReset**, the binding handle in the other thread is also reset. Similarly, freeing the binding handle in one thread by calling **RpcBindingFree** frees the binding handle in the other thread.

If you don't want concurrency, you can design an application to create a copy of a binding handle by calling **RpcBindingCopy**. In this case, an operation to the first binding handle has no effect on the second binding handle.

Note In 16-bit Windows applications, a task may only have a single RPC call outstanding at a time. Binding handles are allocated on a per-task basis and cannot be shared between tasks.

Routines requiring a binding handle as a parameter show a data type of **RPC_BINDING_HANDLE**. Binding-handle parameters are passed by value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

RPC_EP_INQ_HANDLE

The **RPC_EP_INQ_HANDLE** data type declares a handle for an inquiry context. RPC applications use it to view server address information stored in the endpoint map.

```
typedef I_RPC_HANDLE * RPC_EP_INQ_HANDLE;
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcasync.h.

See Also

RpcMgmtEpEltInqBegin, **RpcMgmtEpEltInqNext**, **RpcMgmtEpEltInqDone**

RPC_IF_HANDLE

The **RPC_IF_HANDLE** data type declares an interface handle.

```
typedef void __RPC_FAR * RPC_IF_HANDLE;
```

Remarks

The RPC run-time library uses interface handles to access the interface-specification data structure. The MIDL compiler automatically creates an interface-specification data structure from each IDL file and creates a global variable of type RPC_IF_HANDLE for the interface specification.

The MIDL compiler includes an interface handle in each header file generated for the interface. Functions requiring the interface specification as a parameter show a data type of **RPC_IF_HANDLE**.The form of each interface handle name is as follows:

- *if-name*_**ClientIfHandle** (for the client)
- *if-name*_**ServerIfHandle** (for the server)

if-name

Specifies the interface identifier in the IDL file.

For example:

```
hello_ClientIfHandle
hello_ServerIfHandle
```

Note The maximum length of the interface handle name is 31 characters.

Because the **_ClientIfHandle** and **_ServerIfHandle** parts of the names require 15 characters, the *if-name* element can be no more than 16 characters long.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

RPC_MGR_EPV

The data type **RPC_MGR_EPV** defines a manager entry-point vector.

```
typedef void RPC_MGR_EPV;
typedef _if-name_SERVER_EPV {
  return-type (* Functionname) (param-list);
  ...    // one entry for each function in IDL file
} if-name_SERVER_EPV;
```

Members

if-name
Specifies the name of the interface indicated in the IDL file.

return-type
Specifies the type returned by the function **Functionname** indicated in the IDL file.

Functionname
Specifies the name of the function indicated in the IDL file.

param-list
Specifies the arguments indicated for the function **Functionname** in the IDL file.

Remarks

The manager entry-point vector (EPV) is an array of function pointers. The array contains pointers to the implementations of the functions specified in the IDL file. The number of elements in the array is set to the number of functions specified in the IDL file. An application can also have multiple EPVs, representing multiple implementations of the functions specified in the interface.

The MIDL compiler generates a default **EPV** data type named *if-name*_**SERVER_EPV**, where *if-name* specifies the interface identifier in the IDL file. The MIDL compiler initializes this default **EPV** to contain function pointers for each of the procedures specified in the IDL file.

When the server offers multiple implementations of the same interface, the server application must declare and initialize one variable of type *if-name*_**SERVER_EPV** for each implementation of the interface. Each **EPV** must contain one entry point (function pointer) for each procedure defined in the IDL file.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RpcServerRegisterIfEx

RPC_NS_HANDLE

The data type **RPC_NS_HANDLE** defines a name-service handle.

```
typedef void * RPC_NS_HANDLE;
```

Remarks

A name-service handle is an opaque variable containing information the RPC run-time library uses to return the following RPC data from the name-service database:

- Server-binding handles
- UUIDs of resources offered by server profile members
- Group members

The scope of a name-service handle is from a **Begin** routine through the corresponding **Done** routine.

Applications are responsible for concurrency control of name-service handles across threads.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.

See Also

RpcNsBindingImportBegin, **RpcNsBindingImportDone**, **RpcNsBindingImportNext**, **RpcNsBindingLookupBegin**, **RpcNsBindingLookupDone**, **RpcNsBindingLookupNext**

RPC_STATUS

The data type **RPC_STATUS** represents a platform-specific status code type.

```
typedef long RPC_STATUS;  // Win32 definition
typedef unsigned short RPC_STATUS;  // MS-DOS, Win16 definition
```

Remarks

The **RPC_STATUS** type is returned by most RPC functions and is part of the **RPC_OBJECT_INQ_FN** function type definition.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RPC_OBJECT_INQ_FN

String Binding

The string binding is an unsigned character string composed of strings that represent the binding object UUID, the RPC protocol sequence, the network address, and the endpoint and endpoint options.

```
ObjectUUID@ProtocolSequence:NetworkAddress[Endpoint,Option]
```

Members

ObjectUUID
Specifies the UUID of the object operated on by the remote procedure call. At the server, the RPC run-time library maps the object type to a manager entry-point vector (an array of function pointers) to invoke the correct manager routine. For a discussion of how to map object UUIDs to manager EPVs, see ***RpcServerRegisterIfEx***.

Protocol Sequence
Specifies a character string that represents a valid combination of an RPC protocol (such as **ncacn**), a transport protocol (such as TCP), and a network protocol (such as IP). Microsoft RPC supports the following protocols specified in Protocol Sequence Constants.

NetworkAddress
Specifies the network address of the system to receive remote procedure calls. The format and content of the network address depend on the specified protocol sequence as follows.

Protocol sequence	Network address	Examples
ncacn_nb_tcp	Windows NT/Windows 2000 computer name	myserver
ncacn_nb_ipx	Windows NT/Windows 2000 computer name	myserver
ncacn_nb_nb	Windows NT/Windows 2000 or Windows 95 computer name	myserver
ncacn_ip_tcp	Four-octet internet address, or host name	128.10.2.30 anynode.microsoft.com
ncacn_np	Windows NT/Windows 2000 server name (leading double backslashes are optional)	myserver \\myotherserver
ncacn_spx	IPX internet address, or Windows NT/Windows 2000 server name	~0000000108002B30612C myserver
ncacn_dnet_nsp	Area and node syntax	4.120

Protocol sequence	Network address	Examples
ncacn_at_dsp	Windows NT/Windows 2000 computer name, optionally followed by @ and the AppleTalk zone name. Defaults to @*, the client's zone, if no zone provided	servername@zonename servername
ncacn_vns_spp	StreetTalk server name of the form item@group@organization	printserver@sdkdocs@ microsoft
ncadg_mq	Windows NT/Windows 2000 server name	myserver
ncacn_http	Internet address (either four-octet or friendly name, or local Windows NT/Windows 2000 server name	128.10.2.30 somesvr@anywhere.com mylocalsvr
ncadg_ip_udp	Four-octet internet address, or host name	128.10.2.30 anynode.microsoft.com
ncadg_ipx	IPX internet address, or Windows NT/Windows 2000 server name	~0000000108002B30612C myserver
ncalrpc	Machine name	thismachine

The network-address field is optional. When you do not specify a network address, the string binding refers to your local host. It is possible to specify the name of the local computer when you use the **ncalrpc** protocol sequence, however doing so is completely unnecessary.

Endpoint

Specifies the endpoint, or address, of the process to receive remote procedure calls. An endpoint can be preceded by the keyword **endpoint=** specifying the endpoint is optional if the server has registered its bindings with the endpoint mapper. See *RpcEpRegister*.

The format and content of an endpoint depend on the specified protocol sequence as shown in the following Endpoint/Option Table.

Option

Specifies protocol-specific options. The option field is not required. Each option is specified by a {name, value} pair that uses the syntax *option name=option value*. Options are defined for each protocol sequence as shown in the following Endpoint/Option table.

Protocol sequence	Endpoint	Examples	Option name
ncacn_nb_tcp	Integer between 1 and 254. Many values between 0 and 32 are reserved by Microsoft.	100	None
ncacn_nb_ipx	(as above)	(as above)	None
ncacn_nb_nb	(as above)	(as above)	None
ncacn_ip_tcp	Internet port number.	1025	None
ncacn_np	Windows NT/Windows 2000 named pipe. Name must start with "\\pipe".	\\pipe\\pipename	Security (NT only)
ncacn_spx	Integer between 1 and 65535.	5000	None
ncacn_dnet_nsp	DECnet phase IV object number (must be preceded by the # character), or object name.	mailserver #17	None
ncacn_at_dsp	A character string, up to 22 bytes long.	myservicesendpoint	None
ncacn_vns_spp	Vines SPP port number between 250 and 511.	500	None
ncadg_mq	Integer between 1 and 65535.	5000	None
ncacn_http	Internet port number.	2215	HTTP and RPC proxy server names
ncadg_ip_udp	Internet port number.	1025	Security (32-bit only)
ncadg_ipx	Integer between 1 and 65535.	5000	Security (32-bit only)
ncalrpc	String specifying application or service name. The string cannot include any backslash characters.	my_printer	Security (NT only)

The Security option name, supported for the **ncalrpc**, ncacn_NP, NCADG_IP_UDP, and NCADG_IPX protocol sequences, takes the following option values.

Option name	Option value
Security	{identification I anonymous I impersonation} {dynamic I static} {true I false}

If the security option name is specified, one entry from each of the sets of security option values must also be supplied. The option values must be separated by a single-space character. For example, the following *Option* fields are valid.

```
Security=identification dynamic true
Security=impersonation static true
```

The security option values have the following meanings.

Security option value	Description
Anonymous	The client is anonymous to the server.
Dynamic	A pointer to the security token is maintained. Security settings represent current settings and include changes made after the endpoint was created.
False	Effective = FALSE; all Windows NT/Windows 2000 security settings, including those set to OFF, are included in the token.
Identification	The server has information about the client but cannot impersonate.
Impersonation	The server is the client on the client's behalf.
Static	Security settings associated with the endpoint represent a copy of the security information at the time the endpoint was created. The settings do not change.
True	Effective = TRUE; only Windows NT/Windows 2000 security settings set to ON are included in the token.

For more information about Microsoft Windows NT and Windows 2000 security options, see *Security*.

Remarks

White space is not allowed in string bindings except where required by the **Option** syntax. Default settings for the **NetworkAddress**, **Endpoint**, and **Option** fields vary according to the value of the **ProtocolSequence** member.

For all string-binding fields, a single backslash character (\) is interpreted as an escape character. To specify a single literal backslash character, you must supply two backslash characters (\\).

A string binding contains the character representation of a binding handle and occassionally portions of a binding handle. String bindings are convenient for representing portions of a binding handle, but they can't be used for making remote procedure calls. They must first be converted to a binding handle by calling **RpcBindingFromStringBinding**.

Additionally, a string binding does not contain all of the information from a binding handle. For example, the authentication information, if any, associated with a binding handle is not translated into the string binding returned by calling the **RpcBindingToStringBinding**.

During the development of a distributed application, servers can communicate their binding information to clients using string bindings to establish a client-server relationship without using the endpoint-map database or name-service database. To establish such a relationship, use the function **RpcBindingToStringBinding** to convert one or more binding handles from a binding-handle vector to a string binding, and provide the string binding to the client.

Examples

The following are examples of valid string bindings. In these examples, *obj-uuid* is used for convenience to represent a valid UUID in string form. Instead of showing the UUID 308FB580-1EB2-11CA-923B-08002B1075A7, the examples show *obj-uuid*.

```
obj-uuid@ncadg_mq:mymqserver
obj-uuid@ncacn_http:major7.somewhere.com[2225]
obj_uuid@ncacn_http:major7.somewhere.com[,HttpProxy=proxysvr:80,
    RpcProxy=websvr1.somewhere.com:80]
obj-uuid@ncacn_ip_tcp:16.20.16.27[2001]
obj-uuid@ncacn_ip_tcp:16.20.16.27[endpoint=2001]
obj-uuid@ncacn_nb_nb:
obj-uuid@ncacn_nb_nb:[100]
obj-uuid@ncacn_np:
obj-uuid@ncacn_np:[\\pipe\\p3,Security=impersonation static true]
obj-uuid@ncacn_np:\\\\marketing[\\pipe\\p2\\p3\\p4]
obj-uuid@ncacn_np:\\\\marketing[endpoint=\\pipe\\p2\\p3\\p4]
obj-uuid@ncacn_np:\\\\sales
obj-uuid@ncacn_np:\\\\sales[\\pipe\\p1,Security=identification dynamic true]
obj-uuid@ncalrpc:
obj-uuid@ncalrpc:[object1_name_demonstrating_that_these_can_be_lengthy]
obj-uuid@ncalrpc:[object2_name,Security=anonymous static true]
obj-uuid@ncacn_vns_spp:server@group@org[500]
obj-uuid@ncacn_dnet_nsp:took[elf_server]
obj-uuid@ncacn_dnet_nsp:took[endpoint=elf_server]
obj-uuid@ncadg_ip_udp:128.10.2.30
obj-uuid@ncadg_ip_udp:maryos.microsoft.com[1025]
obj-uuid@ncadg_ipx: ~0000000108002B30612C[5000]
obj-uuid@ncadg_ipx:printserver
obj-uuid@ncacn_spx:annaw[4390]
obj-uuid@ncacn_spx:~0000000108002B30612C
```

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcasync.h.

+ See Also

RpcBindingFromStringBinding, **RpcBindingToStringBinding**, **RpcEpRegister**

String UUID

A string UUID contains the character-array representation of a UUID. A string UUID is composed of multiple fields of hexadecimal characters. Each member has a fixed length, and fields are separated by the hyphen character. For example:

```
989C6E5C-2CC1-11CA-A044-08002B1BB4F5
```

When providing a string UUID as an input parameter to an RPC run-time function, enter the alphabetic hexadecimal characters as either uppercase or lowercase characters. However, when a run-time function returns a string UUID, the hexadecimal characters are always lowercase.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcasync.h.

+ See Also

UUID

RPC Constants

This section describes the following constants used by Microsoft RPC:

- Authentication-Level Constants
- Authentication-Service Constants
- Authorization-Service Constants
- Binding Time-out Constants
- Binding Option Constants

- DCE_C_ERROR_STRING_LEN
- Interface Registration Flags
- Protection Level Constants

Authentication-Level Constants

The authentication-level constants represent the authentication levels passed to the **RpcBindingInqAuthInfo** and **RpcBindingSetAuthInfo** run-time functions through their *AuthnLevel* parameters.

These levels are listed in order of increasing authentication. Each new level adds to the authentication provided by the previous level. If the RPC run-time library does not support the specified level, it automatically upgrades to the next higher supported level.

Constant	Description
RPC_C_AUTHN_LEVEL_DEFAULT	Uses the default authentication level for the specified authentication service.
RPC_C_AUTHN_LEVEL_NONE	Performs no authentication.
RPC_C_AUTHN_LEVEL_CONNECT	Authenticates only when the client establishes a relationship with a server.
RPC_C_AUTHN_LEVEL_CALL	Authenticates only at the beginning of each remote procedure call when the server receives the request. Does not apply to remote procedure calls made using the connection-based protocol sequences (those that start with the prefix "ncacn"). If the protocol sequence in a binding handle is a connection-based protocol sequence and you specify this level, this routine instead uses the RPC_C_AUTHN_LEVEL_PKT constant.
RPC_C_AUTHN_LEVEL_PKT	Authenticates only that all data received is from the expected client. Does not validate the data itself.
RPC_C_AUTHN_LEVEL_PKT_INTEGRITY	Authenticates and verifies that none of the data transferred between client and server has been modified.
RPC_C_AUTHN_LEVEL_PKT_PRIVACY	Authenticates all previous levels and encrypts the argument value of each remote procedure call.

Note RPC_C_AUTHN_LEVEL_CALL, RPC_C_AUTHN_LEVEL_PKT, RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, and RPC_C_AUTHN_LEVEL_PKT_PRIVACY are only supported for clients communicating with a Windows NT/Windows 2000 server. A Windows 95 server can only accept incoming calls at the RPC_C_AUTHN_LEVEL_CONNECT level.

See Also

RpcBindingInqAuthInfo, **RpcBindingSetAuthInfo**

Authentication-Service Constants

The authentication service constants represent the authentication services passed to the **RpcBindingInqAuthInfo** and **RpcBindingSetAuthInfo** run-time functions through their *AuthnSvc* parameters.

The following constants are valid values for the *AuthnSvc* parameter.

Constant	Value	Service
RPC_C_AUTHN_DCE_PRIVATE	1	Distributed Computing Environment (DCE) private key authentication.
RPC_C_AUTHN_DCE_PUBLIC	2	DCE public key authentication (reserved for future use).
RPC_C_AUTHN_DEC_PUBLIC	4	DEC public key authentication (reserved for future use).
RPC_C_AUTHN_DEFAULT	0xffffffff	Default authentication service.
RPC_C_AUTHN_DPA	18	Distributed Password Authentication.
RPC_C_AUTHN_GSS_KERBEROS	16	Uses the Kerberos protocol SSP.
RPC_C_AUTHN_GSS_NEGOTIATE	9	Negotiate between the use of the NTLM and Kerberos protocol SSP.
RPC_C_AUTHN_GSS_SCHANNEL	14	Use the SCHANNEL SSP. This SSP supports SSL, private communication technology (PCT), and transport level security (TLS).
RPC_C_AUTHN_MQ	100	This SSP provides an SSPI-compatible wrapper for the Microsoft Message Queue (MSMQ) transport-level protocol.

(continued)

(continued)

Constant	Value	Service
RPC_C_AUTHN_MSN	17	Authentication protocol SSP used for the Microsoft Network (MSN).
RPC_C_AUTHN_NONE	0	No authentication.
RPC_C_AUTHN_WINNT	10	NT LAN Manager Security Support Provider (NTLM SSP).

Remarks

Specify RPC_C_AUTHN_NONE to turn off authentication for remote procedure calls made over a binding handle.

When you specify RPC_C_AUTHN_DEFAULT, the RPC run-time library uses the RPC_C_AUTHN_DCE_PRIVATE authentication service for remote procedure calls made using the binding handle.

See Also

RpcBindingInqAuthInfo, **RpcBindingSetAuthInfo**

Authorization-Service Constants

The authorization service constants represent the authorization services passed to the **RpcBindingInqAuthInfo** and **RpcBindingSetAuthInfo** run-time functions through their *AuthzSvc* parameters.

The following constants are valid values for the *AuthzSvc* parameter.

Constant	Value	Service
RPC_C_AUTHZ_NONE	0	Server performs no authorization.
RPC_C_AUTHZ_NAME	1	Server performs authorization based on the client's principal name.
RPC_C_AUTHZ_DCE	2	Server performs authorization checking using the client's DCE privilege attribute certificate (PAC) information, which is sent to the server with each remote procedure call made using the binding handle. Generally, access is checked against DCE access control lists (ACLs).
RPC_C_AUTHZ_DEFAULT	0xffffffff	Server uses the default authorization service for the current SSP.

See Also

RpcBindingInqAuthInfo, **RpcBindingSetAuthInfo**

Binding Time-out Constants

The RPC library uses the binding time-out constants to specify the relative amount of time that should be spent to establish a binding to the server before giving up. The following list contains the valid time-out values.

Constant	Value	Description
RPC_C_BINDING_INFINITE_TIMEOUT	10	Keeps trying to establish communications forever.
RPC_C_BINDING_MIN_TIMEOUT	0	Tries the minimum amount of time for the network protocol being used. This value favors response time over correctness in determining whether the server is running.
RPC_C_BINDING_DEFAULT_TIMEOUT	5	Tries an average amount of time for the network protocol being used. This value gives correctness in determining whether a server is running and gives response time equal weight. This is the default value.
RPC_C_BINDING_MAX_TIMEOUT	9	Tries the longest amount of time for the network protocol being used. This value favors correctness in determining whether a server is running over response time.

Note The values in the preceding table are not in seconds. These values represent a relative amount of time on a scale of zero to 10.

Binding Option Constants

Applications set the binding option constants to control how the RPC run-time library processes remote procedure calls. The following table lists each binding property, and the relevant constant values for the binding properties.

Option	Option value	Description
RPC_C_OPT_MQ_DELIVERY	RPC_C_MQ_EXPRESS (The default value.)	Remote calls to the server remain in active memory as they pass through the network.
	RPC_C_MQ_RECOVERABLE	Remote calls to the server are stored on disk until they are delivered and executed. This guarantees delivery even if the server goes down.
RPC_C_OPT_MQ_PRIORITY	0–7 (The default is 3.)	Calls with a higher priority number have precedence over calls with a lower priority number.
RPC_C_OPT_MQ_JOURNAL	RPC_C_MQ_JOURNAL_NONE (The default value.)	Calls are never logged.
	RPC_C_MQ_JOURNAL_DEADLETTER	Calls are logged only when the message cannot be delivered. This can occur when the time specified in RPC_C_MQ_TIME_TO_REACH_QUEUE or RPC_C_MQ_TIME_TO_BE_RECEIVED expires before the message is delivered to the server, or the target queue (of the server) no longer exists.
	RPC_MQ_JOURNAL_ALWAYS	Calls are always logged at the originating computer.
RPC_C_OPT_MQ_ACKNOWLEDGE	FALSE (The default value.)	Control returns to the client process immediately after the MSMQ transport receives the call.

Option	Option value	Description
	TRUE	The remote call does not return control to the client until it receives an acknowledgment that the call has reached the server's receive queue.
RPC_C_OPT_MQ_TIME_TO_REACH_QUEUE	Time in seconds. The default value is INFINITE.	The amount of time allowed for a call to reach the Queue Manager on the target system. If the call times out before reaching the target system, it is discarded.
RPC_C_OPT_MQ_TIME_TO_BE_RECEIVED	Time in seconds. The default value is INFINITE.	The amount of time allowed for a call to reach the server process–receive queue on the target system. If the call times out before the RPC server application receives it, it is discarded.
RPC_C_OPT_MQ_AUTHN_SERVICE	N/A	For RPC internal usage only. Do not use.
RPC_C_OPT_MQ_AUTHN_LEVEL	N/A	For RPC internal usage only. Do not use.
RPC_C_OPT_BINDING_NONCAUSAL	FALSE	Default. Causal call ordering. RPC calls are executed in strict order of submission. See *Remarks*.
	TRUE	Noncausal call ordering. RPC calls are executed independently. See *Remarks*.
RPC_C_OPT_MAX_OPTIONS	Number of array items.	Not needed for application programs. Used internally by Microsoft.

(continued)

(continued)

Option	Option value	Description
RPC_C_DONT_FAIL	FALSE	Not needed for application programs. Used internally by Microsoft.
	TRUE	Not needed for application programs. Used internally by Microsoft.

Remarks

By default, the RPC run-time library executes the calls on a given binding handle from each thread of an application in strict order of submission. This does not guarantee that calls from different threads on the same binding handle are serialized. Multithreaded applications must serialize their RPC calls. If this behavior is too restrictive, you can enable noncausal ordering. When you do, the RPC run-time library executes calls independently. It imposes no ordering on their submission.

One example of an application that might find noncausal ordering useful is a multithreaded program whose threads make calls on the same binding handle. Similarly, a program that uses multiple asynchronous calls on a binding handle will find noncausal ordering a convenient option. Another example might be an Internet proxy program that uses a single thread to handle requests for several clients. In each of these cases, it would be extremely restrictive to try to serialize the remote procedure calls.

See Also

RpcBindingSetOption, **RpcBindingInqOption**

DCE_C_ERROR_STRING_LEN

The constant **DCE_C_ERROR_STRING_LEN** defines the maximum number of characters that an error message string contains.

Interface Registration Flags

The following constants are used in the *Flags* parameter of the **RpcServerRegisterIf2** and **RpcServerRegisterIfEx** functions.

Value	Meaning
0	Standard interface semantics.
RPC_IF_AUTOLISTEN	This is an **auto-listen** interface. The run time begins listening for calls as soon as the interface is registered, and stops listening when the interface is unregistered.
RPC_IF_OLE	Reserved for OLE. Do not use this flag.
RPC_IF_ALLOW_UNKNOWN_AUTHORITY	Accepts certificates from top-level authorities that are not in the server's list of trusted Certificate Authorities.
RPC_IF_ALLOW_SECURE_ONLY	Limits connections to clients that use an authorization level higher than RPC_C_AUTHN_LEVEL_NONE

Protection Level Constants

The OSF/DCE RPC specification has renamed the authentication-level constants. They are now called protection-level constants. Microsoft RPC supports both the old and the new names; these are shown in the following table.

Constant	Value	Description
RPC_C_PROTECT_ LEVEL_DEFAULT	RPC_C_AUTHN_ LEVEL_DEFAULT	Uses the default authentication level for the specified authentication service.
RPC_C_PROTECT_ LEVEL_NONE	RPC_C_AUTHN_ LEVEL_NONE	Performs no authentication.
RPC_C_PROTECT_ LEVEL_CONNEC	RPC_C_AUTHN_ LEVEL_CONNECT	Authenticates only when the client establishes a relationship with a server.

(continued)

(continued)

Constant	Value	Description
RPC_C_PROTECT_ LEVEL_CALL	RPC_C_AUTHN_ LEVEL_CALL	Authenticates only at the beginning of each remote procedure call when the server receives the request. Does not apply to remote procedure calls made using the connection-based protocol sequences (those that start with the prefix "ncacn"). If the protocol sequence in a binding handle is a connection-based protocol sequence and you specify this level, this routine instead uses the RPC_C_AUTHN_LEVEL_PKT constant.
RPC_C_PROTECT_ LEVEL_PKT	RPC_C_AUTHN_ LEVEL_PKT	Authenticates only that all data received is from the expected client. Does not validate the data itself.
RPC_C_PROTECT_ LEVEL_PKT_INTEGRITY	RPC_C_AUTHN_ LEVEL_PKT_ INTEGRITY	Authenticates and verifies that none of the data transferred between client and server has been modified.
RPC_C_PROTECT_ LEVEL_PKT_PRIVACY	RPC_C_AUTHN_ LEVEL_PKT_ PRIVACY	Authenticates all previous levels and encrypts the argument value of each remote procedure call.

See Also

Authentication-Level Constants

Protocol Sequence Constants

Microsoft RPC supports the following protocol sequences.

Protocol sequence	Description	Supporting platforms
ncacn_nb_tcp	Connection-oriented NetBIOS over Transmission Control Protocol (TCP)	Client only: MS-DOS®, Windows® 3.*x* client and Server: Windows® NT
ncacn_nb_ipx	Connection-oriented NetBIOS over Internet Packet Exchange (IPX)	Client only: MS-DOS, Windows 3.*x* Client and Server: Windows NT/Windows 2000

Protocol sequence	Description	Supporting platforms
ncacn_nb_nb	Connection-oriented NetBIOS Enhanced User Interface (NetBEUI)	Client only: MS-DOS, Windows 3.*x* Client and Server: Windows NT/Windows 2000, Windows® 95
ncacn_ip_tcp	Connection-oriented Transmission Control Protocol/Internet Protocol (TCP/IP)	Client only: MS-DOS, Windows 3.*x*, and Apple Macintosh Client and Server: Windows 95 and Windows NT/Windows 2000
ncacn_np	Connection-oriented named pipes	Client only: MS-DOS, Windows 3.*x*, Windows 95 Client and Server: Windows NT/Windows 2000
ncacn_spx	Connection-oriented Sequenced Packet Exchange (SPX)	Client only: MS-DOS, Windows 3.*x* Client and Server: Windows NT/Windows 2000, Windows 95
ncacn_dnet_nsp	Connection-oriented DECnet transport	Client only: MS-DOS, Windows 3.*x*
ncacn_at_dsp	Connection-oriented AppleTalk DSP	Client: Apple Macintosh Server: Windows NT/Windows 2000
ncacn_vns_spp	Connection-oriented Vines scalable parallel processing (SPP) transport	Client only: MS-DOS, Windows 3.*x* Client and Server: Windows NT/Windows 2000
ncadg_ip_udp	Datagram (connectionless) User Datagram Protocol/Internet Protocol (UDP/IP)	Client only: MS-DOS, Windows 3.*x* Client and Server: Windows NT/Windows 2000
ncadg_ipx	Datagram (connectionless) IPX	Client only: MS-DOS, Windows 3.*x* Client and Server: Windows NT/Windows 2000
ncadg_mq	Datagram (connectionless) over the Microsoft® Message Queue Server (MSMQ)	Client only: Windows 95 (DCOM version) Client and Server: Windows NT 4.0, with Service Pack 3, and later; and Windows 2000

(continued)

(continued)

Protocol sequence	Description	Supporting platforms
ncacn_http	Connection-oriented TCP/IP using Microsoft Internet Information Server as HTTP proxy	Client only: Windows 95 (DCOM version) Client and Server: Windows NT 5.0 and later; and Windows 2000
ncalrpc	Local procedure call	Client and Server: Windows NT/Windows 2000 and Windows 95

Note The **ncalrpc** transport supports **RPC_C_AUTHN_WINNT** authentication only. For more information, see *Security* and *Authentication-Service Constants*.

Microsoft RPC supports **RpcBindingCopy** only in client applications, not in server applications.

RPC Return Values

When functions return RPC error-status values, the errors cannot be combined. This table lists the values that can be returned by all RPC functions. See the individual method descriptions for lists of the values each can return.

Manifest	Description
EPT_S_CANT_CREATE	The endpoint-map database cannot be created.
EPT_S_CANT_PERFORM_OP	The operation cannot be performed.
EPT_S_INVALID_ENTRY	The entry is invalid.
EPT_S_NOT_REGISTERED	There are no more endpoints available from the endpoint-map database.
RPC_S_ACCESS_DENIED	The user does not have sufficient privilege to complete the operation.
RPC_S_ADDRESS_ERROR	An addressing error has occurred on the server.
RPC_S_ALREADY_LISTENING	The server is already listening.
RPC_S_ALREADY_REGISTERED	The object UUID has already been registered.
RPC_S_ASYNC_CALL_PENDING	The asynchronous remote procedure call has not yet completed.

Manifest	Description
RPC_S_BINDING_HAS_NO_AUTH	The binding does not contain any authentication information.
RPC_S_BINDING_INCOMPLETE	The binding handle is a required parameter.
RPC_S_BUFFER_TOO_SMALL	The buffer used to transmit data is too small.
RPC_S_CALL_CANCELLED	The remote procedure call exceeded the cancel time out and was cancelled.
RPC_S_CALL_FAILED	The remote procedure call failed.
RPC_S_CALL_FAILED_DNE	The remote procedure call failed and did not execute.
RPC_S_CALL_IN_PROGRESS	A remote procedure call is already in progress for this thread (Windows 3.x only).
RPC_S_CANNOT_SUPPORT	The requested operation is not supported.
RPC_S_CANT_CREATE_ENDPOINT	The endpoint cannot be created.
RPC_S_COMM_FAILURE	Unable to communicate with the server.
RPC_S_DUPLICATE_ENDPOINT	The endpoint is a duplicate.
RPC_S_ENTRY_ALREADY_EXISTS	The entry already exists.
RPC_S_ENTRY_NOT_FOUND	The entry is not found.
RPC_S_FP_DIV_ZERO	A floating-point operation at the server has caused a divide by zero.
RPC_S_FP_OVERFLOW	A floating-point overflow has occurred at the server.
RPC_S_FP_UNDERFLOW	A floating-point underflow has occurred at the server.
RPC_S_GROUP_MEMBER_NOT_FOUND	The group member has not been found.
RPC_S_INCOMPLETE_NAME	The entry name is incomplete.
RPC_S_INTERFACE_NOT_FOUND	The interface has not been found.
RPC_S_INTERNAL_ERROR	An internal error has occurred in a remote procedure call.
RPC_S_INVALID_ARG	The specified argument is not valid.
RPC_S_INVALID_AUTH_IDENTITY	The security context is invalid.

(continued)

(continued)

Manifest	Description
RPC_S_INVALID_BINDING	The binding handle is invalid.
RPC_S_INVALID_BOUND	The array bounds are invalid.
RPC_S_INVALID_ENDPOINT_FORMAT	The endpoint format is invalid.
RPC_S_INVALID_LEVEL	The level parameter is invalid.
RPC_S_INVALID_NAF_ID	The network-address family is invalid.
RPC_S_INVALID_NAME_SYNTAX	The name syntax is invalid.
RPC_S_INVALID_NET_ADDR	The network address is invalid.
RPC_S_INVALID_NETWORK_OPTIONS	The network options are invalid.
RPC_S_INVALID_OBJECT	The object is invalid.
RPC_S_INVALID_RPC_PROTSEQ	The RPC protocol sequence is invalid.
RPC_S_INVALID_SECURITY_DESC	The security descriptor is not in the valid format.
RPC_S_INVALID_STRING_BINDING	The string binding is invalid.
RPC_S_INVALID_STRING_UUID	The string UUID is invalid.
RPC_S_INVALID_TAG	The discriminant value does not match any of the case values. There is no default case.
RPC_S_INVALID_TIMEOUT	The time-out value is invalid.
RPC_S_INVALID_VERS_OPTION	The version option is invalid.
RPC_S_MAX_CALLS_TOO_SMALL	The maximum number of calls is too small.
RPC_S_NAME_SERVICE_UNAVAILABLE	The name service is unavailable.
RPC_S_NO_BINDINGS	There are no bindings.
RPC_S_NO_CALL_ACTIVE	There is no remote procedure call active in this thread.
RPC_S_NO_CONTEXT_AVAILABLE	No security context is available to allow impersonation.
RPC_S_NO_ENDPOINT_FOUND	No endpoint has been found.
RPC_S_NO_ENTRY_NAME	The binding does not contain an entry name.
RPC_S_NO_ENV_SETUP	No environment variable is set up.
RPC_S_NO_INTERFACES	No interfaces are registered.
RPC_S_NO_INTERFACES_EXPORTED	No interfaces have been exported.
RPC_S_NO_MORE_BINDINGS	There are no more bindings.
RPC_S_NO_MORE_ELEMENTS	There are no more elements.

Manifest	Description
RPC_S_NO_MORE_MEMBERS	There are no more members.
RPC_S_NO_NS_PRIVILEGE	There is no privilege for a name-service operation.
RPC_S_NO_PRINC_NAME	No principal name is registered.
RPC_S_NO_PROTSEQS	There are no protocol sequences.
RPC_S_NO_PROTSEQS_REGISTERED	No protocol sequences have been registered.
RPC_S_NOT_ALL_OBJS_UNEXPORTED	Not all objects are unexported.
RPC_S_NOT_CANCELLED	The thread is not cancelled.
RPC_S_NOT_LISTENING	The server is not listening.
RPC_S_NOT_RPC_ERROR	The status code requested is not valid.
RPC_S_NOTHING_TO_EXPORT	There is nothing to export.
RPC_S_OBJECT_NOT_FOUND	The object UUID has not been found.
RPC_S_OK	The call has completed successfully.
RPC_S_OUT_OF_MEMORY	The needed memory is not available.
RPC_S_OUT_OF_RESOURCES	Not enough resources are available to complete this operation.
RPC_S_OUT_OF_THREADS	The RPC run-time library was not able to create another thread.
RPC_S_PROCNUM_OUT_OF_RANGE	The procedure number is out of range.
RPC_S_PROTOCOL_ERROR	An RPC protocol error has occurred.
RPC_S_PROTSEQ_NOT_FOUND	The RPC protocol sequence has not been found.
RPC_S_PROTSEQ_NOT_SUPPORTED	The RPC protocol sequence is not supported.
RPC_S_SEC_PKG_ERROR	There is an error with the security package.
RPC_S_SERVER_NOT_LISTENING	The server is not listening for remote procedure calls.
RPC_S_SERVER_OUT_OF_MEMORY	The server has insufficient memory to complete this operation.
RPC_S_SERVER_TOO_BUSY	The server is too busy to complete this operation.
RPC_S_SERVER_UNAVAILABLE	The server is unavailable.

(continued)

(continued)

Manifest	Description
RPC_S_STRING_TOO_LONG	The string is too long.
RPC_S_TYPE_ALREADY_REGISTERED	The type UUID has already been registered.
RPC_S_UNKNOWN_AUTHN_LEVEL	The authentication level is unknown.
RPC_S_UNKNOWN_AUTHN_SERVICE	The authentication service is unknown.
RPC_S_UNKNOWN_AUTHN_TYPE	The authentication type is unknown.
RPC_S_UNKNOWN_AUTHZ_SERVICE	The authorization service is unknown.
RPC_S_UNKNOWN_IF	The interface is unknown.
RPC_S_UNKNOWN_MGR_TYPE	The manager type is unknown.
RPC_S_UNSUPPORTED_AUTHN_LEVEL	The authentication level is not supported.
RPC_S_UNKNOWN_PRINCIPAL	The principal name is not recognized.
RPC_S_UNSUPPORTED_NAME_SYNTAX	The name syntax is not supported.
RPC_S_UNSUPPORTED_TRANS_SYN	The transfer syntax is not supported by the server.
RPC_S_UNSUPPORTED_TYPE	The type UUID is not supported.
RPC_S_UUID_LOCAL_ONLY	The UUID that is only valid for this computer has been allocated.
RPC_S_UUID_NO_ADDRESS	No network address is available for constructing a UUID.
RPC_S_WRONG_KIND_OF_BINDING	The binding handle is not the correct type.
RPC_S_ZERO_DIVIDE	The server has attempted an integer divide by zero.
RPC_X_BAD_STUB_DATA	The stub has received bad data.
RPC_X_BYTE_COUNT_TOO_SMAL	The byte count is too small.
RPC_X_ENUM_VALUE_OUT_OF_RANGE	The enumeration value is out of range.
RPC_X_ENUM_VALUE_TOO_LARGE	The enumeration constant must be less than 65535.
RPC_X_INVALID_BOUND	The specified bounds of an array are inconsistent.
RPC_X_INVALID_BUFFER	The pointer does not contain the address of a valid data buffer.

Manifest	Description
RPC_X_INVALID_PIPE_OPERATION	The requested pipe operation is not supported.
RPC_X_INVALID_TAG	The discriminant value does not match any of the case values. There is no default case.
RPC_X_NO_MEMORY	Insufficient memory is available.
RPC_X_NO_MORE_ENTRIES	The list of servers available for the [auto_handle] binding has been exhausted.
RPC_X_NULL_REF_POINTER	A null reference pointer has been passed to the stub.
RPC_X_PIPE_APP_MEMORY	Insufficient memory is available for pipe data.
RPC_X_SS_BAD_ES_VERSION	The operation for the serializing handle is not valid.
RPC_X_SS_CANNOT_GET_CALL_HANDLE	The stub is unable to get the call handle.
RPC_X_SS_CHAR_TRANS_OPEN_FAIL	The file designated by DCERPCCHARTRANS cannot be opened.
RPC_X_SS_CHAR_TRANS_SHORT_FILE	The file containing the character-translation table has fewer than 512 bytes.
RPC_X_SS_CONTEXT_DAMAGED	The context handle changed during a call. Only raised on the client side.
RPC_X_SS_CONTEXT_MISMATCH	The context handle does not match any known context handles.
RPC_X_SS_HANDLES_MISMATCH	The binding handles passed to a remote procedure call do not match.
RPC_X_SS_IN_NULL_CONTEXT	A null context handle is passed in an in parameter position.
RPC_X_SS_INVALID_BUFFER	The buffer is not valid for the operation.
RPC_X_SS_WRONG_ES_VERSION	The software version is incorrect.
RPC_X_SS_WRONG_STUB_VERSION	The stub version is incorrect.

CHAPTER 25

RPC Function Reference

This chapter documents the Remote Procedure Call (RPC) run-time functions supported in Microsoft RPC. This chapter describes each function's purpose, syntax, input parameters, and return values. It also provides additional information to help you use RPC functions in an application.

All pointers passed to RPC functions must include the **__RPC_FAR** attribute. For example, the pointer **RPC_BINDING_HANDLE** * becomes **RPC_BINDING_HANDLE __RPC_FAR** * and **char * * Ptr** becomes **char __RPC_FAR * __RPC_FAR * Ptr.**

RPC Functions

Microsoft RPC currently supports the following functions:

- **DceErrorInqText**
- **MesBufferHandleReset**
- **MesDecodeBufferHandleCreate**
- **MesDecodeIncrementalHandleCreate**
- **MesEncodeDynBufferHandleCreate**
- **MesEncodeFixedBufferHandleCreate**
- **MesEncodeIncrementalHandleCreate**
- **MesHandleFree**
- **MesIncrementalHandleReset**
- **MesInqProcEncodingId**
- **RpcAbnormalTermination**
- **RpcAsyncAbortCall**
- **RpcAsyncCancelCall**
- **RpcAsyncCompleteCall**
- **RpcAsyncGetCallStatus**
- **RpcAsyncInitializeHandle**
- **RpcAsyncRegisterInfo**
- **RpcBindingCopy**
- **RpcBindingFree**
- **RpcBindingFromStringBinding**
- **RpcBindingInqAuthClient**
- **RpcBindingInqAuthClientEx**
- **RpcBindingInqAuthInfo**
- **RpcBindingInqAuthInfoEx**
- **RpcBindingInqObject**
- **RpcBindingInqOption**
- **RpcBindingReset**
- **RpcBindingServerFromClient**
- **RpcBindingSetAuthInfo**
- **RpcBindingSetAuthInfoEx**
- **RpcBindingSetObject**
- **RpcBindingSetOption**
- **RpcBindingToStringBinding**
- **RpcBindingVectorFree**
- **RpcCancelThread**
- **RpcCancelThreadEx**
- **RpcCertGeneratePrincipalName**
- **RpcEpRegister**
- **RpcEpRegisterNoReplace**
- **RpcEpResolveBinding**
- **RpcEpUnregister**
- **RpcExceptionCode**

- **RpcIfIdVectorFree**
- **RpcIfInqId**
- **RpcImpersonateClient**
- **RpcMacSetYieldInfo**
- **RpcMgmtEnableIdleCleanup**
- **RpcMgmtEpEltInqBegin**
- **RpcMgmtEpEltInqDone**
- **RpcMgmtEpEltInqNext**
- **RpcMgmtEpUnregister**
- **RpcMgmtInqComTimeout**
- **RpcMgmtInqDefaultProtectLevel**
- **RpcMgmtInqIfIds**
- **RpcMgmtInqServerPrincName**
- **RpcMgmtInqStats**
- **RpcMgmtIsServerListening**
- **RpcMgmtSetAuthorizationFn**
- **RpcMgmtSetCancelTimeout**
- **RpcMgmtSetComTimeout**
- **RpcMgmtSetServerStackSize**
- **RpcMgmtStatsVectorFree**
- **RpcMgmtStopServerListening**
- **RpcMgmtWaitServerListen**
- **RpcNetworkInqProtseqs**
- **RpcNetworkIsProtseqValid**
- **RpcNsBindingExport**
- **RpcNsBindingExportPnP**
- **RpcNsBindingImportBegin**
- **RpcNsBindingImportDone**
- **RpcNsBindingImportNext**
- **RpcNsBindingInqEntryName**
- **RpcNsBindingLookupBegin**
- **RpcNsBindingLookupDone**
- **RpcNsBindingLookupNext**
- **RpcNsBindingSelect**
- **RpcNsBindingUnexport**
- **RpcNsBindingUnexportPnP**
- **RpcNsEntryExpandName**
- **RpcNsEntryObjectInqBegin**
- **RpcNsEntryObjectInqDone**
- **RpcNsEntryObjectInqNext**
- **RpcNsGroupDelete**
- **RpcNsGroupMbrAdd**
- **RpcNsGroupMbrInqBegin**
- **RpcNsGroupMbrInqDone**
- **RpcNsGroupMbrInqNext**
- **RpcNsGroupMbrRemove**
- **RpcNsMgmtBindingUnexport**
- **RpcNsMgmtEntryCreate**
- **RpcNsMgmtEntryDelete**
- **RpcNsMgmtEntryInqIfIds**
- **RpcNsMgmtHandleSetExpAge**
- **RpcNsMgmtInqExpAge**
- **RpcNsMgmtSetExpAge**
- **RpcNsProfileDelete**
- **RpcNsProfileEltAdd**
- **RpcNsProfileEltInqBegin**
- **RpcNsProfileEltInqDone**
- **RpcNsProfileEltInqNext**
- **RpcNsProfileEltRemove**
- **RpcObjectInqType**
- **RpcObjectSetInqFn**
- **RpcObjectSetType**
- **RpcProtseqVectorFree**
- **RpcRaiseException**
- **RpcRevertToSelf**
- **RpcRevertToSelfEx**
- **RpcServerInqBindings**
- **RpcServerInqDefaultPrincName**
- **RpcServerInqIf**
- **RpcServerListen**

- **RpcServerRegisterAuthInfo**
- **RpcServerRegisterIf**
- **RpcServerRegisterIf2**
- **RpcServerRegisterIfEx**
- **RpcServerTestCancel**
- **RpcServerUnregisterIf**
- **RpcServerUseAllProtseqs**
- **RpcServerUseAllProtseqsEx**
- **RpcServerUseAllProtseqsIf**
- **RpcServerUseAllProtseqsIfEx**
- **RpcServerUseProtseq**
- **RpcServerUseProtseqEx**
- **RpcServerUseProtseqEp**
- **RpcServerUseProtseqEpEx**
- **RpcServerUseProtseqIf**
- **RpcServerUseProtseqIfEx**
- **RpcSmAllocate**
- **RpcSmClientFree**
- **RpcSmDestroyClientContext**
- **RpcSmDisableAllocate**
- **RpcSmEnableAllocate**
- **RpcSmFree**
- **RpcSmGetThreadHandle**
- **RpcSmSetClientAllocFree**
- **RpcSmSetThreadHandle**
- **RpcSmSwapClientAllocFree**
- **RpcSsAllocate**
- **RpcSsDestroyClientContext**
- **RpcSsDisableAllocate**
- **RpcSsDontSerializeContext**
- **RpcSsEnableAllocate**
- **RpcSsFree**
- **RpcSsGetThreadHandle**
- **RpcSsSetClientAllocFree**
- **RpcSsSetThreadHandle**
- **RpcSsSwapClientAllocFree**
- **RpcStringBindingCompose**
- **RpcStringBindingParse**
- **RpcStringFree**
- **RpcTestCancel**
- **RpcWinSetYieldInfo**
- **RpcWinSetYieldTimeout**
- **UuidCompare**
- **UuidCreate**
- **UuidCreateSequential**
- **UuidCreateNil**
- **UuidEqual**
- **UuidFromString**
- **UuidHash**
- **UuidIsNil**
- **UuidToString**

DceErrorInqText

The **DceErrorInqText** function returns the message text for a status code.

This function is supported on all 32-bit Windows platforms, except Windows® CE. Note that it is supported in ANSI only on Microsoft® Windows® 95.

```
RPC_STATUS RPC_ENTRY DceErrorInqText(
  unsigned long StatusToConvert,
  unsigned char *ErrorText
);
```

Parameters

StatusToConvert
The status code to convert to a text string.

ErrorText
Returns the text corresponding to the error code.

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Unknown error code.

Return Values

This function returns RPC_S_OK if it is successful, or an error code if not. For a list of valid error codes, see *RPC Return Values*.

Remarks

The **DceErrorInqText** routine fills the string pointed to by the *ErrorText* parameter with a null-terminated character string message for a particular status code.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

MesBufferHandleReset

The **MesBufferHandleReset** function re-initializes the handle for buffer serialization.

```
RPC_STATUS RPC_ENTRY MesBufferHandleReset(
  handle_t Handle,
  unsigned long HandleStyle,
  MIDL_ES_CODE OpCode,
  char **ppBuffer,
  unsigned long BufferSize,
  unsigned long * pEncodedSize
);
```

Parameters

Handle
The handle to be initialized.

HandleStyle
Specifies the style of handle. Valid styles are **MES_FIXED_BUFFER_HANDLE** or **MES_DYNAMIC_BUFFER_HANDLE**.

OpCode
Specifies the operation. Valid codes are **MES_ENCODE** or **MES_DECODE**.

ppBuffer
For **MES_DECODE**, points to a pointer to the buffer containing the data to be decoded.

For **MES_ENCODE**, points to a pointer to the buffer for fixed buffer style, and points to a pointer to return the buffer address for dynamic buffer style of serialization.

BufferSize
The number of bytes of data to be decoded in the buffer. Note that this is used only for the fixed buffer style of serialization.

pEncodedSize
Pointer to the size of the completed encoding. Note that this is used only when the operation is **MES_ENCODE**.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument.

Remarks

The **MesBufferHandleReset** routine is used by applications to re-initialize a buffer style handle and save memory allocations.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Midles.h.
Library: Use Rpcrt4.lib.

See Also

MesEncodeFixedBufferHandleCreate, **MesHandleFree**, **MesEncodeDynBufferHandleCreate**

MesDecodeBufferHandleCreate

The **MesDecodeBufferHandleCreate** function creates a decoding handle and initializes it for a (fixed) buffer style of serialization.

```
RPC_STATUS RPC_ENTRY MesDecodeBufferHandleCreate(
  char *Buffer,
  unsigned long BufferSize,
  handle_t *pHandle
);
```

Parameters

Buffer
Pointer to the buffer containing the data to decode.

BufferSize
The number of bytes of data to decode in the buffer.

pHandle
Pointer to the address to which the handle will be written.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument.
RPC_S_OUT_OF_MEMORY	Out of memory.
RPC_X_INVALID_BUFFER	Invalid buffer.

Remarks

The **MesDecodeBufferHandleCreate** routine is used by applications to create a serialization handle and initialize the handle for the (fixed) buffer style of decoding. When using the fixed buffer style of decoding, the user supplies a single buffer containing all the encoded data. This buffer must have an address which is aligned at 8, and must be a multiple of 8 bytes in size. Further, it must be large enough to hold all of the data to decode.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Midles.h.
Library: Use Rpcrt4.lib.

+ See Also

MesEncodeFixedBufferHandleCreate, **MesHandleFree**

MesDecodeIncrementalHandleCreate

The **MesDecodeIncrementalHandleCreate** function creates a decoding handle for the incremental style of serialization.

```
RPC_STATUS RPC_ENTRY MesDecodeIncrementalHandleCreate(
  void *UserState,
  MIDL_ES_READ ReadFn,
  handle_t *pHandle
);
```

Parameters

UserState
: Pointer to the user-supplied state object that coordinates the user-supplied **Alloc**, **Write**, and **Read** functions.

ReadFn
: Pointer to the **Read** *function*.

pHandle
: Pointer to the newly created handle.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument.
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

The **MesDecodeIncrementalHandleCreate** function is used by applications to create the handle and initialize it for the incremental style of decoding. When using the incremental style of decoding, the user supplies a **Read** function to provide a buffer containing the next part of the data to be decoded. The buffer must be aligned at 8, and the size of the buffer must be a multiple of 8. For additional information on the user-supplied **Alloc**, **Write,** and **Read** functions, see *Serialization Services*.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Midles.h.
Library: Use Rpcrt4.lib.

+ See Also

MesIncrementalHandleReset, **MesHandleFree**

MesEncodeDynBufferHandleCreate

The **MesEncodeDynBufferHandleCreate** function creates an encoding handle and then initializes it for a dynamic buffer style of serialization.

```
RPC_STATUS RPC_ENTRY MesEncodeDynBufferHandleCreate(
  char **ppBuffer,
  unsigned long *pEncodedSize,
  handle_t *pHandle
);
```

Parameters

ppBuffer
Pointer to a pointer to the stub-supplied buffer containing the encoding after serialization is complete.

pEncodedSize
A pointer to the size of the completed encoding. The size will be written to the memory location pointed to by *pEncodedSize* by the subsequent encoding operation(s).

pHandle
Pointer to the address to which the handle will be written.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument.
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

The **MesEncodeDynBufferHandleCreate** routine is used by applications to allocate the memory and initialize the handle for the dynamic buffer style of encoding. When using the dynamic buffer style of encoding, the buffer into which all the encoded data will be placed is supplied by the stub. This buffer will be allocated by the current client memory-management mechanism.

There can be performance implications when using this style for multiple encodings with the same handle. A single buffer is returned from an encoding and data is copied from intermediate buffers. The buffers are released when necessary.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Midles.h.
Library: Use Rpcrt4.lib.

+ See Also

MesBufferHandleReset, **MesHandleFree**

MesEncodeFixedBufferHandleCreate

The **MesEncodeFixedBufferHandleCreate** function creates an encoding handle and then initializes it for a fixed buffer style of serialization.

```
RPC_STATUS RPC_ENTRY MesEncodeFixedBufferHandleCreate(
  char *Buffer,
  unsigned long BufferSize,
  unsigned long *pEncodedSize,
  handle_t *pHandle
);
```

Parameters

Buffer
  Pointer to the user-supplied buffer.

BufferSize
  The size of the user-supplied buffer.

pEncodedSize
  A pointer to the size of the completed encoding. The size will be written to the pointer by the subsequent encoding operation(s).

pHandle
  Pointer to the newly created handle.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument.
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

The **MesEncodeFixedBufferHandleCreate** routine is used by applications to create and initialize the handle for the fixed buffer style of encoding. When using the fixed buffer style of encoding, the user supplies a single buffer into which all the encoded data is placed. This buffer must have an address which is aligned at 8, and must be a multiple of 8 bytes in size. Further, it must be large enough to hold an encoding of all the data, along with an encoding header for each routine being encoded.

When the handle is used for multiple encoding operations, the encoded size is cumulative.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Midles.h.
Library: Use Rpcrt4.lib.

See Also

MesDecodeBufferHandleCreate, **MesHandleFree**

MesEncodeIncrementalHandleCreate

The **MesEncodeIncrementalHandleCreate** function creates an encoding and then initializes it for the incremental style of serialization.

```
RPC_STATUS RPC_ENTRY MesEncodeIncrementalHandleCreate(
  void *UserState,
  MIDL_ES_ALLOC AllocFn,
  MIDL_ES_WRITE WriteFn,
  handle_t *pHandle
);
```

Parameters

UserState
Pointer to the user-supplied state object that coordinates the user-supplied **Alloc, Write**, and **Read** functions.

AllocFn
Pointer to the user-supplied **Alloc** function.

WriteFn
Pointer to the user-supplied **Write** function.

pHandle
Pointer to the newly created handle.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument.
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

The **MesEncodeIncrementalHandleCreate** function is used by applications to create and initialize the handle for the incremental style of encoding or decoding. When using the incremental style of encoding, the user supplies an **Alloc** function to provide an empty buffer into which the encoded data is placed, and a **Write** function to call when the buffer is full or the encoding is complete. For additional information on the user-supplied **Alloc**, **Write**, and **Read** functions, see *Serialization Services*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Midles.h.
Library: Use Rpcrt4.lib.

See Also

MesIncrementalHandleReset, **MesHandleFree**

MesHandleFree

The **MesHandleFree** function frees the memory allocated by the serialization handle.

```
RPC_STATUS RPC_ENTRY MesHandleFree(
  handle_t Handle
);
```

Parameters

Handle
The handle to be freed.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

The **MesHandleFree** routine is used by applications to free the resources of the handle after encoding or decoding data.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Midles.h.
Library: Use Rpcrt4.lib.

See Also

MesEncodeFixedBufferHandleCreate, **MesDecodeBufferHandleCreate**, **MesEncodeDynBufferHandleCreate**, **MesEncodeIncrementalHandleCreate**

MesIncrementalHandleReset

The **MesIncrementalHandleReset** function re-initializes the handle for incremental serialization.

```
RPC_STATUS RPC_ENTRY MesIncrementalHandleReset(
  handle_t Handle,
  void *UserState,
  MIDL_ES_ALLOC AllocFn,
  MIDL_ES_WRITE WriteFn,
  MIDL_ES_READ ReadFn,
  MIDL_ES_CODE OpCode
);
```

Parameters

Handle
The handle to be re-initialized.

UserState
Depending on the function, points to the user-supplied block that coordinates successive calls to the user-supplied **Alloc**, **Write**, and **Read** functions.

AllocFn
Pointer to the user-supplied **Alloc** function. This argument can be NULL if the operation does not require it, or if the handle was previously initiated with the pointer.

WriteFn
Pointer to the user-supplied **Write** function. This argument can be NULL if the operation does not require it, or if the handle was previously initiated with the pointer.

ReadFn
Pointer to the user-supplied **Read** function. This argument can be NULL if the operation does not require it, or if the handle was previously initiated with the pointer.

OpCode
Specifies the operation. Valid operations are **MES_ENCODE** or **MES_DECODE**.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument.
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

The **MesIncrementalHandleReset** routine is used by applications to re-initialize the handle for the incremental style of encoding or decoding. For additional information on the user-supplied **Alloc**, **Write**, and **Read** functions, see *Serialization Services*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Midles.h.
Library: Use Rpcrt4.lib.

See Also

MesEncodeIncrementalHandleCreate, **MesHandleFree**

MesInqProcEncodingId

The **MesInqProcEncodingId** function provides the identity of an encoding.

```
RPC_STATUS RPC_ENTRY MesInqProcEncodingId(
  handle_t Handle,
  PRPC_SYNTAX_IDENTIFIER pInterfaceId,
  unsigned long *pProcNum
);
```

Parameters

Handle
An encoding or decoding handle.

pInterfaceId
Pointer to the address in which the identity of the interface used to encode the data will be written. The *pInterfaceId* consists of the interface universally unique identifier UUID and the version number.

pProcNum
The number of the function used to encode the data.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument.
RPC_S_OUT_OF_MEMORY	Out of memory.
RPC_S_UNKNOWN_IF	Unknown interface.
RPC_S_UNSUPPORTED_TRANS_SYN	Transfer syntax not supported by server.
RPC_X_INVALID_ES_ACTION	Invalid operation for a given handle.
RPC_X_WRONG_ES_VERSION	Incompatible version of the serializing package.
RPC_X_SS_INVALID_BUFFER	Invalid buffer.

Remarks

The **MesInqProcEncodingId** function is used by applications to obtain the identity of the function used to encode the data before calling a function to decode it. When called with an encoding handle, it returns the identity of the last encoding operation. When called with a decoding handle, it returns the identity of the next decoding operation by pre-reading the buffer.

This function can only be used to check the identity of a procedure encoding; it cannot be used to check the identity for a type encoding.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Midles.h.
Library: Use Rpcrt4.lib.

NdrGetUserMarshalInfo

The **NdrGetUserMarshalInfo** function provides additional information to wire_marshal and user_marshal helper functions.

```
RPC_STATUS RPC_ENTRY NdrGetUserMarshalInfo(
  unsigned long __RPC_FAR *pFlags,
  unsigned long InformationLevel,
  NDR_USER_MARSHAL_INFO __RPC_FAR *pMarshalInfo
);
```

Parameters

pFlags
The pointer by the same name that RPC passed to the helper function.

InformationLevel
Indicates the desired level of detail to be received. Different levels imply different sets of information fields. Only level 1 is currently defined.

pMarshalInfo
The address of a memory buffer, supplied by the application, that is to receive the requested information. The buffer must be at least as large as the information structure indicated by *InformationLevel*.

Return Values

Value	Meaning
RPC_S_OK	Call was successful.
RPC_S_INVALID_ARG	At least one of the arguments was invalid.
RPC_X_INVALID_BUFFER	Current marshaling buffer is invalid.

Remarks

The **NdrGetUserMarshalInfo** function is called by the wire_marshal or user_marshal helper functions (provided by the application) <*type*>**_UserSize**, <*type*>**_UserMarshal**, <*type*>**_UserUnmarshal**, and <*type*>**_UserFree** to receive extra information about the state of the call. A common use for this function is to obtain the size of the marshaling buffer for the purpose of checking for end of buffer conditions. Sending incorrectly sized data is a commonly used method of breaching system security.

For a full listing of the information returned by **NdrGetUserMarshalInfo**, see **NDR_USER_MARSHAL_INFO**.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Rpcndr.h.
Library: Use Rpct4.lib.

RpcAbnormalTermination

The **RpcAbnormalTermination** function indicates whether the try block of a **RpcTryFinally** statement terminated normally.

```
BOOL RpcAbnormalTermination(VOID);
```

Parameters

This function has no parameters.

Return Values

Value	Meaning
Zero	Try block terminated normally.
Nonzero	Try block terminated abnormally, either because an exception was raised, or because of a **return**, **continue**, or **break** statement.

Remarks

The **RpcAbnormalTermination** function can only be called from within the *termination-statements* section of an **RpcFinally** termination handler.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpc.h.
Library: Use Rpcrt4.lib.

See Also

RpcFinally

RpcAsyncAbortCall

The server calls **RpcAsyncAbortCall** to abort an asynchronous call.

```
RPC_STATUS  RPC_ENTRY RpcAsyncAbortCall(
  PRPC_ASYNC_STATE pAsync,
  unsigned long ExceptionCode
);
```

Parameters

pAsync

Pointer to the **RPC_ASYNC_STATE** structure that contains asynchronous call information.

ExceptionCode
An application-specific exception code. See *RPC Return Values*.

Return Values

Value	Meaning
RPC_S_OK	Call cancellation successful.
RPC_S_INVALID_ASYNC_HANDLE	Asynchronous handle is invalid.

Remarks

The server calls **RpcAsyncAbortCall** when circumstances require it to abort an asynchronous call before completion. For example, the caller may not have the necessary permissions to make the request, or the server may be too busy to process the call. Use the *ExceptionCode* parameter to specify the reason for the abort. The run-time environment propagates the exception code to the client as a fault.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Rpcasync.h.
Library: Use Rpcrt4.lib.

See Also

Asynchronous RPC, **RPC_ASYNC_STATE**, **RpcAsyncCancelCall**, **RpcAsyncCompleteCall**, **RpcAsyncGetCallHandle**, **RpcAsyncGetCallStatus**, **RpcAsyncInitializeHandle**, **RpcAsyncRegisterInfo**, **RpcServerTestCancel**

RpcAsyncCancelCall

The client calls the **RpcAsyncCancelCall** function to cancel an asynchronous call.

```
RPC_STATUS RPC_ENTRY RpcAsyncCancelCall(
  PRPC_ASYNC_STATE pAsync,
  BOOL fAbortCall
);
```

Parameters

pAsync
Pointer to the **RPC_ASYNC_STATE** structure that contains asynchronous call information.

fAbortCall

If TRUE, the call is cancelled immediately. If FALSE, wait for the server to complete the call.

Return Values

Value	Meaning
RPC_S_OK	Cancellation request was processed.
RPC_S_INVALID_ASYNC_HANDLE	Asynchronous handle is invalid.

Remarks

There are two ways for a client to request cancellation of an asynchronous call—*abortive* and *nonabortive*. In an abortive cancel (*fAbortCall* is TRUE), the **RpcAsyncCancelCall** function sends a cancel notification to the server and client side and the asynchronous call is canceled immediately, without waiting for a response from the server. Note that in a multithreaded application, the thread that originated the call is the only thread that can cancel it.

The server checks for cancel requests from the client by calling **RpcServerTestCancel**. Depending on the state of the call at the time the cancel request was issued and how often the server checks for cancels, the call may or may not complete normally. The client application can call **RpcAsyncCompleteCall** and the return value will indicate whether the call completed, failed, or was canceled. However, the client must still wait for the original call to complete before calling **RPCAsyncCompleteCall**.

In a nonabortive cancel (*fAbortCall* is FALSE) the **RpcAsyncCancelCall** function notifies the server of the cancel and the client waits for the server to complete the call. There is no built-in time-out mechanism. If you want the call to time out, the client should first issue a nonabortive cancel using its own time-out mechanism. If the call times out, then the client can issue an abortive cancel.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Rpcasync.h.
Library: Use Rpcrt4.lib.

See Also

Asynchronous RPC, **RPC_ASYNC_STATE**, **RpcAsyncAbortCall**, **RpcAsyncCompleteCall**, **RpcAsyncGetCallHandle**, **RpcAsyncGetCallStatus**, **RpcAsyncInitializeHandle**, **RpcAsyncRegisterInfo**, **RpcServerTestCancel**

RpcAsyncCompleteCall

The client and the server call the **RpcAsyncCompleteCall** function to complete an asynchronous remote procedure call.

```
RPC_STATUS RPC_ENTRY RpcAsyncCompleteCall(
  PRPC_SYNC_STATE pAsync,
  PVOID Reply
);
```

Parameters

pAsync

Pointer to the **RPC_ASYNC_STATE** structure that contains asynchronous call information.

Reply

Pointer to a buffer containing the return value of the remote procedure call.

Return Values

In addition to the following values, **RpcAsyncCompleteCall** can also return any general RPC or application-specific error.

Value	Meaning
RPC_S_OK	The call was completed successfully.
RPC_S_INVALID_ASYNC_HANDLE	The asynchronous call handle is not valid.
RPC_S_ASYNC_CALL_PENDING	The call has not yet completed.
RPC_S_CALLED_CANCELLED	The call was cancelled.

Remarks

Completes the asynchronous RPC call. Both client and server call this function.

Client: *Reply* points to a buffer that will receive the reply. If the client calls this function before the reply has arrived, the call returns RPC_S_ASYNC_CALL_PENDING. The buffer must be valid and it must be big enough to receive the return value. If this call is successful, the **[out]** and the **[in, out]** parameters are valid.

Server: *Reply* points to a buffer that contains the return value that needs to be sent to the client. The buffer must be valid. Before a call to **RpcAsyncCompleteCall** is made, the **[out]** and **[in**, **out]** parameters must be updated. These parameters, and the asynchronous handle, should not be touched after the call to **RpcAsyncCompleteCall** returns.

Any **[out]** parameters, including **[comm_status]** and **[fault_status]** parameters, are only valid if the return value of **RpcAsyncCompleteCall** is RPC_S_OK.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Rpcasync.h.
Library: Use Rpcrt4.lib.

+ See Also

Asynchronous RPC, Error Handling, RPC_ASYNC_STATE, **RpcAsyncAbortCall**, **RpcAsyncCancelCall**, **RpcAsyncGetCallHandle**, **RpcAsyncGetCallStatus**, **RpcAsyncInitializeHandle**, **RpcAsyncRegisterInfo**, **RpcServerTestCancel**

RpcAsyncGetCallStatus

The client calls the **RpcAsyncGetCallStatus** function to determine the current status of an asynchronous remote call.

```
RPC_STATUS RPC_ENTRY RpcAsyncGetCallStatus(
  PRPC_ASYNC_STATE pAsync
);
```

Parameters

pAsync
Pointer to the **RPC_ASYNC_STATE** structure that contains asynchronous call information.

Return Values

Value	Meaning
RPC_S_OK	The call was completed successfully.
RPC_S_INVALID_ASYNC_HANDLE	The asynchronous call handle is not valid.
RPC_S_ASYNC_CALL_PENDING	The call has not yet completed.
Other error codes	The call failed. The client application must call **RpcAsyncCompleteCall** to receive the application-specific error code.

Remarks

This client-side function returns the current status of the asynchronous call. Note that if the return value is anything other than RPC_S_ASYNC_CALL_PENDING the call is complete.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Rpcasync.h.
Library: Use Rpcrt4.lib.

+ See Also

Asynchronous RPC, RPC_ASYNC_STATE, **RpcAsyncAbortCall**, **RpcAsyncCancelCall**, **RpcAsyncCompleteCall**, **RpcAsyncGetCallHandle**, **RpcAsyncInitializeHandle**, **RpcAsyncRegisterInfo**, **RpcServerTestCancel**

RpcAsyncInitializeHandle

The client calls the **RpcAsyncInitializeHandle** function to initialize the **RPC_ASYNC_STATE** structure so that it can be used to make an asynchronous call.

```
RPC_STATUS RPC_ENTRY RpcAsyncInitializeHandle(
  PRPC_ASYNC_STATE pAsync,
  unsigned int Size
);
```

Parameters

pAsync
Pointer to the **RPC_ASYNC_STATE** structure that contains asynchronous call information.

Size
The size of the **RPC_ASYNC_STATE** structure.

Return Values

Value	Meaning
RPC_S_OK	The call succeeded.
RPC_S_INVALID_ARG	The size is either too small or too large.
RPC_S_INVALID_ASYNC_HANDLE	*pAsync* points to invalid memory.

Remarks

The client creates a new **RPC_ASYNC_STATE** structure and a pointer to that structure and calls **RpcAsyncInitializeHandle** with the pointer as an input parameter. The **RpcAsyncInitializeHandle** function initializes the fields that it uses to maintain the state of an asynchronous remote call. When the call to **RpcAsyncInitializeHandle** returns successfully, the client can set the notification type and any fields related to that notification type in the **RPC_ASYNC_STATE** structure. The client application uses a pointer to this structure to make an asynchronous call.

The client should not attempt to alter the **Size**, **Signature**, **Lock**, and **StubInfo** members of the **RPC_ASYNC_STATE** structure; doing so will invalidate the handle.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Rpcasync.h.
Library: Use Rpcrt4.lib.

See Also

Asynchronous RPC, RPC_ASYNC_STATE, **RpcAsyncAbortCall**, **RpcAsyncCancelCall**, **RpcAsyncCompleteCall**, **RpcAsyncGetCallHandle**, **RpcAsyncGetCallStatus**, **RpcAsyncRegisterInfo**, **RpcServerTestCancel**

RpcAsyncRegisterInfo

The server calls the **RpcAsyncRegisterHandle** function to register an updated handle with the run-time environment.

```
RPC_STATUS RPC_ENTRY RpcAsyncRegisterInfo(
  PRPC_ASYNC_STATE pAsync
);
```

Parameters

pAsync
Pointer to the **RPC_ASYNC_STATE** structure that contains asynchronous call information.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ASYNC_HANDLE	The asynchronous handle is invalid.

Remarks

After the server manager function finishes updating asynchronous information, it calls **RpcAsyncRegisterInfo** to register the updated handle with the run-time environment.

Once registered, a server handle for an asynchronous call remains valid until the server calls the completion function **RpcAsyncCompleteCall**.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Rpcasync.h.
Library: Use Rpcrt4.lib.

See Also

Asynchronous RPC, RPC_ASYNC_STATE, **RpcAsyncAbortCall**, **RpcAsyncCancelCall**, **RpcAsyncCompleteCall**, **RpcAsyncGetCallHandle**, **RpcAsyncGetCallStatus**, **RpcAsyncInitializeHandle**, **RpcServerTestCancel**

RpcBindingCopy

The **RpcBindingCopy** function copies binding information and creates a new binding handle.

```
RPC_STATUS RPC_ENTRY RpcBindingCopy(
  RPC_BINDING_HANDLE SourceBinding,
  RPC_BINDING_HANDLE *DestinationBinding
);
```

Parameters

SourceBinding
Server binding handle whose referenced binding information is copied.

DestinationBinding
Returns a pointer to the server binding handle that refers to the copied binding information.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

The **RpcBindingCopy** function copies the server-binding information referenced by the *SourceBinding* argument. **RpcBindingCopy** uses the *DestinationBinding* argument to return a new server binding handle for the copied binding information. **RpcBindingCopy** also copies the authentication information from the *SourceBinding* argument to the *DestinationBinding* argument.

An application uses **RpcBindingCopy** when it wants to prevent a change being made to binding information by one thread from affecting the binding information used by other threads.

Once an application calls **RpcBindingCopy**, operations performed on the *SourceBinding* binding handle do not affect the binding information referenced by the *DestinationBinding* binding handle. Similarly, operations performed on the *DestinationBinding* binding handle do not affect the binding information referenced by the *SourceBinding* binding handle.

If an application wants one thread's changes to binding information to affect the binding information used by other threads, the application should share a single binding handle across the threads. In this case, the application is responsible for binding-handle concurrency control.

When an application is finished using the binding handle specified by the *DestinationBinding* argument, the application should call the **RpcBindingFree** function to release the memory used by the *DestinationBinding* binding handle and its referenced binding information.

Note Microsoft RPC supports **RpcBindingCopy** only in client applications, not in server applications.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 OSR2 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingFree

RpcBindingFree

The **RpcBindingFree** function releases binding-handle resources.

```
RPC_STATUS RPC_ENTRY RpcBindingFree(
  RPC_BINDING_HANDLE *Binding
);
```

Parameters

Binding
Pointer to the server binding to be freed.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

The **RpcBindingFree** function releases memory used by a server binding handle. Referenced binding information that was dynamically created during program execution is released as well. An application calls the **RpcBindingFree** function when it is finished using the binding handle.

Binding handles are dynamically created by calling the following functions:

- **RpcBindingCopy**
- **RpcBindingFromStringBinding**
- **RpcBindingServerFromClient**
- **RpcServerInqBindings**
- **RpcNsBindingImportNext**
- **RpcNsBindingSelect**

If the operation successfully frees the binding, the *Binding* argument returns a value of NULL.

Note Microsoft RPC supports **RpcBindingFree** only in client applications, not in server applications.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingCopy, **RpcBindingFromStringBinding**, **RpcBindingVectorFree**, **RpcNsBindingImportNext**, **RpcNsBindingLookupNext**, **RpcNsBindingSelect**, **RpcServerInqBindings**

RpcBindingFromStringBinding

The **RpcBindingFromStringBinding** function returns a binding handle from a string representation of a binding handle.

```
RPC_STATUS RPC_ENTRY RpcBindingFromStringBinding(
  unsigned char *StringBinding,
  RPC_BINDING_HANDLE *Binding
);
```

Parameters

StringBinding
Pointer to a string representation of a binding handle.

Binding
Returns a pointer to the server binding handle.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_STRING_BINDING	Invalid string binding.
RPC_S_PROTSEQ_NOT_SUPPORTED	Protocol sequence not supported on this host.
RPC_S_INVALID_RPC_PROTSEQ	Invalid protocol sequence.
RPC_S_INVALID_ENDPOINT_FORMAT	Invalid endpoint format.
RPC_S_STRING_TOO_LONG	String too long.
RPC_S_INVALID_NET_ADDR	Invalid network address.
RPC_S_INVALID_ARG	Invalid argument.
RPC_S_INVALID_NAF_ID	Invalid network address–family identifier.

Remarks

The **RpcBindingFromStringBinding** function creates a server binding handle from a string representation of a binding handle. The *StringBinding* argument does not have to contain an object UUID. In this case, the returned binding contains a nil UUID. If the provided *StringBinding* argument does not contain an endpoint field, the returned *Binding* argument is a partially-bound binding handle. If the provided *StringBinding* argument contains an endpoint field, the endpoint is considered to be a well-known endpoint. If the provided *StringBinding* argument does not contain a host address field, the returned *Binding* argument references the local host.

An application creates a string binding by calling the **RpcStringBindingCompose** function or by providing a character-string constant.

When an application is finished using the *Binding* argument, the application should call the **RpcBindingFree** function to release the memory used by the binding handle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingCopy, **RpcBindingFree**, **RpcBindingToStringBinding**, **RpcStringBindingCompose**

RpcBindingInqAuthClient

A server application calls the **RpcBindingInqAuthClient** function to obtain the principal name or privilege attributes of the authenticated client that made the remote procedure call.

```
RPC_STATUS RPC_ENTRY RpcBindingInqAuthClient(
  RPC_BINDING_HANDLE ClientBinding,
  RPC_AUTHZ_HANDLE *Privs,
  unsigned char **ServerPrincName,
  unsigned long *AuthnLevel,
  unsigned long *AuthnSvc,
  unsigned long *AuthzSvc
);
```

Parameters

ClientBinding
The client binding handle of the client that made the remote procedure call. This value can be zero. See Remarks.

Privs
Returns a pointer to a handle to the privileged information for the client application that made the remote procedure call on the *ClientBinding* binding handle.

The server application must cast the *ClientBinding* binding handle to the data type specified by the *AuthzSvc* argument. The data referenced by this argument is read-only and should not be modified by the server application. If the server wants to preserve any of the returned data, the server must copy the data into server-allocated memory.

The data that the *Privs* parameter points to comes directly from the SSP Therefore, the format of the data is specific to the SSP. For more information on SSPs, see *Security Support Providers (SSPs)*.

ServerPrincName

Returns a pointer to a pointer to the server principal name specified by the client application that made the remote procedure call on the *ClientBinding* binding handle. The content of the returned name and its syntax are defined by the authentication service in use. For the SCHANNEL SSP, the principal name is in Microsoft-standard (msstd) format. For further information on msstd format, see *Principal Names*.

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *ServerPrincName* argument. In this case, the application does not call the **RpcStringFree** function.

AuthnLevel

Returns a pointer to the level of authentication requested by the client application that made the remote procedure call on the *ClientBinding* binding handle.

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *AuthnLevel* argument.

AuthnSvc

Returns a pointer to the authentication service requested by the client application that made the remote procedure call on the *ClientBinding* binding handle. For a list of the RPC-supported authentication levels, see *Authentication-Level Constants*.

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *AuthnSvc* argument.

AuthzSvc

Returns a pointer to the authorization service requested by the client application that made the remote procedure call on the *Binding* binding handle. For a list of possible returns, see ***RpcMgmtInqDefaultProtectLevel***.

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *AuthzSvc* argument. This parameter is not used by the RPC_C_AUTHN_WINNT authentication service. The returned value will always be RPC_S_AUTHZ_NONE.

Remarks

A server application calls the **RpcBindingInqAuthClient** function to obtain the principal name or privilege attributes of the authenticated client that made the remote procedure call. In addition, **RpcBindingInqAuthClient** returns the authentication service, authentication level, and server principal name specified by the client. The server can use the returned data for authorization purposes.

The RPC run-time library allocates memory for the returned *ServerPrincName* argument. The application is responsible for calling the **RpcStringFree** function for the returned argument string.

For clients using the MIDL **auto_handle** or **implicit_handle** attributes, the server application should use zero as the value for the *ClientBinding* parameter. Using zero retrieves the authentication and authorization information from the currently executing remote procedure call.

Return Values

value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_S_BINDING_HAS_NO_AUTH	Binding has no authentication information.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

RpcBindingSetAuthInfo, **RpcStringFree**

RpcBindingInqAuthClientEx

A server application calls the **RpcBindingInqAuthClientEx** function to obtain extended information about the client program that made the remote procedure call.

```
RPC_STATUS RPC_ENTRY RpcBindingInqAuthClient(
  RPC_BINDING_HANDLE ClientBinding,
  RPC_AUTHZ_HANDLE *Privs,
  unsigned char **ServerPrincName,
  unsigned long *AuthnLevel,
  unsigned long *AuthnSvc,
  unsigned long *AuthzSvc,
  unsigned long flags
);
```

Parameters

ClientBinding
The client binding handle of the client that made the remote procedure call. This value can be zero. See Remarks.

Privs

Returns a pointer to a handle to the privileged information for the client application that made the remote procedure call on the *ClientBinding* binding handle.

The server application must cast the *ClientBinding* binding handle to the data type specified by the *AuthzSvc* argument. The data referenced by this argument is read-only and should not be modified by the server application. If the server wants to preserve any of the returned data, the server must copy the data into server-allocated memory.

The data that the *Privs* parameter points to comes directly from the SSP; therefore, the format of the data is specific to the SSP. For more information on SSPs, see *Security Support Providers (SSPs).*

ServerPrincName

Returns a pointer to a pointer to the server principal name specified by the client application that made the remote procedure call on the *ClientBinding* binding handle. The content of the returned name and its syntax are defined by the authentication service in use. For the SCHANNEL SSP, the principal name is in msstd format. For further information on msstd format, see *Principal Names.*

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *ServerPrincName* argument. In this case, the application does not call the **RpcStringFree** function.

AuthnLevel

Returns a pointer to the level of authentication requested by the client application that made the remote procedure call on the *ClientBinding* binding handle.

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *AuthnLevel* argument.

AuthnSvc

Returns a pointer to the authentication service requested by the client application that made the remote procedure call on the *ClientBinding* binding handle. For a list of the RPC-supported authentication levels, see *Authentication-Level Constants.*

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *AuthnSvc* argument.

AuthzSvc

Returns a pointer to the authorization service requested by the client application that made the remote procedure call on the *Binding* binding handle. For a list of possible returns, see ***RpcMgmtInqDefaultProtectLevel***.

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *AuthzSvc* argument. This parameter is not used by the RPC_C_AUTHN_WINNT authentication service. The returned value will always be RPC_S_AUTHZ_NONE.

flags

Controls the format of the principal name. This parameter can be set to the following value.

Value	Meaning
RPC_C_FULL_CERT_CHAIN	Passes back the principal name in fullsic format.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_S_BINDING_HAS_NO_AUTH	Binding has no authentication information.

Remarks

A server application calls the **RpcBindingInqAuthClient** function to obtain the principal name or privilege attributes of the authenticated client that made the remote procedure call. In addition, **RpcBindingInqAuthClient** returns the authentication service, authentication level, and server principal name specified by the client. The server can use the returned data for authorization purposes.

The RPC run-time library allocates memory for the returned *ServerPrincName* argument. The application is responsible for calling the **RpcStringFree** function for the returned argument string.

For clients using the MIDL **[auto_handle]** or **[implicit_handle]** attributes, the server application should use zero as the value for the *ClientBinding* parameter. Using zero retrieves the authentication and authorization information from the currently executing remote procedure call.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingInqAuthClient, **RpcBindingSetAuthInfo**, **RpcStringFree**

RpcBindingInqAuthInfo

The **RpcBindingInqAuthInfo** function returns authentication and authorization information from a binding handle.

```
RPC_STATUS RPC_ENTRY RpcBindingInqAuthInfo(
  RPC_BINDING_HANDLE Binding,
  RPC_CHAR *ServerPrincName,
  unsigned long *AuthnLevel,
  unsigned long *AuthnSvc,
  RPC_AUTH_IDENTITY_HANDLE *AuthIdentity,
  unsigned long *AuthzSvc
);
```

Parameters

Binding

The server binding handle from which authentication and authorization information is returned.

ServerPrincName

Returns a pointer to a pointer to the expected principal name of the server referenced in *Binding*. The content of the returned name and its syntax are defined by the authentication service in use.

Specify a null value to prevent **RpcBindingInqAuthInfo** from returning the *ServerPrincName* argument. In this case, the application does not call the **RpcStringFree** function.

AuthnLevel

Returns a pointer to the level of authentication used for remote procedure calls made using *Binding*. See Note.

Specify a null value to prevent the function from returning the *AuthnLevel* argument.

The level returned in the *AuthnLevel* argument may be different from the level specified when the client called the **RpcBindingSetAuthInfo** function. This discrepancy occurs when the RPC run-time library does not support the authentication level specified by the client and automatically upgrades to the next higher authentication level.

AuthnSvc

Returns a pointer to the authentication service specified for remote procedure calls made using *Binding*. See Note.

Specify a null value to prevent **RpcBindingInqAuthInfo** from returning the *AuthnSvc* argument.

AuthIdentity

Returns a pointer to a handle to the data structure that contains the client's authentication and authorization credentials specified for remote procedure calls made using *Binding*.

Specify a null value to prevent **RpcBindingInqAuthInfo** from returning the *AuthIdentity* argument.

AuthzSvc

Returns a pointer to the authorization service requested by the client application that made the remote procedure call on *Binding* See Note.

Specify a null value to prevent **RpcBindingInqAuthInfo** from returning the *AuthzSvc* argument.

Note For a list of the RPC-supported authentication services, see *Authentication-Service Constants*.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_BINDING_HAS_NO_AUTH	Binding has no authentication information.

Remarks

A client application calls the **RpcBindingInqAuthInfo** function to view the authentication and authorization information associated with a server binding handle. A similar function, **RpcBindingInqAuthInfoEx** additionally provides security quality-of-service information on the binding handle.

The RPC run-time library allocates memory for the returned *ServerPrincName* argument. The application is responsible for calling the **RpcStringFree** function for that returned argument string.

When a client application does not know a server's principal name, calling **RpcBindingInqAuthInfo** after making a remote procedure call provides the server's principal name. For example, clients that import from a group or profile may not know a server's principal name when calling the **RpcBindingSetAuthInfo** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingInqAuthClient, **RpcBindingInqAuthInfoEx**, **RpcBindingSetAuthInfo**, **RpcBindingInqOption**, **RpcStringFree**

RpcBindingInqAuthInfoEx

The **RpcBindingInqAuthInfoEx** function returns authentication, authorization, and security quality-of-service information from a binding handle. The function is supported with limited functionality (no security quality-of-service information) on Windows® NT 4.0 and with full functionality on Windows 2000, and Windows® 95.

```
RPC_STATUS RPC_ENTRY RpcBindingInqAuthInfoEx(
  RPC_BINDING_HANDLE Binding,
  RPC_CHAR PAPI *ServerPrincName,
  unsigned long PAPI *AuthnLevel,
  unsigned long PAPI *AuthnSvc,
  RPC_AUTH_IDENTITY_HANDLE PAPI *AuthIdentity,
  unsigned long PAPI *AuthzSvc,
  unsigned long RpcQosVersion,
  RPC_SECURITY_QOS *SecurityQos
);
```

Parameters

Binding

The server binding handle from which authentication and authorization information is returned.

ServerPrincName

Returns a pointer to a pointer to the expected principal name of the server referenced in *Binding*. The content of the returned name and its syntax are defined by the authentication service in use.

Specify a null value to prevent **RpcBindingInqAuthInfoEx** from returning the *ServerPrincName* argument. In this case, the application does not call the **RpcStringFree** function.

AuthnLevel

Returns a pointer to the level of authentication used for remote procedure calls made using *Binding*. For a list of the RPC-supported authentication levels, see *Authentication-Level Constants*. Specify a null value to prevent the function from returning the *AuthnLevel* argument.

The level returned in the *AuthnLevel* argument may be different from the level specified when the client called the **RpcBindingSetAuthInfoEx** function. This discrepancy happens when the RPC run-time library does not support the authentication level specified by the client and automatically upgrades to the next higher authentication level.

AuthnSvc

Returns a pointer to the authentication service specified for remote procedure calls made using *Binding*. For a list of the RPC-supported authentication services, see *Authentication-Service Constants*.

Specify a null value to prevent **RpcBindingInqAuthInfoEx** from returning the *AuthnSvc* argument.

AuthIdentity
Returns a pointer to a handle to the data structure that contains the client's authentication and authorization credentials specified for remote procedure calls made using *Binding*.

Specify a null value to prevent **RpcBindingInqAuthInfoEx** from returning the *AuthIdentity* argument.

AuthzSvc
Returns a pointer to the authorization service requested by the client application that made the remote procedure call on *Binding*. For a list of the RPC-supported authentication services, see *Authentication-Service Constants*.

Specify a null value to prevent **RpcBindingInqAuthInfoEx** from returning the *AuthzSvc* argument.

RpcQosVersion
Passes value of current version (needed for forward compatibility if extensions are made to this function). Always set this parameter to RPC_C_SECURITY_QOS_VERSION.

SecurityQos
Returns pointer to the **RPC_SECURITY_QOS** structure, which defines quality-of-service settings. Not supported in Windows NT 4.0.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_BINDING_HAS_NO_AUTH	Binding has no authentication information.

Remarks

A client application calls the **RpcBindingInqAuthInfoEx** function to view the authentication and authorization information associated with a server binding handle. This function provides the ability to inquire about the security quality of service on the binding handle. It is otherwise identical to **RpcBindingInqAuthInfo**.

The RPC run-time library allocates memory for the returned *ServerPrincName* argument. The application is responsible for calling the **RpcStringFree** function for that returned argument string.

When a client application does not know a server's principal name, calling **RpcBindingInqAuthInfoEx** after making a remote procedure call provides the server's principal name. For example, clients that import from a group or profile may not know a server's principal name when calling the **RpcBindingSetAuthInfoEx** function.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RPC_SECURITY_QOS, **RpcBindingSetAuthInfoEx**, **RpcStringFree**

RpcBindingInqObject

The **RpcBindingInqObject** function returns the object UUID from a binding handle.

```
RPC_STATUS RPC_ENTRY RpcBindingInqObject(
  RPC_BINDING_HANDLE Binding,
  UUID *ObjectUuid
);
```

Parameters

Binding
A client or server binding handle.

ObjectUuid
Returns a pointer to the object UUID found in the *Binding* argument. *ObjectUuid* is a unique identifier of an object to which a remote procedure call can be made.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle

Remarks

An application calls the **RpcBindingInqObject** function to see the object UUID associated with a client or server binding handle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingSetObject

RpcBindingInqOption

RPC client processes use **RpcBindingInqOption** to determine current values of the binding options that are specific to the **ncadg_mq** message-queuing transport.

```
RPC_STATUS RPC_ENTRY RpcBindingInqOption(
  RPC_BINDING_HANDLE hBinding,
  unsigned long Option,
  unsigned long *pOptionValue
);
```

Parameters

hBinding
The server binding about which to determine binding-option values.

Option
The binding handle property to inquire about.

pOptionValue
The memory location to place the value for the specified *Option*

Note For a list of binding options and their possible values, see *Binding Option Constants*.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_CANNOT_SUPPORT	The function is not supported for either the operating system or the transport. Note that calling **RpcBindingInqOption** on binding handles that use any protocol sequence other than **ncadg_mq** will fail and return this value.

Remarks

When you use the Microsoft Message Queue Server (MSMQ) as the transport mechanism for your RPC application, and you use asynchronous **message** calls, the client process can control certain binding-specific options that affect queue operation.

Client processes call **RpcBindingInqOption** to determine the current settings of the message options. To inquire about authentication settings, the client must call **RpcBindingInqAuthClient**.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingSetOption, **RpcBindingInqOption**, **RpcBindingSetAuthInfo**, **RpcBindingInqAuthClient**, **message**, RPC Message Queuing

RpcBindingReset

The **RpcBindingReset** function resets a binding handle so that the host is specified but the server on that host is unspecified.

```
RPC_STATUS RPC_ENTRY RpcBindingReset(
  RPC_BINDING_HANDLE Binding
);
```

Parameters

Binding
The server binding handle to reset.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

A client calls the **RpcBindingReset** function to disassociate a particular server instance from the server binding handle specified in the *Binding* argument.
The **RpcBindingReset** function dissociates a server instance by removing the endpoint portion of the server address in the binding handle. The host remains unchanged in the binding handle. The result is a partially-bound server binding handle.

RpcBindingReset does not affect the *Binding* argument's authentication information, if there is any.

If a client is willing to be serviced by any compatible server instance on the host specified in the binding handle, the client calls the **RpcBindingReset** function before making a remote procedure call using the *Binding* binding handle.

When the client makes the next remote procedure call using the reset (partially-bound) binding, the client's RPC run-time library uses a well-known endpoint from the client's interface specification, if any. Otherwise, the client's run-time library automatically communicates with the endpoint-mapping service on the specified remote host to obtain the endpoint of a compatible server from the endpoint-map database. If a compatible server is located, the RPC run-time library updates the binding with a new endpoint. If a compatible server is not found, the remote procedure call fails. For calls using a connection protocol (**ncacn**), the RPC_S_NO_ENDPOINT_FOUND status code is returned to the client. For calls using a datagram protocol (**ncadg**), the RPC_S_COMM_FAILURE status code is returned to the client.

Server applications should register all binding handles by calling **RpcEpRegister** and **RpcEpRegisterNoReplace** if the server wants to be available to clients that make a remote procedure call on a reset binding handle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcEpRegister, **RpcEpRegisterNoReplace**

RpcBindingServerFromClient

An application calls **RpcBindingServerFromClient** to convert a client binding handle into a partially-bound server binding handle.

```
RPC_STATUS RPC_ENTRY RpcBindingServerFromClient(
  RPC_BINDING_HANDLE ClientBinding,
  RPC_BINDING_HANDLE *ServerBinding
);
```

Parameters

ClientBinding
Client binding handle to convert to a server binding handle.

ServerBinding
Returns a server binding handle.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_S_CANNOT_SUPPORT	Cannot determine the client's host (not TCP or SPX).

Remarks

The following protocol sequences support **RpcBindingServerFromClient**:

- **ncadg_ip_udp**
- **ncadg_ipx**
- **ncacn_ip_tcp**
- **ncacn_spx**.
- **ncacn_np** (effective with Windows 2000)

An application gets a client binding handle from the RPC run-time. When the remote procedure call arrives at a server, the run-time creates a client binding handle that contains information about the calling client. The run-time passes this handle to the server manager function as the first argument.

Calling **RpcBindingServerFromClient** converts this client handle to a server handle that has these properties:

- The server handle is a partially-bound handle. It contains a network address for the calling client, but lacks an endpoint.
- The server handle contains the same object UUID used by the calling client. This can be the nil UUID. For more information on how a client specifies an object UUID for a call, see **RpcBindingsetObject**, **RpcNsBindingImportBegin**, **RpcNsBindingLookupBegin**, and **RpcBindingFromStringBinding**.
- The server handle contains no authentication information.

The client application must call **RpcBindingFree** to free the resources used by the server binding handle once it is no longer needed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingFree, **RpcBindingSetObject**, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingImportBegin**, **RpcNsBindingLookupBegin**, **RpcBindingFromStringBinding**

RpcBindingSetAuthInfo

The **RpcBindingSetAuthInfo** function sets a binding handle's authentication and authorization information.

```
RPC_STATUS RPC_ENTRY RpcBindingSetAuthInfo(
  RPC_BINDING_HANDLE Binding,
  unsigned char *ServerPrincName,
  unsigned long AuthnLevel,
  unsigned long AuthnSvc,
  RPC_AUTH_IDENTITY_HANDLE AuthIdentity,
  unsigned long AuthzService
);
```

Parameters

Binding
: Server binding handle to which authentication and authorization information is to be applied.

ServerPrincName
: Pointer to the expected principal name of the server referenced by *hBinding*. The content of the name and its syntax are defined by the authentication service in use. Set this parameter to NULL when accessing the Microsoft Message Queue Server (MSMQ) security.

AuthnLevel
: Level of authentication to be performed on remote procedure calls made using *hBinding* . For a list of the RPC-supported authentication levels, see the list of *Authentication-Level Constants*.

Under MSMQ security this can be RPC_C_AUTHN_LEVEL_NONE, RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, or RPC_C_AUTHN_LEVEL_PKT_PRIVACY. If you specify any other level it will be converted silently to the next higher supported level.

AuthnSvc
: Authentication service to use. See Note.

Specify RPC_C_AUTHN_NONE to turn off authentication for remote procedure calls made using *hBinding* .

If RPC_C_AUTHN_DEFAULT is specified, the RPC run-time library uses the RPC_C_AUTHN_WINNT authentication service for remote procedure calls made using *hBinding* .

The only authentication services allowed for MSMQ security are RPC_C_AUTHN_NONE and RPC_C_AUTHN_MQ. If you use RPC_C_AUTHN_WINNT or one of the other security providers your **[message]** calls will not be authenticated or encrypted.

AuthIdentity

A handle for the structure that contains the client's authentication and authorization credentials appropriate for the selected authentication and authorization service. When using the RPC_C_AUTHN_WINNT authentication service *AuthIdentity* should be a pointer to a **SEC_WINNT_AUTH_IDENTITY** structure (defined in Rpcdce.h).

When you select the RPC_C_AUTHN_GSS_SCHANNEL authentication service the *AuthIdentity* parameter should be a pointer to a **SCHANNEL_CRED** structure (defined in Schannel.h). Specify a null value to use the security login context for the current address space. Pass the value RPC_C_NO_CREDENTIALS to use an anonymous log-in context.

AuthzService

The authorization service implemented by the server for the interface of interest. See Note

The validity and trustworthiness of authorization data, like any application data, depends on the authentication service and authentication level selected. This parameter is ignored when using the RPC_C_AUTHN_WINNT authentication service.

Note For more information see *Authentication-Service Constants*.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_S_UNKNOWN_AUTHN_SERVICE	Unknown authentication service.

Remarks

A client application calls the **RpcBindingSetAuthInfo** function to set up a server binding handle for making authenticated remote procedure calls. A client is not required to call this function.

Unless a client calls **RpcBindingSetAuthInfo**, no remote procedure calls on the *hBinding* binding handle are authenticated. A server can call **RpcBindingInqAuthClient** from within a remote procedure call to determine if that call has been authenticated.

For information on platform-specific issues related to the **RpcBindingSetAuthInfo** function, see *MS DOS Considerations* and *Windows 95/98 Considerations*.

All versions of Microsoft RPC for Windows 9*x* and Windows NT (4.*x* and earlier) maintain a pointer to the *AuthIdentity* parameter for as long as the binding handle exists. Therefore, your program should ensure it is not on the stack and is not freed until the binding handle is freed. If the binding handle is copied, or if a context handle is created from the binding handle, then the *AuthIdentity* pointer will also be copied.

The **RpcBindingSetAuthInfo** function in Microsoft RPC for Windows 2000 takes a snapshot of the credentials. Therefore, the memory dedicated to the *AuthIdentity* parameter can be freed before the binding handle. The exception to this is when your application uses **RpcBindingSetAuthInfo** with RPC_C_QOS_IDENTITY_DYNAMIC and also specifies a non-NULL value for *AuthIdentity.*

Because of the varying requirements of different versions of Microsoft RPC, Microsoft recommends that your application maintains a pointer to the *AuthIdentity* parameter for as long as the binding handle exists. Doing so increases the applications portability.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

MSMQ Security Services, **RpcBindingSetOption**, **RpcBindingInqAuthInfo**, **RpcServerRegisterAuthInfo**

RpcBindingSetAuthInfoEx

The **RpcBindingSetAuthInfoEx** function sets a binding handle's authentication, authorization, and security quality-of-service information.

```
RPC_STATUS RPC_ENTRY RpcBindingSetAuthInfoEx(
  RPC_BINDING_HANDLE Binding,
  unsigned char PAPI *ServerPrincName,
  unsigned long AuthnLevel,
  unsigned long AuthnSvc,
  RPC_AUTH_IDENTITY_HANDLE AuthIdentity,
  unsigned long AuthzSvc,
  RPC_SECURITY_QOS *SecurityQOS
);
```

Parameters

Binding
: Server binding handle into which authentication and authorization information is set.

ServerPrincName
: Pointer to the expected principal name of the server referenced by *Binding*. The content of the name and its syntax are defined by the authentication service in use.

AuthnLevel
: Level of authentication to be performed on remote procedure calls made using *Binding*. For a list of the RPC-supported authentication levels, see *Authentication-Level Constants*.

AuthnSvc
: Authentication service to use. See Note.

Specify RPC_C_AUTHN_NONE to turn off authentication for remote procedure calls made using *Binding*.

If RPC_C_AUTHN_DEFAULT is specified, the RPC run-time library uses the RPC_C_AUTHN_WINNT authentication service for remote procedure calls made using *Binding*.

AuthIdentity
: Handle for the structure that contains the client's authentication and authorization credentials appropriate for the selected authentication and authorization service.

When using the RPC_C_AUTHN.WINNT authentication service *AuthIdentity* should be a pointer to a **SEC_WINNT_AUTH_IDENTITY** structure (defined in Rpcdce.h).

Specify a null value to use the security login context for the current address space. Pass the value RPC_C_NO_CREDENTIALS to use an anonymous log-in context.

AuthzSvc
: Authorization service implemented by the server for the interface of interest. The validity and trustworthiness of authorization data, like any application data, depends on the authentication service and authentication level selected. This parameter is ignored when using the RPC_C_AUTHN_WINNT authentication service. See Note.

SecurityQOS
: Pointer to the **RPC_SECURITY_QOS** structure, which defines the security quality-of-service.

Note For a list of the RPC-supported authentication services, see *Authentication-Service Constants*.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.

Value	Meaning
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_S_UNKNOWN_AUTHN_SERVICE	Unknown authentication service.

Remarks

A client application calls the **RpcBindingSetAuthInfoEx** function to set up a server binding handle for making authenticated remote procedure calls. This function provides the capability to set security quality-of-service information on the binding handle. It is otherwise identical to **RpcBindingSetAuthInfo**.

Unless a client calls **RpcBindingSetAuthInfoEx**, all remote procedure calls on *Binding* are unauthenticated. A client is not required to call this function.

All versions of Microsoft RPC for Windows 9*x* and Windows NT (4.*x* and earlier) maintain a pointer to the *AuthIdentity* parameter for as long as the binding handle exists. Therefore, your program should ensure it is not on the stack and is not freed until the binding handle is freed. If the binding handle is copied, or if a context handle is created from the binding handle, then the *AuthIdentity* pointer will also be copied.

The **RpcBindingSetAuthInfoEx** function in Microsoft RPC for Windows 2000 takes a "snapshot" of the credentials. Therefore, the memory dedicated to the *AuthIdentity* parameter can be freed before the binding handle. The exception to this is when your application uses **RpcBindingSetAuthInfoEx** with RPC_C_QOS_IDENTITY_DYNAMIC and also specifies a non-NULL value for *AuthIdentity.*

Because of the varying requirements of different versions of Microsoft RPC, Microsoft recommends that your application maintains a pointer to the *AuthIdentity* parameter for as long as the binding handle exists. Doing so increases the applications portability.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 OSR2 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RPC_SECURITY_QOS, **RpcBindingInqAuthInfoEx**, **RpcServerRegisterAuthInfo**

RpcBindingSetObject

The **RpcBindingSetObject** function sets the object UUID value in a binding handle.

```
RPC_STATUS RPC_ENTRY RpcBindingSetObject(
  RPC_BINDING_HANDLE Binding,
  UUID *ObjectUuid
);
```

Parameters

Binding
Server binding into which the *ObjectUuid* is set.

ObjectUuid
Pointer to the UUID of the object serviced by the server specified in the *Binding* argument. *ObjectUuid* is a unique identifier of an object to which a remote procedure call can be made.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

An application calls the **RpcBindingSetObject** function to associate an object UUID with a server binding handle. The set-object operation replaces the previously associated object UUID with the UUID in the *ObjectUuid* argument.

To set the object UUID to the nil UUID, specify a null value or the nil UUID for the *ObjectUuid* argument.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingFromStringBinding, **RpcBindingInqObject**

RpcBindingSetOption

The **RpcBindingSetOption** function enables client applications to specify message-queuing options on a binding handle. This function is supported on Windows NT, Windows 2000. Windows 95, with the second release of DCOM 95, and Windows 98 also support this function.

```
RPC_STATUS RPC_ENTRY RpcBindingSetOption(
  RPC_BINDING_HANDLE hBinding,
  unsigned long Option,
  unsigned long OptionValue
);
```

Parameters

hBinding
Server binding to modify.

Option
Binding property to modify. See *Note*.

OptionValue
The new value for the binding property. See *Note*.

Note For a list of binding options and their possible values, see *Binding Option Constants*.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_CANNOT_SUPPORT	The function is not supported for either the operating system or the transport. Note that calling **RpcBindingSetOption** on binding handles that use any protocol sequence other than **ncadg_mq** will fail and return this value.

Remarks

RPC client processes use **RpcBindingSetOption** to control the delivery quality-of-service, call logging, and call lifetimes. Changing the binding-handle properties will affect all remote calls until the properties are changed by another call to **RpcBindingSetOption**. You can also call **RpcBindingSetAuthInfo** to set security options for the binding handle.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 OSR2 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

message, RPC Message Queuing, **RpcBindingInqOption**, **RpcBindingSetAuthInfo**, **RpcBindingInqAuthClient**

RpcBindingToStringBinding

The **RpcBindingToStringBinding** function returns a string representation of a binding handle.

```
RPC_STATUS RPC_ENTRY RpcBindingToStringBinding(
  RPC_BINDING_HANDLE Binding,
  unsigned char **StringBinding
);
```

Parameters

Binding
Client or server binding handle to convert to a string representation of a binding handle.

StringBinding
Returns a pointer to a pointer to the string representation of the binding handle specified in the *Binding* argument.

Specify a null value to prevent **RpcBindingToStringBinding** from returning the *StringBinding* argument. In this case, the application does not call the **RpcStringFree** function.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.

Remarks

The **RpcBindingToStringBinding** function converts a client or server binding handle to its string representation.

The RPC run-time library allocates memory for the string returned in the *StringBinding* argument. The application is responsible for calling the **RpcStringFree** function to deallocate that memory.

If the binding handle in the *Binding* argument contained a nil object UUID, the object UUID field is not included in the returned string.

To parse the returned *StringBinding* argument, call the **RpcStringBindingParse** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingFromStringBinding, **RpcStringBindingParse**, **RpcStringFree**

RpcBindingVectorFree

The **RpcBindingVectorFree** function frees the binding handles contained in the vector and the vector itself.

```
RPC_STATUS RPC_ENTRY RpcBindingVectorFree(
  RPC_BINDING_VECTOR **BindingVector
);
```

Parameters

BindingVector
Pointer to a pointer to a vector of server binding handles. On return, the pointer is set to NULL.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

An application calls the **RpcBindingVectorFree** function to release the memory used to store a vector of server binding handles. The function frees both the binding handles and the vector itself.

A server obtains a vector of binding handles by calling the **RpcServerInqBindings** function. A client obtains a vector of binding handles by calling the **RpcNsBindingLookupNext** function.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcNsBindingLookupNext, **RpcServerInqBindings**

RpcCancelThread

The **RpcCancelThread** function cancels a thread.

```
RPC_STATUS RPC_ENTRY RpcCancelThread(
  HANDLE ThreadHandle
);
```

Parameters

ThreadHandle
The handle of the thread to cancel.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_ACCESS_DENIED	Thread handle does not have privilege.
RPC_S_CANNOT_SUPPORT	Called by an MS-DOS or Windows 3.*x* client.

Remarks

The **RpcCancelThread** function allows one client thread to cancel an RPC in progress on another client thread. When the function is called, the server run-time is informed of the **cancel** operation. The server stub can determine if the call has been canceled by calling **RpcTestCancel**. If the call has been canceled, the server stub should clean up and return control to the client.

By default, the client waits forever for the server to return control after a cancel. To reduce this time, call **RpcMgmtSetCancelTimeout**, specifying the number of seconds to wait for a response. If the server does not return within this interval, the call fails at the client with an RPC_S_CALL_FAILED exception. The server stub continues to execute.

If you are using the named pipes protocol, **ncacn_np**, you must specify a finite time-out.

Note You can use **RpcCancelThread** with any of the connection-oriented protocols (**ncacn_***) except **ncacn_http**, and with any of the datagram protocols except **ncadg_mq** and **ncalrpc**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

RpcCancelThreadEx

The **RpcCancelThread** function stops the execution of a thread.

```
RPC_STATUS RPC_ENTRY RpcCancelThread(
  HANDLE ThreadHandle,
  LONG Timeout
);
```

Parameters

ThreadHandle
The handle of the thread to cancel.

Timeout
Number of seconds to wait for the thread to be canceled before this function returns. To specify that a client waits an indefinite amount of time, pass the value RPC_C_CANCEL_INFINITE_TIMEOUT.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_ACCESS_DENIED	Thread handle does not have privilege.
RPC_S_CANNOT_SUPPORT	Called by an MS-DOS or Windows 3.*x* client.

Remarks

The **RpcCancelThread** function allows one client thread to cancel an RPC in progress on another client thread. When the function is called, the server run-time is informed of the **cancel** operation. The server stub can determine if the call has been canceled by calling **RpcTestCancel**. If the call has been canceled, the server stub should clean up and return control to the client.

Using the *Timeout* parameter, your application can specify the number of seconds to wait for a response. If the server does not return within this interval, the call fails at the client with an RPC_S_CALL_FAILED exception. The server stub continues to execute.

If you are using the named pipes protocol, **ncacn_np**, you must specify a finite time-out.

Note You can use **RpcCancelThread** with any of the connection-oriented protocols (**ncacn_***) except **ncacn_http**, and with any of the datagram protocols except **ncadg_mq** and **ncalrpc**.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

RpcCertGeneratePrincipalName

Server programs use the **RpcCertGeneratePrincipalName** function to generate principal names for security certificates.

```
RPC_STATUS RpcCertGeneratePrincipalName(
  PCCERT_CONTEXT Context,
  DWORD Flags,
  UCHAR **pBuffer
);
```

Parameters

Context
Pointer to the security-certificate context.

Flags
Currently, the only valid flag for this parameter is RPC_C_FULL_CERT_CHAIN. Using this flag causes the principal name to be generated in fullsic format.

pBuffer
Pointer to a pointer. The **RpcCertGeneratePrincipalName** function sets this to point at a null-terminated string that contains the principal name.

Remarks

By default, the principal name that the **RpcCertGeneratePrincipalName** function passes back is in msstd format. To generate a name in fullsic format, pass RPC_C_FULL_CERT_CHAIN as the value for the *Flags* parameter.

Your application must call **RpcStringFree** to release the memory for the string which contains the principal name.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 OSR2 or later.
Header: Declared in Rpcssl.h.
Library: Use Rpcrt4.lib.

+ See Also

Principal Names

RpcEpRegister

The **RpcEpRegister** function adds to or replaces server address information in the local endpoint-map database. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcEpRegister(
  RPC_IF_HANDLE IfSpec,
  RPC_BINDING_VECTOR *BindingVector,
  UUID_VECTOR *UuidVector,
  unsigned char *Annotation
);
```

Parameters

IfSpec
Specifies an interface to register with the local endpoint-map database.

BindingVector
Pointer to a vector of binding handles over which the server can receive remote procedure calls.

UuidVector
Pointer to a vector of object UUIDs offered by the server. The server application constructs this vector.A null argument value indicates there are no object UUIDs to register.

Annotation
Pointer to the character-string comment applied to each cross-product element added to the local endpoint-map database. The string can be up to 64 characters long, including the null terminating character. Specify a null value or a null-terminated string ("\0") if there is no annotation string.

The annotation string is used by applications for information only. RPC does not use this string to determine which server instance a client communicates with or for enumerating elements in the endpoint-map database.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_BINDINGS	No bindings.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

The **RpcEpRegister** function adds or replaces entries in the local host's endpoint-map database. For an existing database entry that matches the provided interface specification, binding handle, and object UUID, this function replaces the entry's endpoint with the endpoint in the provided binding handle.

A server uses **RpcEpRegister** rather than **RpcEpRegisterNoReplace** when only a single instance of the server will run on the server's host. In other words, use this function when no more than one server instance will offer the same interface UUID, object UUID, and protocol sequence at any one time.

When entries are not replaced, stale data accumulates each time a server instance stops running without calling **RpcEpUnregister**. Stale entries increase the likelihood that a client will receive endpoints to nonexistent servers. The client will spend time trying to communicate with a nonexistent server before obtaining another endpoint.

Using **RpcEpRegister** to replace existing endpoint-map database entries reduces the likelihood that a client will be given the endpoint of a nonexistent server instance. A server application calls this function to register endpoints specified by calling any of the following functions:

- **RpcServerUseAllProtseqs**
- **RpcServerUseProtseq**
- **RpcServerUseProtseqEp**

A server that calls only **RpcServerUseAllProtseqsIf** or **RpcServerUseProtseqIf** does not need to call **RpcEpRegister**. In this case, the client's run-time library uses an endpoint from the client's interface specification to fill in a partially-bound binding handle.

If the server also exports to the name-service database, the server calls **RpcEpRegister** with the same *IfSpec*, *BindingVector*, and *UuidVector* that the server uses when calling the **RpcNsBindingExport** function.

For automatically started servers running over a connection-oriented protocol, the RPC run-time library automatically generates a dynamic endpoint. In this case, the server can call **RpcServerInqBindings** followed by **RpcEpRegister** to make itself available to multiple clients. Otherwise, the automatically started server is known only to the client for which the server was started. Each element added to the endpoint-map database logically contains the following:

- Interface UUID
- Interface version (major and minor)
- Binding handle
- Object UUID (optional)
- Annotation (optional)

RpcEpRegister creates a cross-product from the *IfSpec*, *BindingVector*, and *UuidVector* arguments and adds each element in the cross-product as a separate registration in the endpoint-map database.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingFromStringBinding, **RpcEpRegisterNoReplace**, **RpcEpUnregister**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

RpcEpRegisterNoReplace

The **RpcEpRegisterNoReplace** function adds server-address information to the local endpoint-map database.

```
RPC_STATUS RPC_ENTRY RpcEpRegisterNoReplace(
  RPC_IF_HANDLE IfSpec,
  RPC_BINDING_VECTOR *BindingVector,
  UUID_VECTOR *UuidVector,
  unsigned char *Annotation
);
```

Parameters

IfSpec

Specifies an interface to register with the local endpoint-map database.

BindingVector

Pointer to a vector of binding handles over which the server can receive remote procedure calls.

UuidVector

Pointer to a vector of object UUIDs offered by the server. The server application constructs this vector.

A null argument value indicates there are no object UUIDs to register.

Annotation

Pointer to the character-string comment applied to each cross-product element added to the local endpoint-map database. The string can be up to 64 characters long, including the null-terminating character. Specify a null value or a null-terminated string ("\0") if there is no annotation string.

The annotation string is used by applications for information only. RPC does not use this string to determine which server instance a client communicates with or to enumerate elements in the endpoint-map database.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_BINDINGS	No bindings.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

The **RpcEpRegisterNoReplace** function adds entries to the local host's endpoint-map database. This function does not replace existing database entries.

A server uses **RpcEpRegisterNoReplace** rather than **RpcEpRegister** when multiple instances of the server will run on the same host. In other words, use this function when more than one server instance will offer the same interface UUID, object UUID, and protocol sequence at any one time.

Because entries are not replaced when calling **RpcEpRegisterNoReplace**, servers must unregister themselves before they stop running. Otherwise, stale data accumulates each time a server instance stops running without calling **RpcEpUnregister**. Stale entries increase the likelihood that a client will receive endpoints to nonexistent servers. The client will spend time trying to communicate with a nonexistent server before obtaining another endpoint.

A server application calls **RpcEpRegisterNoReplace** to register endpoints specified by calling any of the following functions:

- **RpcServerUseAllProtseqs**
- **RpcServerUseProtseq**
- **RpcServerUseProtseqEp**

A server that calls only **RpcServerUseAllProtseqsIf** or **RpcServerUseProtseqIf** is not required to call **RpcEpRegisterNoReplace**. In this case, the client's run-time library uses an endpoint from the client's interface specification to fill in a partially-bound binding handle.

If the server also exports to the name-service database, the server calls **RpcEpRegisterNoReplace** with the same *IfSpec*, *BindingVector*, and *UuidVector* arguments that the server uses when calling the **RpcNsBindingExport** function.

For automatically started servers running over a connection-oriented protocol, the RPC run-time library automatically generates a dynamic endpoint. In this case, the server can call **RpcServerInqBindings** followed by **RpcEpRegisterNoReplace** to make itself available to multiple clients. Otherwise, the automatically started server is known only to the client for which the server was started.

Each element added to the endpoint-map database logically contains the following:

- Interface UUID
- Interface version (major and minor)
- Binding handle
- Object UUID (optional)
- Annotation (optional)

RpcEpRegisterNoReplace creates a cross-product from the *IfSpec*, *BindingVector*, and *UuidVector* arguments and adds each element in the cross-product as a separate registration in the endpoint-map database.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingFromStringBinding, **RpcEpRegister**, **RpcEpUnregister**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

RpcEpResolveBinding

The **RpcEpResolveBinding** function resolves a partially-bound server binding handle into a fully-bound server binding handle.

```
RPC_STATUS RPC_ENTRY RpcEpResolveBinding(
  RPC_BINDING_HANDLE Binding,
  RPC_IF_HANDLE IfSpec
);
```

Parameters

Binding
Partially-bound server binding handle to resolve to a fully-bound server binding handle.

IfSpec
Stub-generated structure specifying the interface of interest.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

An application calls the **RpcEpResolveBinding** function to resolve a partially-bound server binding handle into a fully-bound binding handle.

Resolving binding handles requires an interface UUID and an object UUID (which may be nil). The RPC run-time library asks the endpoint-mapping service on the host specified by the *Binding* argument to look up an endpoint for a compatible server instance. To find the endpoint, the endpoint-mapping service looks in the endpoint-map database for the interface UUID in the *IfSpec* argument and the object UUID in the *Binding* argument, if any.

How the resolve-binding operation functions depends on whether the specified binding handle is partially- or fully-bound. When the client specifies a partially-bound handle, the resolve-binding operation has the possible outcomes on the following page.

- If no compatible server instances are registered in the endpoint-map database, the resolve-binding operation returns the EPT_S_NOT_REGISTERED status code.
- If a compatible server instance is registered in the endpoint-map database, the resolve-binding operation returns a fully-bound binding and the RPC_S_OK status code.

When the client specifies a fully-bound binding handle, the resolve-binding operation returns the specified binding handle and the RPC_S_OK status code. The resolve-binding operation does not contact the endpoint-mapping service.

In neither the partially- nor the fully-bound binding case does the resolve-binding operation contact a compatible server instance.

For information on platform-specific issues related to the **RpcEpResolveBinding** function, see *MS DOS Considerations* and *Windows 95/98 Considerations*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingFromStringBinding, **RpcBindingReset**, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingImportBegin**, **RpcNsBindingImportDone**, **RpcNsBindingImportNext**

RpcEpUnregister

The **RpcEpUnregister** function removes server-address information from the local endpoint-map database. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcEpUnregister(
  RPC_IF_HANDLE IfSpec,
  RPC_BINDING_VECTOR *BindingVector,
  UUID_VECTOR *UuidVector
);
```

Parameters

IfSpec
Specifies an interface to unregister from the local endpoint-map database.

BindingVector
Pointer to a vector of binding handles to unregister.

UuidVector

Pointer to an optional vector of object UUIDs to unregister. The server application constructs this vector. **RpcEpUnregister** unregisters all endpoint-map database elements that match the specified *IfSpec* and *BindingVector* arguments and the object UUID(s).

A null argument value indicates there are no object UUIDs to unregister.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_BINDINGS	No bindings.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

The **RpcEpUnregister** function removes elements from the local host's endpoint-map database. A server application calls this function only when the server has previously registered endpoints and the server wants to remove that address information from the endpoint-map database.

Specifically, **RpcEpUnregister** allows a server application to remove its own endpoint-map database elements (server-address information) based on the interface specification or on both the interface specification and the object UUID(s) of the resource(s) offered.

The server calls the **RpcServerInqBindings** function to obtain the required *BindingVector* argument. To unregister selected endpoints, the server can prune the binding vector prior to calling this function.

RpcEpUnregister creates a cross-product from the *IfSpec*, *BindingVector*, and *UuidVector* arguments and removes each element in the cross-product from the endpoint-map database. Use **RpcEpUnregister** cautiously: removing elements from the endpoint-map database may make servers unavailable to client applications that have not previously communicated with the server.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcEpRegister, **RpcEpRegisterNoReplace**, **RpcNsBindingUnexport**, **RpcServerInqBindings**

RpcExceptionCode

The **RpcExceptionCode** function retrieves a code that identifies the type of exception that occurred.

```
unsigned long RpcExceptionCode(VOID);
```

Parameters

This function has no parameters.

Return Values

Possible return values include the set of error codes returned by the RPC functions with the prefixes "RPC_S_" and "RPC_X" and the set of exceptions returned by the Windows operating system. For a partial listing of these codes, see *RPC Return Values*.

Remarks

The **RpcExceptionCode** function can only be called from within the *expression* and *exception statements* of an **RpcTryExcept** exception handler.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpc.h.
Library: Use Rpcrt4.lib.

See Also

Exception Handling, **RpcExcept**, **RpcFinally**

RpcIfIdVectorFree

The **RpcIfIdVectorFree** function frees the vector and the interface-identification structures contained in the vector. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcIfIdVectorFree(
  RPC_IF_ID_VECTOR **IfIdVec
);
```

Parameters

IfIdVec

The address of a pointer to a vector of interface information. On return, the pointer is set to NULL.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument.

Remarks

An application calls the **RpcIfIdVectorFree** function to release the memory used to store a vector of interface identifications. **RpcIfIdVectorFree** frees memory containing the interface identifications and the vector itself. On return, this function sets the *IfIdVec* argument to NULL.

An application obtains a vector of interface identifications by calling the **RpcNsMgmtEntryInqIfIds** and **RpcMgmtInqIfIds** functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcIfInqId, **RpcMgmtInqIfIds**, **RpcNsMgmtEntryInqIfIds**

RpcIfInqId

The **RpcIfInqId** function returns the interface-identification part of an interface specification.

```
RPC_STATUS RPC_ENTRY RpcIfInqId(
  RPC_IF_HANDLE RpcIfHandle,
  RPC_IF_ID *RpcIfId
);
```

Parameters

RpcIfHandle

Stub-generated structure specifying the interface to query.

RpcIfId

Returns a pointer to the interface identification. The application provides memory for the returned data.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

An application calls the **RpcIfInqId** function to obtain a copy of the interface identification from the provided interface specification.

The returned interface identification consists of the interface UUID and interface version numbers (major and minor) specified in the *IfSpec* argument from the IDL file.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcServerInqIf, **RpcServerRegisterIf**

RpcImpersonateClient

A server thread that is processing client remote procedure calls can call the **RpcImpersonateClient** function to impersonate the active client.

```
RPC_STATUS RPC_ENTRY RpcImpersonateClient(
  RPC_BINDING_HANDLE BindingHandle
);
```

Parameters

BindingHandle

Binding handle on the server that represents a binding to a client. The server impersonates the client indicated by this handle. If a value of zero is specified, the server impersonates the client that is being served by this server thread.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_CALL_ACTIVE	No client is active on this server thread.
RPC_S_CANNOT_SUPPORT	The function is not supported for either the operating system, the transport, or this security subsystem.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_S_NO_CONTEXT_AVAILABLE	The server does not have permission to impersonate the client.

Remarks

In a multithreaded application, if the call to **RpcImpersonateClient** is with a handle to another client thread, you must call **RpcRevertToSelfEx** with the handle to that thread to end impersonation.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpc.h.
Library: Use Rpcrt4.lib.

See Also

Client Impersonation, **RpcRevertToSelf**

RpcMacSetYieldInfo

The **RpcMacSetYieldInfo** function configures Macintosh client applications to yield to other applications during remote procedure calls.

```
RPC_STATUS RPC_ENTRY RpcMacSetYieldInfo(
  MACYIELDCALLBACK pfnCallback
);
```

Parameters

pfnCallback
Pointer to a callback function.

Return Values

Value	Meaning
RPC_S_OK	The information was set successfully.

Remarks

Register a yielding function by calling **RpcMacSetYieldInfo** with a pointer to the callback (yielding) function. If a yielding function is not registered, an RPC will not yield on the Mac.

The yielding function must yield until **pStatus* is not equal to 1. For example:

```
void RPC_ENTRY MacCallbackFunc (short *pStatus)
{
    MSG msg;
    while (*pStatus == 1)
    {
        if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
}
```

Note that Rpc.h must be included before Winerror.h (or any files that include it, such as Winbase.h, Windows.h, and so on).

! Requirements

Windows NT/2000: Unsupported.
Windows 95/98: Unsupported.
Version: Requires Macintosh client.
Header: Declared in Rpc.h.
Library: Use Rpcrt4.lib.

RpcMgmtEnableIdleCleanup

The **RpcMgmtEnableIdleCleanup** function closes idle resources, such as network connections, on the client. This function is supported on all 32-bit and 16-bit Windows platforms. It is not supported on MS-DOS.

```
RPC_STATUS RPC_ENTRY RpcMgmtEnableIdleCleanup(VOID);
```

Parameters

This function has no parameters.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_OUT_OF_THREADS	Out of threads.
RPC_S_OUT_OF_RESOURCES	Out of resources.
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

Connection-oriented protocols set five seconds as the default waiting period to determine whether a resource is idle.

Note **RpcMgmtEnableIdleCleanup** is a Microsoft extension to the OSF-DCE RPC specification.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcServerUnregisterIf

RpcMgmtEpEltInqBegin

The **RpcMgmtEpEltInqBegin** function creates an inquiry context for viewing the elements in an endpoint map. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtEpEltInqBegin(
  RPC_BINDING_HANDLE EpBinding,
  unsigned long InquiryType,
  RPC_IF_ID *IfId,
  unsigned long VersOption,
  UUID *ObjectUuid,
  RPC_EP_INQ_HANDLE *InquiryContext
);
```

Parameters

EpBinding

Host whose endpoint-map elements will be viewed. Specify NULL to view elements from the local host.

InquiryType

Integer value that indicates the type of inquiry to perform on the endpoint map. The following are valid inquiry types:

Value	Description
RPC_C_EP_ALL_ELTS	Returns every element from the endpoint map. The *IfId*, *VersOption*, and *ObjectUuid* parameters are ignored.
RPC_C_EP_MATCH_BY_IF	Searches the endpoint map for elements that contain the interface identifier specified by the *IfId* and *VersOption* values.
RPC_C_EP_MATCH_BY_OBJ	Searches the endpoint map for elements that contain the object UUID specified by *ObjectUuid*.
RPC_C_EP_MATCH_BY_BOTH	Searches the endpoint map for elements that contain the interface identifier and object UUID specified by *IfId*, *VersOption*, and *ObjectUuid*.

IfId

Interface identifier of the endpoint-map elements to be returned by **RpcMgmtEpEltInqNext**. This parameter is only used when *InquiryType* is either RPC_C_EP_MATCH_BY_IF or RPC_C_EP_MATCH_BY_BOTH. Otherwise, it is ignored.

VersOption

Specifies how **RpcMgmtEpEltInqNext** uses the *IfId* parameter. This parameter is only used when *InquiryType* is either RPC_C_EP_MATCH_BY_IF or RPC_C_EP_MATCH_BY_BOTH. Otherwise, it is ignored. The following are valid values for this parameter:

Value	Description
RPC_C_VERS_ALL	Returns endpoint-map elements that offer the specified interface UUID, regardless of the version numbers.
RPC_C_VERS_COMPATIBLE	Returns endpoint-map elements that offer the same major version of the specified interface UUID and a minor version greater than or equal to the minor version of the specified interface UUID.

(continued)

(continued)

Value	Description
RPC_C_VERS_EXACT	Returns endpoint-map elements that offer the specified version of the specified interface UUID.
RPC_C_VERS_MAJOR_ONLY	Returns endpoint-map elements that offer the same major version of the specified interface UUID and ignores the minor version.
RPC_C_VERS_UPTO	Returns endpoint-map elements that offer a version of the specified interface UUID less than or equal to the specified major and minor version.

ObjectUuid
The object UUID that **RpcMgmtEpEltInqNext** looks for in endpoint-map elements. This parameter is used only when *InquiryType* is either RPC_C_EP_MATCH_BY_OBJ or RPC_C_EP_MATCH_BY_BOTH.

InquiryContext
Returns an inquiry context for use with **RpcMgmtEpEltInqNext** and **RpcMgmtEpEltInqDone**. See ***RPC_EP_INQ_HANDLE***.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

The **RpcMgmtEpEltInqBegin** function creates an inquiry context for viewing server-address information stored in the endpoint map. Using *InquiryType* and *VersOption*, an application specifies which of the following endpoint-map elements are to be returned from calls to **RpcMgmtEpEltInqNext**:

- All elements
- Those elements with the specified interface identifier
- Those elements with the specified object UUID
- Those elements with both the specified interface identifier and object UUID

Before calling **RpcMgmtEpEltInqNext**, the application must first call this function to create an inquiry context. After viewing the endpoint-map elements, the application calls **RpcMgmtEpEltInqDone** to delete the inquiry context.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcEpRegister

RpcMgmtEpEltInqDone

The **RpcMgmtEpEltInqDone** function deletes the inquiry context for viewing the elements in an endpoint map. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtEpEltInqDone(
  RPC_EP_INQ_HANDLE *InquiryContext
);
```

Parameters

InquiryContext
Inquiry context to delete and returns the value NULL.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

The **RpcMgmtEpEltInqDone** function deletes an inquiry context created by **RpcMgmtEpEltInqBegin**. An application calls this function after viewing local endpoint-map elements using **RpcMgmtEpEltInqNext**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcEpRegister

RpcMgmtEpEltInqNext

The **RpcMgmtEpEltInqNext** function returns one element from an endpoint map. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtEpEltInqNext(
  RPC_EP_INQ_HANDLE InquiryContext,
  RPC_IF_ID *IfId,
  RPC_BINDING_HANDLE *Binding,
  UUID *ObjectUuid,
  unsigned char **Annotation
);
```

Parameters

InquiryContext
Specifies an inquiry context. The inquiry context is returned from **RpcMgmtEpEltInqBegin**.

IfId
Returns the interface identifier of the endpoint-map element.

Binding
Optional. Returns the binding handle from the endpoint-map element.

ObjectUuid
Optional. Returns the object UUID from the endpoint-map element.

Annotation
Optional. Returns the annotation string for the endpoint-map element. When there is no annotation string in the endpoint-map element, the empty string ("") is returned.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

The **RpcMgmtEpEltInqNext** function returns one element from the endpoint map. Elements selected depend on the inquiry context. The selection criteria are determined by *InquiryType* of the **RpcMgmtEpEltInqBegin** function that returned *InquiryContext*.

An application can view all the selected endpoint-map elements by repeatedly calling **RpcMgmtEpEltInqNext**. When all the elements have been viewed, this function returns an RPC_X_NO_MORE_ENTRIES status. The returned elements are unordered.

When the respective arguments are non-NULL, the RPC run-time function library allocates memory for *Binding* and *Annotation* on each call to this function. The application is responsible for calling **RpcBindingFree** for each returned *Binding* and **RpcStringFree** for each returned *Annotation*.

After viewing the endpoint-map elements, the application must call **RpcMgmtEpEltInqDone** to delete the inquiry context.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

RpcEpRegister

RpcMgmtEpUnregister

The **RpcMgmtEpUnregister** function removes server address information from an endpoint map. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtEpUnregister(
  RPC_BINDING_HANDLE EpBinding,
  RPC_IF_ID *IfId,
  RPC_BINDING_HANDLE Binding,
  UUID *ObjectUuid
);
```

Parameters

EpBinding
Host whose endpoint-map elements are to be unregistered. To remove elements from the same host as the calling application, the application specifies a value of NULL. To remove elements from another host, the application specifies a server binding handle for any server residing on that host. Note that the application can specify the same binding handle it is using to make other remote procedure calls.

IfId
Interface identifier to remove from the endpoint map.

Binding
Binding handle to remove.

ObjectUuid
Optional object UUID to remove. The value NULL indicates there is no object UUID to remove.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_CANT_PERFORM_OP	Cannot perform the requested operation.

Remarks

The **RpcMgmtEpUnregister** function unregisters an element from the endpoint map. A management program calls this function to remove addresses of servers that are no longer available, or to remove addresses of servers that support objects that are no longer offered.

The *EpBinding* parameter must be a full binding. The object UUID associated with the *EpBinding* parameter must be a nil UUID. Specifying a non-nil UUID causes the function to fail with the status code EPT_S_CANT_PERFORM_OP. Other than the host information and object UUID, all information in this argument is ignored.

An application calls **RpcMgmtEpEltInqNext** to view local endpoint-map elements. The application can then remove the elements using **RpcMgmtEpUnregister**.

Note Use this function with caution. Removing elements from the local endpoint map may make servers unavailable to client applications that do not already have a fully-bound binding handle to the server.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later. Not supported on Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcEpRegister, **RpcEpUnregister**

RpcMgmtInqComTimeout

The **RpcMgmtInqComTimeout** function returns the binding-communications time-out value in a binding handle.

```
RPC_STATUS RPC_ENTRY RpcMgmtInqComTimeout(
  RPC_BINDING_HANDLE Binding,
  unsigned int *Timeout
);
```

Parameters

Binding
Specifies a binding.

Timeout
Returns a pointer to the time-out value from the *Binding* argument.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

A client application calls **RpcMgmtInqComTimeout** to view the time-out value in a server binding handle. The time-out value specifies the relative amount of time that should be spent to establish a binding to the server before giving up. For a table of the time-out values, see *Binding Time-out Constants*.

A client also calls **RpcMgmtSetComTimeout** to change the time-out value.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcMgmtInqStats, **RpcMgmtSetComTimeout**, Binding Time-out Constants

RpcMgmtInqDefaultProtectLevel

The **RpcMgmtInqDefaultProtectLevel** function returns the default authentication level for an authentication service.

```
RPC_STATUS RPC_ENTRY RpcMgmtInqDefaultProtectLevel(
  unsigned int AuthnSvc,
  unsigned int *AuthnLevel
);
```

Parameters

AuthnSvc

Authentication service for which to return the default authentication level. Possible values are as follows.

Value	Description
RPC_C_AUTHN_NONE	No authentication.
RPC_C_AUTHN_WINNT	32-bit Windows authentication service.

AuthnLevel

Returns the default authentication level for the specified authentication service. The authentication level determines the degree to which authenticated communications between the client and server are protected. Possible values are as follows:

Value	Description
RPC_C_AUTHN_LEVEL_DEFAULT	Uses the default authentication level for the specified authentication service.
RPC_C_AUTHN_LEVEL_NONE	Performs no authentication.
RPC_C_AUTHN_LEVEL_CONNECT	Authenticates only when the client establishes a relationship with a server.
RPC_C_AUTHN_LEVEL_CALL	Authenticates only at the beginning of each remote procedure call when the server receives the request. Does not apply to remote procedure calls made using the connection-based protocol sequences that start with the prefix "**ncacn**". If the protocol sequence in a binding is a connection-based protocol sequence and you specify this level, this function instead uses the RPC_C_AUTHN_LEVEL_PKT constant.
RPC_C_AUTHN_LEVEL_PKT	Authenticates that all data received is from the expected client.
RPC_C_AUTHN_LEVEL_PKT_INTEGRITY	Authenticates and verifies that none of the data transferred between client and server has been modified.
RPC_C_AUTHN_LEVEL_PKT_PRIVACY	Authenticates all previous levels and encrypts the argument value of each remote procedure call.

Note RPC_C_AUTHN_LEVEL_CALL, RPC_C_AUTHN_LEVEL_PKT, RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, and RPC_C_AUTHN_LEVEL_PKT_PRIVACY are only supported for clients communicating with a Windows NT and Windows 2000 server. A Windows 95 server can only accept incoming calls at the RPC_C_AUTHN_LEVEL_CONNECT level.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_UNKNOWN_AUTH_SERVICE	Unknown authentication service.

Remarks

An application calls the **RpcMgmtInqDefaultProtectLevel** function to obtain the default authentication level for a specified authentication service.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

RpcMgmtInqIfIds

The **RpcMgmtInqIfIds** function returns a vector containing the identifiers of the interfaces offered by the server. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtInqIfIds(
  RPC_BINDING_HANDLE Binding,
  RPC_IF_ID_VECTOR **IfIdVector
);
```

Parameters

Binding
To receive interface identifiers about a remote application, specify a server binding handle for that application. To receive interface information about your own application, specify a value of NULL.

IfIdVector
Returns the address of an interface identifier vector.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

An application calls the **RpcMgmtInqIfIds** function to obtain a vector of interface identifiers about the specified server from the RPC run-time library.

The RPC run-time library allocates memory for the interface identifier vector. The application is responsible for calling the **RpcIfIdVectorFree** function to release the memory used by this vector.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

RpcMgmtInqServerPrincName

The **RpcMgmtInqServerPrincName** function returns a server's principal name. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtInqServerPrincName(
  RPC_BINDING_HANDLE Binding,
  unsigned int AuthnSvc,
  unsigned char **ServerPrincName
);
```

Parameters

Binding
To receive the principal name for a server, specify a server binding handle for that server. To receive the principal name for your own (local) application, specify a value of NULL.

AuthnSvc
Authentication service for which a principal name is returned. Possible values are as follow:

Value	Description
RPC_C_AUTHN_NONE	No authentication.
RPC_C_AUTHN_WINNT	Windows NT/Windows 2000 authentication service.

ServerPrincName
Returns a principal name that is registered for the authentication service in *AuthnSvc* by the server referenced in *Binding*. If multiple names are registered, only one name is returned.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

An application calls the **RpcMgmtInqServerPrincName** function to obtain the principal name of a server that is registered for a specified authentication service.

The RPC run-time library allocates memory for the string returned in *ServerPrincName*. The application is responsible for calling the **RpcStringFree** function to release the memory used by this function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

RpcMgmtInqStats

The **RpcMgmtInqStats** function returns RPC run-time statistics. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtInqStats(
  RPC_BINDING_HANDLE Binding,
  RPC_STATS_VECTOR **Statistics
);
```

Parameters

Binding

To receive statistics about a remote application, specify a server binding handle for that application. To receive statistics about your own (local) application, specify a value of NULL.

Statistics

Returns a pointer to a pointer to the statistics about the server specified by the *Binding* argument. Each statistic is an **unsigned long** value.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

An application calls the **RpcMgmtInqStats** function to obtain statistics about the specified server from the RPC run-time library.

Each array element in the returned statistics vector contains an **unsigned long** value. The following table describes the statistics indexed by the specified constant.

Statistic	Description
RPC_C_STATS_CALLS_IN	Number of remote procedure calls received by the server.
RPC_C_STATS_CALLS_OUT	Number of remote procedure calls initiated by the server.
RPC_C_STATS_PKTS_IN	Number of network packets received by the server.
RPC_C_STATS_PKTS_OUT	Number of network packets sent by the server.

The RPC run-time library allocates memory for the statistics vector. The application is responsible for calling the **RpcMgmtStatsVectorFree** function to release the memory used by the statistics vector.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcEpResolveBinding, **RpcMgmtStatsVectorFree**

RpcMgmtIsServerListening

The **RpcMgmtIsServerListening** function tells whether a server is listening for remote procedure calls. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtIsServerListening(
  RPC_BINDING_HANDLE Binding
);
```

Parameters

Binding

To determine whether a remote application is listening for remote procedure calls, specify a server binding handle for that application. To determine whether your own (local) application is listening for remote procedure calls, specify a value of NULL.

Return Values

Value	Meaning
RPC_S_OK	Server listening for remote procedure calls.
RPC_S_SERVER_NOT_LISTENING	Server not listening for remote procedure calls.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

An application calls the **RpcMgmtIsServerListening** function to determine whether the server specified in the *Binding* argument is listening for remote procedure calls.

The **RpcMgmtIsServerListening** function returns a value of TRUE if the server has called **RpcServerListen**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcEpResolveBinding, **RpcServerListen**

RpcMgmtSetAuthorizationFn

The **RpcMgmtSetAuthorizationFn** function establishes an authorization function for processing remote calls to a server's management functions.

```
RPC_STATUS RPC_ENTRY RpcMgmtSetAuthorizationFn(
  RPC_MGMT_AUTHORIZATION_FN AuthorizationFn
);
```

Parameters

AuthorizationFn

Specifies an authorization function. The RPC server run-time library automatically calls this function whenever the server run-time receives a client request to execute one of the remote management functions. The server must implement this function. Applications specify a value of NULL to unregister a previously registered authorization function. After such a call, default authorizations are used.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

Server applications call the **RpcMgmtSetAuthorizationFn** function to establish an authorization function that controls access to the server's remote management functions. When a server has not called **RpcMgmtSetAuthorizationFn**, or calls with a null value for *AuthorizationFn*, the server run-time library uses the following default authorizations.

Remote function	Default authorization
RpcMgmtInqIfIds	Enabled
RpcMgmtInqServerPrincName	Enabled
RpcMgmtInqStats	Enabled
RpcMgmtIsServerListening	Enabled
RpcMgmtStopServerListening	Disabled

In the preceding table, "Enabled" indicates that all clients can execute the remote function, and "Disabled" indicates that all clients are prevented from executing the remote function.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcMgmtInqStats, **RpcMgmtIsServerListening**, **RpcMgmtStopServerListening**, **RpcMgmtWaitServerListen**, **RPC_MGMT_AUTHORIZATION_FN**

RpcMgmtSetCancelTimeout

The **RpcMgmtSetCancelTimeout** function sets the lower bound on the time to wait before timing out after forwarding a cancel:

```
RPC_STATUS RPC_ENTRY RpcMgmtSetCancelTimeout(
  signed int Seconds
);
```

Parameters

Seconds

An integer specifying the number of seconds to wait for a server to acknowledge a cancel command. To specify that a client waits an indefinite amount of time, supply the value RPC_C_CANCEL_INFINITE_TIMEOUT.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_CANNOT_SUPPORT	Called from an MS-DOS or Windows 3.*x* client.

Remarks

An application calls the **RpcMgmtSetCancelTimeout** function to reset the amount of time the run-time library waits for a server to acknowledge a cancel. The application specifies either to wait forever or to wait a specified length of time in seconds. If the value of *Seconds* is 0 (zero), the call is immediately abandoned upon a cancel command and control returns to the client application. The default value is RPC_C_CANCEL_INFINITE_TIMEOUT, which specifies waiting indefinitely for the call to complete.

The value for the cancel command time-out applies to all remote procedure calls made in the current thread. To change the time-out value, a multithreaded client must call this function in each thread of execution.

Note This function is only supported for Windows NT and Windows 2000 clients.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

RpcMgmtSetComTimeout

The **RpcMgmtSetComTimeout** function sets the binding-communications time-out value in a binding handle.

```
RPC_STATUS RPC_ENTRY RpcMgmtSetComTimeout(
  RPC_BINDING_HANDLE Binding,
  unsigned int Timeout
);
```

Parameters

Binding
The server binding handle whose time-out value is set.

Timeout
The communications time-out value.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_INVALID_TIMEOUT	Invalid time-out value.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

A client application calls **RpcMgmtSetComTimeout** to change the communications time-out value for a server binding handle. The time-out value specifies the relative amount of time that should be spent to establish a relationship to the server before giving up. Depending on the protocol sequence for the specified binding handle, the time-out value acts only as a hint to the RPC run-time library.

After the initial relationship is established, subsequent communications for the binding handle revert to not less than the default time-out for the protocol service. This means that after setting a short initial time-out establishing a connection, calls in progress will not be timed out any more aggressively than the default.

The time-out value can be any integer value from 0 to 10. For convenience, constants are provided for certain values in the time-out range. For a list of the RPC-defined values that an application can use for the time-out argument, see *Binding Time-out Constants*.

Note The values passed through the *Timeout* parameter are not in seconds. These values represent a relative amount of time on a scale of zero to 10.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcMgmtInqComTimeout

RpcMgmtSetServerStackSize

The **RpcMgmtSetServerStackSize** function specifies the stack size for each server thread. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtSetServerStackSize(
  unsigned int ThreadStackSize
);
```

Parameters

ThreadStackSize

The stack size in bytes allocated for each thread created by **RpcServerListen**. This value is applied to all threads created for the server. Select this value based on the stack requirements of the remote procedures offered by the server.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument.

Remarks

A server application calls the **RpcMgmtSetServerStackSize** function to specify the thread stack size to use when the RPC run-time library creates call threads for executing remote procedure calls. The *MaxCalls* argument in the **RpcServerListen** function specifies the number of call threads created.

Servers that know the stack requirements of all the manager functions in the interfaces it offers can call the **RpcMgmtSetServerStackSize** function to ensure that each call thread has the necessary stack size.

Calling **RpcMgmtSetServerStackSize** is optional. However, when used, it must be called before the server calls **RpcServerListen**. If a server does not call **RpcMgmtSetServerStackSize**, the default per thread stack size from the underlying threads package is used.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcServerListen

RpcMgmtStatsVectorFree

The **RpcMgmtStatsVectorFree** function frees a statistics vector. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtStatsVectorFree(
  RPC_STATS_VECTOR **StatsVector
);
```

Parameters

StatsVector
Pointer to a pointer to a statistics vector. On return, the pointer is set to NULL.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

An application calls the **RpcMgmtStatsVectorFree** function to release the memory used to store statistics.

An application obtains a vector of statistics by calling the **RpcMgmtInqStats** function.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcMgmtInqStats

RpcMgmtStopServerListening

The **RpcMgmtStopServerListening** function tells a server to stop listening for remote procedure calls. This function will not affect auto-listen interfaces. See ***RpcServerRegisterIfEx*** for more details. This server-side function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtStopServerListening(
  RPC_BINDING_HANDLE Binding
);
```

Parameters

Binding

To direct a remote application to stop listening for remote procedure calls, specify a server binding handle for that application. To direct your own (local) application to stop listening for remote procedure calls, specify a value of NULL.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.

Remarks

An application calls the **RpcMgmtStopServerListening** function to direct a server to stop listening for remote procedure calls. If *DontWait* was TRUE, the application should call **RpcMgmtWaitServerListen** to wait for all calls to complete.

When it receives a stop-listening request, the RPC run-time library stops accepting new remote procedure calls for all registered interfaces. Executing calls are allowed to complete, including callbacks. After all calls complete, the **RpcServerListen** function returns to the caller. If *DontWait* is TRUE, the application calls **RpcMgmtWaitServerListen** for all calls to complete.

Note From the client-side, **RpcMgmtStopServerListening** is disabled by default. To enable this function, create an authorization function in your server application that returns TRUE (to allow a remote shutdown) whenever **RpcMgmtStopServerListening** is called. Use **RpcMgmtSetAuthorizationFn** to give the client access to the management function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcEpResolveBinding, **RpcMgmtWaitServerListen**, **RpcServerListen**, **RpcServerRegisterIfEx**

RpcMgmtWaitServerListen

The **RpcMgmtWaitServerListen** function performs the **wait** operation usually associated with **RpcServerListen**. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcMgmtWaitServerListen(VOID);
```

Parameters

This function has no parameters.

Return Values

Value	Meaning
RPC_S_OK	All remote procedure calls are complete.
RPC_S_ALREADY_LISTENING	Another thread has called **RpcMgmtWaitServerListen** and has not yet returned.
RPC_S_NOT_LISTENING	The server application must call **RpcServerListen** before calling **RpcMgmtWaitServerListen**.

Remarks

When the **RpcServerListen** flag parameter *DontWait* has a nonzero value, the **RpcServerListen** function returns to the server application without performing the wait operation. In this case, the wait can be performed by **RpcMgmtWaitServerListen**.

Applications must call **RpcServerListen** with a nonzero value for the *DontWait* parameter before calling **RpcMgmtWaitServerListen**. The **RpcMgmtWaitServerListen** function returns after the server application calls **RpcMgmtStopServerListening** and all active remote procedure calls complete, or after a fatal error occurs in the RPC run-time library.

Note **RpcMgmtWaitServerListen** is a Microsoft extension to the DCE API set.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcMgmtStopServerListening, **RpcServerListen**

RpcNetworkInqProtseqs

The **RpcNetworkInqProtseqs** function returns all protocol sequences supported by both the RPC run-time library and the operating system. This function is supported, for servers, on all 32-bit Windows platforms, except Windows CE. Client applications should use **RpcNetworkIsProtseqValid** For a list of Microsoft RPC's supported protocol sequences, see *String Binding*.

```
RPC_STATUS RPC_ENTRY RpcNetworkInqProtseqs(
  RPC_PROTSEQ_VECTOR **ProtSeqVector
);
```

Parameters

ProtSeqVector
Returns a pointer to a pointer to a protocol sequence vector.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_PROTSEQS	No supported protocol sequences.

Remarks

A server application calls the **RpcNetworkInqProtseqs** function to obtain a vector containing the protocol sequences supported by both the RPC run-time library and the operating system. If there are no supported protocol sequences, this function returns the RPC_S_NO_PROTSEQS status code and a *ProtSeqVector* argument value of NULL.

The server is responsible for calling the **RpcProtseqVectorFree** function to release the memory used by the vector.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

RpcNetworkIsProtseqValid

The **RpcNetworkIsProtseqValid** function tells whether the specified protocol sequence is supported by both the RPC run-time library and the operating system. The function is supported, for clients, on all 32-bit Windows platforms, except Windows CE. Server applications should use **RpcNetworkInqProtseqs**.

```
RPC_STATUS RPC_ENTRY RpcNetworkIsProtseqValid(
  unsigned char *Protseq
);
```

Parameters

Protseq

Pointer to a string identifier of the protocol sequence to be checked.

If the *Protseq* argument is not a valid protocol sequence string, **RpcNetworkIsProtseqValid** returns RPC_S_INVALID_RPC_PROTSEQ.

Return Values

Value	Meaning
RPC_S_OK	Call successful.; protocol sequence supported
RPC_S_PROTSEQ_NOT_SUPPORTED	Protocol sequence not supported on this host.
RPC_S_INVALID_RPC_PROTSEQ	Invalid protocol sequence.

Remarks

An application calls the **RpcNetworkIsProtseqValid** function to determine whether an individual protocol sequence is available for making remote procedure calls.

A protocol sequence is valid if both the RPC run-time library and the operating system support the specified protocols. For a list of Microsoft RPC's supported protocol sequences, see *String Binding*. An application calls **RpcNetworkInqProtseqs** to see all of the supported protocol sequences.

Note **RpcNetworkIsProtseqValid** is available for client applications, not for server applications.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNetworkInqProtseqs

RpcNsBindingExport

The **RpcNsBindingExport** function establishes a name service–database entry with multiple binding handles and multiple objects for a server.

```
RPC_STATUS RPC_ENTRY RpcNsBindingExport(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName,
  RPC_IF_HANDLE IfSpec,
  RPC_BINDING_VECTOR *BindingVec,
  UUID_VECTOR *ObjectUuidVec
);
```

Parameters

EntryNameSyntax
Indicates the syntax of the *EntryName* argument.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName
Pointer to the entry name to which binding handles and object UUIDs are exported. You cannot provide a null or empty string. The client and the server must both use the same entry name.

IfSpec
Specifies a stub-generated data structure specifying the interface to export. A null value indicates there are no binding handles to export (only object UUIDs are to be exported) and *BindingVec* is ignored.

BindingVec
Pointer to server bindings to export. A null value indicates there are no binding handles to export (only object UUIDs are to be exported).

ObjectUuidVec
Pointer to a vector of object UUIDs offered by the server. The server application constructs this vector. A null value indicates there are no object UUIDs to export (only binding handles are to be exported).

Return Values

Value	**Meaning**
RPC_S_OK	Call successful.
RPC_S_NOTHING_TO_EXPORT	Nothing to export.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.

Value	Meaning
RPC_S_NO_NS_PRIVILEGE	No privilege for name-service operation.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsBindingExport** function allows a server application to publicly offer an interface in the name-service database for use by any client application.

Effective with Windows 2000, the RPC run-time environment uses the Active Directory as its name-service database. This means that authorized exported entries persist in the name service, and are visible even after rebooting. Unauthorized exports do not persist. See *Access Control* in the Security section of the Microsoft Platform SDK for more information on authorization and Access Control Lists.

To export an interface, the server application calls the **RpcNsBindingExport** routine with an interface and the server binding handles a client can use to access the server. A server application also calls the **RpcNsBindingExport** function to publicly offer the object UUID(s) of resource(s) it offers, if any, in the name-service database.

A server can export interfaces and objects in a single call to **RpcNsBindingExport**, or it can export them separately. If the name-service database entry specified by *EntryName* does not exist, **RpcNsBindingExport** tries to create it. In this case, the server application must have the privilege to create the entry. In addition to calling **RpcNsBindingExport**, a server that called the **RpcServerUseAllProtseqs** or **RpcServerUseProtseq** function must also register with the local endpoint-map database by calling either **RpcEpRegister** or **RpcEpRegisterNoReplace**.

A server is not required to export any of its interfaces to the name-service database. When a server does not export, only clients that privately know that server's binding information can access its interfaces. For example, a client that has the information needed to construct a string binding can call the **RpcBindingFromStringBinding** to create a binding handle for making remote procedure calls to a server.

Before calling **RpcNsBindingExport**, a server must do the following:

- Register one or more protocol sequences with the local RPC run-time library by calling one of the following functions:
  - **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsEx**
  - **RpcServerUseProtseq**, **RpcServerUseProtseqEx**
  - **RpcServerUseAllProtseqsIf**, **RpcServerUseAllProtseqsIfEx**
  - **RpcServerUseProtseqIf**, **RpcServerUseProtseqIf**
  - **RpcServerUseProtseqEp**, **RpcServerUseProtseqEp**
- Obtain a list of server bindings by calling the **RpcServerInqBindings** function.

The vector returned from the **RpcServerInqBindings** function becomes the *Binding* argument for **RpcNsBindingExport**. To prevent a binding from being exported, set the selected vector element to a null value.

If a server exports to the same name-service database entry multiple times, the second and subsequent calls to **RpcNsBindingExport** add the binding information and object UUIDs when that data is different from the binding information already in the server entry. Existing data is not removed from the entry.

To remove binding handles and object UUIDs from the name-service database, a server application calls the **RpcNsBindingUnexport** function.

A server entry must have at least one binding handle to exist. As a result, exporting only UUIDs to a non-existing entry has no effect, and unexporting all binding handles deletes the entry.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingFromStringBinding, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingUnexport**, **RpcServerInqBindings**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

RpcNsBindingExportPnP

The **RpcNsBindingExportPnP** function establishes a name-service database entry with multiple binding handles and multiple objects for a server that supports Plug and Play.

```
RPC_STATUS RPC_ENTRY RpcNsBindingExportPnP(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName,
  RPC_IF_HANDLE IfSpec,
  UUID_VECTOR *ObjectUuidVec
);
```

Parameters

EntryNameSyntax

Indicates the syntax of the *EntryName* argument.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName

Pointer to the entry name to which binding handles and object UUIDs are exported. You cannot provide a null or empty string.

To use the entry name specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultEntry**, provide a null pointer or an empty string. In this case, the *EntryNameSyntax* parameter is ignored and the run-time library uses the default syntax.

IfSpec

Specifies a stub-generated data structure specifying the interface to export. A null value indicates there are no binding handles to export (only object UUIDs are to be exported) and *BindingVec* is ignored.

ObjectUuidVec

Pointer to a vector of object UUIDs offered by the server. The server application constructs this vector. A null value indicates there are no object UUIDs to export (only binding handles are to be exported).

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NOTHING_TO_EXPORT	Nothing to export.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_NO_NS_PRIVILEGE	No privilege for name-service operation.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsBindingExportPnP** function allows a server application to publicly offer an interface in the name-service database that supports Plug and Play bindings for use by any client application.

Note that the server application should not explicitly supply the binding vector when exporting Plug and Play bindings. The bindings are automatically updated when there is a change in the bindings due to a Plug and Play event.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 OSR2 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on all platforms.

See Also

RpcNsBindingExport, **RpcNsBindingUnexportPnP**

RpcNsBindingImportBegin

The **RpcNsBindingImportBegin** function creates an import context for importing client-compatible binding handles for servers that offer the specified interface and object.

```
RPC_STATUS RPC_ENTRY RpcNsBindingImportBegin(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName,
  RPC_IF_HANDLE IfSpec,
  UUID *ObjUuid,
  RPC_NS_HANDLE *ImportContext
);
```

Parameters

EntryNameSyntax
Indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, specify RPC_C_NS_SYNTAX_DEFAULT.

EntryName
Pointer to an entry name at which the search for compatible binding handles begins.

To use the entry name specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultEntry**, provide a null pointer or an empty string. In this case, the *EntryNameSyntax* parameter is ignored and the run-time library uses the default syntax.

IfSpec

Specifies a stub-generated data structure indicating the interface to import. If the interface specification has not been exported or is of no concern to the caller, specify a null value for this argument. In this case, the bindings returned are only guaranteed to be of a compatible and supported protocol sequence and to contain the specified object UUID. The contacted server might not support the desired interface.

ObjUuid

Pointer to an optional object UUID.

For a nonzero UUID, compatible binding handles are returned from an entry only if the server has exported the specified object UUID.

When *ObjUuid* has a null pointer value or a nil UUID, the returned binding handles contain one of the object UUIDs exported by the compatible server. If the server did not export any object UUIDs, the returned compatible binding handles contain a nil object UUID.

ImportContext

Specifies a returned name-service handle for use with the **RpcNsBindingImportNext** and **RpcNsBindingImportDone** functions.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.
RPC_S_INVALID_OBJECT	Invalid object.

Remarks

Before calling the **RpcNsBindingImportNext** function, the client application must first call **RpcNsBindingImportBegin** to create an import context. The arguments to this function control the operation of the **RpcNsBindingImportNext** function.

When finished importing binding handles, the client application calls the **RpcNsBindingImportDone** function to delete the import context.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsBindingImportDone, **RpcNsBindingImportNext**

RpcNsBindingImportDone

The **RpcNsBindingImportDone** function signals that a client has finished looking for a compatible server and deletes the import context.

```
RPC_STATUS RPC_ENTRY RpcNsBindingImportDone(
  RPC_NS_HANDLE *ImportContext
);
```

Parameters

ImportContext

Pointer to a name-service handle to free. The name-service handle *ImportContext* points to is created by calling the **RpcNsBindingImportBegin** function.

An argument value of NULL is returned.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

Typically, a client application calls **RpcNsBindingImportDone** after completing remote procedure calls to a server using a binding handle returned from the **RpcNsBindingImportNext** function. However, a client application is responsible for calling **RpcNsBindingImportDone** for each import context that was created by calling the **RpcNsBindingImportBegin**, regardless of the status returned from **RpcNsBindingImportNext** or the success in making remote procedure calls.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

See Also

RpcNsBindingImportBegin, **RpcNsBindingImportNext**

RpcNsBindingImportNext

The **RpcNsBindingImportNext** function looks up an interface (and optionally an object from a name-service database) and returns a binding handle of a compatible server, if found.

```
RPC_STATUS RPC_ENTRY RpcNsBindingImportNext(
  RPC_NS_HANDLE ImportContext,
  RPC_BINDING_HANDLE *Binding
);
```

Parameters

ImportContext
Specifies a name-service handle returned from the **RpcNsBindingImportBegin** function.

Binding
Returns a pointer to a client-compatible server binding handle for a server.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_MORE_BINDINGS	No more bindings.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsBindingImportNext** function returns one client-compatible server binding handle for a server that offers the interface and object UUID specified by the *IfSpec* and *ObjUuid* arguments in the **RpcNsBindingImportBegin** function. The function communicates only with the name-service database, not directly with servers.

Effective with Windows 2000, the RPC environment uses the Active Directory as its name-service database and the order in which the run-time environment performs the search is as follows:

- Search in the local cache. If there is no entry,
- Search in the Active Directory. If there is no entry,
- Send broadcast requests to all other directory services in the domain.

  Note that if the entry exists in the Active Directory, but there is no information associated with the entry, the run-time environment does not issue this broadcast request.

The compatible binding handle that is returned *always* contains an object UUID, the value of which depends on the *ObjUuid* argument in the **RpcNsBindingImportBegin** function. If a non-null object UUID was specified, the returned binding handle contains that object UUID. If, however, a null object UUID or null value was specified, the object UUID that is returned is a result of the following possibilities:

- If the server did not export any object UUIDs, the returned binding handle contains a nil object UUID.
- If the server exported one object UUID, the returned binding handle contains that object UUID.
- If the server exported multiple object UUIDs, the returned binding handle contains one of the object UUIDs. The import-next operation selects the returned object UUID in a non-deterministic fashion. As a result, a different object UUID can be returned for each compatible binding handle from a single server entry.

The **RpcNsBindingImportNext** function selects and returns one server binding handle from the compatible binding handles found. The client application can use that binding handle to attempt a remote procedure call to the server. If the client fails to establish a relationship with the server, it can call **RpcNsBindingImportNext** again.

Each time the client calls **RpcNsBindingImportNext**, the function returns another server binding handle. The returned binding handles are unordered. A client application calls the **RpcNsBindingInqEntryName** function to obtain the name-service database in the entry name from which the binding handle came. When the search reaches the end of the name-service database, **RpcNsBindingInqEntryName** returns a status of RPC_S_NO_MORE_BINDINGS and returns a binding argument value of NULL.

The **RpcNsBindingImportNext** function allocates storage for the data referenced by the returned *Binding* argument. When a client application finishes with the binding handle, it must call **RpcBindingFree** to deallocate the storage. Each call to **RpcNsBindingImportNext** requires a corresponding call to **RpcBindingFree**.

The client is responsible for calling the **RpcNsBindingImportDone** function, which deletes the import context. The client also calls **RpcNsBindingImportDone** before calling **RpcNsBindingImportBegin** to start a new search for compatible servers. Because the order of binding handles returned is different for each new search, the order in which binding handles are returned to an application can be different each time the application is run.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

+ See Also

RpcBindingFree, **RpcNsBindingImportBegin**, **RpcNsBindingImportDone**, **RpcNsBindingInqEntryName**, **RpcNsBindingLookupBegin**, **RpcNsBindingLookupDone**, **RpcNsBindingLookupNext**, **RpcNsBindingSelect**

RpcNsBindingInqEntryName

The **RpcNsBindingInqEntryName** function returns the entry name from which the binding handle came.

```
RPC_STATUS RPC_ENTRY RpcNsBindingInqEntryName(
  RPC_BINDING_HANDLE Binding,
  unsigned long EntryNameSyntax,
  unsigned char **EntryName
);
```

Parameters

Binding
Specifies the binding handle whose name-service database entry name is returned.

EntryNameSyntax
Indicates the syntax used in the *EntryName* argument.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName
Returns the address of a pointer to the name of the name-service database entry in which *Binding* was found.

Specify a null value to prevent **RpcNsBindingInqEntryName** from returning the *EntryName* argument. In this case, the application does not call the **RpcStringFree** function.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_NO_ENTRY_NAME	No entry name for binding.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.

Remarks

The **RpcNsBindingInqEntryName** function returns the name of the name service–database entry name from which a client compatible–binding handle came.

The RPC run-time library allocates memory for the string returned in the *EntryName* argument. The application is responsible for calling the **RpcStringFree** function to deallocate that memory.

An entry name is associated only with binding handles returned from the **RpcNsBindingImportNext**, **RpcNsBindingLookupNext**, and **RpcNsBindingSelect** functions.

If the binding handle specified in the *Binding* argument was not returned from a name-service database entry (for example, if the binding handle was created by calling **RpcBindingFromStringBinding**), **RpcNsBindingInqEntryName** returns an empty string ("\0") and an RPC_S_NO_ENTRY_NAME status code.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingFromStringBinding, **RpcNsBindingImportNext**, **RpcNsBindingLookupNext**, **RpcNsBindingSelect**, **RpcStringFree**

RpcNsBindingLookupBegin

The **RpcNsBindingLookupBegin** function creates a lookup context for an interface and an object.

```
RPC_STATUS RPC_ENTRY RpcNsBindingLookupBegin(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName,
  RPC_IF_HANDLE IfSpec,
  UUID *ObjUuid,
  unsigned long BindingMaxCount,
  RPC_NS_HANDLE *LookupContext
);
```

Parameters

EntryNameSyntax

Indicates the syntax of the *EntryName* argument.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName

Pointer to an entry name at which the search for compatible bindings begins.

To use the entry name specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultEntry**, provide a null pointer or an empty string. In this case, the *EntryNameSyntax* parameter is ignored and the run-time library uses the default syntax.

IfSpec

Specifies a stub-generated structure indicating the interface to look up. If the interface specification has not been exported or is of no concern to the caller, specify a null value for this argument. In this case, the bindings returned are only guaranteed to be of a compatible and supported protocol sequence and to contain the specified object UUID. The desired interface might not be supported by the contacted server.

ObjUuid

Pointer to an optional object UUID.

For a nonzero UUID, compatible binding handles are returned from an entry only if the server has exported the specified object UUID.

For a null pointer value or a nil UUID for this argument, the returned binding handles contain one of the object UUIDs exported by the compatible server. If the server did not export any object UUIDs, the returned compatible binding handles contain a nil object UUID.

BindingMaxCount

Specifies the maximum number of bindings to return in the *BindingVec* argument from the **RpcNsBindingLookupNext** function.

Specify a value of zero to use the default count of RPC_C_BINDING_MAX_COUNT_DEFAULT.

LookupContext

Returns a pointer to a name-service handle for use with the **RpcNsBindingLookupNext** and **RpcNsBindingLookupDone** functions.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.

(continued)

(continued)

Value	Meaning
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.
RPC_S_INVALID_OBJECT	Invalid object.

Remarks

The **RpcNsBindingLookupBegin** function creates a lookup context for locating client-compatible binding handles to servers that offer the specified interface and object.

Before calling **RpcNsBindingLookupNext**, the client application must first call **RpcNsBindingLookupBegin** to create a lookup context. The arguments to this function control the operation of the **RpcNsBindingLookupNext** function.

Effective with Windows 2000, the RPC environment uses the Active Directory as its name-service database and the order in which the run-time environment performs the search is as follows:

- Search in the local cache.
- If entry not found in local cache, search that machine's Active Directory.
- If entry not found on local machine, send broadcast requests to all other Active Directory services in the domain.

  Note that if the entry exists in the Active Directory, but there is no information associated with the entry, the run-time environment will not issue this broadcast request.

When finished locating binding handles, the client application calls the **RpcNsBindingLookupDone** function to delete the lookup context.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsBindingLookupDone, **RpcNsBindingLookupNext**

RpcNsBindingLookupDone

The **RpcNsBindingLookupDone** function signifies that a client has finished looking for compatible servers and deletes the lookup context.

```
RPC_STATUS RPC_ENTRY RpcNsBindingLookupDone(
  RPC_NS_HANDLE *LookupContext
);
```

Parameters

LookupContext

Pointer to the name-service handle to free. The name-service handle *LookupContext* points to is created by calling the **RpcNsBindingLookupBegin** function.

An argument value of NULL is returned.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

The **RpcNsBindingLookupDone** function frees a lookup context created by calling the **RpcNsBindingLookupBegin** function.

Typically, a client application calls **RpcNsBindingLookupDone** after completing remote procedure calls to a server using a binding handle returned from the **RpcNsBindingLookupNext** function. However, a client application is responsible for calling **RpcNsBindingLookupDone** for each created lookup context, regardless of the status returned from the **RpcNsBindingLookupNext** function or the success in making remote procedure calls.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

See Also

RpcNsBindingLookupBegin, **RpcNsBindingLookupNext**

RpcNsBindingLookupNext

The **RpcNsBindingLookupNext** function returns a list of compatible binding handles for a specified interface and optionally an object.

```
RPC_STATUS RPC_ENTRY RpcNsBindingLookupNext(
  RPC_NS_HANDLE LookupContext,
  RPC_BINDING_VECTOR **BindingVec
);
```

Parameters

LookupContext
Specifies the name-service handle returned from the **RpcNsBindingLookupBegin** function.

BindingVec
Returns the address of a pointer to a vector of client-compatible server binding handles.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_MORE_BINDINGS	No more bindings.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name-service unavailable.

Remarks

The **RpcNsBindingLookupNext** function returns a vector of client-compatible server binding handles for a server offering the interface and object UUID specified by the *IfSpec* and *ObjUuid* parameters in the **RpcNsBindingLookupBegin** function. (Compare this to **RpcNsBindingImportNext**, which returns a single compatible server binding handle.)

The **RpcNsBindingLookupNext** function communicates only with the name-service database, not directly with servers.

Effective with Windows 2000, the RPC environment uses Active Directory as its name-service database and the order in which the run-time environment performs the search is as follows:

- Search the local cache.
- If entry not found in local cache, search that machine's Active Directory.
- If entry not found on local machine, send broadcast requests to all other Active Directory services in the domain.

  Note that if the entry exists in the Active Directory, but there is no information associated with the entry, the run-time environment will not issue this broadcast request.

On successive calls, the **RpcNsBindingLookupNext** function traverses name-service database entries, collecting client-compatible server binding handles from each entry.

When the Microsoft® Active Directory is the name-service database, **RpcNsBindingLookupNext** traverses the database only if the given entry name is null and the default entry (in the registry) is undefined or empty. Also, since mixed entries are not permitted in the Active Directory, the function searches for server entry names only, not group or profile names.

When the DCE Cell Directory Service (CDS) is the name-service database, and the entry at which the search begins contains binding handles in addition to group or profile names, **RpcNsBindingLookupNext** returns the binding handles from *EntryName* before searching the group or profile. This means that the function can return a partially full vector before processing the members of the group or profile.

The compatible binding handle that is returned *always* contains an object UUID, the value of which depends on the *ObjUuid* argument in the **RpcNsBindingImportBegin** function. If a non-null object UUID was specified, the returned binding handle contains that object UUID. If, however, a null object UUID or null value was specified, the object UUID that is returned is a result of the following possibilities:

- If the server did not export any object UUIDs, the returned binding handle contains a nil object UUID.
- If the server exported one object UUID, the returned binding handle contains that object UUID.
- If the server exported multiple object UUIDs, the returned binding handle contains one of the object UUIDs. The import-next operation selects the returned object UUID in a non-deterministic fashion. As a result, a different object UUID can be returned for each compatible binding handle from a single server entry.

From the returned vector of server binding handles, the client application can employ its own criteria for selecting individual binding handles, or the application can call the **RpcNsBindingSelect** function to select a binding handle. The **RpcBindingToStringBinding** and **RpcStringBindingParse** functions will be helpful to a client creating its own selection criteria.

The client application can use the selected binding handle to attempt to make a remote procedure call to the server. If the client fails to establish a relationship with the server, it can select another binding handle from the vector. When all of the binding handles in the vector have been used, the client application calls **RpcNsBindingLookupNext** again.

Each time the client calls **RpcNsBindingLookupNext**, the function returns another vector of binding handles. The binding handles returned in each vector are unordered. The vectors returned from multiple calls to this function are also unordered.

A client calls the **RpcNsBindingInqEntryName** function to obtain the name-service database server entry name that the binding came from.

When the search reaches the end of the name-service database, **RpcNsBindingLookupNext** returns a status of RPC_S_NO_MORE_BINDINGS and returns a *BindingVec* value of NULL.

The **RpcNsBindingLookupNext** function allocates storage for the data referenced by the returned *BindingVec* argument. When a client application finishes with the vector, it must call the **RpcBindingVectorFree** function to deallocate the storage. Each call to **RpcNsBindingLookupNext** requires a corresponding call to **RpcBindingVectorFree**.

The client is responsible for calling the **RpcNsBindingLookupDone** function to delete the lookup context, or if you want the application to start a new search for compatible servers.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

See Also

RpcBindingToStringBinding, **RpcBindingVectorFree**, **RpcNsBindingInqEntryName**, **RpcNsBindingLookupBegin**, **RpcNsBindingLookupDone**, **RpcStringBindingParse**

RpcNsBindingSelect

The **RpcNsBindingSelect** function returns a binding handle from a list of compatible binding handles.

```
RPC_STATUS RPC_ENTRY RpcNsBindingSelect(
  RPC_BINDING_VECTOR *BindingVec,
  RPC_BINDING_HANDLE *Binding
);
```

Parameters

BindingVec
: Pointer to the vector of client-compatible server binding handles from which a binding handle is selected. The returned binding vector no longer references the selected binding handle, which is returned separately in the *Binding* argument.

Binding
: Pointer to a selected binding handle.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_MORE_BINDINGS	No more bindings.

Remarks

Each time the client calls the **RpcNsBindingSelect** function, the function operation returns another binding handle from the vector.

When all of the binding handles have been returned from the vector, the function returns a status of RPC_S_NO_MORE_BINDINGS and returns a *Binding* value of NULL.

The **select** operation allocates storage for the data referenced by the returned *Binding* parameter. When a client finishes with the binding handle, it should call the **RpcBindingFree** function to deallocate the storage. Each call to **RpcNsBindingSelect** requires a corresponding call to the **RpcBindingFree** function.

Clients can create their own select routines implementing application-specific selection criteria. In this case, **RpcStringBindingParse** provides access to the fields of a binding.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

+ See Also

RpcBindingFree, **RpcNsBindingLookupNext**, **RpcStringBindingParse**

RpcNsBindingUnexport

The **RpcNsBindingUnexport** function removes the binding handles for an interface and objects from an entry in the name-service database.

```
RPC_STATUS RPC_ENTRY RpcNsBindingUnexport(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName,
  RPC_IF_HANDLE IfSpec,
  UUID_VECTOR *ObjectUuidVec
);
```

Parameters

EntryNameSyntax

Indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName

Pointer to the entry name from which to remove binding handles and object UUIDs.

IfSpec

An interface specification for the binding handles to be removed from the name service database. A null argument value indicates not to unexport any binding handles (only object UUIDs are to be unexported).

ObjectUuidVec

Pointer to a vector of object UUIDs that the server no longer wants to offer. The application constructs this vector. A null value indicates there are no object UUIDs to unexport (only binding handles are to be unexported).

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_VERS_OPTION	Invalid version option.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.
RPC_S_INTERFACE_NOT_FOUND	Interface not found.
RPC_S_NOT_ALL_OBJS_UNEXPORTED	Not all objects unexported.

Remarks

The **RpcNsBindingUnexport** function allows a server application to remove the binding handles and object UUIDs of resources from a name service database entry. A server application can unexport the specified interface and objects in a single call to **RpcNsBindingUnexport**, or it can unexport them separately. Only the binding handles that match the interface UUID and the major and minor interface version numbers found in the *IfSpec* argument are unexported. Use the **RpcNsMgmtBindingUnexport** function to remove multiple versions of an interface.

Effective with Windows 2000, the RPC run-time environment uses the Active Directory as its name-service database. This means that an authorized unexported entries will be removed both from the local cache and from the Active Directory.

Unauthorized unexports will only be removed from the local cache. See *Access Control* in the Security section of the Platform SDK documentation for more information on authorization and Access Control Lists.

If **RpcNsBindingUnexport** does not find any binding handles for the specified interface, the function returns an RPC_S_INTERFACE_NOT_FOUND status code and does not unexport the object UUIDs, if any were specified.

If one or more binding handles for the specified interface are found and unexported without error, **RpcNsBindingUnexport** unexports the specified object UUIDs, if any.

If any of the specified object UUIDs were not found, **RpcNsBindingUnexport** returns the RPC_S_NOT_ALL_OBJS_UNEXPORTED status code.

In addition to calling **RpcNsBindingUnexport**, a server should also call the **RpcEpUnregister** function to unregister the endpoints the server previously registered with the local endpoint-map database.

Once created, a server entry persists, even when all of the binding handles and UUIDs are removed. A server entry must have at least one binding handle to exist. As a result, exporting only UUIDs to a nonexisting entry has no effect, and unexporting all binding handles deletes the entry.

Use **RpcNsBindingUnexport** judiciously. To keep an automatically activated server available, you must leave its binding handles in the name-service database between the times when server processes are activated. However, with dynamic bindings, if you do not unexport binding handles, the Active Directory can become so large as to be unmanageable.

Therefore, before you call this function, keep in mind how long you expect the server to be unavailable, and the type of binding in use. If you are using static bindings, reserve this function for when you expect a server to be unavailable for an extended time—for example, when it is being permanently removed from service.

Note Name-service databases are designed to be relatively stable. In replicated name-service databases, frequent use of the **RpcNsBindingExport** and **RpcNsBindingUnexport** functions causes the name-service database to repeatedly remove and replace the same entry and can cause performance problems.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcEpUnregister, **RpcNsBindingExport**

RpcNsBindingUnexportPnP

The **RpcNsBindingUnexportPnP** function removes the binding handles for Plug and Play interfaces and objects from an entry in the name-service database.

```
RPC_STATUS RPC_ENTRY RpcNsBindingUnexportPnP(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName,
  RPC_IF_HANDLE IfSpec,
  UUID_VECTOR *ObjectUuidVec
);
```

Parameters

EntryNameSyntax
Indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName
Pointer to the entry name from which to remove binding handles and object UUIDs.

IfSpec
An interface specification for the binding handles to be removed from the name service database. A null argument value indicates not to unexport any binding handles (only object UUIDs are to be unexported).

ObjectUuidVec
Pointer to a vector of object UUIDs that the server no longer wants to offer. The application constructs this vector. A null value indicates there are no object UUIDs to unexport (only binding handles are to be unexported).

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_VERS_OPTION	Invalid version option.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.

Value	Meaning
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.
RPC_S_INTERFACE_NOT_FOUND	Interface not found.
RPC_S_NOT_ALL_OBJS_UNEXPORTED	Not all objects unexported.

Remarks

The **RpcNsBindingUnexportPnP** function allows a server application to remove the binding handles and object UUIDs of Plug and Play-compatible resources from a name service database entry. A server application can unexport the specified interface and objects in a single call to **RpcNsBindingUnexportPnP**, or it can unexport them separately. Only the binding handles that match the interface UUID and the major and minor interface version numbers found in the *IfSpec* argument are unexported.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 OSR2 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on all platforms.

See Also

RpcNsBindingUnexport, **RpcNsBindingExportPnP**

RpcNsEntryExpandName

The **RpcNsEntryExpandName** function expands a name-service entry name. This function is supported by the DCE Cell Directory Service and the Active Directory in Windows 2000, and Windows 98.

```
RPC_STATUS RPC_ENTRY RpcNsEntryExpandName(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName,
  unsigned char **ExpandedName
);
```

Parameters

EntryNameSyntax

Indicates the syntax of the *EntryName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName
Pointer to the entry name to expand.

ExpandedName
Returns a pointer to a pointer to the expanded version of *EntryName*.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INCOMPLETE_NAME	Incomplete name.

Remarks

An application calls the **RpcNsEntryExpandName** function to obtain a fully expanded entry name.

The RPC run-time library allocates memory for the returned *ExpandedName* parameter. The application is responsible for calling the **RpcStringFree** function for that returned string.

The returned expanded entry name accounts for local name translations and for differences in locally defined naming schema.

Note The Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcStringFree

RpcNsEntryObjectInqBegin

The **RpcNsEntryObjectInqBegin** function creates an inquiry context for the objects of a name-service database entry.

```
RPC_STATUS RPC_ENTRY RpcNsEntryObjectInqBegin(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName,
  RPC_NS_HANDLE *InquiryContext
);
```

Parameters

EntryNameSyntax

Indicates the syntax to use in the *EntryName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName

Pointer to the name-service database entry name for which object UUIDs are to be viewed.

InquiryContext

Returns a pointer to a name-service handle for use with the **RpcNsEntryObjectInqNext** and **RpcNsEntryObjectInqDone** functions.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsEntryObjectInqBegin** function creates an inquiry context for viewing the object UUIDs exported to *EntryName*.

Before calling the **RpcNsEntryObjectInqNext** function, the application must first call **RpcNsEntryObjectInqBegin** to create an inquiry context.

When finished viewing the object UUIDs, the application calls the **RpcNsEntryObjectInqDone** function to delete the inquiry context.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsBindingExport, **RpcNsEntryObjectInqDone**, **RpcNsEntryObjectInqNext**

RpcNsEntryObjectInqDone

The **RpcNsEntryObjectInqDone** function deletes the inquiry context for a name-service database entry's objects.

```
RPC_STATUS RPC_ENTRY RpcNsEntryObjectInqDone(
  RPC_NS_HANDLE *InquiryContext
);
```

Parameters

InquiryContext

Pointer to a name-service handle specifying the object UUIDs exported to the *EntryName* parameter specified in the **RpcNsEntryObjectInqBegin** function.

An argument value of NULL is returned.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

The **RpcNsEntryObjectInqDone** function frees an inquiry context created by calling the **RpcNsEntryObjectInqBegin** function.

An application calls **RpcNsEntryObjectInqDone** after viewing exported object UUIDs using the **RpcNsEntryObjectInqNext** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

See Also

RpcNsEntryObjectInqBegin, **RpcNsEntryObjectInqNext**

RpcNsEntryObjectInqNext

The **RpcNsEntryObjectInqNext** function returns one object at a time from a name-service database entry.

```
RPC_STATUS RPC_ENTRY RpcNsEntryObjectInqNext(
  RPC_NS_HANDLE InquiryContext,
  UUID *ObjUuid
);
```

Parameters

InquiryContext
Specifies a name-service handle that indicates the object UUIDs for a name-service database entry.

ObjUuid
Returns a pointer to an exported object UUID.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_MORE_MEMBERS	No more members.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name-service unavailable.

Remarks

The **RpcNsEntryObjectInqNext** function returns one of the object UUIDs exported to the name-service database entry specified by the *EntryName* parameter in the **RpcNsEntryObjectInqBegin** function.

An application can view all of the exported object UUIDs by repeatedly calling **RpcNsEntryObjectInqNext**. When all the object UUIDs have been viewed, this function returns an RPC_S_NO_MORE_MEMBERS status code. The returned object UUIDs are unordered.

The application supplies the memory for the object UUID returned in the *ObjUuid* parameter.

After viewing the object UUIDs, the application must call the **RpcNsEntryObjectInqDone** function to release the inquiry context.

The order in which object UUIDs are returned can be different for each viewing of an entry. This means that the order in which object UUIDs are returned to an application can be different each time the application is run.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

+ See Also

RpcNsBindingExport, **RpcNsEntryObjectInqBegin**, **RpcNsEntryObjectInqDone**

RpcNsGroupDelete

The **RpcNsGroupDelete** function deletes a group attribute.

```
RPC_STATUS RPC_ENTRY RpcNsGroupDelete(
  unsigned long GroupNameSyntax,
  unsigned char *GroupName
);
```

Parameters

GroupNameSyntax
Specifies an integer value that indicates the syntax of the next parameter, *GroupName*. This parameter can be set to one of the following values:

Value	Meaning
RPC_C_NS_SYNTAX_DEFAULT	Use the syntax specified in the registry value **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**
RPC_C_NS_SYNTAX_DCE	Use DCE syntax.

GroupName
Pointer to the name of the name-service group to delete.

Return Values

This function returns one of the following values.

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsGroupDelete** function deletes the group attribute from the specified name service–database entry.

Neither the specified name service–database entry nor the group members are deleted.

Note This DCE function is not supported by Microsoft Locator. Windows NT and Windows 2000 support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsGroupMbrAdd, **RpcNsGroupMbrRemove**

RpcNsGroupMbrAdd

The **RpcNsGroupMbrAdd** function adds an entry name to a group. If necessary, it creates the entry.

```
RPC_STATUS RPC_ENTRY RpcNsGroupMbrAdd(
  unsigned long GroupNameSyntax,
  unsigned char *GroupName,
  unsigned long MemberNameSyntax,
  unsigned char *MemberName
);
```

Parameters

GroupNameSyntax

Indicates the syntax of the *GroupName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

GroupName

Pointer to the name of the RPC group to receive a new member.

MemberNameSyntax

Indicates the syntax to use in the *MemberName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

MemberName

Pointer to the name of the new RPC group member.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsGroupMbrAdd** adds a name service–database entry name as a member to the RPC group attribute.

If the *GroupName* entry does not exist, **RpcNsGroupMbrAdd** tries to create the entry with a group attribute and adds the group member specified by *MemberName*. In this case, the application must have the privilege to create the entry. Otherwise, a management application with the necessary privilege should create the entry by calling **RpcNsMgmtEntryCreate** before the application is run.

Note Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsGroupMbrRemove, **RpcNsMgmtEntryCreate**

RpcNsGroupMbrInqBegin

The **RpcNsGroupMbrInqBegin** function creates an inquiry context for viewing group members.

```
RPC_STATUS RPC_ENTRY RpcNsGroupMbrInqBegin(
  unsigned long GroupNameSyntax,
  unsigned char *GroupName,
  unsigned long MemberNameSyntax,
  RPC_NS_HANDLE *InquiryContext
);
```

Parameters

GroupNameSyntax
Indicates the syntax of the *GroupName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

GroupName
Pointer to the name of the RPC group to view.

MemberNameSyntax
Indicates the syntax of the return argument, *MemberName*, in the **RpcNsGroupMbrInqNext** function.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

InquiryContext
Returns a pointer to a name-service handle for use with the **RpcNsGroupMbrInqNext** and **RpcNsGroupMbrInqDone** functions.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsGroupMbrInqBegin** function creates an inquiry context for viewing the members of an RPC group. Before calling **RpcNsGroupMbrInqNext**, the application must first call **RpcNsGroupMbrInqBegin** to create an inquiry context. When finished viewing the RPC group members, the application calls **RpcNsGroupMbrInqDone** to delete the inquiry context.

Note Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsGroupMbrAdd, **RpcNsGroupMbrInqDone**, **RpcNsGroupMbrInqNext**

RpcNsGroupMbrInqDone

The **RpcNsGroupMbrInqDone** function deletes the inquiry context for a group.

```
RPC_STATUS RPC_ENTRY RpcNsGroupMbrInqDone(
  RPC_NS_HANDLE *InquiryContext
);
```

Parameters

InquiryContext
Pointer to a name-service handle to free. A value of NULL is returned.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NS_HANDLE	Invalid name-service handle.

Remarks

The **RpcNsGroupMbrInqDone** function frees an inquiry context created by calling the **RpcNsGroupMbrInqBegin** function. An application calls **RpcNsGroupMbrInqDone** after viewing RPC group members using the **RpcNsGroupMbrInqNext** function.

Note Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

See Also

RpcNsGroupMbrInqBegin, **RpcNsGroupMbrInqNext**

RpcNsGroupMbrInqNext

The **RpcNsGroupMbrInqNext** function returns one entry name from a group at a time.

```
RPC_STATUS RPC_ENTRY RpcNsGroupMbrInqNext(
  RPC_NS_HANDLE InquiryContext,
  unsigned char **MemberName
);
```

Parameters

InquiryContext
Specifies a name-service handle.

MemberName
Returns the address of a pointer to an RPC group member name. The syntax of the returned name was specified by the *MemberNameSyntax* parameter in the **RpcNsGroupMbrInqBegin** function.

Specify a null value to prevent **RpcNsGroupMbrInqNext** from returning the *MemberName* parameter. In this case, the application does not call the **RpcStringFree** function.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NS_HANDLE	Invalid name-service handle.
RPC_S_NO_MORE_MEMBERS	No more members.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsGroupMbrInqNext** function returns one member of the RPC group specified by the *GroupName* parameter in **RpcNsGroupMbrInqBegin**. An application can view all the members of an RPC group set by repeatedly calling **RpcNsGroupMbrInqNext**. When all the group members have been viewed, this function returns an RPC_S_NO_MORE_MEMBERS status code. The returned group members are unordered.

On each call to **RpcNsGroupMbrInqNext** that returns a member name, the RPC run-time library allocates memory for the returned *MemberName*. The application is responsible for calling **RpcStringFree** for each returned *MemberName* string. After viewing the RPC group's members, the application must call **RpcNsGroupMbrInqDone** to release the inquiry context.

The order in which group members are returned can be different for each viewing of a group. This means that the order in which group members are returned to an application can be different each time the application is run.

Note Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsGroupMbrInqBegin, **RpcNsGroupMbrInqDone**, **RpcStringFree**

RpcNsGroupMbrRemove

The **RpcNsGroupMbrRemove** function removes an entry name from a group.

```
RPC_STATUS RPC_ENTRY RpcNsGroupMbrRemove(
  unsigned long GroupNameSyntax,
  unsigned char *GroupName,
  unsigned long MemberNameSyntax,
  unsigned char *MemberName
);
```

Parameters

GroupNameSyntax

Indicates the syntax of the *GroupName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

GroupName

Pointer to the name of the RPC group from which to remove the member name.

MemberNameSyntax

Indicates the syntax to use in the *MemberName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

MemberName

Pointer to the name of the member to remove from the RPC group attribute in the entry *GroupName*.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.
RPC_S_GROUP_MEMBER_NOT_FOUND	Group member not found.

Remarks

The **RpcNsGroupMbrRemove** function removes a member from the RPC group attribute in the *GroupName* argument.

Note Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsGroupMbrAdd

RpcNsMgmtBindingUnexport

The **RpcNsMgmtBindingUnexport** function removes multiple binding handles and objects from an entry in the name-service database.

```
RPC_STATUS RPC_ENTRY RpcNsMgmtBindingUnexport(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName,
  RPC_IF_ID *IfId,
  unsigned long VersOption,
  UUID_VECTOR *ObjectUuidVec
);
```

Parameters

EntryNameSyntax
Indicates the syntax of the *EntryName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName
Pointer to the name of the entry from which to remove binding handles and object UUIDs.

IfId
Pointer to an interface identification. A null argument value indicates that binding handles are not to be unexported—only object UUIDs are to be unexported.

VersOption
Specifies how the **RpcNsMgmtBindingUnexport** function uses the *VersMajor* and *VersMinor* members of the structure pointed to by the *IfId* parameter.

The following table describes valid values for the *VersOption* parameter.

***VersOption* values**	**Description**
RPC_C_VERS_ALL	Unexports all bindings for the interface UUID in *IfId*, regardless of the version numbers. For this value, specify 0 for both the major and minor versions in *IfId*.
RPC_C_VERS_IF_ID	Unexports the bindings for the compatible interface UUID in *IfId* with the same major version and with a minor version greater than or equal to the minor version in *IfId*.
RPC_C_VERS_EXACT	Unexports the bindings for the interface UUID in *IfId* with the same major and minor versions as in *IfId*.
RPC_C_VERS_MAJOR_ONLY	Unexports the bindings for the interface UUID in *IfId* with the same major version as in *IfId* (ignores the minor version). For this value, specify 0 for the minor version in *IfId*.
RPC_C_VERS_UPTO	Unexports the bindings that offer a version of the specified interface UUID less than or equal to the specified major and minor version. (For example, if the *IfId* contained V2.0 and the name service–database entry contained binding handles with the versions 1.3, 2.0, and 2.1, the **RpcNsMgmtBindingUnexport** function would unexport the binding handles with versions 1.3 and 2.0.)

ObjectUuidVec

Pointer to a vector of object UUIDs that the server no longer wants to offer. The application constructs this vector. A null value indicates there are no object UUIDs to unexport—only binding handles are to be unexported.

Return Values

Value	**Meaning**
RPC_S_OK	Call successful.
RPC_S_INVALID_VERS_OPTION	Invalid version option.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.

(continued)

(continued)

Value	Meaning
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.
RPC_S_INTERFACE_NOT_FOUND	Interface not found.
RPC_S_NOT_ALL_OBJS_UNEXPORTED	Not all objects unexported.

Remarks

The **RpcNsMgmtBindingUnexport** function allows a management application to remove one of the following from a name service–database entry:

- All the binding handles for a specified interface UUID, qualified by the interface version numbers (major and minor)
- One or more object UUIDs of resources
- Both binding handles and object UUIDs of resources

A management application can unexport interfaces and objects in a single call to **RpcNsMgmtBindingUnexport**, or it can unexport them separately. If **RpcNsMgmtBindingUnexport** does not find any binding handles for the specified interface, the function returns an RPC_S_INTERFACE_NOT_FOUND status code and does not unexport the object UUIDs, if any were specified.

If one or more binding handles for the specified interface are found and unexported without error, **RpcNsMgmtBindingUnexport** unexports any specified object UUIDs. If any of the specified object UUIDs were not found, **RpcNsMgmtBindingUnexport** returns RPC_S_NOT_ALL_OBJS_UNEXPORTED.

In addition to calling **RpcNsMgmtBindingUnexport**, a management application should also call the **RpcMgmtEpUnregister** function to unregister the servers that have registered with the endpoint-map database.

Note Name-service databases are designed to be relatively stable. In replicated name services, frequent use of the **RpcNsBindingExport** and **RpcNsBindingUnexport** functions causes the name service to repeatedly remove and replace the same entry, which can cause performance problems.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcMgmtEpUnregister, **RpcNsBindingExport**, **RpcNsBindingUnexport**

RpcNsMgmtEntryCreate

The **RpcNsMgmtEntryCreate** function creates a name service–database entry.

```
RPC_STATUS RPC_ENTRY RpcNsMgmtEntryCreate(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName
);
```

Parameters

EntryNameSyntax

Indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName

Pointer to the name of the entry to create.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_ALREADY_EXISTS	Name-service entry already exists.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsMgmtEntryCreate** function creates an entry in the name-service database. A management application can call **RpcNsMgmtEntryCreate** to create a name service–database entry for use by another application that does not itself have the necessary name service–database privileges to create an entry.

Note Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsMgmtEntryDelete

RpcNsMgmtEntryDelete

The **RpcNsMgmtEntryDelete** function deletes a name service–database entry.

```
RPC_STATUS RPC_ENTRY RpcNsMgmtEntryDelete(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName
);
```

Parameters

EntryNameSyntax
Indicates the syntax of the *EntryName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName
Pointer to the name of the entry to delete.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.
RPC_S_NOT_RPC_ENTRY	Not an RPC entry.

Remarks

Management applications use the **RpcNsMgmtEntryDelete** function only when an entry is no longer needed—for example, when a server is being permanently removed from service.

Because name-service databases are designed to be relatively stable, frequent use of **RpcNsMgmtEntryDelete** in client or server applications can result in performance problems. Creating and deleting entries in client or server applications causes the name-service database to repeatedly remove and replace the same entry. This can lead to performance problems in replicated name-service databases.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsMgmtEntryCreate

RpcNsMgmtEntryInqIfIds

The **RpcNsMgmtEntryInqIfIds** function returns the list of interfaces exported to a name service–database entry. It also returns an interface-identification vector containing the interfaces of binding handles exported by a server to *EntryName*. This function uses an expiration age of 0, causing an immediate update of the local copy of name-service data.

```
RPC_STATUS RPC_ENTRY RpcNsMgmtEntryInqIfIds(
  unsigned long EntryNameSyntax,
  unsigned char *EntryName,
  RPC_IF_ID_VECTOR **IfIdVec
);
```

Parameters

EntryNameSyntax
Indicates the syntax of the *EntryName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

EntryName
Pointer to the name service–database entry name for which an interface-identification vector is returned.

IfIdVec
Returns an address of a pointer to the interface-identification vector.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsMgmtEntryInqIfIds** function returns an interface-identification vector containing the interfaces of binding handles exported by a server to *EntryName*. This function uses an expiration age of 0, causing an immediate update of the local copy of name-service data. The calling application is responsible for calling the **RpcIfIdVectorFree** function to release memory used by the vector.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcIfIdVectorFree, **RpcIfInqId**, **RpcNsBindingExport**

RpcNsMgmtHandleSetExpAge

The **RpcNsMgmtHandleSetExpAge** function sets the expiration age of a name-service handle for local copies of name-service data.

```
RPC_STATUS RPC_ENTRY RpcNsMgmtHandleSetExpAge(
  RPC_NS_HANDLE NsHandle,
  unsigned long ExpirationAge
);
```

Parameters

NsHandle

Specifies a name-service handle for which an expiration age is set. A name-service handle is returned from a name service **begin** operation.

ExpirationAge

Integer value in seconds that sets the expiration age of local name-service data read by all next routines using the specified *NsHandle* parameter.

An expiration age of 0 causes an immediate update of the local name-service data.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsMgmtHandleSetExpAge** function sets a handle-expiration age for a specified name–service handle (*NsHandle*). The expiration age is the amount of time that a local copy of data from a name-service attribute can exist before a request from the application for the attribute requires updating the local copy. When an application begins running, the RPC run-time library specifies a default expiration age of two hours. The default is global to the application. A handle-expiration age applies only to a specific name-service handle and temporarily overrides the current global expiration age.

A handle-expiration age is used exclusively by Pointer **next** operations (which read data from name-service attributes). A **next** operation typically starts by looking for a local copy of the attribute data being requested by an application. In the absence of a local copy, the **next** operation creates one with fresh attribute data from the name-service database. If a local copy already exists, the operation compares its actual age to the expiration age being used by the application (which, in this case, is the expiration age set for the name-service handle). If the actual age exceeds the handle-expiration age, the operation automatically tries to update the local copy with fresh attribute data. If updating is impossible, the old local data remains in place and the **next** operation fails, returning the RPC_S_NAME_SERVICE_UNAVAILABLE status code.

The scope of a handle-expiration age is a single series of **next** operations. The **RpcNsMgmtHandleSetExpAge** function operates within the following context:

- A **begin** operation creates a name-service handle.
- A call to the **RpcNsMgmtHandleSetExpAge** function creates an expiration age for the handle.
- A series of **next** operations for the name-service handle uses the handle expiration age.
- A **done** operation for the name-service handle deletes both the handle and its expiration age.

Tip Typically, you should avoid using **RpcNsMgmtHandleSetExpAge**. Instead, you should rely on the application's global expiration age. Setting the handle-expiration age to a small value causes the name service **next** operations to frequently update local data for any name-service attribute requested by your application. For example, setting the expiration age to 0 forces the **next** operation to update local data for the name-service attribute requested by your application. Therefore, setting a small handle-expiration age can create performance problems for your application. Furthermore, if your application is using a remote name-service server, a small expiration age can adversely affect network performance for all applications.

Limit use of **RpcNsMgmtHandleSetExpAge** to the following situations:

- When you must always get accurate name-service data.

  For example, during management operations to update a profile, you may need to always see the profile's current contents. In this case, before beginning to inquire about a profile, your application should call the **RpcNsMgmtHandleSetExpAge** function and specify 0 for the *ExpirationAge* argument.
- When a request using the default expiration age has failed, and your application needs to retry the operation.

  For example, a client application using name service **import** operations should first try to obtain bindings using the application's default expiration age. However, sometimes the **import-next** operation returns either no binding handles or an insufficient number of them. In this case, the client could retry the **import** operation and, after the **RpcNsBindingImportBegin** call, include an **RpcNsMgmtHandleSetExpAge** call and specify 0 for the *ExpirationAge* argument. When the client calls the **import-next** function again, the small handle-expiration age causes the **import-next** operation to update the local attribute data.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

See Also

RpcNsBindingImportBegin, **RpcNsMgmtInqExpAge**, **RpcNsMgmtSetExpAge**

RpcNsMgmtInqExpAge

The **RpcNsMgmtInqExpAge** function returns the global expiration age for local copies of name-service data.

```
RPC_STATUS RPC_ENTRY RpcNsMgmtInqExpAge(
  unsigned long *ExpirationAge
);
```

Parameters

ExpirationAge

Pointer to the default expiration age, in seconds. This value is used by all name service **next** operations.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

The **RpcNsMgmtInqExpAge** function returns the expiration age that the application is using. The expiration age is the amount of time in seconds that a local copy of data from a name-service attribute can exist before a request from the application for the attribute requires updating the local copy. When an application begins running, the RPC run-time library specifies a default expiration age of two hours. The default is global to the application.

An expiration age is used by Pointer **next** operations (which read data from name-service attributes). A **next** operation typically starts by looking for a local copy of the attribute data being requested by an application. In the absence of a local copy, the **next** operation creates one with fresh attribute data from the name-service database. If a local copy already exists, the operation compares its actual age to the expiration age being used by the application. If the actual age exceeds the expiration age, the operation automatically tries to update the local copy with fresh attribute data. If updating is impossible, the old local data remains in place and the **next** operation fails.

Applications typically should use only the default expiration age. For special cases, however, an application can substitute a user-supplied global expiration age for the default by calling **RpcNsMgmtSetExpAge**. The **RpcNsMgmtInqExpAge** function returns the current global expiration age, whether a default or a user-supplied value. An application can also override the global expiration age temporarily by calling the **RpcNsMgmtHandleSetExpAge** function.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

See Also

RpcNsMgmtHandleSetExpAge, **RpcNsMgmtSetExpAge**

RpcNsMgmtSetExpAge

The **RpcNsMgmtSetExpAge** function modifies the application's global expiration age for local copies of name-service data.

```
RPC_STATUS RPC_ENTRY RpcNsMgmtSetExpAge(
  unsigned long ExpirationAge
);
```

Parameters

ExpirationAge

Pointer to the default expiration age, in seconds. This value is used by all name service–**next** operations. An expiration age of 0 causes an immediate update of the local name-service data.

To reset the expiration age to an RPC-assigned default value of two hours, specify a value of RPC_C_NS_DEFAULT_EXP_AGE.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsMgmtSetExpAge** function modifies the global expiration age of an application. The expiration age is the amount of time that a local copy of data from a name-service attribute can exist before a request from the application for the attribute requires updating the local copy. When an application begins running, the RPC run-time library specifies a default expiration age of two hours. The default is global to the application. Typically, you should avoid using **RpcNsMgmtSetExpAge**. Instead, you should rely on the default expiration age.

An expiration age is used by Pointer **next** operations (which read data from name-service attributes). A **next** operation typically starts by looking for a local copy of the attribute data being requested by an application. In the absence of a local copy, the **next** operation creates one with fresh attribute data from the name-service database. If a local copy already exists, the operation compares its actual age to the expiration age being used by the application. If the actual age exceeds the expiration age, the operation automatically tries to update the local copy with fresh attribute data. If updating is impossible, the old local data remains in place and the next operation fails, returning the RPC_S_NAME_SERVICE_UNAVAILABLE status code.

Setting the expiration age to a small value causes the Pointer **next** operations to frequently update local data for any name-service attribute requested by your application. For example, setting the expiration age to 0 forces all **next** operations to update local data for the name-service attribute requested by your application. Therefore, setting small expiration ages can create performance problems for your application and increase network traffic. Furthermore, if your application is using a remote name-service server, a small expiration age can adversely affect network performance for all applications.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

See Also

RpcNsMgmtHandleSetExpAge

RpcNsProfileDelete

The **RpcNsProfileDelete** function deletes a profile attribute:

```
RPC_STATUS RPC_ENTRY RpcNsProfileDelete(
  unsigned long ProfileNameSyntax,
  unsigned char *ProfileName
);
```

Parameters

ProfileNameSyntax
Specifies an integer value that indicates the syntax of the next argument, *ProfileName*.

To use the syntax specified in the registry value **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

ProfileName
Pointer to the name of the profile to delete.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.

(continued)

(continued)

Value	Meaning
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsProfileDelete** function deletes the profile attribute from the specified name-service entry (*ProfileName*). Neither *ProfileName* nor the entry names included as members in each profile element are deleted.

Tip Use **RpcNsProfileDelete** cautiously; deleting a profile can have the unwanted effect of breaking a hierarchy of profiles.

Note This DCE function is not supported by Microsoft Locator. Windows NT and Windows 2000 support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsProfileEltAdd, **RpcNsProfileEltRemove**

RpcNsProfileEltAdd

The **RpcNsProfileEltAdd** function adds an element to a profile. If necessary, it creates the entry.

```
RPC_STATUS RPC_ENTRY RpcNsProfileEltAdd(
  unsigned long ProfileNameSyntax,
  unsigned char *ProfileName,
  RPC_IF_ID *IfId,
  unsigned long MemberNameSyntax,
  unsigned char *MemberName,
  unsigned long Priority,
  unsigned char *Annotation
);
```

Parameters

ProfileNameSyntax

Indicates the syntax of the *ProfileName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

ProfileName

Pointer to the name of the profile to receive a new element.

IfId

Pointer to the interface identification of the new profile element. To add or replace the default profile element, specify a null value.

MemberNameSyntax

Indicates the syntax of the *MemberName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

MemberName

Pointer to a name service–entry name to include in the new profile element.

Priority

Integer value (0 through 7) that indicates the relative priority for using the new profile element during the **import** and **lookup** operations. A value of 0 is the highest priority; a value of 7 is the lowest priority. When adding a default profile member, use a value of 0.

Annotation

Pointer to an annotation string stored as part of the new profile element. Specify a null value or a null-terminated string if there is no annotation string.

The string is used by applications for informational purposes only. For example, an application can use this string to store the interface-name string specified in the IDL file. RPC does not use the annotation string during **lookup** or **import** operations or for enumerating profile elements.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsProfileEltAdd** function adds an element to the profile attribute of the name-service entry specified by *ProfileName*. If the *ProfileName* entry does not exist, **RpcNsProfileEltAdd** tries to create the entry with a profile attribute and adds the profile element specified by the *IfId*, *MemberName*, *Priority*, and *Annotation* parameters. In this case, the application must have the privilege to create the entry. Otherwise, a management application with the necessary privileges should create the entry by calling the **RpcNsMgmtEntryCreate** function before the application is run.

If an element with the specified member name and interface identification is already in the profile, **RpcNsProfileEltAdd** updates the element's priority and annotation string using the values provided in the *Priority* and *Annotation* parameters.

Note The Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcIfInqId, **RpcNsMgmtEntryCreate**, **RpcNsProfileEltRemove**

RpcNsProfileEltInqBegin

The **RpcNsProfileEltInqBegin** function creates an inquiry context for viewing the elements in a profile.

```
RPC_STATUS RPC_ENTRY RpcNsProfileEltInqBegin(
  unsigned long ProfileNameSyntax,
  unsigned char *ProfileName,
  unsigned long InquiryType,
  RPC_IF_ID *IfId,
  unsigned long VersOption,
  unsigned long MemberNameSyntax,
  unsigned char *MemberName,
  RPC_NS_HANDLE *InquiryContext
);
```

Parameters

ProfileNameSyntax

Indicates the syntax of the *ProfileName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

ProfileName

Pointer to the name of the profile to view.

InquiryType

Indicates the type of inquiry to perform on the profile. The following table lists valid inquiry types.

Inquiry type	Description
RPC_C_PROFILE_DEFAULT_ELT	Searches the profile for the default profile element, if any. The *IfId*, *VersOption*, and *MemberName* parameters are ignored.
RPC_C_PROFILE_ALL_ELTS	Returns every element from the profile. The *IfId*, *VersOption*, and *MemberName* parameters are ignored.
RPC_C_PROFILE_MATCH_BY_IF	Searches the profile for elements that contain the interface identification specified by *IfId* and *VersOption*. The *MemberName* parameter is ignored.
RPC_C_PROFILE_MATCH_BY_MBR	Searches the profile for elements that contain *MemberName*. The *IfId* and *VersOption* parameters are ignored.
RPC_C_PROFILE_MATCH_BY_BOTH	Searches the profile for elements that contain the interface identification and member identified by the *IfId*, *VersOption*, and *MemberName* parameters.

IfId

Pointer to the interface identification of the profile elements to be returned by the **RpcNsProfileEltInqNext** function.

The *IfId* parameter is used only when specifying a value of RPC_C_PROFILE_MATCH_BY_IF or RPC_C_PROFILE_MATCH_BY_BOTH for the *InquiryType* parameter. Otherwise, *IfId* is ignored and a null value can be specified.

VersOption

Specifies how the **RpcNsProfileEltInqNext** function uses the *IfId* parameter. This parameter is used only when specifying a value of RPC_C_PROFILE_MATCH_BY_IF or RPC_C_PROFILE_MATCH_BY_BOTH for *InquiryType*. Otherwise, this parameter is ignored and a 0 value can be specified.

The following table describes valid values for *VersOption*.

Values	Description
RPC_C_VERS_ALL	Returns profile elements that offer the specified interface UUID, regardless of the version numbers. For this value, specify 0 for both the major and minor versions in *IfId*.
RPC_C_VERS_COMPATIBLE	Returns profile elements that offer the same major version of the specified interface UUID and a minor version greater than or equal to the minor version of the specified interface UUID.
RPC_C_VERS_EXACT	Returns profile elements that offer the specified version of the specified interface UUID.
RPC_C_VERS_MAJOR_ONLY	Returns profile elements that offer the same major version of the specified interface UUID (ignores the minor version). For this value, specify 0 for the minor version in *IfId*.
RPC_C_VERS_UPTO	Returns profile elements that offer a version of the specified interface UUID less than or equal to the specified major and minor version. (For example, if the *IfId* contained V2.0 and the profile contained elements with V1.3, V2.0, and V2.1, the **RpcNsProfileEltInqNext** function returns elements with V1.3 and V2.0.)

MemberNameSyntax

Indicates the syntax of the *MemberName* parameter, and of the return parameter *MemberName* in the **RpcNsProfileEltInqNext** function.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

MemberName

Pointer to the member name that the **RpcNsProfileEltInqNext** function looks for in profile elements. The *MemberName* parameter is used only when specifying a value of RPC_C_PROFILE_MATCH_BY_MBR or RPC_C_PROFILE_MATCH_BY_BOTH for *InquiryType*. Otherwise, *MemberName* is ignored and a null value can be specified.

InquiryContext

Returns a pointer to a name-service handle for use with the **RpcNsProfileEltInqNext** and **RpcNsProfileEltInqDone** functions.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_VERS_OPTION	Invalid version option.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsProfileEltInqBegin** function creates an inquiry context for viewing the elements in a profile.

Using the *InquiryType* parameter, an application specifies which of the following profile elements are to be returned from calls to **RpcNsProfileEltInqNext**:

- The default element
- All elements
- Elements with the specified interface identification
- Elements with the specified member name
- Elements with both the specified interface identification and member name

Before calling **RpcNsProfileEltInqNext**, the application must first call **RpcNsProfileEltInqBegin** to create an inquiry context.

When finished viewing the profile elements, the application calls the **RpcNsProfileEltInqDone** function to delete the inquiry context.

Note Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcIfInqId, **RpcNsProfileEltInqDone**, **RpcNsProfileEltInqNext**

RpcNsProfileEltInqDone

The **RpcNsProfileEltInqDone** function deletes the inquiry context for viewing the elements in a profile.

```
RPC_STATUS RPC_ENTRY RpcNsProfileEltInqDone(
  RPC_NS_HANDLE *InquiryContext
);
```

Parameters

InquiryContext

Pointer to a name-service handle to free. The name-service handle that *InquiryContext* points to is created by calling the **RpcNsProfileEltInqBegin** function.

An argument value of NULL is returned.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

The **RpcNsProfileEltInqDone** function frees an inquiry context created by calling **RpcNsProfileEltInqBegin**.

An application calls **RpcNsProfileEltInqDone** after viewing profile elements using the **RpcNsProfileEltInqNext** function.

Note Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.

See Also

RpcNsProfileEltInqBegin, RpcNsProfileEltInqNext

RpcNsProfileEltInqNext

The **RpcNsProfileEltInqNext** function returns one element at a time from a profile.

```
RPC_STATUS RPC_ENTRY RpcNsProfileEltInqNext(
  RPC_NS_HANDLE InquiryContext,
  RPC_IF_ID *IfId,
  unsigned char **MemberName,
  unsigned long *Priority,
  unsigned char **Annotation
);
```

Parameters

InquiryContext

Specifies a name-service handle returned from the **RpcNsProfileEltInqBegin** function.

IfId

Returns a pointer to the interface identification of the profile element.

MemberName

Returns a pointer to a pointer to the profile element's member name. The syntax of the returned name was specified by the *MemberNameSyntax* parameter in the **RpcNsProfileEltInqBegin** function.

Specify a null value to prevent **RpcNsProfileEltInqNext** from returning the *MemberName* argument. In this case, the application does not call the **RpcStringFree** function.

Priority

Returns a pointer to the profile-element priority.

Annotation

Returns a pointer to a pointer to the annotation string for the profile element. If there is no annotation string in the profile element, the string \0 is returned.

Specify a null value to prevent **RpcNsProfileEltInqNext** from returning the *Annotation* parameter. In this case, the application does not need to call the **RpcStringFree** function.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.
RPC_S_NO_MORE_ELEMENTS	No more elements.

Remarks

The **RpcNsProfileEltInqNext** function returns one element from the profile specified by the *ProfileName* parameter in **RpcNsProfileEltInqBegin**. Regardless of the value of *InquiryType* in **RpcNsProfileEltInqBegin**, **RpcNsProfileEltInqNext** returns all the components (interface identification, member name, priority, annotation string) of a profile element.

An application can view all the selected profile entries by repeatedly calling the **RpcNsProfileEltInqNext** function. When all the elements have been viewed, this function returns a RPC_S_NO_MORE_ELEMENTS status code. The returned elements are unordered.

On each call to **RpcNsProfileEltInqNext** that returns a profile element, the RPC run-time library allocates memory for the returned member name and annotation string. The application is responsible for calling the **RpcStringFree** function for each returned member name and annotation string. After viewing the profile's elements, the application must call **RpcNsProfileEltInqDone** to release the inquiry context.

Note Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNsProfileEltInqBegin, **RpcNsProfileEltInqDone**, **RpcStringFree**

RpcNsProfileEltRemove

The **RpcNsProfileEltRemove** function removes an element from a profile.

```
RPC_STATUS RPC_ENTRY RpcNsProfileEltRemove(
  unsigned long ProfileNameSyntax,
  unsigned char *ProfileName,
  RPC_IF_ID *IfId,
  unsigned long MemberNameSyntax,
  unsigned char *MemberName
);
```

Parameters

ProfileNameSyntax

Indicates the syntax of the *ProfileName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

ProfileName

Pointer to the name of the profile from which to remove an element.

IfId

Pointer to the interface identification of the profile element to be removed.

Specify a null value to remove the default profile member.

MemberNameSyntax

Indicates the syntax of the *MemberName* parameter.

To use the syntax specified in the registry value entry **HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\DefaultSyntax**, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

MemberName

Pointer to the name service–entry name in the profile element to remove.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.

Remarks

The **RpcNsProfileEltRemove** function removes a profile element from the profile attribute in the *ProfileName* entry. This function requires an exact match of the *MemberName* and *IfId* parameters to remove a profile element. The entry (*MemberName*), included as a member in the profile element, is not deleted.

Tip Use **RpcNsProfileEltRemove** cautiously: removing elements from a profile can have the unwanted effect of breaking a hierarchy of profiles.

Note Windows 2000 Active Directory supports this function. Earlier versions of Windows NT support the use of this function with Cell Directory Service (CDS) only.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in Rpcnsi.h.
Library: Use Rpcns4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

RpcNsProfileDelete, **RpcNsProfileEltAdd**

RpcObjectInqType

The **RpcObjectInqType** function returns the type of an object. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcObjectInqType(
  UUID *ObjUuid,
  UUID *TypeUuid
);
```

Parameters

ObjUuid

Pointer to the object UUID whose associated type UUID is returned.

TypeUuid

Returns a pointer to the type UUID of the *ObjUuid* argument.

Specify an argument value of NULL to prevent the return of a type UUID. In this way, an application can determine (from the returned status) whether *ObjUuid* is registered without specifying an output type UUID variable.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_OBJECT_NOT_FOUND	Object not found

Remarks

A server application calls **RpcObjectInqType** to obtain the type UUID of an object. If the object was registered with the RPC run-time library using the **RpcObjectSetType** function, the registered type is returned.

Optionally, an application can privately maintain an object/type registration. In this case, if the application has provided an object inquiry function (see under ***RpcObjectSetInqFn***). The RPC run-time library uses that function to determine an object's type.

The **RpcObjectInqType** function obtains the type UUID as described in the following table.

Object UUID registered	Inquiry function registered	Return value
Yes (**RpcObjectSetType**)	Ignored	The object's registered type UUID
No	Yes. (**RpcObjectSetInqFn**)	The type UUID returned from the inquiry function
No	No	The nil UUID

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcObjectSetInqFn, **RpcObjectSetType**

RpcObjectSetInqFn

The **RpcObjectSetInqFn** function registers an object-inquiry function. A null value turns off a previously registered object-inquiry function. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcObjectSetInqFn(
  RPC_OBJECT_INQ_FN InquiryFn
);
```

Parameters

InquiryFn

Specifies an object-type inquiry function. See **RPC_OBJECT_INQ_FN**. When an application calls **RpcObjectInqType** and the RPC run-time library finds that the specified object is not registered, the run-time library automatically calls **RpcObjectSetInqFn** to determine the object's type.

Return Values

This function returns the following value.

Value	Meaning
RPC_S_OK	Call successful.

Remarks

A server application calls **RpcObjectSetInqFn** to override the default mapping function that maps object UUIDs to type UUIDs, which determine an object's type. If an application privately maintains an object/type registration, the specified inquiry function returns the type UUID of an object.

The RPC run-time library automatically calls the inquiry function when the application calls **RpcObjectInqType** and the object of interest was not previously registered with **RpcObjectSetType**. The *TypeUuid* and *Status* values of the **RPC_OBJECT_INQ_FN** function are returned as the output from **RpcObjectInqType**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcObjectInqType, **RpcObjectSetType**

RpcObjectSetType

The **RpcObjectSetType** function assigns the type of an object. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcObjectSetType(
  UUID *ObjUuid,
  UUID *TypeUuid
);
```

Parameters

ObjUuid
Pointer to an object UUID to associate with the type UUID in the *TypeUuid* argument.

TypeUuid
Pointer to the type UUID of the *ObjUuid* argument.

Specify an argument value of NULL or a nil UUID to reset the object type to the default association of object UUID/nil-type UUID.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_OBJECT	Invalid object.
RPC_S_ALREADY_REGISTERED	Object already registered.

Remarks

A server application calls **RpcObjectSetType** to assign a type UUID to an object UUID. By default, the RPC run-time library automatically assigns all object UUIDs with the nil-type UUID. A server application that contains one implementation of an interface (one manager Entry-Point Vector [EPV]) does not need to call **RpcObjectSetType** provided that the server registered the interface with the nil-type UUID (see under ***RpcServerRegisterIf***).

A server application that contains multiple implementations of an interface (multiple manager EPVs—that is, multiple type UUIDs) calls **RpcObjectSetType** once for each different object UUID/non-nil type UUID association the server supports. Associating each object with a type UUID tells the RPC run-time library which manager EPV (interface implementation) to use when the server receives a remote procedure call for a non-nil object UUID.

The RPC run-time library allows an application to set the type for an unlimited number of objects. To remove the association between an object UUID and its type UUID (established by calling **RpcObjectSetType**), a server calls **RpcObjectSetType** again, specifying a null value or a nil UUID for the *TypeUuid* argument. This resets the object UUID/type UUID association to the default association of object UUID/nil-type UUID. A server cannot assign a type to the nil object UUID. The RPC run-time library automatically assigns the nil object UUID a nil-type UUID.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcServerRegisterIf

RpcProtseqVectorFree

The **RpcProtseqVectorFree** function frees the protocol sequences contained in the vector and the vector itself. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcProtseqVectorFree(
  RPC_PROTSEQ_VECTOR **ProtSeqVector
);
```

Parameters

ProtSeqVector

Pointer to a pointer to a vector of protocol sequences. On return, the pointer is set to NULL.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

A server calls **RpcProtseqVectorFree** to release the memory used to store a vector of protocol sequences and the individual protocol sequences. **RpcProtseqVectorFree** sets the *ProtSeqVector* argument to a null value.

For a list of Microsoft RPC supported protocol sequences, see *String Binding*.

A server obtains a vector of protocol sequences by calling **RpcNetworkInqProtseqs**.

Note **RpcProtseqVectorFree** is available for server applications, not client applications, using Microsoft RPC.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcNetworkInqProtseqs

RpcRaiseException

Use the **RpcRaiseException** function to raise an exception. The function does not return to the caller.

```
void RPC_ENTRY RpcRaiseException (
  RPC_STATUS Exception
);
```

Parameters

Exception

Specifies the exception code for the exception. The following exception codes are defined.

Exception code	Description
RPC_S_ACCESS_DENIED	Access denied.
RPC_S_ADDRESS_ERROR	An addressing error occurred in the RPC server.
RPC_S_ALREADY_LISTENING	Server already listening.
RPC_S_ALREADY_REGISTERED	Object already registered.
RPC_S_BINDING_HAS_NO_AUTH	Binding has no authentication.
RPC_S_BINDING_INCOMPLETE	The binding handle is a required parameter.
RPC_S_BUFFER_TOO_SMALL	Insufficient buffer.
RPC_S_CALL_CANCELLED	The remote procedure call exceeded the cancel time-out and was canceled.
RPC_S_CALL_FAILED	Call failed.
RPC_S_CALL_FAILED_DNE	Call failed and did not execute.
RPC_S_CALL_IN_PROGRESS	Call already in progress for this thread.
RPC_S_CANNOT_SUPPORT	Operation is not supported.
RPC_S_CANT_CREATE_ENDPOINT	Cannot create endpoint.
RPC_S_COMM_FAILURE	Unable to communicate with the server.
RPC_S_DUPLICATE_ENDPOINT	Endpoint already exists.
RPC_S_ENTRY_ALREADY_EXISTS	Name-service entry already exists.
RPC_S_ENTRY_NOT_FOUND	Name-service entry not found.
RPC_S_FP_DIV_ZERO	A floating-point operation in the server caused a division by zero.
RPC_S_FP_OVERFLOW	Floating-point overflow has occurred in the RPC server.
RPC_S_FP_UNDERFLOW	Floating-point underflow has occurred in the server.
RPC_S_GROUP_MEMBER_NOT_FOUND	Group member not found.

(continued)

(continued)

Exception code	Description
RPC_S_INCOMPLETE_NAME	Incomplete name.
RPC_S_INTERFACE_NOT_FOUND	Interface not found.
RPC_S_INTERNAL_ERROR	Internal error.
RPC_S_INVALID_ARG	Invalid argument.
RPC_S_INVALID_AUTH_IDENTITY	Invalid authentication.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_INVALID_BOUND	Invalid bound.
RPC_S_INVALID_ENDPOINT_FORMAT	Invalid endpoint format.
RPC_S_INVALID_INQUIRY_CONTEXT	Invalid inquiry context.
RPC_S_INVALID_INQUIRY_TYPE	Invalid inquiry type.
RPC_S_INVALID_LEVEL	Invalid parameter.
RPC_S_INVALID_NAF_IF	Invalid network-address family identifier.
RPC_S_INVALID_NAME_SYNTAX	Invalid name syntax.
RPC_S_INVALID_NET_ADDR	Invalid network address.
RPC_S_INVALID_NETWORK_OPTIONS	Invalid network options.
RPC_S_INVALID_OBJECT	Invalid object.
RPC_S_INVALID_RPC_PROTSEQ	Invalid protocol sequence.
RPC_S_INVALID_SECURIT_DESC	Invalid security descriptor.
RPC_S_INVALID_STRING_BINDING	Invalid string binding.
RPC_S_INVALID_STRING_UUID	Invalid string UUID.
RPC_S_INVALID_TAG	Invalid tag.
RPC_S_INVALID_TIMEOUT	Invalid time-out value.
RPC_S_INVALID_VERS_OPTION	Invalid version option.
RPC_S_MAX_CALLS_TOO_SMALL	Maximum-calls value too small.
RPC_S_NAME_SERVICE_UNAVAILABLE	Name service unavailable.
RPC_S_NO_BINDINGS	No bindings.
RPC_S_NO_CALL_ACTIVE	No remote procedure active in this thread.
RPC_S_NO_CONTEXT_AVAILABLE	No security context available to perform impersonation.
RPC_S_NO_ENDPOINT_FOUND	No endpoint found.
RPC_S_NO_ENTRY_NAME	No entry name for binding.
RPC_S_NO_ENV_SETUP	No environment variable set up.
RPC_S_NO_INTERFACES	No interfaces registered.
RPC_S_NO_INTERFACES_EXPORTED	No interfaces have been exported.

Exception code	Description
RPC_S_NO_MORE_BINDINGS	No more bindings.
RPC_NO_MORE_ELEMENTS	No more elements.
RPC_S_NO_MORE_MEMBERS	No more members.
RPC_S_NO_NS_PRIVILEGE	No privilege for name-service operation.
RPC_S_NO_PRINC_NAME	No principal name registered.
RPC_S_NO_PROTSEQS	No supported protocol sequences.
RPC_S_NO_PROTSEQS_REGISTERED	No protocol sequences registered.
RPC_S_NOT_ALL_OBJS_UNEXPORTED	Not all objects unexported.
RPC_S_NOT_CANCELLED	Thread not canceled.
RPC_S_NOT_LISTENING	Server not listening.
RPC_S_NOT_RPC_ERROR	Requested status code not valid.
RPC_S_NOTHING_TO_EXPORT	Nothing to export.
RPC_S_OBJECT_NOT_FOUND	Object not found.
RPC_S_OK	Call succeeded.
RPC_S_OUT_OF_MEMORY	Out of memory.
RPC_S_OUT_OF_RESOURCES	Out of resources.
RPC_S_OUT_OF_THREADS	Out of threads.
RPC_S_PROCNUM_OUT_OF_RANGE	Procedure number out of range.
RPC_S_PROTOCOL_ERROR	An RPC protocol error occurred.
RPC_S_PROTSEQ_NOT_FOUND	Protocol sequence not found.
RPC_S_PROTSEQ_NOT_SUPPORTED	Protocol sequence not supported.
RPC_S_SERVER_OUT_OF_MEMORY	Server out of memory.
RPC_S_SERVER_TOO_BUSY	Server too busy.
RPC_S_SERVER_UNAVAILABLE	Server unavailable.
RPC_S_STRING_TOO_LONG	String too long.
RPC_S_TYPE_ALREADY_REGISTERED	Type UUID already registered.
RPC_S_UNKNOWN_AUTHN_LEVEL	Unknown authentication level.
RPC_S_UNKNOWN_AUTHN_SERVICE	Unknown authentication service.
RPC_S_UNKNOWN_AUTHN_TYPE	Unknown authentication type.
RPC_S_UNKNOWN_IF	Unknown interface.
RPC_S_UNKNOWN_MGR_TYPE	Unknown manager type.
RPC_S_UNSUPPORTED-TRANS_SYN	Transfer syntax not supported by the server.
RPC_S_UNSUPPORTED_NAME_SYNTAX	Unsupported name syntax.
RPC_S_UNSUPPORTED_TYPE	Unsupported UUID type.

(continued)

(continued)

Exception code	Description
RPC_S_UUID_LOCAL_ONLY	The UUID valid for this computer has been allocated.
RPC_S_UUID_NO_ADDRESS	No network address available to use to construct a UUID.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_S_ZERO_DIVIDE	Attempt to divide an integer by zero.
RPC_X_BAD_STUB_DATA	Stub received bad data.
RPC_X_BYTE_COUNT_TOO_SMALL	Byte count too small.
RPC_X_ENUM_VALUE_OUT_OF RANGE	The enumeration value out of range.
RPC_X_ENUM_VALUE_TOO_LARGE	The enumeration value out of range.
RPC_X_INVALID_BOUND	Specified bounds of an array inconsistent.
RPC_X_INVALID_TAG	Discriminant value does not match any case values; no default case.
RPC_X_NO_MEMORY	Insufficient memory available to set up necessary data structures.
RPC_X_NO_MORE_ENTRIES	List of servers available for *AutoHandle* binding has been exhausted.
RPC_X_NULL_REF_POINTER	A null reference pointer passed to the stub.
RPC_X_SS_BAD_ES_VERSION	Operation for the serializing handle not valid.
RPC_X_SS_CANNOT_GET_CALL_HANDLE	Stub unable to get the remote procedure call handle.
RPC_X_SS_CHAR_TRANS_OPEN_FAIL	File designated by DCERPCCHARTRANS cannot be opened.
RPC_X_SS_CHAR_TRANS_SHORT_FILE	File containing character-translation table has fewer than 512 bytes.
RPC_X_SS_CONTEXT_DAMAGED	Only raised on caller side; UUID in, **out** context handle changed during call.
RPC_X_SS_CONTEXT_MISMATCH	Only raised in the invoked function; UUID in handle does not correspond to any known context.
RPC_X_SS_HANDLES_MISMATCH	Binding handles passed to a remote procedure call don't match.
RPC_X_SS_IN_NULL_CONTEXT	Null context handle passed in parameter position.
RPC_X_SS_INVALID_BUFFER	Buffer not valid for operation.
RPC_X_SS_WRONG_ES_VERSION	Software version incorrect.
RPC_X_SS_WRONG_STUB_VERSION	Stub version incorrect.

Return Values

This function does not return a value.

Remarks

RpcRaiseException raises an exception. The exception handler can then handle the exception.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcAbnormalTermination, **RpcExcept**, **RpcFinally**

RpcRevertToSelf

After calling **RpcImpersonateClient** and completing any tasks that require client impersonation, the server calls **RpcRevertToSelf** to end impersonation and to reestablish its own security identity.

```
RPC_STATUS RPC_ENTRY RpcRevertToSelf(VOID);
```

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_CALL_ACTIVE	Server does not have a client to impersonate.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_S_CANNOT_SUPPORT	Not supported for this operating system, this transport, or this security subsystem.

Remarks

In a multithreaded application, if the call to **RpcImpersonateClient** is with a handle to another client thread, you must call **RpcRevertToSelfEx** with the handle to that thread to end impersonation.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

Client Impersonation, **RpcImpersonateClient**

RpcRevertToSelfEx

The **RpcRevertToSelfEx** function allows a server to impersonate a client and then revert in a multithreaded operation where the call to impersonate a client can come from a thread other than the thread originally dispatched from the RPC.

```
RPC_STATUS RPC_ENTRY RpcRevertToSelfEx(
  RPC_BINDING_HANDLE BindingHandle
);
```

Parameters

BindingHandle
Specifies a binding handle on the server that represents a binding to the client that the server impersonated. A value of zero specifies the client handle of the current thread; in this case, the functionality of **RpcRevertToSelfEx** is identical to that of the **RpcRevertToSelf** function.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_CALL_ACTIVE	Server does not have a client to impersonate.
RPC_S_INVALID_BINDING	Invalid binding handle.
RPC_S_WRONG_KIND_OF_BINDING	Wrong kind of binding for operation.
RPC_S_CANNOT_SUPPORT	Not supported for this operating system, this transport, or this security subsystem.

Remarks

After calling **RpcImpersonateClient** and completing any tasks that require client impersonation, the server calls **RpcRevertToSelfEx** to end impersonation and to reestablish its own security identity. For example, consider a primary thread, called thread1, which is dispatched from a remote client and wakes up a worker thread, called thread2. If thread2 requires that the server impersonate the client, the server calls **RpcImpersonateClient**(THREAD1_CALL_HANDLE), performs the required task, calls **RpcRevertToSelfEx**(THREAD1_CALL_HANDLE) to end the impersonation, and then wakes up thread1.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

Client Impersonation, **RpcImpersonateClient**, **RpcRevertToSelf**

RpcServerInqBindings

The **RpcServerInqBindings** function returns the binding handles over which remote procedure calls can be received. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerInqBindings(
  RPC_BINDING_VECTOR **BindingVector
);
```

Parameters

BindingVector
Returns a pointer to a pointer to a vector of server binding handles.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_BINDINGS	No bindings.

Remarks

A server application calls **RpcServerInqBindings** to obtain a vector of server binding handles. The RPC run-time library creates binding handles when a server application calls the following functions to register protocol sequences:

- **RpcServerUseAllProtseqs**
- **RpcServerUseProtseq**
- **RpcServerUseAllProtseqsIf**

- **RpcServerUseProtseqIf**
- **RpcServerUseProtseqEp**

The returned binding vector can contain binding handles with dynamic endpoints or binding handles with well-known endpoints, depending on which of the above functions the server application called.

A server uses the vector of binding handles for exporting to the name service, for registering with the local endpoint-map database, or for conversion to string bindings. If there are no binding handles (no registered protocol sequences), this routine returns the RPC_S_NO_BINDINGS status code and a *BindingVector* argument value of NULL. The server is responsible for calling the **RpcBindingVectorFree** function to release the memory used by the vector.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingVectorFree, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingExport**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

RpcServerInqDefaultPrincName

The **RpcServerInqDefaultPrincName** function obtains the default principal name from the server.

```
RPC_STATUS RPC_ENTRY RpcServerInqDefaultPrincName(
  unsigned long AuthnSvc,
  RPC_CHAR **PrincName
);
```

Parameters

AuthnSvc
Specifies an authentication service to use when the server receives a request for a remote procedure call.

PrincName
Pointer to the principal name to use for the server when authenticating remote procedure calls using the service specified by the *AuthnSvc* argument. The authentication service that is in use defines the content of the name and its syntax.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_OUT_OF_MEMORY	Insufficient memory to complete the operation.

Remarks

In an environment that only uses NetWare, the server application calls the **RpcServerInqDefaultPrincName** function to obtain the name of the NetWare server, when authenticated RPC is required. The value obtained from this function is then passed to **RpcServerRegisterAuthInfo**.

Requirements

Windows NT/2000: Unsupported.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingSetAuthInfo, **RpcServerRegisterAuthInfo**

RpcServerInqIf

The **RpcServerInqIf** function returns the manager entry-point vector (EPV) registered for an interface. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerInqIf(
  RPC_IF_HANDLE IfSpec,
  UUID *MgrTypeUuid,
  RPC_MGR_EPV **MgrEpv
);
```

Parameters

IfSpec
Specifies the interface whose manager EPV is returned.

MgrTypeUuid
Pointer to the manager type UUID whose manager EPV is returned.

Specifying an argument value of NULL (or a nil UUID) signifies to return the manager EPV registered with *IfSpec* and the nil manager type UUID.

MgrEpv
Returns a pointer to the manager EPV for the requested interface.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_UNKNOWN_IF	Unknown interface.
RPC_S_UNKNOWN_MGR_TYPE	Unknown manager type.

Remarks

A server application calls the **RpcServerInqIf** function to determine the manager EPV for a registered interface and manager type UUID.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcServerRegisterIf

RpcServerListen

The **RpcServerListen** function signals the RPC run-time library to listen for remote procedure calls. This function will not affect auto-listen interfaces; use the **RpcServerRegisterIfEx** function if you need that functionality. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerListen(
  unsigned int MinimumCallThreads,
  unsigned int MaxCalls,
  unsigned int DontWait
);
```

Parameters

MinimumCallThreads
  Specifies the minimum number of call threads.

MaxCalls
  Specifies the recommended maximum number of concurrent remote procedure calls the server can execute. To allow efficient performance, the RPC run-time libraries interpret the *MaxCalls* parameter as a suggested limit rather than as an absolute upper bound.

  Use RPC_C_LISTEN_MAX_CALLS_DEFAULT to specify the default value.

DontWait
Specifies a flag controlling the return from **RpcServerListen**. A value of nonzero indicates that **RpcServerListen** should return immediately after completing function processing. A value of zero indicates that **RpcServerListen** should not return until the **RpcMgmtStopServerListening** function has been called and all remote calls have completed.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_ALREADY_LISTENING	Server already listening
RPC_S_NO_PROTSEQS_REGISTERED	No protocol sequences registered.
RPC_S_MAX_CALLS_TOO_SMALL	Maximum calls value too small.

Remarks

A server calls **RpcServerListen** when the server is ready to process remote procedure calls. RPC allows a server to simultaneously process multiple calls. The *MaxCalls* argument recommends the maximum number of concurrent remote procedure calls the server should execute.

The *MaxCalls* value should be equal to or greater than the largest *MaxCalls* value specified to the functions **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**, **RpcServerUseAllProtseqs**, and **RpcServerUseAllProtseqsIf**.

A server application is responsible for concurrency control between the server manager routines because each routine executes in a separate thread.

When the *DontWait* parameter has a value of zero, the RPC run-time library continues listening for remote procedure calls (that is, the routine does not return to the server application) until one of the following events occurs:

- One of the server application's manager routines calls **RpcMgmtStopServerListening.**
- A client calls a remote procedure provided by the server that directs the server to call **RpcMgmtStopServerListening.**
- A client calls **RpcMgmtStopServerListening** with a binding handle to the server.

After it receives a stop-listening request, the RPC run-time library stops accepting new remote procedure calls for all registered interfaces. Executing calls are allowed to complete, including callbacks. After all calls complete, **RpcServerListen** returns to the caller.

When the *DontWait* parameter has a nonzero value, **RpcServerListen** returns to the server immediately after processing all the instructions associated with the function.

You can use the **RpcMgmtWaitServerListen** function to perform the wait operation usually associated with **RpcServerListen**.

Note The Microsoft RPC implementation of **RpcServerListen** includes two additional parameters that do not appear in the DCE specification: *DontWait* and *MinimumCallThreads*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcMgmtStopServerListening, **RpcMgmtWaitServerListen**, **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

RpcServerRegisterAuthInfo

The **RpcServerRegisterAuthInfo** function registers authentication information with the RPC run-time library. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerRegisterAuthInfo(
  unsigned char *ServerPrincName,
  unsigned long AuthnSvc,
  RPC_AUTH_KEY_RETRIEVAL_FN GetKeyFn,
  void *Arg
);
```

Parameters

ServerPrincName
Pointer to the principal name to use for the server when authenticating remote procedure calls using the service specified by the *AuthnSvc* argument. The content of the name and its syntax are defined by the authentication service in use. For more information, see *Principal Names*.

AuthnSvc
Specifies an authentication service to use when the server receives a request for a remote procedure call.

GetKeyFn

Specifies the address of a server-application-provided routine that returns encryption keys. See ***RPC_AUTH_KEY_RETRIEVAL_FN***.

Specify a NULL argument value to use the default method of encryption-key acquisition. In this case, the authentication service specifies the default behavior. Set this parameter to NULL when using the RPC_C_AUTHN_WINNT authentication service.

Authentication service	GetKeyFn	Arg	Run-time behavior
RPC_C_AUTHN_DPA	Ignored	Ignored	Does not support.
RPC_C_AUTHN_GSS_ KERBEROS	Ignored	Ignored	Does not support.
RPC_C_AUTHN_GSS_ NEGOTIATE	Ignored	Ignored	Does not support.
RPC_C_AUTHN_GSS_ SCHANNEL	Ignored	Ignored	Does not support.
RPC_C_AUTHN_MQ	Ignored	Ignored	Does not support.
RPC_C_AUTHN_MSN	Ignored	Ignored	Does not support.
RPC_C_AUTHN_WINNT	Ignored	Ignored	Does not support.
RPC_C_AUTHN_DCE_ PRIVATE	NULL	Non-null	Uses default method of encryption-key acquisition from specified key table; specified argument is passed to default acquisition function.
RPC_C_AUTHN_DCE_ PRIVATE	Non-null	NULL	Uses specified encryption-key acquisition function to obtain keys from default key table.
RPC_C_AUTHN_DCE_ PRIVATE	Non-null	Non-null	Uses specified encryption-key acquisition function to obtain keys from specified key table; specified argument is passed to acquisition function.
RPC_C_AUTHN_DEC_ PUBLIC	Ignored	Ignored	Reserved for future use.

The RPC run-time library passes the *ServerPrincName* argument value from **RpcServerRegisterAuthInfo** as the *ServerPrincName* argument value to the *GetKeyFn* acquisition function. The RPC run-time library automatically provides a value for the key version (*KeyVer*) argument. For a *KeyVer* argument value of zero, the acquisition function must return the most recent key available. The retrieval function returns the authentication key in the *Key* argument.

If the acquisition function called from **RpcServerRegisterAuthInfo** returns a status other than RPC_S_OK, then this function fails and returns an error code to the server application. If the acquisition function called by the RPC run-time library while authenticating a client's remote procedure call request returns a status other than RPC_S_OK, the request fails and the RPC run-time library returns an error code to the client application.

Arg

Pointer to an argument to pass to the *GetKeyFn* routine, if specified. This parameter can also be used to pass a pointer to an **SCHANNEL_CRED** structure to specify explicit credentials if the authentication service is set to SCHANNEL.

If the *Arg* parameter is set to NULL, this function will use the default certificate or credential if it has been set up in the directory service.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_UNKNOWN_AUTHN_SERVICE	Unknown authentication service.

Remarks

A server application calls **RpcServerRegisterAuthInfo** to register an authentication service to use for authenticating remote procedure calls. A server calls this routine once for each authentication service and/or principal name the server wants to register.

The authentication service that a client application specifies (using **RpcBindingSetAuthInfo** or **RpcServerRegisterAuthInfo**) must be one of the authentication services specified by the server application. Otherwise, the client's remote procedure call fails and an RPC_S_UNKNOWN_AUTHN_SERVICE status code is returned.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingSetAuthInfo

RpcServerRegisterIf

The **RpcServerRegisterIf** function registers an interface with the RPC run-time library. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerRegisterIf(
  RPC_IF_HANDLE IfSpec,
  UUID *MgrTypeUuid,
  RPC_MGR_EPV *MgrEpv
);
```

Parameters

IfSpec
Specifies a MIDL-generated structure indicating the interface to register.

MgrTypeUuid
Pointer to a type UUID to associate with the *MgrEpv* argument. Specifying a null argument value (or a nil UUID) registers *IfSpec* with a nil-type UUID.

MgrEpv
Specifies the manager routines' entry-point vector (EPV). To use the MIDL-generated default EPV, specify a null value.

Remarks

A server can register an unlimited number of interfaces with the RPC run-time library. Registration makes an interface available to clients using a binding handle to the server. To register an interface, the server application code calls **RpcServerRegisterIf**. For each implementation of an interface that a server offers, it must register a separate manager EPV.

When calling **RpcServerRegisterIf**, the server provides the following information:

- Interface specification

  The interface specification is a data structure that the MIDL compiler generates. The server specifies the interface using the *IfSpec* argument.

- Manager type UUID and manager EPV

  The manager type UUID and the manager EPV determine which manager routine executes when a server receives a remote procedure call request from a client.

  The server specifies the manager type UUID and EPV using the *MgrTypeUuid* and *MgrEpv* arguments. Note that when specifying a non-nil manager-type UUID, the server must also call the **RpcObjectSetType** function to register objects of this non-nil type.

If your server application needs to register an auto-listen interface or use a callback function for authentication purposes, use **RpcServerRegisterIfEx**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

Registering the Interface, **RpcBindingFromStringBinding**, **RpcBindingSetObject**, **RpcNsBindingExport**, **RpcNsBindingImportBegin**, **RpcNsBindingLookupBegin**, **RpcObjectSetType**, **RpcServerUnregisterIf**, **RpcServerRegisterIf2**, **RpcServerRegisterIfEx**

RpcServerRegisterIf2

The **RpcServerRegisterIf2** function registers an interface with the RPC run-time library.

```
RPC_STATUS RPC_ENTRY RpcServerRegisterIf2(
  RPC_IF_HANDLE IfSpec,
  UUID *MgrTypeUuid,
  RPC_MGR_EPV *MgrEpv,
  unsigned int Flags,
  unsigned int MaxCalls,
  unsigned int MaxRpcSize,
  RPC_IF_CALLBACK_FN *IfCallbackFn
);
```

Parameters

IfSpec
Specifies a MIDL-generated structure indicating the interface to register.

MgrTypeUuid
Pointer to a type UUID to associate with the *MgrEpv* argument. Specifying a null argument value (or a nil UUID) registers *IfSpec* with a nil-type UUID.

MgrEpv
Specifies the manager routines' entry-point vector (EPV). To use the MIDL-generated default EPV, specify a null value.

Flags
For a list of flag values, see *Interface Registration Flags*.

MaxCalls
Specifies the maximum number of concurrent remote procedure call requests the server can accept. The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater and can vary for each protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value. A call on an **auto-listen** interface uses the value of *MaxCalls* specified for that interface. Calls on other interfaces are governed by the value of the process-wide *MaxCalls* specified in **RpcServerListen**.

MaxRpcSize

Specifies the maximum size, in bytes, of incoming data blocks. This parameter may be used to help prevent malicious denial-of-service attacks. If the data block of a remote procedure call is larger than *MaxRpcSize*, the RPC run-time library rejects the call and sends an RPC_S_ACCESS_DENIED error to the client. Specifying a value of (unsigned int)–1 for this parameter removes the limit on the size of incoming data blocks.

IfCallbackFn

Specifies a security-callback function, or NULL for no callback. Each registered interface can have a different callback function. See *Remarks.*

Remarks

The parameters and effects of the **RpcServerRegisterIf2** function extend those of the **RpcServerRegisterIf** function. The difference is the ability to register an **auto-listen** interface and to specify a security-callback function.

The server application code calls **RpcServerRegisterIf2** to register an interface. To register an interface, the server provides the following information:

- Interface specification.

  The interface specification is a data structure that the MIDL compiler generates.
- Manager type UUID and manager EPV

  The manager type UUID and the manager EPV determine which manager routine executes when a server receives a remote procedure call request from a client. For each implementation of an interface offered by a server, it must register a separate manager EPV.

  Note that when specifying a non-nil, manager type UUID, the server must also call **RpcObjectSetType** to register objects of this non-nil type.

Specifying the RPC_IF_AUTOLISTEN flags marks the interface as an **auto-listen** interface. The run time begins listening for calls as soon as the interface is registered, and stops listening when the interface is unregistered. A call to **RpcServerUnregisterIf** for this interface will wait for the completion of all pending calls on this interface. Calls to the **RpcServerListen** and the **RpcMgmtStopServerListening** functions will not affect the interface, nor will a call to the **RpcServerUnregisterIf** function with *IfSpec* set to the value NULL. This allows a DLL to register RPC interfaces or remove them from the registry without changing the main application's RPC state.

Specifying a security-callback function allows the server application to restrict access to its interfaces on a per-client basis. Remember that, by default, security is optional; the server run-time will dispatch unsecured calls even if the server has called the

RpcServerRegisterAuthInfo function. If the server wants to accept only authenticated clients, each server stub must call **RpcBindingInqAuthClient** function to retrieve the security level, or attempt to impersonate the client with the **RpcImpersonateClient** function.

When a server application specifies a security-callback function for its interface(s), the RPC run time automatically rejects unauthenticated calls to that interface. In addition, the run time records the interfaces that each client has used. When a client makes an RPC to an interface that it has not used during the current communication session, the RPC run-time library will call the interface's security-callback function.

For the signature for the callback function, see ***RPC_IF_CALLBACK_FN***.

The callback function should return RPC_S_OK, if the client is allowed to call methods in this interface. Any other return code will cause the client to receive the exception RPC_S_ACCESS_DENIED.

In some cases, the RPC run time may call the security-callback function more than once per client per interface. Be sure your callback function can handle this possibility.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 OSR2 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on all platforms.

See Also

RpcServerRegisterIf, **RpcServerRegisterIfEx**

RpcServerRegisterIfEx

The **RpcServerRegisterIfEx** function registers an interface with the RPC run-time library. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerRegisterIfEx(
  RPC_IF_HANDLE IfSpec,
  UUID *MgrTypeUuid,
  RPC_MGR_EPV *MgrEpv,
  unsigned int Flags,
  unsigned int MaxCalls,
  RPC_IF_CALLBACK_FN *IfCallback
);
```

Parameters

IfSpec
Specifies a MIDL-generated structure indicating the interface to register.

MgrTypeUuid
Pointer to a type UUID to associate with the *MgrEpv* argument. Specifying a null argument value (or a nil UUID) registers *IfSpec* with a nil-type UUID.

MgrEpv
Specifies the manager routines' Entry-Point Vector (EPV). To use the MIDL-generated default EPV, specify a null value.

Flags
For a list of flag values, see *Interface Registration Flags*.

MaxCalls
Specifies the maximum number of concurrent remote procedure call requests the server can accept. The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater and can vary for each protocol sequence. Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value. A call on an auto-listen interface uses the value of MaxCalls specified for that interface. Calls on other interfaces are governed by the value of the process-wide MaxCalls specified in ***RpcServerListen***.

IfCallback
Specifies a security-callback function, or NULL for no callback. Each registered interface can have a different callback function. See *Remarks* for more details.

Remarks

The parameters and effects of **RpcServerRegisterIfEx** subsume those of **RpcServerRegisterIf**. The difference is the ability to register an auto-listen interface and to specify a security-callback function.

The server application code calls **RpcServerRegisterIfEx** to register an interface. To register an interface, the server provides the following information:

- Interface specification.

  The interface specification is a data structure that the MIDL compiler generates.
- Manager type UUID and manager EPV

  The manager type UUID and the manager EPV determine which manager routine executes when a server receives a remote procedure call request from a client. For each implementation of an interface offered by a server, it must register a separate manager EPV.

  Note that when specifying a non-nil, manager type UUID, the server must also call **RpcObjectSetType** to register objects of this non-nil type.

Specifying the RPC_IF_AUTOLISTEN flags marks the interface as an **auto-listen** interface. The run time begins listening for calls as soon as the interface is registered, and stops listening when the interface is unregistered. A call to **RpcServerUnregisterIf** for this interface will wait for the completion of all pending calls on this interface. Calls to **RpcServerListen** and **RpcMgmtStopServerListening** will not affect the interface, nor will a call to **RpcServerUnregisterIf** with *IfSpec* == NULL. This allows a DLL to register RPC interfaces or remove them from the registry without changing the main application's RPC state.

Specifying a security-callback function allows the server application to restrict access to its interfaces on a per-client basis. Remember that, by default, security is optional; the server run time will dispatch unsecured calls even if the server has called **RpcServerRegisterAuthInfo**. If the server wants to accept only authenticated clients, each server stub must call **RpcBindingInqAuthClient** to retrieve the security level, or attempt to impersonate the client with **RpcImpersonateClient**.

When a server application specifies a security-callback function for its interface(s), the RPC run time automatically rejects unauthenticated calls to that interface. In addition, the run-time records the interfaces that each client has used. When a client makes an RPC to an interface that it has not used during the current communication session, the RPC run-time library will call the interface's security-callback function.

For the signature for the callback function, see ***RPC_IF_CALLBACK_FN***.

The callback function should return RPC_S_OK if the client is allowed to call methods in this interface. Any other return code will cause the client to receive the exception RPC_S_ACCESS_DENIED.

In some cases, the RPC run time may call the security-callback function more than once per client–per interface. Be sure your callback function can handle this possibility.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

Registering the Interface, **RpcBindingFromStringBinding**, **RpcBindingSetObject**, **RpcNsBindingExport**, **RpcNsBindingImportBegin**, **RpcNsBindingLookupBegin**, **RpcObjectSetType**, **RpcServerRegisterIf**, **RpcServerUnregisterIf**

RpcServerTestCancel

The server calls **RpcServerTestCancel** to test for client **cancel** requests.

```
RPC_STATUS RPC_ENTRY RpcServerTestCancel(
  RPC_BINDING_HANDLE BindingHandle
);
```

Parameters

BindingHandle
Specifies the thread on which to test for **cancel** commands.

Return Values

Value	Meaning
RPC_S_OK	The call was canceled.
RPC_S_NO_CALL_ACTIVE	There is no active call on the current thread.
RPC_S_CALL_IN_PROGRESS	The call was not canceled.
RPC_S_INVALID_BINDING	The handle is not valid.

Remarks

The server calls **RpcServerTestCancel** to find out if the client has requested cancellation of an outstanding call. The *Binding Handle* parameter specifies the call on which to test. If the parameter has a value of zero, the call on the current thread is tested. The server can call the **RpcServerTestCancel(RpcAsyncGetCallHandle(**pAsync**))** function to test for **cancel** message using the asynchronous handle to obtain the binding handle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

Asynchronous RPC, RPC_ASYNC_STATE, **RpcAsyncAbortCall**, **RpcAsyncCancelCall**, **RpcAsyncCompleteCall**, **RpcAsyncGetCallHandle**, **RpcAsyncGetCallStatus**, **RpcAsyncInitializeHandle**, **RpcAsyncRegisterInfo**

RpcServerUnregisterIf

The **RpcServerUnregisterIf** function removes an interface from the RPC run-time library registry. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerUnregisterIf(
  RPC_IF_HANDLE IfSpec,
  UUID *MgrTypeUuid,
  unsigned int WaitForCallsToComplete
);
```

Parameters

IfSpec

Specifies the interface to remove from the registry.

Specify a null value to remove all interfaces previously registered with the type UUID value specified in the *MgrTypeUuid* argument.

MgrTypeUuid

Pointer to the type UUID of the manager entry-point vector (EPV) to remove from the registry. The value of *MgrTypeUuid* should be the same value as was provided in a call to the **RpcServerRegisterIf** function or the **RpcServerRegisterIfEx** function.

Specify a null value to remove the interface specified in the *IfSpec* argument for all previously registered type UUIDs from the registry.

Specify a nil UUID to remove the MIDL-generated default manager EPV from the registry. In this case, all manager EPVs registered with a non-nil type UUID remain registered.

WaitForCallsToComplete

Specifies a flag that indicates whether to remove the interface from the registry immediately or to wait until all current calls are complete.

Specify a value of zero to disregard calls in progress and remove the interface from the registry immediately. Specify any nonzero value to wait until all active calls complete.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_UNKNOWN_MGR_TYPE	Unknown manager type
RPC_S_UNKNOWN_IF	Unknown interface

Remarks

A server calls **RpcServerUnregisterIf** to remove the association between an interface and a manager EPV. To specify the manager EPV to remove in the *MgrTypeUuid* argument, provide the type UUID value that was specified in a call to **RpcServerRegisterIf**. After it is removed from the registry, an interface is no longer available to client applications.

When an interface is removed from the registry, the RPC run-time library stops accepting new calls for that interface. Calls that are currently executing on the interface are allowed to complete, including callbacks.

The following table summarizes the behavior of **RpcServerUnregisterIf**.

IfSpec	MgrTypeUuid	Behavior
Non-null	Non-null	Removes from the registry the manager EPV associated with the specified arguments.
Non-null	NULL	Removes all manager EPVs associated with the *IfSpec* argument.
NULL	Non-null	Removes all manager EPVs associated with the *MgrTypeUuid* argument.
NULL	NULL	Removes all manager EPVs. This call has the effect of preventing the server from receiving any new remote procedure calls because all the manager EPVs for all interfaces have been unregistered.

Note If the value of *IfSpec* is NULL, this function will leave **auto-listen** interfaces registered. **Auto-listen** interfaces must be removed from the registry individually. See ***RpcServerRegisterIfEx*** for more details.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcServerRegisterIf, **RpcServerRegisterIfEx**

RpcServerUseAllProtseqs

The **RpcServerUseAllProtseqs** function tells the RPC run-time library to use all supported protocol sequences for receiving remote procedure calls. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerUseAllProtseqs(
  unsigned int MaxCalls,
  void *SecurityDescriptor
);
```

Parameters

MaxCalls

Specifies the maximum number of concurrent remote procedure call requests the server can accept.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater and can vary for each protocol sequence. Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

SecurityDescriptor

Pointer to an optional parameter provided for the Windows NT and Windows 2000 security subsystem. Note that this parameter does not appear in the DCE specification for this API.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_PROTSEQS	No supported protocol sequences.
RPC_S_OUT_OF_MEMORY	Insufficient memory available.
RPC_S_INVALID_SECURITY_DESC	Security descriptor invalid.

Remarks

A server application calls **RpcServerUseAllProtseqs** to register all supported protocol sequences with the RPC run-time library. To receive remote procedure calls, a server must register at least one protocol sequence with the RPC run-time library. For a list of Microsoft RPC supported protocol sequences, see the reference topic *String Binding*.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence. Each binding handle contains an endpoint dynamically generated by the RPC run-time library or the operating system.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to be able to handle. See *Server-Side Binding* for a description of the functions that a server will typically call after registering protocol sequences. To selectively register protocol sequences, a server calls **RpcServerUseProtseq**, **RpcServerUseProtseqIf**, or **RpcServerUseProtseqEp**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingToStringBinding, **RpcBindingVectorFree**, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerListen**, **RpcServerRegisterIf**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

RpcServerUseAllProtseqsEx

The **RpcServerUseAllProtseqsEx** function tells the RPC run-time library to use all supported protocol sequences for receiving remote procedure calls.

```
RPC_STATUS RPC_ENTRY RpcServerUseAllProtseqsEx(
  unsigned int MaxCalls,
  void *SecurityDescriptor,
  PRPC_POLICY Policy
);
```

Parameters

MaxCalls

Specifies the maximum number of concurrent remote procedure call requests the server can accept.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater and can vary for each protocol sequence. Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

SecurityDescriptor

Pointer to an optional parameter provided for the Windows NT and Windows 2000 security subsystem.

Policy

Pointer to the **RPC_POLICY** structure, which allows you to override the default policies for dynamic port allocation and binding to Network Interface Cards (NICs) on multihomed computers (computers with multiple network cards).

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_PROTSEQS	No supported protocol sequences.
RPC_S_OUT_OF_MEMORY	Insufficient memory available.
RPC_S_INVALID_SECURITY_DESC	Security descriptor is invalid.

Remarks

The parameters and effects of **RpcServerUseAllProtseqsEx** subsume those of **RpcServerUseAllProtseqs**. The difference is the *Policy* parameter, which allows you to restrict port allocation for dynamic ports and allows multihomed machines to selectively bind to specified NICs.

Setting the *NICFlags* field of the **RPC_POLICY** structure to zero makes this extended API functionally equivalent to the original **RpcServerUseAllProtseqs**, and the server will bind to NICs based on the settings in the system registry. For information on how the registry settings define the available Internet and intranet ports, see *Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding.*

Note The flag settings in the *Policy* field are effective only when the **ncacn_ip_tcp** or **ncadg_ip_udp** protocol sequence is in use. For all other protocol sequences, the RPC run-time ignores these values.

A server application calls **RpcServerUseAllProtseqsEx** to register all supported protocol sequences with the RPC run-time library. To receive remote procedure calls, a server must register at least one protocol sequence with the RPC run-time library.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence. Each binding handle contains an endpoint dynamically generated by the RPC run-time library or the operating system.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle. To selectively register protocol sequences, a server calls **RpcServerUseProtseqEx**, **RpcServerUseProtseqIfEx**, or **RpcServerUseProtseqEpEx**. See *Server-Side Binding* for a description of the routines that a server will typically call after registering protocol sequences.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqEpEx**, **RpcServerUseProtseqIfEx**

RpcServerUseAllProtseqsIf

The **RpcServerUseAllProtseqsIf** function tells the RPC run-time library to use all specified protocol sequences and endpoints in the interface specification for receiving remote procedure calls. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerUseAllProtseqsIf(
  unsigned int MaxCalls,
  RPC_IF_HANDLE IfSpec,
  void *SecurityDescriptor
);
```

Parameters

MaxCalls

Specifies the maximum number of concurrent remote procedure call requests the server can accept.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater and can vary for each protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

IfSpec

Specifies the interface containing the protocol sequences and corresponding endpoint information to use in creating binding handles.

SecurityDescriptor

Pointer to an optional parameter provided for the Microsoft Windows NT and Windows 2000 security subsystem. Note that this parameter does not appear in the DCE specification for this API.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_PROTSEQS	No supported protocol sequences.
RPC_S_INVALID_ENDPOINT_FORMAT	Invalid endpoint format.
RPC_S_OUT_OF_MEMORY	Out of memory.
RPC_S_DUPLICATE_ENDPOINT	Endpoint is duplicate.
RPC_S_INVALID_SECURITY_DESC	Security descriptor invalid.
RPC_S_INVALID_RPC_PROTSEQ	RPC protocol sequence invalid.

Remarks

A server application calls **RpcServerUseAllProtseqsIf** to register with the RPC run-time library all protocol sequences and associated endpoint address information provided in the IDL file. For a list of RPC-supported protocol sequences, see *String Binding*.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle. See *Server-Side Binding* for a description of the functions that a server will typically call after registering protocol sequences. To register selected protocol sequences specified in the IDL file, a server calls **RpcServerUseProtseqIf**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

RpcBindingVectorFree, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerListen**, **RpcServerRegisterIfEx**, **RpcServerRegisterIf**, **RpcServerUseAllProtseqs**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

RpcServerUseAllProtseqsIfEx

The **RpcServerUseAllProtseqsIfEx** function tells the RPC run-time library to use all the specified protocol sequences and endpoints in the interface specification for receiving remote procedure calls.

```
RPC_STATUS RPC_ENTRY RpcServerUseAllProtseqsIfEx(
  unsigned int MaxCalls,
  RPC_IF_HANDLE IfSpec,
  void *SecurityDescriptor,
  PRPC_POLICY Policy
);
```

Parameters

MaxCalls

Specifies the maximum number of concurrent remote procedure call requests the server can accept. The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater and can vary for each protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

IfSpec

Specifies the interface containing the protocol sequences and corresponding endpoint information to use in creating binding handles.

SecurityDescriptor

Pointer to an optional parameter provided for the Windows NT and Windows 2000 security subsystem.

Policy

Pointer to the **RPC_POLICY** structure, which contains flags to restrict port allocation for dynamic ports and allow multihomed computers to selectively bind to network interface cards.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_NO_PROTSEQS	No supported protocol sequences.
RPC_S_INVALID_ENDPOINT_FORMAT	Invalid endpoint format.
RPC_S_OUT_OF_MEMORY	Out of memory.
RPC_S_DUPLICATE_ENDPOINT	Endpoint is a duplicate.
RPC_S_INVALID_SECURITY_DESC	Security descriptor is invalid.
RPC_S_INVALID_RPC_PROTSEQ	RPC protocol sequence is invalid.

Remarks

The parameters and effects of **RpcServerUseAllProtseqsIfEx** subsume those of **RpcServerUseAllProtseqsIf**. The difference is the *Policy* field, which allows you to restrict port allocation for dynamic ports and allows multihomed machines to selectively bind to network interface cards.

Setting the *NICFlags* field of the **RPC_POLICY** structure to zero makes this extended function functionally equivalent to the original **RpcServerUseAllProtseqsIf,** and the server will bind to NICs based on the settings in the system registry. For information on how the registry settings define the available Internet and intranet ports, see *Configuring the Windows NT/Windows 2000 Registry for Port Allocations and Selective Binding.*

Note The flag settings in the *Policy* field are effective only when the **ncacn_ip_tcp** or **ncadg_ip_udp** protocol sequence is in use. For all other protocol sequences, the RPC run time ignores these values.

A server application calls **RpcServerUseAllProtseqsIfEx** to register with the RPC run-time library all protocol sequences and associated endpoint address information provided in the IDL file.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle. To register selected protocol sequences specified in the IDL file, a server calls **RpcServerUseProtseqIfEx**. See *Server-Side Binding* for a description of the routines that a server will typically call after registering protocol sequences.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding, **RpcServerUseAllProtseqsEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqEpEx**, **RpcServerUseProtseqIfEx**

RpcServerUseProtseq

The **RpcServerUseProtseq** function tells the RPC run-time library to use the specified protocol sequence for receiving remote procedure calls. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerUseProtseq(
  unsigned char *ProtSeq,
  unsigned int MaxCalls,
  void *SecurityDescriptor
);
```

Parameters

ProtSeq

Pointer to a string identifier of the protocol sequence to register with the RPC run-time library.

MaxCalls

Specifies the maximum number of concurrent remote procedure call requests the server wants to handle.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

SecurityDescriptor

Pointer to an optional parameter provided for the Windows NT and Windows 2000 security subsystem. Note that this parameter does not appear in the DCE specification for this API.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_PROTSEQ_NOT_SUPPORTED	Protocol sequence not supported on this host.
RPC_S_INVALID_RPC_PROTSEQ	Invalid protocol sequence.
RPC_S_OUT_OF_MEMORY	Out of memory.
RPC_S_INVALID_SECURITY_DESC	Security descriptor invalid.

Remarks

A server application calls **RpcServerUseProtseq** to register one protocol sequence with the RPC run-time library. To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call **RpcServerUseProtseq** multiple times to register additional protocol sequences. For more information, see *String Binding* and *Server-Side Binding*.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence. Each binding handle contains an endpoint dynamically generated by the RPC run-time library.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle. See *Server-Side Binding* for a description of the functions that a server will typically call after registering protocol sequences. To register all protocol sequences, a server calls **RpcServerUseAllProtseqs**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

RpcBindingVectorFree, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNetworkIsProtseqValid**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerListen**, **RpcServerRegisterIfEx**, **RpcServerRegisterIf**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

RpcServerUseProtseqEx

The **RpcServerUseProtseqEx** function tells the RPC run-time library to use the specified protocol sequence for receiving remote procedure calls.

```
RPC_STATUS RPC_ENTRY RpcServerUseProtseqEx(
  unsigned char *ProtSeq,
  unsigned int MaxCalls,
  void *SecurityDescriptor,
  PRPC_POLICY Policy
);
```

Parameters

ProtSeq
Pointer to a string identifier of the protocol sequence to register with the RPC run-time library.

MaxCalls
Specifies the maximum number of concurrent remote procedure call requests the server wants to handle. The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

SecurityDescriptor
Pointer to an optional parameter provided for the Windows NT and Windows 2000 security subsystem.

Policy
Pointer to the **RPC_POLICY** structure, which contains flags to restrict port allocation for dynamic ports and allow multihomed computers to selectively bind to network interface cards.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_PROTSEQ_NOT_SUPPORTED	Protocol sequence is not supported on this host.
RPC_S_INVALID_RPC_PROTSEQ	Invalid protocol sequence.
RPC_S_OUT_OF_MEMORY	Out of memory.
RPC_S_INVALID_SECURITY_DESC	Security descriptor is invalid.

Remarks

The parameters and effects of **RpcServerUseProtseqEx** subsume those of **RpcServerUseProtseq**. The difference is the *Policy* field, which allows you to restrict port allocation for dynamic ports and allows multihomed machines to selectively bind to network interface cards.

Setting the *NICFlags* field of the **RPC_POLICY** structure to zero makes this extended function functionally equivalent to the original **RpcServerUseProtseq**, and the server will bind to NICs based on the settings in the system registry. For information, see *Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding*.

Note The flag settings in the *Policy* field are effective only when the **ncacn_ip_tcp** or **ncadg_ip_udp** protocol sequence is in use. For all other protocol sequences, the RPC run time ignores these values.

A server application calls **RpcServerUseProtseqEx** to register one protocol sequence with the RPC run-time library. To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call **RpcServerUseProtseqEx** multiple times to register additional protocol sequences.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence. Each binding handle contains an endpoint dynamically generated by the RPC run-time library.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle. To register all protocol sequences, a server calls **RpcServerUseAllProtseqsEx** routine.

See *Server-Side Binding*.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding, **RpcServerUseAllProtseqsEx**, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEpEx**, **RpcServerUseProtseqIfEx**

RpcServerUseProtseqEp

The **RpcServerUseProtseqEp** function tells the RPC run-time library to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerUseProtseqEp(
  unsigned char *Protseq,
  unsigned int MaxCalls,
  unsigned char *Endpoint,
  void *SecurityDescriptor
);
```

Parameters

Protseq
Pointer to a string identifier of the protocol sequence to register with the RPC run-time library.

MaxCalls
Specifies the maximum number of concurrent remote procedure call requests the server wants to handle. The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

Endpoint
Pointer to the endpoint-address information to use in creating a binding for the protocol sequence specified in the *Protseq* argument.

SecurityDescriptor
Pointer to an optional parameter provided for the Windows NT and Windows 2000 security subsystem. Note that this parameter does not appear in the DCE specification for this API.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_PROTSEQ_NOT_SUPPORTED	Protocol sequence not supported on this host.
RPC_S_INVALID_RPC_PROTSEQ	Invalid protocol sequence.
RPC_S_INVALID_ENDPOINT_FORMAT	Invalid endpoint format.
RPC_S_OUT_OF_MEMORY	Out of memory.
RPC_S_DUPLICATE_ENDPOINT	Endpoint is duplicate.
RPC_S_INVALID_SECURITY_DESC	Security descriptor invalid.

Remarks

A server application calls **RpcServerUseProtseqEp** to register one protocol sequence with the RPC run-time library. With each protocol sequence registration, **RpcServerUseProtseqEp** includes the specified endpoint-address information.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call this routine multiple times to register additional protocol sequences and endpoints. For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server needs to be able to handle. See *Server-Side Binding* and *String Binding*.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingVectorFree, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerListen**, **RpcServerRegisterIf**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqIf**

RpcServerUseProtseqEpEx

The **RpcServerUseProtseqEpEx** function tells the RPC run-time library to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls.

```
RPC_STATUS RPC_ENTRY RpcServerUseProtseqEpEx(
  unsigned char *Protseq,
  unsigned int MaxCalls,
  unsigned char *Endpoint,
  void *SecurityDescriptor,
  PRPC_POLICY Policy
);
```

Parameters

Protseq
Pointer to a string identifier of the protocol sequence to register with the RPC run-time library.

MaxCalls
The maximum number of concurrent remote procedure call requests the server can handle.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence. Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

Endpoint
Pointer to the endpoint-address information to use in creating a binding for the protocol sequence specified by *Protseq*.

SecurityDescriptor
Pointer to an optional parameter provided for the Microsoft Windows NT and Windows 2000 security subsystem.

Policy
Pointer to the **RPC_POLICY** structure, which contains flags that set transport-specific attributes. In the case of the **ncadg_mq** transport, these flags specify the properties of the server process–receive queue. In the case of the **ncacn_ip_tcp** or **ncadg_ip_udp** transports, these flags restrict port allocation for dynamic ports and allow multihomed computers to selectively bind to network interface cards.

The flag settings in the *Policy* field are effective only when the **ncacn_ip_tcp**, **ncadg_ip_udp**, or **ncadg_mq** protocol sequences are in use. For all other protocol sequences, the RPC run time ignores these values.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_PROTSEQ_NOT_SUPPORTED	Protocol sequence is not supported on this host.
RPC_S_INVALID_RPC_PROTSEQ	Invalid protocol sequence.
RPC_S_INVALID_ENDPOINT_FORMAT	Invalid endpoint format.
RPC_S_OUT_OF_MEMORY	Out of memory.
RPC_S_DUPLICATE_ENDPOINT	Endpoint is a duplicate.
RPC_S_INVALID_SECURITY_DESC	Security descriptor is invalid.

Remarks

The parameters and effects of **RpcServerUseProtseqEpEx** subsume those of **RpcServerUseProtseqEp**. The difference is the *Policy* argument, which allows you to set specific policies at the endpoints. Setting the *NICFlags* field of the **RPC_POLICY** structure to zero makes this extended function equivalent to the original **RpcServerUseProtseqEp** when used with the **ncacn_ip_tcp** or **ncadg_ip_udp** transports.

A server application calls **RpcServerUseProtseqEpEx** to register one protocol sequence with the RPC run-time library. With each protocol sequence registration, **RpcServerUseProtseqEpEx** includes the specified endpoint-address information.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call this routine many times to register additional protocol sequences and endpoints. For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server can handle. See *Server-Side Binding*, *String Binding*, *Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding*, and *RPC Message Queuing* and the MIDL reference pages ***message*** and ***ncadg_mq***.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RPC_POLICY, **RpcServerUseAllProtseqsEx**, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqIfEx**

RpcServerUseProtseqIf

The **RpcServerUseProtseqIf** function tells the RPC run-time library to use the specified protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls. This function is supported on all 32-bit Windows platforms, except Windows CE.

```
RPC_STATUS RPC_ENTRY RpcServerUseProtseqIf(
  unsigned char *Protseq,
  unsigned int MaxCalls,
  RPC_IF_HANDLE IfSpec,
  void *SecurityDescriptor
);
```

Parameters

Protseq
Pointer to a string identifier of the protocol sequence to register with the RPC run-time library.

MaxCalls
Specifies the maximum number of concurrent remote procedure call requests the server needs to be able to handle.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

IfSpec
Specifies the interface containing endpoint information to use in creating a binding for the protocol sequence specified in the *Protseq* argument.

SecurityDescriptor
Pointer to an optional parameter provided for the Microsoft Windows NT and Windows 2000 security subsystem.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_PROTSEQ_NOT_FOUND	The endpoint for this protocol sequence not specified in the IDL file.
RPC_S_PROTSEQ_NOT_SUPPORTED	Protocol sequence not supported on this host.
RPC_S_INVALID_RPC_PROTSEQ	Invalid protocol sequence.
RPC_S_INVALID_ENDPOINT_FORMAT	Invalid endpoint format.
RPC_S_OUT_OF_MEMORY	Out of memory.
RPC_S_INVALID_SECURITY_DESC	Security descriptor invalid.

Remarks

A server application calls **RpcServerUseProtseqIf** to register one protocol sequence with the RPC run-time library. With each protocol-sequence registration, the routine includes the endpoint-address information provided in the IDL file.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call this function multiple times to register additional protocol sequences.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server needs to be able to handle. See *Server-Side Binding* for a description of the routines that a server will typically call after registering protocol sequences. For a list of Microsoft RPC supported protocol sequences, see *String Binding*. To register all protocol sequences from the IDL file, a server calls **RpcServerUseAllProtseqsIf**.

Note The Microsoft RPC implementation of **RpcServerUseProtseqIf** includes a new, additional parameter, *SecurityDescriptor*, that does not appear in the DCE specification.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingVectorFree, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerListen**, **RpcServerRegisterIf**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**

RpcServerUseProtseqIfEx

The **RpcServerUseProtseqIfEx** function tells the RPC run-time library to use the specified protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls.

```
RPC_STATUS RPC_ENTRY RpcServerUseProtseqIfEx(
  unsigned char *Protseq,
  unsigned int MaxCalls,
  RPC_IF_HANDLE IfSpec,
  void *SecurityDescriptor,
  PRPC_POLICY Policy
);
```

Parameters

Protseq

Pointer to a string identifier of the protocol sequence to register with the RPC run-time library.

MaxCalls

Specifies the maximum number of concurrent remote procedure call requests the server wants to handle.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

IfSpec

Specifies the interface containing endpoint information to use in creating a binding for the protocol sequence specified in the *Protseq* argument.

SecurityDescriptor

Pointer to an optional parameter provided for the Microsoft Windows NT and Windows 2000 security subsystem.

Policy

Pointer to the **RPC_POLICY** structure, which contains flags to restrict port allocation for dynamic ports and that allow multihomed computers to selectively bind to network interface cards.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_PROTSEQ_NOT_FOUND	The endpoint for this protocol sequence is not specified in the IDL file.
RPC_S_PROTSEQ_NOT_SUPPORTED	Protocol sequence is not supported on this host.
RPC_S_INVALID_RPC_PROTSEQ	Invalid protocol sequence.
RPC_S_INVALID_ENDPOINT_FORMAT	Invalid endpoint format.
RPC_S_OUT_OF_MEMORY	Out of memory.
RPC_S_INVALID_SECURITY_DESC	Security descriptor is invalid.

Remarks

The parameters and effects of **RpcServerUseProtseqIfEx** extend those of **RpcServerUseProtseqIf**. The difference is the *Policy* parameter, which allows you to restrict port allocation for dynamic ports and allows multihomed computers to selectively bind to network interface cards.

Setting the *NICFlags* field of the **RPC_POLICY** structure to 0 makes this extended API functionally equivalent to the original **RpcServerUseProtseqIf**, and the server will bind to NICs based on the settings in the system registry. For information on how the registry settings define the available Internet and intranet ports, see *Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding*.

Note The flag settings in the *Policy* field are effective only when the **ncacn_ip_tcp** or **ncadg_ip_udp** protocol sequence is in use; for all other protocol sequences, the RPC run time ignores these values.

A server application calls **RpcServerUseProtseqIfEx** to register one protocol sequence with the RPC run-time library. With each protocol-sequence registration, the routine includes the endpoint-address information provided in the IDL file.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call this routine multiple times to register additional protocol sequences.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server needs to handle. To register all protocol sequences from the IDL file, a server calls **RpcServerUseAllProtseqsIfEx**.

See *Server-Side Binding* for a description of the routines that a server will typically call after registering protocol sequences. For a list of Microsoft RPC supported protocol sequences, see *String Binding*.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Configuring the Windows NT and Windows 2000 Registry for Port Allocations and Selective Binding, **RpcServerUseAllProtseqsEx**, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqEpEx**

RpcSmAllocate

The **RpcSmAllocate** function allocates memory within the RPC stub memory management function and returns a pointer to the allocated memory or NULL.

```
void * RPC_ENTRY RpcSmAllocate(
  size_t Size,
  RPC_STATUS *pStatus
);
```

Parameters

Size
Specifies the size of memory to allocate (in bytes).

pStatus
Specifies a pointer to the returned status.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_OUT_OF_MEMORY	Out of Memory.

Remarks

The **RpcSmAllocate** routine allows an application to allocate memory within the RPC stub memory–management environment. Prior to calling **RpcSmAllocate**, the memory-management environment must already be established. For memory management called within the stub, the server stub itself may establish the necessary environment. See ***RpcSmEnableAllocate*** for more information. When using **RpcSmAllocate** to allocate memory not called from the stub, the application must call **RpcSmEnableAllocate** to establish the required memory-management environment.

The **RpcSmAllocate** routine returns a pointer to the allocated memory if the call is successful. Otherwise, a NULL is returned.

When the stub establishes the memory management, it frees any memory allocated by **RpcSmAllocate**. The application can free such memory before returning to the calling stub by calling **RpcSmFree**.

By contrast, when the application establishes the memory management, it must free any memory allocated. It does so by calling either **RpcSmFree** or **RpcSmDisableAllocate**.

To manage the same memory within the stub memory–management environment, multiple threads can call **RpcSmAllocate** and **RpcSmFree**. In this case, the threads must share the same stub memory management thread handle. Applications pass thread handles from thread to thread by calling **RpcSmGetThreadHandle** and **RpcSmSetThreadHandle**.

See ***Memory Management*** or a complete discussion of the various memory management conditions supported by RPC.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcSmEnableAllocate, **RpcSmDisableAllocate**, **RpcSmFree**, **RpcSmGetThreadHandle**, **RpcSmSetThreadHandle**

RpcSmClientFree

The **RpcSmClientFree** function frees memory returned from a client stub.

```
RPC_STATUS RPC_ENTRY RpcSmClientFree(
  void *NodeToFree
);
```

Parameters

NodeToFree
Specifies a pointer to memory returned from a client stub.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

The **RpcSmClientFree** function releases memory allocated and returned from a client stub. The memory management handle of the thread calling this function must match the handle of the thread that made the RPC call. Use **RpcSmGetThreadHandle** and **RpcSmSetThreadHandle** to pass handles from thread to thread.

Note that using **RpcSmClientFree** allows a function to free dynamically-allocated memory returned by an RPC call without knowing the memory-management environment from which it was called.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmFree, **RpcSmGetThreadHandle**, **RpcSmSetClientAllocFree**, **RpcSmSetThreadHandle**, **RpcSmSwapClientAllocFree**

RpcSmDestroyClientContext

The **RpcSmDestroyClientContext** function reclaims the client memory resources for a context handle and makes the context handle NULL.

```
RPC_STATUS RPC_ENTRY RpcSmDestroyClientContext(
  void **ContextHandle
);
```

Parameters

ContextHandle
Specifies the context handle that can no longer be used.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_X_SS_CONTEXT_MISMATCH	Invalid handle.

Remarks

Client applications use **RpcSmDestroyClientContext** to reclaim resources from an inactive context handle. Applications can call **RpcSmDestroyClientContext** after a communications error makes the context handle unusable.

Note that when this function reclaims the memory resources, it also makes the context handle NULL.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmFree, **RpcSmGetThreadHandle**, **RpcSmSetClientAllocFree**, **RpcSmSetThreadHandle**, **RpcSmSwapClientAllocFree**

RpcSmDisableAllocate

The **RpcSmDisableAllocate** function frees resources and memory within the stub memory–management environment.

```
RPC_STATUS RPC_ENTRY RpcSmDisableAllocate(VOID);
```

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

The **RpcSmDisableAllocate** function frees all the resources used by a call to **RpcSmEnableAllocate**. It also releases memory allocated by a call to **RpcSmAllocate** after the call to **RpcSmEnableAllocate** and marked for deletion by the **RpcSmFree** function.

Note that **RpcSmEnableAllocate** and **RpcSmDisableAllocate** must be used together as matching pairs.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmAllocate, **RpcSmEnableAllocate**

RpcSmEnableAllocate

The **RpcSmEnableAllocate** function establishes the stub memory–management environment.

```
RPC_STATUS RPC_ENTRY RpcSmEnableAllocate(VOID);
```

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

In cases where the stub memory management is not enabled by the server stub itself, applications call **RpcSmEnableAllocate** to establish the stub memory–management environment. This environment must be established prior to making a call to **RpcSmAllocate**. In OSF-compatibility (/**osf**) mode, for server manager code called from the stub, the memory-management environment may be established by the server stub itself by using pointer manipulation or the **enable_allocate** attribute. In default (Microsoft-extended) mode, the environment is established only upon request by using the **enable_allocate** attribute. Otherwise, call **RpcSmEnableAllocate** before calling **RpcSmAllocate**. See ***Memory Management***, for a complete discussion of the memory management conditions used by RPC.

RpcSmGetThreadHandle and **RpcSmSetThreadHandle**

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmAllocate, **RpcSmDisableAllocate**

RpcSmFree

The **RpcSmFree** function releases memory allocated by **RpcSmAllocate**.

```
RPC_STATUS RPC_ENTRY RpcSmFree(
  void *NodeToFree
);
```

Parameters

NodeToFree
Specifies a pointer to memory allocated by **RpcSmAllocate** or **RpcSsAllocate**.

Return Values

The function **RpcSmFree** returns the following value.

Value	Meaning
RPC_S_OK	Call successful.

Remarks

Applications use **RpcSmFree** to free memory allocated by **RpcSmAllocate**. In cases where the stub allocates the memory for the application, **RpcSmFree** can also be used to release memory. See ***Memory Management*** for a complete discussion of memory management conditions supported by RPC.

To improve performance, the **RpcSmFree** function only marks memory for release. Memory is not actually released until your application calls the **RpcSmDisableAllocate** function. To release memory immediately, invoke the **midl_user_free** function.

Note that the handle of the thread calling **RpcSmFree** must match the handle of the thread that allocated the memory by calling **RpcSmAllocate.**. Use **RpcSmGetThreadHandle** and **RpcSmSetThreadHandle** to pass handles from thread to thread.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmAllocate, **RpcSmGetThreadHandle**, **RpcSmSetThreadHandle**, **midl_user_allocate**

RpcSmGetThreadHandle

The **RpcSmGetThreadHandle** function returns a thread handle, or NULL, for the stub memory–management environment.

```
RPC_SS_THREAD_HANDLE RPC_ENTRY RpcSmGetThreadHandle(
  RPC_STATUS *pStatus
);
```

Parameters

pStatus
Specifies a pointer to the returned status.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

Applications call **RpcSmGetThreadHandle** to obtain a thread handle for the stub memory–management environment. A thread used to manage memory for the stub memory–management environment uses **RpcSmGetThreadHandle** to receive a handle for its memory environment. In this way, another thread that calls **RpcSmSetThreadHandle** by using this handle can then use the same memory-management environment.

The same memory management thread handle must be used by multiple threads calling **RpcSmAllocate** and **RpcSmFree** in order to manage the same memory. Before spawning new threads to manage the same memory, the thread that established the memory-management environment (parent thread) calls **RpcSmGetThreadHandle** to obtain a thread handle for this environment. Then, the spawned threads call **RpcSmSetThreadHandle** with the new manager handle provided by the parent thread.

Typically a server manager procedure calls **RpcSmGetThreadHandle** before additional threads are spawned. The stub sets up the memory-management environment for the manager procedure, and the manager calls **RpcSmGetThreadHandle** to make this environment available to the other threads.

A thread can also call **RpcSmGetThreadHandle** and **RpcSmSetThreadHandle** to save and restore its memory-management environment.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmAllocate, **RpcSmFree**, **RpcSmSetThreadHandle**

RpcSmSetClientAllocFree

The **RpcSmSetClientAllocFree** function enables the memory allocation and release mechanisms used by the client stubs.

```
RPC_STATUS RPC_ENTRY RpcSmSetClientAllocFree(
  RPC_CLIENT_ALLOC *pfnAllocate,
  RPC_CLIENT_FREE *pfnFree
);
```

Parameters

pfnAllocate
The function used to allocate memory.

pfnFree
The function used to release memory and used with the function specified by *pfnAllocate*.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

By overriding the default routines used by the client stub to manage memory, **RpcSmSetClientAllocFree** establishes the memory allocation and memory-freeing mechanisms. Note that the default routines are **free** and **malloc**, unless the remote call occurs within manager code. In this case, the default memory–management functions are **RpcSmFree** and **RpcSmAllocate**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmAllocate, **RpcSmFree**

RpcSmSetThreadHandle

The **RpcSmSetThreadHandle** function sets a thread handle for the stub memory–management environment.

```
RPC_STATUS RPC_ENTRY RpcSmSetThreadHandle(
  RPC_SS_THREAD_HANDLE Handle
);
```

Parameters

Handle

Specifies a thread handle returned by a call to **RpcSmGetThreadHandle**.

Return Values

Value	**Meaning**
RPC_S_OK	Call successful.

Remarks

An application calls **RpcSmSetThreadHandle** to set a thread handle for the stub memory–management environment. A thread used to manage memory for the stub memory–management environment calls **RpcSmGetThreadHandle** to obtain a handle for its memory environment. In this way, another thread that calls **RpcSmSetThreadHandle** by using this handle can then use the same memory-management environment.

The same memory management–thread handle must be used by multiple threads calling **RpcSmAllocate** and **RpcSmFree** to manage the same memory. Before spawning new threads to manage the same memory, the thread that established the memory-management environment (parent thread) calls **RpcSmGetThreadHandle** to obtain a thread handle for this environment. Then, the spawned threads call **RpcSmSetThreadHandle** with the new manager handle provided by the parent thread.

Note that **RpcSmSetThreadHandle** is usually called by a thread spawned by a server-manager procedure. The stub sets up the memory-management environment for the manager procedure, and the manager calls **RpcSmGetThreadHandle** to obtain a thread handle. Then, each spawned thread calls **RpcSmGetThreadHandle** to get access to the manager's memory-management environment.

A thread can also call **RpcSmGetThreadHandle** and **RpcSmSetThreadHandle** to save and restore its memory-management environment.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmAllocate, **RpcSmGetThreadHandle**, **RpcSmFree**

RpcSmSwapClientAllocFree

The **RpcSmSwapClientAllocFree** function exchanges the client stub's memory-allocation and memory-freeing mechanisms with those supplied by the client.

```
RPC_STATUS RPC_ENTRY RpcSmSwapClientAllocFree(
  RPC_CLIENT_ALLOC *pfnAllocate,
  RPC_CLIENT_FREE *pfnFree,
  RPC_CLIENT_ALLOC **pfnOldAllocate,
  RPC_CLIENT_FREE **pfnOldFree
);
```

Parameters

pfnAllocate
Specifies a new memory-allocation function.

pfnFree
Specifies a new memory-releasing function.

pfnOldAllocate
Returns the previous memory-allocation function before the call to this function.

pfnOldFree
Returns the previous memory-releasing function before the call to this function.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_ARG	Invalid argument(s).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmAllocate, **RpcSmFree**, **RpcSmSetClientAllocFree**

RpcSsAllocate

The **RpcSsAllocate** function allocates memory within the RPC stub memory–management function, and returns a pointer to the allocated memory or NULL.

```
void __RPC_FAR  * RPC_ENTRY RpcSsAllocate(
  size_t Size
);
```

Parameters

Size
Specifies the size of memory to allocate (in bytes).

Return Values

Value	Meaning
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

The **RpcSsAllocate** function allows an application to allocate memory within the RPC stub memory–management function. Prior to calling **RpcSsAllocate**, the memory-management environment must already be established. For memory management called within the stub, the stub itself usually establishes the necessary environment. See *Memory Management* for a complete discussion of the various memory management models supported by RPC. When using **RpcSsAllocate** to allocate memory not called from the stub, the application must call **RpcSsEnableAllocate** to establish the required memory-management environment.

The **RpcSsAllocate** routine returns a pointer to the allocated memory, if the call was successful. Otherwise, it raises an exception.

When the stub establishes the memory management, it frees any memory allocated by **RpcSsAllocate.** The application can free such memory before returning to the calling stub by calling **RpcSsFree**.

By contrast, when the application establishes the memory management, it must free any allocated memory. It does so by calling either **RpcSsFree** or **RpcSsDisableAllocate**.

To manage the same memory within the stub memory–management environment, multiple threads can call **RpcSsAllocate** and **RpcSsFree**. In this case, the threads must share the same stub memory management–thread handle. Applications pass thread handles from thread-to-thread by calling **RpcSsGetThreadHandle** and **RPCSsSetThreadHandle**.

Note The **RpcSsAllocate** routine raises exceptions, unlike **RpcSmAllocate**, which returns the error code.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcSmAllocate, **RpcSsDisableAllocate**, **RpcSsEnableAllocate**, **RpcSsFree**, **RpcSsGetThreadHandle**, **RpcSsSetThreadHandle**

RpcSsDestroyClientContext

The **RpcSsDestroyClientContext** function destroys, without contacting the server, a context handle no longer needed by the client.

```
void RPC_ENTRY RpcSsDestroyClientContext(
  void **ContextHandle
);
```

Parameters

ContextHandle

Specifies the context handle to be destroyed. The handle is set to NULL before **RpcSsDestroyClientContext** returns.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_X_SS_CONTEXT_MISMATCH	Invalid context handle.

Remarks

RpcSsDestroyClientContext is used by the client application to reclaim the memory resources used to maintain a context handle on the client. This function is used when *ContextHandle* is no longer valid, such as when a communication failure has occurred and the server is no longer available. The context handle is set to NULL.

Do not use **RpcSsDestroyClientContext** to replace a server function that closes the context handle.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcBindingReset

RpcSsDisableAllocate

The **RpcSsDisableAllocate** function frees resources and memory within the stub memory–management environment.

```
void RPC_ENTRY RpcSsDisableAllocate(VOID);
```

Remarks

RpcSsDisableAllocate frees all the resources used by a call to **RpcSsEnableAllocate**. It also releases memory that was allocated by a call to **RpcSsAllocate** after the call to **RpcSsEnableAllocate**.

RpcSsEnableAllocate and **RpcSsDisableAllocate** must be used together as matching pairs.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcSmDisableAllocate, **RpcSsAllocate**, **RpcSsEnableAllocate**

RpcSsDontSerializeContext

The **RpcSsDontSerializeContext** function disables run-time serialization of multiple calls dispatched to server-manager routines on the same context handle.

```
void RPC_ENTRY RpcSsDontSerializeContext(void);
```

Remarks

The **RpcSsDontSerializeContext** function prevents the run time from performing this serialization service, allowing multiple calls to be dispatched on a given context handle. Calling this function does not disable serialization entirely—when a context run down occurs, your context run-down routine will not run until all outstanding client requests have completed. Changes to the context handle state, including freeing the context handle typically, must continue to be serialized.

It is recommended that, if your distributed application invokes the **RpcSsDontSerializeContext** function, the call should be made before the server program begins handling remote procedure calls.

Tip Typically, the RPC run-time serializes calls on the same context handle dispatched to server manager routines. Context handles are maintained on a per-client basis and typically represent the server-side state. This means that your server manager does not have to guard against another thread from the same client changing the context or against the context running down while a call is dispatched.

Note After it is called, the **RpcSsDontSerializeContext** function is not revertible for the life of the process.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

Server Context Rundown Routine, Multi-threaded Clients and Context Handles

RpcSsEnableAllocate

The **RpcSsEnableAllocate** function establishes the stub memory–management environment.

```
void RPC_ENTRY RpcSsEnableAllocate(VOID);
```

Return Values

Value	Meaning
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

In cases where the stub memory management is not enabled by the stub itself, an application calls **RpcSsEnableAllocate** to establish the stub memory–management environment. This environment must be established prior to making a call to **RpcSsAllocate**. For server manager code called from the stub, the memory-management environment is usually established by the stub itself. Otherwise, call **RpcSsEnableAllocate** before calling **RpcSsAllocate**. See *Memory Management* for a complete discussion of the memory-management conditions used by RPC. To learn how spawned threads use a stub memory–management environment, see ***RpcSsGetThreadHandle*** and ***RpcSsSetThreadHandle*** later in this section.

Note The **RpcSsEnableAllocate** function raises exceptions, while the **RpcSmEnableAllocate** function returns the error code.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmEnableAllocate, **RpcSsAllocate**, **RpcSsDisableAllocate**

RpcSsFree

The **RpcSsFree** function releases memory allocated by **RpcSsAllocate**.

```
void RPC_ENTRY RpcSsFree(
  void *NodeToFree
);
```

Parameters

NodeToFree
Specifies a pointer to memory allocated by **RpcSsAllocate** or **RpcSmAllocate**.

Remarks

An application uses **RpcSsFree** to free memory that was allocated with **RpcSsAllocate**. In cases where the stub allocates the memory for the environment, **RpcSsFree** can also be used to release memory. See ***Memory Management*** for a complete discussion of memory-management conditions supported by RPC.

Note that the handle of the thread calling **RpcSsFree** must match the handle of the thread that allocated the memory by calling **RpcSsAllocate**. Use **RpcSsGetThreadHandle** and **RpcSsSetThreadHandle** to pass handles from thread to thread.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcSmFree, **RpcSsAllocate**, **RpcSsGetThreadHandle**, **RpcSsSetThreadHandle**

RpcSsGetThreadHandle

The **RpcSsGetThreadHandle** function returns a thread handle for the stub memory–management environment.

```
RPC_SS_THREAD_HANDLE RPC_ENTRY RpcSsGetThreadHandle(VOID);
```

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

An application calls **RpcSsGetThreadHandle** to obtain a thread handle for the stub memory–management environment. A thread used to manage memory for the stub memory–management environment uses **RpcSsGetThreadHandle** to receive a handle for its memory environment. In this way, another thread that calls **RpcSsSetThreadHandle** by using this handle can then use the same memory-management environment.

The same thread handle must be used by multiple threads calling **RpcSsAllocate** and **RpcSsFree** to manage the same memory. Before spawning new threads to manage the same memory, the thread that established the memory-management environment (parent thread) calls **RpcSsGetThreadHandle** to obtain a thread handle for this environment. Then, the spawned threads call **RpcSsSetThreadHandle** with the handle provided by the parent thread.

Typically, a server manager procedure calls **RpcSsGetThreadHandle** before additional threads are spawned. The stub sets up the memory-management environment for the manager procedure, and the manager calls **RpcSsGetThreadHandle** to make this environment available to the other threads.

A thread can also call **RpcSsGetThreadHandle** and **RpcSsSetThreadHandle** to save and restore its memory-management environment.

Note **RpcSsGetThreadHandle** raises exceptions, while **RpcSmGetThreadHandle** returns the error code.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmGetThreadHandle, **RpcSsAllocate**, **RpcSsFree**, **RpcSsSetThreadHandle**

RpcSsSetClientAllocFree

The **RpcSsSetClientAllocFree** function enables the memory allocation and release mechanisms used by the client stubs.

```
void RPC_ENTRY RpcSsSetClientAllocFree(
  RPC_CLIENT_ALLOC *pfnAllocate,
  RPC_CLIENT_FREE *pfnFree
);
```

Parameters

pfnAllocate
Specifies the memory-allocation function.

pfnFree
Specifies the memory-releasing function used with the memory-allocation function specified by *pfnAllocate*.

Return Values

Value	Meaning
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

By overriding the default routines used by the client stub to manage memory, **RpcSsSetClientAllocFree** establishes the memory allocation and memory freeing mechanisms. Note that the default routines are **free** and **malloc**, unless the remote call occurs within manager code. In this case, the default memory–management routines are **RpcSsFree** and **RpcSsAllocate**.

Note that when **RpcSsSetClientAllocFree** reclaims the memory resources, it also makes the context handle NULL.

Note **RpcSsSetClientAllocFree** raises exceptions, unlike **RpcSmSetClientAllocFree**, which returns the error code.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmSetClientAllocFree, **RpcSsAllocate**, **RpcSsFree**

RpcSsSetThreadHandle

The **RpcSsSetThreadHandle** function sets a thread handle for the stub memory–management environment.

```
void RPC_ENTRY RpcSsSetThreadHandle(
  RPC_SM_THREAD_HANDLE Handle
);
```

Parameters

Handle
Specifies a thread handle returned by a call to **RpcSsGetThreadHandle**.

Remarks

An application calls **RpcSsSetThreadHandle** to set a thread handle for the stub memory–management environment. A thread used to manage memory for the stub memory–management environment calls **RpcSsGetThreadHandle** to obtain a handle for its memory environment. In this way, another thread that calls **RpcSsSetThreadHandle** by using this handle can then use the same memory-management environment.

The same thread handle must be used by multiple threads calling **RpcSsAllocate** and **RpcSsFree** in order to manage the same memory. Before spawning new threads to manage the same memory, the thread that established the memory-management environment (parent thread) calls **RpcSsGetThreadHandle** to obtain a thread handle for this environment. Then, the spawned threads call **RpcSsSetThreadHandle** with the handle provided by the parent thread.

Typically, a thread spawned by a server manager procedure calls **RpcSsSetThreadHandle**. The stub sets up the memory-management environment for the manager procedure, and the manager calls **RpcSsGetThreadHandle** to obtain a thread handle. Then, each spawned thread calls **RpcSsGetThreadHandle** to get access to the manager's memory-management environment.

A thread can also call **RpcSsGetThreadHandle** and **RpcSsSetThreadHandle** to save and restore its memory-management environment.

Note The **RpcSsSetThreadHandle** routine raises exceptions, while the **RpcSmSetThreadHandle** routine returns the error code.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

See Also

RpcSmSetThreadHandle, **RpcSsAllocate**, **RpcSsFree**, **RpcSsGetThreadHandle**

RpcSsSwapClientAllocFree

The **RpcSsSwapClientAllocFree** function exchanges the memory allocation and release mechanisms used by the client stubs with those supplied by the client.

```
void RPC_ENTRY RpcSsSwapClientAllocFree(
  RPC_CLIENT_ALLOC *pfnAllocate,
  RPC_CLIENT_FREE *pfnFree,
  RPC_CLIENT_ALLOC **pfnOldAllocate,
  RPC_CLIENT_FREE **pfnOldFree
);
```

Parameters

pfnAllocate
Specifies a new function to allocate memory.

pfnFree
Specifies a new function to release memory.

pfnOldAllocate
Returns the previous memory-allocation function.

pfnOldFree
Returns the previous memory-freeing function.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_OUT_OF_MEMORY	Out of memory.

Remarks

RpcSsSwapClientAllocFree exchanges the current memory allocation and memory freeing mechanisms with those supplied by the client.

Note **RpcSsSwapClientAllocFree** raises exceptions, unlike **RpcSmSwapClientAllocFree**, which returns the error code.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcSmSwapClientAllocFree, **RpcSsAllocate**, **RpcSsFree**, **RpcSsSetClientAllocFree**

RpcStringBindingCompose

The **RpcStringBindingCompose** function creates a string binding handle.

```
RPC_STATUS RPC_ENTRY RpcStringBindingCompose(
  unsigned char *ObjUuid,
  unsigned char *ProtSeq,
  unsigned char *NetworkAddr,
  unsigned char *EndPoint,
  unsigned char *Options,
  unsigned char **StringBinding
);
```

Parameters

ObjUuid

Pointer to a null-terminated string representation of an object UUID. For example, the string 6B29FC40-CA47-1067-B31D-00DD010662DA represents a valid UUID.

ProtSeq

Pointer to a null-terminated string representation of a protocol sequence. See Note.

NetworkAddr

Pointer to a null-terminated string representation of a network address. The network-address format is associated with the protocol sequence. See Note.

EndPoint

Pointer to a null-terminated string representation of an endpoint. The endpoint format and content are associated with the protocol sequence. For example, the endpoint associated with the protocol sequence **ncacn_np** is a pipe name in the format \pipe\pipename. See Note.

Options

Pointer to a null-terminated string representation of network options. The option string is associated with the protocol sequence. See Note.

StringBinding

Returns a pointer to a pointer to a null-terminated string representation of a binding handle.

Specify a NULL value to prevent **RpcStringBindingCompose** from returning the *StringBinding* argument. In this case, the application does not call **RpcStringFree**. See Note.

Note For more information, see *String Binding*.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_STRING_UUID	String representation of the UUID not valid.

Remarks

An application calls **RpcStringBindingCompose** routine to combine an object UUID, a protocol sequence, a network address, an endpoint and other network options into a string representation of a binding handle.

The RPC run-time library allocates memory for the string returned in the *StringBinding* argument. The application is responsible for calling **RpcStringFree** to deallocate that memory.

Specify a null parameter value or provide an empty string (\0) for each input string that has no data.

Literal backslash characters within C-language strings must be quoted. The actual C string for the server name appears as \\servername, and the actual C string for a pipe name appears as \\pipe\\pipename.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingFromStringBinding, **RpcBindingToStringBinding**, **RpcStringBindingParse**, **RpcStringFree**

RpcStringBindingParse

The **RpcStringBindingParse** function returns the object UUID part and the address parts of a string binding as separate strings. An application calls **RpcStringBindingParse** to parse a string representation of a binding handle into its component fields. The **RpcStringBindingParse** function returns the object UUID part and the address parts of a string binding as separate strings.

```
RPC_STATUS RPC_ENTRY RpcStringBindingParse(
  unsigned char *StringBinding,
  unsigned char **ObjectUuid,
  unsigned char **ProtSeq,
  unsigned char **NetworkAddr,
  unsigned char **EndPoint,
  unsigned char **NetworkOptions
);
```

Parameters

StringBinding

Pointer to a null-terminated string representation of a binding.

ObjectUuid

Returns a pointer to a pointer to a null-terminated string representation of an object UUID.

Specify a NULL value to prevent **RpcStringBindingParse** from returning the *ObjectUuid* parameter. In this case, the application does not call **RpcStringFree**.

ProtSeq

Returns a pointer to a pointer to a null-terminated string representation of a protocol sequence. For a list of Microsoft RPC supported protocol sequences, see *String Binding*.

Specify a NULL value to prevent **RpcStringBindingParse** from returning the *ProtSeq* parameter. In this case, the application does not call **RpcStringFree**.

NetworkAddr

Returns a pointer to a pointer to a null-terminated string representation of a network address. Specify a NULL value to prevent **RpcStringBindingParse** from returning the *NetworkAddr* parameter. In this case, the application does not call **RpcStringFree**.

EndPoint

Returns a pointer to a pointer to a null-terminated string representation of an endpoint. Specify a NULL value to prevent **RpcStringBindingParse** from returning the *EndPoint* parameter. In this case, the application does not call **RpcStringFree**.

NetworkOptions

Returns a pointer to a pointer to a null-terminated string representation of network options.

Specify a NULL value to prevent **RpcStringBindingParse** from returning the *NetworkOptions* parameter. In this case, the application does not call **RpcStringFree**.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_STRING_BINDING	Invalid string binding.

Remarks

An application calls **RpcStringBindingParse** routine to parse a string representation of a binding handle into its component fields.

The RPC run-time library allocates memory for each component string returned. The application is responsible for calling **RpcStringFree** once for each returned string to deallocate the memory for that string.

If any field of the *StringBinding* argument is empty, **RpcStringBindingParse** returns an empty string (\0) in the corresponding output parameter.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcBindingFromStringBinding, **RpcBindingToStringBinding**, **RpcStringBindingCompose**, **RpcStringFree**

RpcStringFree

The **RpcStringFree** function frees a character string allocated by the RPC run-time library.

```
RPC_STATUS RPC_ENTRY RpcStringFree(
  unsigned char **String
);
```

Parameters

String
Pointer to a pointer to the character string to free.

Return Values

Value	Meaning
RPC_S_OK	Call successful.

Remarks

An application is responsible for calling **RpcStringFree** once for each character string allocated and returned by calls to other RPC run-time library routines.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

RpcBindingToStringBinding, **RpcNsBindingInqEntryName**, **RpcStringBindingParse**

RpcTestCancel

The **RpcTestCancel** function checks for a cancel indication.

```
RPC_STATUS RPC_ENTRY RpcTestCancel(VOID);
```

Return Values

Value	Meaning
RPC_S_OK	Call has been canceled.
Other values	Call has not been canceled.

It is not unusual for the **RpcTestCancel** function to return the value ERROR_ACCESS_DENIED. This indicates that the remote procedure call has not been canceled.

Remarks

An application server stub calls **RpcTestCancel** to determine whether a call has been canceled. If the call has been canceled, RPC_S_OK is returned; otherwise, another value is returned.

This function should be called periodically by the server stub so that it can respond to cancels in a timely fashion. If the function returns RPC_S_OK, the stub should clean up its data structures and return to the client.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

RpcWinSetYieldInfo

The **RpcWinSetYieldInfo** function configures Microsoft Windows 3.*x* client applications to yield to other applications during remote procedure calls.

```
RPC_STATUS
RpcWinSetYieldInfo(
  HWND hWnd,
  BOOL fCustomYield,
  WORD wMsg,
  DWORD dwOtherInfo
);
```

Parameters

hWnd

Identifies the application window that receives messages relating to yielding.

Applications should usually specify the parent window of the dialog box. Standard yield applications receive messages for both the start and end of the yield period. Custom yield applications receive messages that indicate when the RPC operation has completed.

fCustomYield

Specifies the yielding method. The following values are defined:

Value	Yield method
TRUE	Custom yield.
FALSE	Standard yield.

wMsg

Specifies the message that is posted by the RPC run-time library to notify the application of RPC events. The message value should be in the range beginning with WM_USER. If a zero value is specified, no message is posted.

For standard-yield applications, the message indicates the beginning or end of the yield period. This allows the application to refrain from performing operations that are illegal during an RPC operation. Standard-yield applications use the following values of *wParam* and *lParam* with this message.

Parameter	Value	Description
wParam	1	Yield period beginning.
wParam	0	Yield period ending.
lParam	–	Unused.

For a custom-yield application, the *wMsg* message notifies the application that the RPC operation is complete. When the application receives this message, it should immediately return control to the RPC run-time library by having the callback function return. The values of *wParam* and *lParam* are set to zero and are not used.

dwOtherInfo

Specifies additional information about the yielding behavior.

For standard-yield applications, *dwOtherInfo* contains an optional **HANDLE** to an application-supplied dialog-box resource. This handle is passed as the second parameter to the **DialogBoxIndirect** function. If the handle specified by *dwOtherInfo* is zero, the default dialog box supplied by the RPC run-time library is used.

For custom-yield applications, *dwOtherInfo* contains the procedure-instance address of the application supplied–callback function.

Return Values

Value	Meaning
RPC_S_OK	Information was set successfully.
RPC_S_OUT_OF_MEMORY	Memory could not be allocated to store the information for this task.

Remarks

The **RpcWinSetYieldInfo** function supports two yielding methods:

- Standard yield method. The RPC run-time library provides a standard modal dialog box that includes a single push-button control with an IDCANCEL identifier. The dialog box prevents direct user input, such as mouse and keyboard events, from being sent to the application. The application continues to receive messages while the dialog box is present. The IDCANCEL message indicates that the application user wants to end the remote procedure.
- Custom yield method. The application provides a callback function that the RPC run-time library calls while a remote operation is in progress. The callback function must retrieve messages from the message queue (including mouse and keyboard messages) and must process messages (both queued and nonqueued). The RPC run-time library posts a message to the application's queue when the RPC operation is complete. The callback function returns a boolean value to the RPC run-time library.

When a conventional RPC client application makes a remote procedure call, the MIDL-generated stub calls the RPC run-time library and the library calls the appropriate transport. These calls are synchronous and block until the server side sends back a response. In the cooperatively multitasked Windows 3.*x* environment, an active, blocked application prevents Windows and other Windows applications from running. The **RpcWinSetYieldInfo** function allows you to direct the application to yield to Windows and other Windows applications while waiting for an RPC operation to finish.

Windows RPC client applications can be organized into three classes that correspond to levels of yielding support: no yielding, standard yielding, and custom yielding.

- Some applications do not yield. RPC calls block until completion.
- Standard-yield applications are RPC-aware applications that yield but do not need to perform special handling.
- Custom-yield applications are those that are RPC aware and want to perform special handling while an RPC operation is in progress.

You can replace the provided dialog-box resource with an application-specified dialog-box resource. The resource must use the same style as the default and must contain a single push-button control with an IDCANCEL ID. The dialog-box function is part of the RPC run-time library and cannot be replaced.

To yield in a well-behaved manner from within the context of a pending RPC operation, applications must observe the following rules:

- Do not make another RPC call. If the RPC run-time library detects that a new call is being made during the yielding period, it returns an error to the caller. This is particularly important if the application makes RPC calls in response to common messages, such as WM_PAINT.
- Do not exit the application. Do not close the window specified by the *hWnd* handle parameter. Your application can process WM_CLOSE messages in the window procedure and not call **DefWindowProc** during the yielding period.
- Return FALSE in response to WM_QUERYENDSESSION messages. Alternatively, a custom-yield application can use this message as a signal to cause *YieldFunctionName* to return FALSE to the RPC run-time library and end the yielding period.

There is no guarantee that any code that supports yielding will be invoked. Whether or not an application yields depends on the specific call, the current state of the underlying system, and the implementation of the underlying RPC transport. Applications should not rely on this code to do anything other than manage yielding.

The **RpcWinSetYieldInfo** function can be called more than once by an application. Each call simply replaces the information stored in the previous calls.

Note This function is only available for 16-bit Windows client applications. A 32-bit application should take advantage of the preemptive multitasking and multithreading support that the 32-bit Windows operating systems provide.

! Requirements

Windows NT/2000: Unsupported.
Windows 95/98: Unsupported.
Version: Requires 16-bit client only.
Header: Declared in Rpc.h.
Library: Use Rpcrt4.lib.

See Also

DefWindowProc, **DialogBoxIndirect**, **MakeProcInstance**, **YieldFunctionName**

RpcWinSetYieldTimeout

The **RpcWinSetYieldTimeout** function configures the amount of time an RPC call will wait for the server to respond before invoking the application's RPC yielding mechanism. This function is only available for Windows 3.*x* applications.

```
RPC_STATUS RPC_ENTRY RpcWinSetYieldTimeout(
  unsigned int Timeout
);
```

Parameters

Timeout

Specifies the time-out value in milliseconds. If this function is not called, the default is 500 milliseconds.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_CANNOT_SUPPORT	**RpcWinSetYieldInfo** must be called prior to **RpcWinSetYieldTimeout**.

Remarks

Depending on the type of yielding specified in **RpcWinSetYieldInfo**, this can either produce a dialog box or signal the application.

If the *Timeout* value is small, the yielding mechanism can be invoked too often. This results in loss of performance. Conversely, if the value specified for *Timeout* is too large, the application and system will be frozen for the time-out period. To avoid this, use time-outs in the range of 500 to 2000 milliseconds.

The **RpcWinSetYieldTimeout** function can be called more than once by an application. Each call simply replaces the information stored in the previous calls.

Requirements

Windows NT/2000: Unsupported.
Windows 95/98: Unsupported.
Version: Requires 16-bit client only.
Header: Declared in Rpc.h.
Library: Use Rpcrt4.lib.

See Also

RpcWinSetYieldInfo

UuidCompare

An application calls **UuidCompare** routine to compare two UUIDs and determine their order. The returned value gives the order.

```
signed int RPC_ENTRY UuidCompare(
  UUID *Uuid1,
  UUID *Uuid2,
  RPC_STATUS *Status
);
```

Parameters

Uuid1

Specifies a pointer to a UUID. This UUID is compared with the UUID specified in the *Uuid2* argument.

Uuid2

Specifies a pointer to a UUID. This UUID is compared with the UUID specified in the *Uuid1* argument.

Status

Returns any errors that may occur, and will typically be set by the function to RPC_S_OK upon return.

Return Values

Values	Meaning
–1	The *Uuid1* argument is less than the *Uuid2* argument.
0	The *Uuid1* argument is equal to the *Uuid2* argument.
1	The *Uuid1* argument is greater than the *Uuid2* argument.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

UuidCreate

UuidCreate

The **UuidCreate** function creates a new UUID.

```
RPC_STATUS RPC_ENTRY UuidCreate(
  UUID *Uuid
);
```

Parameters

Uuid
Returns a pointer to the created UUID.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_UUID_LOCAL_ONLY	The UUID is guaranteed to be unique to this computer only.
RPC_S_UUID_NO_ADDRESS	Cannot get Ethernet or token-ring hardware address for this computer.

Remarks

For security reasons, it is often desirable to keep ethernet/token ring addresses on networks from becoming available outside a company or organization. The **UuidCreate** function generates a UUID that cannot be traced to the ethernet/token ring address of the computer on which it was generated. It also cannot be associated with other UUIDs created on the same computer. If you do not need this level of security, your application can use the **UuidCreateSequential** function.

In Windows NT 4.0, Windows 2000, Windows 95, DCOM release, and Windows 98, **UuidCreate** returns RPC_S_UUID_LOCAL_ONLY when the originating computer does not have an ethernet/token ring (IEEE 802.*x*) address. In this case, the generated UUID is a valid identifier, and is guaranteed to be unique among all UUIDs generated on the computer. However, the possibility exists that another computer without an ethernet/token ring address generated the identical UUID. Therefore you should never use this UUID to identify an object that is not strictly local to your computer. Computers with ethernet/token ring addresses generate UUIDs that are guaranteed to be globally unique.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

UuidFromString, **UuidToString**

UuidCreateSequential

The **UuidCreateSequential** function creates a new UUID.

```
RPC_STATUS RPC_ENTRY UuidCreateSequential(
  UUID *Uuid
);
```

Parameters

Uuid

Returns a pointer to the created UUID.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_UUID_LOCAL_ONLY	The UUID is guaranteed to be unique to this computer only.
RPC_S_UUID_NO_ADDRESS	Cannot get Ethernet or token-ring hardware address for this computer.

Remarks

In Windows NT 4.0, Windows 2000, Windows 95, DCOM release, and Windows 98, **UuidCreateSequential** returns RPC_S_UUID_LOCAL_ONLY when the originating computer does not have an ethernet/token ring (IEEE 802.*x*) address. In this case, the generated UUID is a valid identifier, and is guaranteed to be unique among all UUIDs generated on the computer. However, the possibility exists that another computer without an ethernet/token ring address generated the identical UUID. Therefore you should never use this UUID to identify an object that is not strictly local to your computer. Computers with ethernet/token ring addresses generate UUIDs that are guaranteed to be globally unique.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpct4.lib.

See Also

UuidFromString, **UuidToString**

UuidCreateNil

The **UuidCreateNil** function creates a nil-valued UUID.

```
RPC_ENTRY UuidCreateNil(
  UUID *Nil_Uuid
);
```

Parameters

Nil_Uuid
  Returns a nil-valued UUID.

Return Values

Returns any errors that may occur. The parameter is typically set by the function to RPC_S_OK upon return.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

UuidEqual

An application calls **UuidEqual** function to compare two UUIDs and determine whether they are equal.

```
int RPC_ENTRY UuidEqual(
  UUID *Uuid1,
  UUID *Uuid2,
  RPC_STATUS *Status
);
```

Parameters

Uuid1
  Specifies a pointer to a UUID. This UUID is compared with the UUID specified in the *Uuid2* argument.

Uuid2
Specifies a pointer to a UUID. This UUID is compared with the UUID specified in the *Uuid1* argument.

Status
Returns any errors that may occur, and will usually be set by the function to RPC_S_OK upon return.

Return Values

Value	Meaning
TRUE	The *Uuid1* argument is equal to the *Uuid2* argument.
FALSE	The *Uuid1* argument is not equal to the *Uuid2* argument.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

UuidCreate

UuidFromString

The **UuidFromString** function converts a string to a UUID.

```
RPC_STATUS RPC_ENTRY UuidFromString(
  unsigned char *StringUuid,
  UUID *Uuid
);
```

Parameters

StringUuid
Pointer to a string representation of a UUID.

Uuid
Returns a pointer to a UUID in binary form.

Remarks

An application calls **UuidFromString** function to convert a string UUID to a binary UUID.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_INVALID_STRING_UUID	The string UUID is invalid.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

UuidToString

UuidHash

An application calls the **UuidHash** function to generate a hash value for a specified UUID.

```
unsigned short RPC_ENTRY UuidHash(
  UUID *Uuid,
  RPC_STATUS *Status
);
```

Parameters

Uuid
Specifies the UUID for which a hash value is created.

Status
Returns any errors that may occur, and will usually be set by the function to RPC_S_OK upon return.

Remarks

An application calls **UuidHash** to generate a hash value for a specified UUID. The hash value returned is implementation dependent and may vary from implementation to implementation.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

+ See Also

UuidCreate

UuidIsNil

An application calls **UuidIsNil** function to determine whether the specified UUID is a nil-valued UUID.

```
int RPC_ENTRY UuidIsNil(
  UUID *Uuid,
  RPC_STATUS *Status
);
```

Parameters

Uuid
Specifies a UUID to test for nil value.

Status
Returns any errors that may occur, and will typically be set by the function to RPC_S_OK upon return.

Remarks

This function acts as though the application called **UuidCreateNil**, and then called the **UuidEqual** to compare the returned nil-value UUID to the UUID specified in the *Uuid* argument.

Upon completion, one of the following values is returned.

Returned Value	Meaning
TRUE	The *Uuid* argument is a nil-valued UUID.
FALSE	The *Uuid* argument is not a nil-valued UUID.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.

See Also

UuidCreate

UuidToString

The **UuidToString** function converts a UUID to a string.

```
RPC_STATUS RPC_ENTRY UuidToString(
  UUID *Uuid,
  unsigned char **StringUuid
);
```

Parameters

Uuid

Pointer to a binary UUID.

StringUuid

Returns a pointer to a pointer to the string representation of the UUID specified in the *Uuid* argument.

Specify a NULL value to prevent **UuidToString** from returning the *StringUuid* parameter. In this case, the application does not call **RpcStringFree**.

Return Values

Value	Meaning
RPC_S_OK	Call successful.
RPC_S_OUT_OF_MEMORY	No memory.

Remarks

An application calls **UuidToString** to convert a binary UUID to a string UUID. The RPC run-time library allocates memory for the string returned in the *StringUuid* argument. The application is responsible for calling **RpcStringFree** to deallocate that memory.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.
Library: Use Rpcrt4.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

RpcStringFree, **UuidFromString**

CHAPTER 26

RPC Callback and Notification Functions

MACYIELDCALLBACK

The **MACYIELDCALLBACK** function is required on the Mac for cooperative multitasking.

```
typedef void  (RPC_ENTRY *MACYIELDCALLBACK)(
    short *pYeildStatus
) ;
```

Parameters

pYeildStatus

A pointer to a short integer containing the current yield status. The application should yield control when this parameter equals 1.

Return Values

This function does not return a value.

Remarks

Register a yielding function by calling **RpcMacSetYieldInfo** with a pointer to the callback (yielding) function. If a yielding function is not registered, an RPC will not yield on the Mac.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Rpc.h.

See Also

RpcMacSetYieldInfo

RPC_AUTH_KEY_RETRIEVAL_FN

```
typedef void (* RPC_AUTH_KEY_RETRIEVAL_FN)(
    void *Arg,
    unsigned char *ServerPrincName,
    unsigned long KeyVer,
    void **Key,
    RPC_STATUS *Status
);
```

Parameters

Arg
: Pointer to a user-defined argument to the user-supplied encryption key acquisition function. The RPC run-time library uses the *Arg* argument supplied to **RpcServerRegisterAuthInfo**.

ServerPrincName
: Pointer to the principal name to use for the server when authenticating remote procedure calls. The RPC run-time library uses the *ServerPrincName* parameter supplied to **RpcServerRegisterAuthInfo**.

KeyVer
: Specifies the value that the RPC run-time library automatically provides for the key-version parameter. When the value is zero, the acquisition function must return the most recent key available.

Key
: Pointer to a pointer to the authentication key returned by the user-supplied function.

Status
: Pointer to the status returned by the acquisition function when it is called by the RPC run-time library to authenticate the client RPC request. If the status is other than RPC_S_OK, the request fails and the run-time library returns the error status to the client application.

Remarks

An authorization key-retrieval function specifies the address of a server-application-provided routine returning encryption keys.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RpcServerRegisterAuthInfo

RPC_IF_CALLBACK_FN

The RPC_IF_CALLBACK_FN is a prototype for a security-callback function that your application supplies. Your program can provide a callback function for each interface it defines.

```
typedef RPC_STATUS RPC_IF_CALLBACK_FN(
    IN RPC_IF_ID *Interface,
    IN void *Context
);
```

Parameters

Interface

Contains the UUID and version of the interface in question.

Context

A server binding handle representing the client. The callback function may pass this handle to **RpcImpersonateClient** or **RpcBindingServerFromClient** to gain information about the client.

Return Values

The callback function should return RPC_S_OK if the client is allowed to call methods in this interface. Any other return code will cause the client to receive the exception RPC_S_ACCESS_DENIED.

Remarks

In some cases, the RPC run time may call the security-callback function more than once per client per interface. Be sure your callback function can handle this possibility.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RpcServerRegisterIfEx

RPC_MGMT_AUTHORIZATION_FN

The RPC_MGMT_AUTHORIZATION_FN enables server programs to implement custom RPC authorization techniques.

```
typedef int (__RPC_API *RPC_MGMT_AUTHORIZATION_FN) (
    RPC_BINDING_HANDLE ClientBinding,
    ULONG RequestedMgmtOperation,
    RPC_STATUS *Status
);
```

Parameters

ClientBinding

The client/server binding handle.

RequestedMgmtOperation

The value for *RequestedMgmtOperation* depends on the remote function requested, as shown in the following table.

Called remote function	***RequestedMgmtOperation* value**
RpcMgmtInqIfIds	RPC_C_MGMT_INQ_IF_IDS
RpcMgmtInqServerPrincName	RPC_C_MGMT_INQ_PRINC_NAME
RpcMgmtInqStats	RPC_C_MGMT_INQ_STATS
RpcMgmtIsServerListening	RPC_C_MGMT_IS_SERVER_LISTEN
RpcMgmtStopServerListening	RPC_C_MGMT_STOP_SERVER_LISTEN

The authorization function must handle all of these values.

Status

If *Status* is either 0 (zero) or RPC_S_OK, the *Status* value RPC_S_ACCESS_DENIED is returned to the client by the remote management function. If the authorization function returns any other value for *Status*, that *Status* value is returned to the client by the remote management function.

Return Values

Returns TRUE if the calling client is allowed access to the requested management function. If the authorization function returns FALSE, the management function cannot execute. In this case, the function returns a *Status* value to the client:

Remarks

When a client requests one of the server's remote management functions, the server run-time library calls the authorization function with *ClientBinding* and *RequestedMgmtOperation*. The authorization function uses these parameters to determine whether the calling client can execute the requested management function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

Authorization Functions, **RpcMgmtSetAuthorizationFn**

RPC_OBJECT_INQ_FN

```
typedef void RPC_OBJECT_INQ_FN(
    UUID *ObjectUuid,
    UUID *TypeUuid,
    RPC_STATUS *Status
);
```

Parameters

ObjectUuid
Pointer to the variable that specifies the object UUID that is to be mapped to a type UUID.

TypeUuid
Pointer to the address of the variable that is to contain the type UUID derived from the object UUID. The type UUID is returned by the function.

Status
Pointer to a return value for the function.

Remarks

You can replace the default mapping function that maps object UUIDs to type UUIDs by calling **RpcObjectSetInqFn** and supplying a pointer to a function of type RPC_OBJECT_INQ_FN. The supplied function must match the function prototype specified by the type definition: a function with three parameters and the function return value of void.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

RpcObjectSetInqFn

RPCNOTIFICATION_ROUTINE

The **RPCNOTIFICATION_ROUTINE** function provides programs that utilize asynchronous RPC with the ability to customize responses to asynchronous events.

```
typedef void RPC_ENTRY RPCNOTIFICATION_ROUTINE(
    struct _RPC_ASYNC_STATE *pAsync,
    void *Context,
    RPC_ASYNC_EVENT Event
);
```

Parameters

pAsync

Pointer to a structure that contains the current state of the asynchronous RPC run-time library. For details, see ***RPC_ASYNC_STATE***.

Context

Reserved for future use. Windows 2000 currently sets this parameter to NULL.

Event

A value from the RPC_ASYNC_EVENT enumerated type that identifies the current asynchronous event.

Remarks

For each asynchronous remote procedure call that a client program executes, it can specify an asynchronous procedure call (APC). The RPC run-time library will invoke the APC when the asynchronous remote procedure call completes. The APC function must match the prototype specified by **RPCNOTIFICATION_ROUTINE**.

Return Values

This function does not return a value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcdce.h.

See Also

Asynchronous RPC, RPC_ASYNC_STATE

YieldFunctionName

The **YieldFunctionName** function is a placeholder name for the application-supplied function name provided as a parameter to the **RpcWinSetYieldInfo** routine.

```
BOOL FAR PASCAL  YieldFunctionName(VOID);
```

Remarks

The callback function must retrieve messages from the message queue (including mouse and keyboard messages) and must process messages, both queued and nonqueued.

The **YieldFunctionName** function should return TRUE when the application is notified that the RPC operation has completed (by receiving the *wMsg* message). It is an error for **YieldFunctionName** to return TRUE if it has not been notified that the RPC operation has completed.

The **YieldFunctionName** function should return FALSE if the user wants to cancel the RPC operation in progress. The RPC run-time library then attempts to abort the current operation, which is likely to result in the RPC call returning an error to the application. Note that due to race conditions, the operation can complete successfully even if **YieldFunctionName** returns FALSE.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpcndr.h.

See Also

RpcWinSetYieldInfo

CHAPTER 27

RPC Macros

This chapter details the macros that are used with the RPC run-time library.

- Portability Macros
- **RpcAsyncGetCallHandle**
- **RpcEndExcept**
- **RpcEndFinally**
- **RpcExcept**
- **RpcFinally**
- **RpcTryExcept**
- **RpcTryFinally**

Portability Macros

The RPC tools achieve model, calling, and naming-convention independence by associating data types and function-return types in the generated stub files and header files with definitions that are specific to each platform. These macro definitions ensure that any data types and functions that require the designation of **__far** are specified as far objects.

RPC macros are defined as follows.

Definition	Description
__RPC_API	Applied to calls made by the stub to the user application. Both functions are in the same executable program.
__RPC_FAR	Applied to the standard macro definition for pointers. This macro definition should appear as part of the signature of all user-supplied functions.
__RPC_STUB	Applied to calls made from the run-time library to the stub. These calls can be considered private.
__RPC_USER	Applied to calls made by the run-time library to the user application. These cross the boundary between a DLL and an application.
RPC_ENTRY	Applied to calls made by the application or stub to the run-time library. This macro definition is applied to all RPC run-time functions.

Figure 27-1 shows the macro definitions that the MIDL compiler applies to function calls between RPC components:

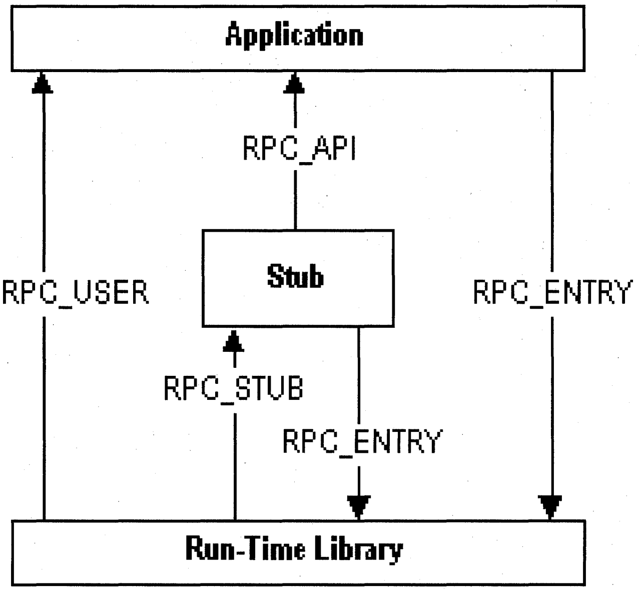


Figure 27-1: Macro Definitions Applied by the MIDL Compiler.

To link correctly with the Microsoft RPC run-time libraries, stubs, and support routines, some user-supplied functions must also include these macros in the function definition. Use the macro __**RPC_API** when you define the functions associated with memory management, user-defined binding handles, and the **transmit_as** attribute, and use the macro __**RPC_USER** when you define the context run-down routine associated with the context handle. Specify the functions as:

__RPC_USER midl_user_allocate(...)
__RPC_USER midl_user_free(...)
__RPC_USER *handletype*_**bind(...)**
__RPC_USER *handletype*_**unbind(...)**
__RPC_USER *type*_**to_local**
__RPC_USER *type*_**from_local**
__RPC_USER *type*_**to_xmit(...)**
__RPC_USER *type*_**from_xmit(...)**
__RPC_USER type_**free_local**
__RPC_USER *type*_**free_inst(...)**
__RPC_USER *type*_**free_xmit(...)**
__RPC_USER context_rundown(...)

Note All pointer parameters in these functions must be specified using the macro **__RPC_FAR**.

These are the two approaches that can be used to select an application's memory model:

1. To use a single memory model for all files, compile all source files using the same memory-model compiler switches. For example, to develop a small-model application, compile both the application and the stub source code using the C-compiler switch **/AS**, as in the following:

```
cl -c /AS myfunc.c
cl -c /AS clstub_c.c
```

2. To use different memory models for the application source files and the support source files (stubs files), use the RPC macros when you define function prototypes in the IDL file. Compile the distributed-application source files using one compiler memory-model setting and compile the support files using another compiler memory-model setting. Use the same memory model for all of the files generated by the compiler.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpc.h.

RpcAsyncGetCallHandle

The **RpcAsyncGetCallHandle** macro returns the binding handle on an asynchronous remote procedure call. This macro is supported in Windows 2000, Windows 95 with the second DCOM OSR, and Windows 98.

```
PVOID RpcAsyncGetCallHandle(
    PRPC_ASYNC_STATE pAsync
);
```

Parameters

pAsync
Pointer to the RPC_ASYNC_STATE structure that contains asynchronous call information.

Remarks

Given an asynchronous handle, it returns the corresponding binding handle. For example, the **RpcServerTestCancel** function uses **RpcAsyncGetCallHandle**(*pAsync*) as a parameter to test for cancel requests.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in Rpcasync.h.
Library: Use Rpcrt4.lib.

+ See Also

Asynchronous RPC, RPC_ASYNC_STATE, **RpcAsyncAbortCall**, **RpcAsyncCancelCall**, **RpcAsyncCompleteCall**, **RpcAsyncGetCallStatus**, **RpcAsyncInitializeHandle**, **RpcAsyncRegisterInfo**, **RpcServerTestCancel**

RpcEndExcept

Use the **RpcEndExcept** statement to terminate all **RpcTryExcept** statements.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpc.h.
Library: Use Rpcrt4.lib.

+ See Also

RpcTryExcept, **RpcExcept**

RpcEndFinally

Use the **RpcEndFinally** statement to terminate all **RpcTryFinally** statements.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpc.h.
Library: Use Rpcrt4.lib.

See Also

RpcTryFinally, **RpcFinally**

RpcExcept

The **RpcExcept** statement provides structured exception handling for RPC applications.

```
RpcTryExcept
{
    guarded statements
}
RpcExcept(expression)
{
    exception statements
}
RpcEndExcept;
```

Parameters

guarded statements
Specifies program statements that are guarded or monitored for exceptions during execution.

expression
An expression that is evaluated when an exception occurs. If *expression* evaluates to a nonzero value, the exception statements are executed. If *expression* evaluates to a zero value, unwinding continues to the next **RpcTryExcept** or **RpcTryFinally** function.

exception statements
Statements that are executed when the expression evaluates to a nonzero value.

Remarks

If an exception does not occur, the *expression* and *exception statements* are skipped and execution continues at the statement following the **RpcEndExcept** statement.

The compound statement after the **RpcTryExcept** clause is the body or guarded section. The compound statement after the **RpcExcept** clause is the exception handler. The handler specifies a set of actions to be taken if an exception is raised during execution of the body of the guarded section. Execution proceeds as follows:

1. The guarded section is executed.
2. If no exception occurs during execution of the guarded section, execution continues at the statement after the __except clause.

3. If an exception occurs during execution of the guarded section or in any routine the guarded section calls, the __except expression is evaluated and the value determines how the exception is handled. There are three values:
   - EXCEPTION_CONTINUE_EXECUTION (–1) Exception is dismissed. Continue execution at the point where the exception occurred.
   - EXCEPTION_CONTINUE_SEARCH (0) Exception is not recognized. Continue to search up the stack for a handler, first for containing try-except statements, then for handlers with the next highest precedence.
   - Exception is recognized. Transfer control to the exception handler by executing the __except compound statement, then continue execution after the __except block.

Because the **RpcExcept** expression is evaluated as a C expression, it is limited to a single value, the conditional-expression operator, or the comma operator. If more extensive processing is required, the expression can call a routine that returns one of the three values listed above.

RpcExceptionCode can be used in both *expression* and *exception statements* to determine which exception occurred.

The following restrictions apply:

- Jumping (through a **goto**) into *guarded statements* is not allowed.
- Jumping (through a **goto**) into *exception statements* is not allowed.
- Returning or jumping (through a **goto**) from *guarded statements* is not allowed.
- Returning or jumping (through a **goto**) from *exception statements* is not allowed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpc.h.
Library: Use Rpcrt4.lib.

See Also

RpcExceptionCode, **RpcFinally**, **RpcRaiseException**

RpcFinally

The **RpcFinally** statement provides developers with the ability to create termination handlers.

```
RpcTryFinally
{
    guarded statements
}
RpcFinally
{
    termination statements
}
RpcEndFinally;
```

Parameters

guarded statements
Specifies statements that are executed while exceptions are being monitored. If an exception occurs during the execution of these statements, *termination statements* will be executed. Unwinding then continues to the next **RpcTryExcept** or **RpcTryFinally** statements.

termination statements
Specifies statements that are executed when an exception occurs. After the termination statements are complete, the exception is raised again.

Remarks

Use the **RpcFinally** statement to build termination handlers. All termination handlers created with **RpcFinally** execute whether or not an exception occurs. Your program can call the **RpcAbnormalTermination** function in *termination statements* to determine whether *termination statements* is being executed because an exception occurred. A nonzero return from **RpcAbnormalTermination** indicates that an exception occurred. A value of zero indicates that no exception occurred.

The following restrictions apply:

- Jumping (through a **goto**) into *guarded statements* is not allowed.
- Jumping (through a **goto**) into *termination statements* is not allowed.
- Returning or jumping (through a **goto**) from *guarded statements* is not allowed.
- Returning or jumping (through a **goto**) from *termination statements* is not allowed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpc.h.
Library: Use Rpcrt4.lib.

See Also

RpcAbnormalTermination

RpcTryExcept

The **RpcTryExcept** statement provides structured exception handling for RPC applications. If any of the program statements in the **RpcTryExcept** cause an exception, the statements in the **RpcExcept** code block are executed. All **RpcTryExcept** statements must be terminated by the **RpcEndExcept** statement.

For more information, see ***RpcExcept***.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpc.h.

See Also

RpcExcept, **RpcTryFinally**

RpcTryFinally

The **RpcTryFinally** statement provides structured termination handling. It an exception occurs during the execution statements of the code block associated with the **RpcTryFinally** statement, the statements in the code block associated with the **RpcFinally** statement are exectured. All **RpcTryFinally** statements must be terminated by an **RpcEndFinally** statement.

For more information, see ***RpcFinally***.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Rpc.h.

See Also

RpcFinally, **RpcEndFinally**

CHAPTER 28

Windows Networking (WNet)

The Win32® API provides the Windows networking (WNet) functions so that you can implement networking capabilities in your application without making allowances for a particular network provider or physical network implementation. This is because the WNet functions are network independent.

About Windows Networking

Applications can use the WNet functions to add and cancel network connections and to retrieve information about the current configuration of the network.

Figure 28-1 shows the structure of a typical network.

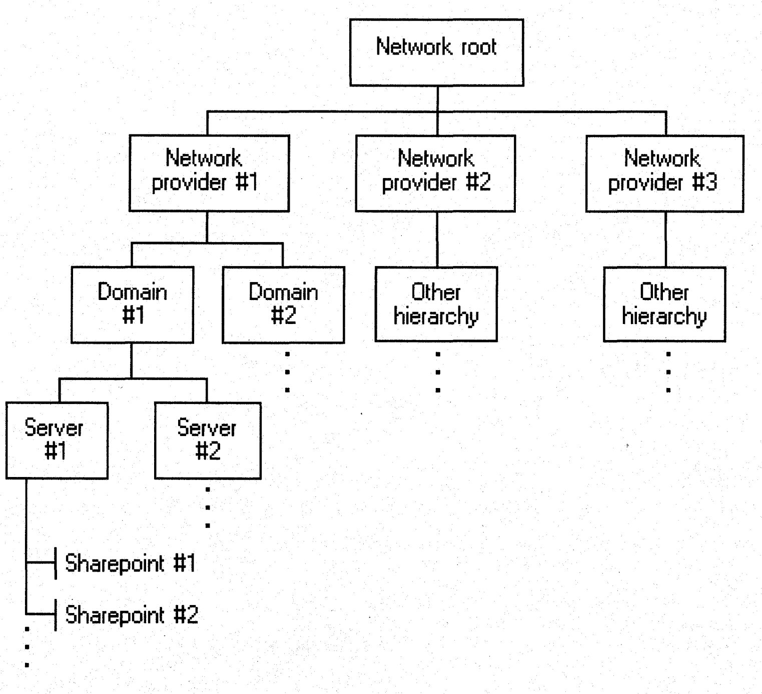


Figure 28-1: Typical Network Structure.

In the preceding figure, the hierarchy for Microsoft® Windows NT® Server/Windows® 2000 Server resources is given in detail as Network provider #1. Network resources from other providers have different hierarchical systems. An application does not need information about the hierarchy before it begins to work with a network. It can proceed from the network root (that is, the topmost container resource) and retrieve information about the network's resources as the information is required.

Network resources that contain other resources are called *containers*. Container resources are in boxes in the preceding figure.

Resources that do not contain other resources are called *objects*. In the preceding figure, Sharepoint #1 and Sharepoint #2 are objects. A *sharepoint* is an object that is accessible across the network. Examples of sharepoints include printers and shared directories.

WNet Functions

The Windows Networking functions can be grouped as follows:

- Connection functions
- Enumeration functions
- Information functions
- User functions

Connection Functions

Call the following WNet connection functions to connect a local device to a network resource, and to cancel network connections.

Function	Description
MultinetGetConnectionPerformance	Returns information about the expected performance of a connection to a network resource.
WNetAddConnection	Connects a local device to a network resource. (Provided for compatibility with 16-bit versions of Windows.)
WNetAddConnection2	Connects a local device to a network resource.
WNetAddConnection3	Connects a local device to a network resource. This function includes one more parameter than the **WNetAddConnection2** function, a handle to a window that the network provider can use as an owner window for dialog boxes.

Function	Description
WNetCancelConnection	Cancels a network connection. (Provided for compatibility with 16-bit versions of Windows.)
WNetCancelConnection2	Cancels a network connection, providing the ability to update the user profile with information about persistent connections.
WNetConnectionDialog	Starts a general browsing dialog box for connecting to network resources.
WNetConnectionDialog1	Starts a general browsing dialog box for connecting to network resources, using a **CONNECTDLGSTRUCT** structure.
WNetDisconnectDialog	Starts a general browsing dialog box for disconnecting from network resources.
WNetDisconnectDialog1	Starts a general browsing dialog box for disconnecting from network resources, using a **DISCDLGSTRUCT** structure.
WNetGetConnection	Retrieves the name of the network resource associated with a local device.
WNetGetUniversalName	When given a drive-based path for a network resource, returns a more universal form of the name.
WNetUseConnection	Connects a local device to a network resource; automatically selects an unused local device to redirect to the network resource.

Note The **WNetAddConnection** and **WNetCancelConnection** functions are supported for compatibility with Windows for Workgroups. However, new applications should use **WNetAddConnection2** or **WNetAddConnection3**, and **WNetCancelConnection2**.

Enumeration Functions

Call the following WNet functions to enumerate network resources.

Function	Description
WNetCloseEnum	Ends a network resource enumeration.
WNetEnumResource	Continues an enumeration of network resources started by the **WNetOpenEnum** function.
WNetOpenEnum	Starts an enumeration of network resources.

Information Functions

Call the following WNet informational and utility functions to retrieve network provider and other information.

Function	Description
WNetGetLastError	Returns the most recent error code set by a WNet function, the one reported by a network provider.
WNetGetNetworkInformation	Returns extended information about a specific network provider.
WNetGetProviderName	Returns the provider name for a specific type of network.
WNetGetResourceInformation	Returns the network provider that owns a resource, and obtains information about the resource type.
WNetGetResourceParent	Returns the parent of a network resource.

User Functions

Call the following WNet function to retrieve the name of the user associated with a local device.

Function	Description
WNetGetUser	Returns the current default user name, or the user name that established the connection.

Many of the WNet functions use a **NETRESOURCE** structure to store information about a network resource.

Windows Networking Operations

An application can use the WNet functions to browse, add, or cancel network connections anywhere in the network hierarchy.

A *persistent connection* is a network connection that the system automatically restores when the user logs on. You can call the **WNetAddConnection2** (or **WNetAddConnection3**) and **WNetCancelConnection2** functions to control whether a network connection is persistent from one session to the next.

To find the default user name or the user name used to establish a network connection, call the **WNetGetUser** function.

In addition to calling the WNet functions, a process can also use mailslots and named pipes to communicate with another process. For more information, see *Mailslots* and *Pipes*.

Using Windows Networking

The WNet functions enable your application to query and control network connections directly, or to give direct control of the network connections to the user.

Using the Connections Dialog Box

Windows NT/Windows 2000

The **WNetConnectionDialog** function creates a dialog box that allows the user to browse and connect to network resources. You can also call the **WNetConnectionDialog1** function to create a connections dialog box. **WNetConnectionDialog1** requires a **CONNECTDLGSTRUCT** structure.

The **WNetDisconnectDialog** function creates a dialog box that allows the user to disconnect from network resources.

The following code sample demonstrates how to call the **WNetConnectionDialog** function to create a dialog box that displays disk resources. If the call fails, the sample calls an application-defined error handler.

```
DWORD dwResult;
//
// Call the WNetConnectionDialog function.
//
dwResult = WNetConnectionDialog(hwnd, RESOURCETYPE_DISK);
//
// Call an application-defined error handler
//  to process errors.
//
if(dwResult != NO_ERROR)
{
    NetErrorHandler(hwnd, dwResult, (LPSTR)"WNetConnectionDialog");
    return FALSE;
}
```

For more information about using an application-defined error handler, see *Retrieving Network Errors*.

Enumerating Network Resources

Windows NT/Windows 2000

To begin the enumeration of a network container resource, your application should perform the following steps:

1. Pass the address of a **NETRESOURCE** structure that represents the resource to the **WNetOpenEnum** function.

2. Allocate a buffer large enough to hold the array of **NETRESOURCE** structures that the **WNetEnumResource** function returns, plus the strings to which their members point.
3. Pass the resource handle returned by **WNetOpenEnum** to the **WNetEnumResource** function.
4. Close the resource handle when it is no longer needed by calling the **WNetCloseEnum** function.

You can continue enumerating a container resource described in the array of **NETRESOURCE** structures retrieved by **WNetEnumResource**. If the **dwUsage** member of the **NETRESOURCE** structure is equal to RESOURCEUSAGE_CONTAINER, pass the address of the structure to the **WNetOpenEnum** function to open the container and continue the enumeration. If **dwUsage** is equal to RESOURCEUSAGE_CONNECTABLE, the application can pass the structure's address to the **WNetAddConnection2** function or the **WNetAddConnection3** function to connect to the resource.

The following example illustrates an application-defined function (EnumerateFunc) that enumerates all the resources on a network. The sample specifies NULL for the pointer to the **NETRESOURCE** structure because when **WNetOpenEnum** receives a NULL pointer, it retrieves a handle to the root of the network.

First the sample calls the **WNetOpenEnum** function to begin the enumeration. The sample calls the **GlobalAlloc** function to allocate the required buffer, and the **ZeroMemory** function to initialize the buffer with zeroes. Then the sample calls the **WNetEnumResource** function to continue the enumeration. Whenever the **dwUsage** member of a **NETRESOURCE** structure retrieved by **WNetEnumResource** is equal to RESOURCEUSAGE_CONTAINER, the EnumerateFunc function calls itself recursively and uses a pointer to that structure in its call to **WNetOpenEnum**. Finally, the sample calls the **GlobalFree** function to free the allocated memory, and the **WNetCloseEnum** to end the enumeration. EnumerateFunc calls an application-defined error handler to process errors, and the **TextOut** function for printing.

```
BOOL WINAPI EnumerateFunc(HWND hwnd,
                          HDC hdc,
                          LPNETRESOURCE lpnr)
{
  DWORD dwResult, dwResultEnum;
  HANDLE hEnum;
  DWORD cbBuffer = 16384;      // 16K is a good size
  DWORD cEntries = -1;         // enumerate all possible entries
  LPNETRESOURCE lpnrLocal;     // pointer to enumerated structures
  DWORD i;
  //
  // Call the WNetOpenEnum function to begin the enumeration.
```

```
//
dwResult = WNetOpenEnum(RESOURCE_GLOBALNET, // all network resources
                        RESOURCETYPE_ANY,   // all resources
                        0,        // enumerate all resources
                        lpnr,     // NULL first time the function
                                  // is called
                        &hEnum);  // handle to the resource

if (dwResult != NO_ERROR)
{
  //
  // Process errors with an application-defined error handler.
  //
  NetErrorHandler(hwnd, dwResult, (LPSTR)"WNetOpenEnum");
  return FALSE;
}
//
// Call the GlobalAlloc function to allocate resources.
//
lpnrLocal = (LPNETRESOURCE) GlobalAlloc(GPTR, cbBuffer);

do
{
  //
  // Initialize the buffer.
  //
  ZeroMemory(lpnrLocal, cbBuffer);
  //
  // Call the WNetEnumResource function to continue
  //  the enumeration.
  //
  dwResultEnum = WNetEnumResource(hEnum,      // resource handle
                                  &cEntries,  // defined locally as -1
                                  lpnrLocal,  // LPNETRESOURCE
                                  &cbBuffer); // buffer size
  //
  // If the call succeeds, loop through the structures.
  //
  if (dwResultEnum == NO_ERROR)
  {
    for(i = 0; i < cEntries; i++)
    {
      // Call an application-defined function to
      //  display the contents of the NETRESOURCE structures.
```

(continued)

(continued)

```
      //
      DisplayStruct(hdc, &lpnrLocal[i]);

      // If the NETRESOURCE structure represents a container resource,
      //  call the EnumerateFunc function recursively.

      if(RESOURCEUSAGE_CONTAINER == (lpnrLocal[i].dwUsage
                                     & RESOURCEUSAGE_CONTAINER))
        if(!EnumerateFunc(hwnd, hdc, &lpnrLocal[i]))
          TextOut(hdc, 10, 10, "EnumerateFunc returned FALSE.", 29);
    }
  }
  // Process errors.
  //
  else if (dwResultEnum != ERROR_NO_MORE_ITEMS)
  {
    NetErrorHandler(hwnd, dwResultEnum, (LPSTR)"WNetEnumResource");
    break;
  }
}
//
// End do.
//
while(dwResultEnum != ERROR_NO_MORE_ITEMS);
//
// Call the GlobalFree function to free the memory.
//
GlobalFree((HGLOBAL)lpnrLocal);
//
// Call WNetCloseEnum to end the enumeration.
//
dwResult = WNetCloseEnum(hEnum);

if(dwResult != NO_ERROR)
{
  //
  // Process errors.
  //
  NetErrorHandler(hwnd, dwResult, (LPSTR)"WNetCloseEnum");
  return FALSE;
}

return TRUE;
}
```

For more information about using an application-defined error handler, see *Retrieving Network Errors*.

Adding a Network Connection

Windows NT/Windows 2000

To make a connection to a network resource described by a **NETRESOURCE** structure, an application can call the **WNetAddConnection2**, the **WNetAddConnection3**, or the **WNetUseConnection** function. The following example demonstrates use of the **WNetAddConnection2** function.

The code sample calls the **WNetAddConnection2** function, specifying that the system should update the user's profile with the information, creating a "remembered" or persistent connection. The sample calls an application-defined error handler to process errors, and the **TextOut** function for printing.

```
DWORD dwResult;
NETRESOURCE nr;
//
// Call the WNetAddConnection2 function to make the connection,
//   specifying a persistent connection.
//
dwResult = WNetAddConnection2(&nr, // NETRESOURCE from enumeration
    (LPSTR) NULL,                  // no password
    (LPSTR) NULL,                  // logged-in user
    CONNECT_UPDATE_PROFILE);       // update profile with connect information

// Process errors.
//  The local device is already connected to a network resource.
//
if (dwResult == ERROR_ALREADY_ASSIGNED)
{
    TextOut(hdc, 10, 10, "Already connected to specified resource.", 40);
    return FALSE;
}

//  An entry for the local device already exists in the user profile.
//
else if (dwResult == ERROR_DEVICE_ALREADY_REMEMBERED)
{
    TextOut(hdc, 10, 10,
        "Attempted reassignment of remembered device.", 44);
    return FALSE;
}
else if(dwResult != NO_ERROR)
```

(continued)

(continued)

```
{
    //
    // Call an application-defined error handler.
    //
    NetErrorHandler(hwnd, dwResult, (LPSTR)"WNetAddConnection2");
    return FALSE;
}

//
// Otherwise, report a successful connection.
//
TextOut(hdc, 10, 10, "Connected to specified resource.", 32);
```

The **WNetAddConnection** function is supported for compatibility with earlier versions of Windows for Workgroups. New applications should call the **WNetAddConnection2** function or the **WNetAddConnection3** function.

For more information about using an application-defined error handler, see *Retrieving Network Errors*.

Assigning a Drive to a Share

Windows NT/Windows 2000

The following example demonstrates how to connect a drive letter to a remote server share with a call to the **WNetAddConnection2** function. The sample informs the user whether or not the call was successful.

To test the following code sample, perform the following steps:

1. Change the following lines to valid strings:

```
szUserName[32] = "myUserName",
szPassword[32] = "myPassword",
szLocalName[32] = "Q:",
szRemoteName[MAX_PATH] = "\\\\products2\\relsys";
```

2. Add the file to a console application called AddConn2.
3. Link the library MPR.LIB to the compiler list of libraries.
4. Compile and run the program AddConn2.EXE:

```
#include <windows.h>
#include <stdio.h>
#include <winnetwk.h>

void main()
{
```

```
NETRESOURCE nr;
DWORD res;
TCHAR szUserName[32] = "MyUserName",
    szPassword[32] = "MyPassword",
    szLocalName[32] = "Q:",
    szRemoteName[MAX_PATH] = "\\\\products2\\relsys";
//
// Assign values to the NETRESOURCE structure.
//
nr.dwType = RESOURCETYPE_ANY;
nr.lpLocalName = (LPTSTR) &szLocalName;
nr.lpRemoteName = (LPTSTR) &szRemoteName;
nr.lpProvider = NULL;
//
// Call the WNetAddConnection2 function to assign
//   a drive letter to the share.
//
res = WNetAddConnection2(&nr, (LPTSTR) &szPassword, (LPTSTR) &szUserName,
FALSE);
//
// If the call succeeds, inform the user; otherwise,
//  print the error.
//
if(res == NO_ERROR)
  printf("Connection added \n", szRemoteName);
else
  printf("Error: %ld\n", res);
  return;
}
```

Determining the Location of a Share

Windows NT/Windows 2000

The following example demonstrates how to call the **WNetGetUniversalName** function to determine the location of a share on a redirected drive.

First the code sample calls the **WNetGetUniversalName** function, specifying the **UNIVERSAL_NAME_INFO** information level to retrieve a pointer to a Universal Naming Convention (UNC) name string for the resource. Then the sample calls **WNetGetUniversalName** a second time, specifying the **REMOTE_NAME_INFO** information level to retrieve two additional network connection information strings. If the calls are successful, the sample prints the location of the share.

To test the following code sample, perform the following steps:

1. Name the code sample GetUni.cpp.
2. Add the sample to a console application called GetUni.

3. Link the libraries Shell32.lib, Mpr.lib, and NetApi32.lib to the compiler list of libraries.
4. From the command prompt, change to the GetUni directory.
5. Compile GetUni.cpp.
6. Run the file GetUni.exe followed by a drive letter and colon, like this:

```
GetUni H:\

#define  STRICT
#include <windows.h>
#include <stdio.h>

#define BUFFSIZE = 1000

void main( int argc, char *argv[] )
{
  DWORD cbBuff = 1000    // Size of Buffer
  TCHAR szBuff[1000];    // Buffer to receive information
  REMOTE_NAME_INFO  * prni = (REMOTE_NAME_INFO *)   &szBuff;
  // Pointers to head of buffer
  UNIVERSAL_NAME_INFO * puni = (UNIVERSAL_NAME_INFO *) &szBuff;
  DWORD res;

  if((argc < 2) | (lstrcmp(argv[1], "/?") == 0))
  {
    printf("Syntax:  GetUni DrivePathAndFilename\n"
         "Example: GetUni U:\\WINNT\\SYSTEM32\\WSOCK32.DLL\n");
    return;
  }

  // Call the WNetGetUniversalName function specifying the
  //  UNIVERSAL_NAME_INFO_LEVEL option.
  //
  printf("Call WNetGetUniversalName using UNIVERSAL_NAME_INFO_LEVEL.\n");
  if((res = WNetGetUniversalName((LPTSTR)argv[1],
   UNIVERSAL_NAME_INFO_LEVEL,
   //
   //The structure is written to this block of memory.
   //
   (LPVOID) &szBuff,
   &cbBuff)) != NO_ERROR)
    //
    // If the call fails, print the error; otherwise, print
    //  the location of the share,
    //  using the pointer to UNIVERSAL_NAME_INFO_LEVEL.
    //
    printf("Error: %ld\n\n", res);

else
printf("Universal Name: \t%s\n\n",
puni->lpUniversalName);
  //
  // Call the WNetGetUniversalName function,
  //  using the REMOTE_NAME_INFO_LEVEL option.
  //
  printf("Call WNetGetUniversalName using REMOTE_NAME_INFO_LEVEL.\n");
  if((res = WNetGetUniversalName((LPTSTR)argv[1],
     REMOTE_NAME_INFO_LEVEL,
        (LPVOID) &szBuff,    // Structure is written to this
                             // block of memory
      &cbBuff)) != NO_ERROR)
    //
    // If the call fails, print the error; otherwise, print
    //  the location of the share, using
    //  the pointer to REMOTE_NAME_INFO_LEVEL.
    //
    printf("Error: %ld\n", res);
  else
    printf("Universal Name: \t%s\n"
         "Connection Name:\t%s\n"
       "Remaining Path: \t%s\n",
          prni->lpUniversalName,
          prni->lpConnectionName,
          prni->lpRemainingPath);
  return;
}
```

Retrieving the Connection Name

Windows NT/Windows 2000

To retrieve the name of the network resource associated with a local device, an application can call the **WNetGetConnection** function, as shown in the following example. The sample calls an application-defined error handler to process errors, and the **TextOut** function for printing.

```
CHAR szDeviceName[80];
DWORD dwResult, cchBuff = sizeof(szDeviceName);

// Call the WNetGetConnection function.
```

(continued)

(continued)

```
//
dwResult = WNetGetConnection("z:",
    (LPSTR) szDeviceName,
    &cchBuff);

switch (dwResult)
{
    //
    // Print the connection name or process errors.
    //
    case NO_ERROR:
        TextOut(hdc, 10, 10, (LPSTR) szDeviceName,
            lstrlen((LPSTR) szDeviceName));
        break;
    //
    // The device is not a redirected device.
    //
    case ERROR_NOT_CONNECTED:
        TextOut(hdc, 10, 10, "Device z: not connected.", 24);
    //
    // The device is not currently connected,
    //  but it is a persistent connection.
    //
    case ERROR_CONNECTION_UNAVAIL:
        TextOut(hdc, 10, 10, "Connection unavailable.", 23);
    //
    // Call an application-defined error handler.
    //
    default:
        NetErrorHandler(hwnd, dwResult, (LPSTR)"WNetGetConnection");
        return FALSE;
}
```

For more information about using an application-defined error handler, see *Retrieving Network Errors*.

Retrieving the User Name

Windows NT/Windows 2000

To retrieve the name of the user associated either with a local device connected to a network resource or with the name of a network, an application can call the **WNetGetUser** function. The following example uses the device name to retrieve the name of the user. The sample calls an application-defined error handler to process errors, and the **TextOut** function for printing.

```
CHAR szUserName[80];
DWORD dwResult, cchBuff = 80;

// Call the WNetGetUser function.
//
dwResult = WNetGetUser("z:",
    (LPSTR) szUserName,
    &cchBuff);

// If the call succeeds, print the user name.
//
if(dwResult == NO_ERROR)
    TextOut(hdc, 10, 10,
        (LPSTR) szUserName,
        lstrlen((LPSTR) szUserName));

// Call an application-defined error handler.
//
else {
    NetErrorHandler(hwnd, dwResult, (LPSTR)"WNetGetUser");
    return FALSE;
}
```

For more information about using an application-defined error handler, see *Retrieving Network Errors.*

Canceling a Network Connection

Windows NT/Windows 2000

To cancel a connection to a network resource, an application can call the **WNetCancelConnection2** function, as shown in the following example. The call to **WNetCancelConnection2** specifies that a network connection should no longer be persistent. The sample calls an application-defined error handler to process errors, and the **TextOut** function for printing.

```
DWORD dwResult;

// Call the WNetCancelConnection2 function, specifying
//  that the connection should no longer be a persistent one.
//
dwResult = WNetCancelConnection2("z:",
    CONNECT_UPDATE_PROFILE, // remove connection from profile
    FALSE);                 // fail if open files or jobs
```

(continued)

(continued)

```
// Process errors.
//  The device is not a local redirected device.
//
if (dwResult == ERROR_NOT_CONNECTED)
{
    TextOut(hdc, 10, 10, "Drive z: not connected.", 23);
    return FALSE;
}

// Call an application-defined error handler.
//
else if(dwResult != NO_ERROR)
{
    NetErrorHandler(hwnd, dwResult, (LPSTR)"WNetCancelConnection2");
    return FALSE;
}
//
// Otherwise, report canceling the connection.
//
TextOut(hdc, 10, 10, "Connection closed for z:.", 25);
```

The **WNetCancelConnection** function is supported for compatibility with earlier versions of Windows for Workgroups. For new applications, use **WNetCancelConnection2**.

For more information about using an application-defined error handler, see *Retrieving Network Errors*.

Retrieving Network Errors

Windows NT/Windows 2000

The WNet functions return error codes for compatibility with Windows for Workgroups. For compatibility with the Win32 API, the function also sets the error code value returned by **GetLastError**.

When one of the WNet functions returns ERROR_EXTENDED_ERROR, an application can call the **WNetGetLastError** function to retrieve additional information about the error. This information is usually specific to the network provider.

The following example illustrates an application-defined error-handling function (NetErrorHandler). The function takes three arguments: a window handle, the error code returned by one of the WNet functions, and the name of the function that produced the error. If the error code is ERROR_EXTENDED_ERROR, NetErrorHandler calls **WNetGetLastError** to get extended error information and prints the information. The sample calls the **MessageBox** function to process messages.

```
BOOL WINAPI NetErrorHandler(HWND hwnd,
                            DWORD dwErrorCode,
                            LPSTR lpszFunction)
{
    DWORD dwWNetResult, dwLastError;
    CHAR szError[256];
    CHAR szCaption[256];
    CHAR szDescription[256];
    CHAR szProvider[256];

    // The following code performs standard error-handling.

    if (dwErrorCode != ERROR_EXTENDED_ERROR)
    {
        wsprintf((LPSTR) szError, "%s failed; \nResult is %ld",
            lpszFunction, dwErrorCode);
        wsprintf((LPSTR) szCaption, "%s error", lpszFunction);
        MessageBox(hwnd, (LPSTR) szError, (LPSTR) szCaption, MB_OK);
        return TRUE;
    }

    // The following code performs error-handling when the
    //  ERROR_EXTENDED_ERROR return value indicates that the
    //  WNetGetLastError function can retrieve additional information.

    else
    {
        dwWNetResult = WNetGetLastError(&dwLastError, // error code
            (LPSTR) szDescription,  // buffer for error description
            sizeof(szDescription),  // size of error buffer
            (LPSTR) szProvider,     // buffer for provider name
            sizeof(szProvider));    // size of name buffer

        //
        // Process errors.
        //
        if(dwWNetResult != NO_ERROR) {
            wsprintf((LPSTR) szError,
                "WNetGetLastError failed; error %ld", dwWNetResult);
            MessageBox(hwnd, (LPSTR) szError,
                "WNetGetLastError", MB_OK);
            return FALSE;
        }
```

(continued)

(continued)

```
        //
        // Otherwise, print the additional error information.
        //
        wsprintf((LPSTR) szError,
            "%s failed with code %ld;\n%s",
            (LPSTR) szProvider, dwLastError, (LPSTR) szDescription);
        wsprintf((LPSTR) szCaption, "%s error", lpszFunction);
        MessageBox(hwnd, (LPSTR) szError, (LPSTR) szCaption, MB_OK);
        return TRUE;
    }
}
```

Windows Networking Reference

The following elements are used in Windows networking:

- Windows Networking Functions
- Windows Networking Structures

Windows Networking Functions

The following functions are used in Windows networking:

MultinetGetConnectionPerformance
WNetAddConnection2
WNetAddConnection3
WNetCancelConnection
WNetCancelConnection2
WNetCloseEnum
WNetConnectionDialog
WNetConnectionDialog1
WNetDisconnectDialog
WNetDisconnectDialog1
WNetEnumResource
WNetGetConnection
WNetGetLastError
WNetGetNetworkInformation
WNetGetProviderName
WNetGetResourceInformation
WNetGetResourceParent
WNetGetUniversalName
WNetGetUser
WNetOpenEnum
WNetUseConnection

Obsolete Functions

The following function is provided only for compatibility with 16-bit versions of Windows:

WNetAddConnection

MultinetGetConnectionPerformance

The **MultinetGetConnectionPerformance** function returns information about the expected performance of a connection used to access a network resource. The function returns the information in a **NETCONNECTINFOSTRUCT** structure.

Note that the **MultinetGetConnectionPerformance** function can be used only to request information for a local device that is redirected to a network resource, or for a network resource to which there is currently a connection:

```
DWORD MultinetGetConnectionPerformance(
  LPNETRESOURCE lpNetResource,
          // network resource
  LPNETCONNECTINFOSTRUCT lpNetConnectInfoStruct
          // data buffer
);
```

Parameters

lpNetResource

[in] Pointer to a **NETRESOURCE** structure that specifies the network resource. The following members have specific meanings in this context.

Member	Meaning
lpLocalName	Pointer to a buffer that specifies a local device, such as "F:" or "LPT1", that is redirected to a network resource to be queried. If this member is NULL or an empty string, the network resource is specified in the **lpRemoteName** member. If this flag specifies a local device, **lpRemoteName** is ignored.
lpRemoteName	Pointer to a network resource to query. The resource must currently have an established connection. For example, if the resource is a file on a file server, then having the file open will ensure the connection.
lpProvider	Usually set to NULL, but can be a pointer to the owner (provider) of the resource if the network on which the resource resides is known. If the **lpProvider** member is not NULL, the system attempts to return information only about the named network.

lpNetConnectInfoStruct

[out] Pointer to the **NETCONNECTINFOSTRUCT** structure that receives the data.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one or more of the following error values.

Value	Meaning
ERROR_NOT_SUPPORTED	The network resource does not supply this information.
ERROR_NOT_CONNECTED	The **lpLocalName** member does not specify a redirected device, or the **lpRemoteName** member does not specify the name of a resource that is currently connected.
ERROR_NO_NET_OR_BAD_PATH	The operation could not be completed, either because a network component is not started, or because the specified resource name is not recognized.
ERROR_BAD_DEVICE	The local device specified by the **lpLocalName** member is invalid.
ERROR_BAD_NET_NAME	The resource specified by the **lpRemoteName** member was not recognized by any network.
ERROR_INVALID_PARAMETER	Either the *lpNetConnectInfoStruct* parameter does not point to a **NETCONNECTINFOSTRUCT** structure in which the **cbStructure** member is filled with the structure size, or both the **lpLocalName** and **lpRemoteName** members are not specified.
ERROR_NO_NETWORK	The network is unavailable.
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call **WNetGetLastError**.

Remarks

The information returned by the **MultinetGetConnectionPerformance** function is an estimate only. Network traffic and routing can affect the accuracy of the results returned.

A typical way to use this function would be to open a file on a network server (which would ensure that there is a connection to the file), call this function, and use the results to make decisions about how to manage file I/O. For example, you can decide whether to read the entire file into a temporary file on the client or directly access the file on the server.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, Windows Networking Functions, **NETCONNECTINFOSTRUCT**, **NETRESOURCE**

WNetAddConnection

The **WNetAddConnection** function enables the calling application to connect a local device to a network resource. A successful connection is persistent, meaning that the system automatically restores the connection during subsequent logon operations.

Note This function is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should call the **WNetAddConnection2** or the **WNetAddConnection3** function.

```
DWORD WNetAddConnection(
  LPCTSTR lpRemoteName, // network device name
  LPCTSTR lpPassword,   // password
  LPCTSTR lpLocalName   // local device name
);
```

Parameters

lpRemoteName
[in] Pointer to a constant null-terminated string that specifies the network resource to connect to.

lpPassword
[in] Pointer to a constant null-terminated string that specifies the password to be used to make a connection. This parameter is usually the password associated with the current user.

If this parameter is NULL, the default password is used. If the string is empty, no password is used.

Windows 95/98: This parameter must be NULL or an empty string.

lpLocalName
[in] Pointer to a constant null-terminated string that specifies the name of a local device to be redirected, such as "F:" or "LPT1". The string is treated in a case-insensitive manner. If the string is NULL, a connection to the network resource is made without redirecting the local device.

Return Values

If the function succeeds, the return value is NO_ERROR.

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

Value	Meaning
ERROR_ACCESS_DENIED	Access is denied.
ERROR_ALREADY_ASSIGNED	The device specified in the *lpLocalName* parameter is already connected.
ERROR_BAD_DEV_TYPE	The device type and the resource type do not match.
ERROR_BAD_DEVICE	The value specified in the *lpLocalName* parameter is invalid.
ERROR_BAD_NET_NAME	The value specified in the *lpRemoteName* parameter is not valid or cannot be located.
ERROR_BAD_PROFILE	The user profile is in an incorrect format.
ERROR_CANNOT_OPEN_PROFILE	The system is unable to open the user profile to process persistent connections.
ERROR_DEVICE_ALREADY_REMEMBERED	An entry for the device specified in the *lpLocalName* parameter is already in the user profile.
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call the **WNetGetLastError** function.
ERROR_INVALID_PASSWORD	The specified password is invalid.
ERROR_NO_NET_OR_BAD_PATH	The operation cannot be performed because a network component is not started or because a specified name cannot be used.
ERROR_NO_NETWORK	The network is unavailable.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetAddConnection2**, **WNetAddConnection3**, **WNetCancelConnection**, **WNetCancelConnection2**, **WNetGetConnection**

WNetAddConnection2

The **WNetAddConnection2** function makes a connection to a network resource. The function can redirect a local device to the network resource.

The **WNetAddConnection2** function supersedes the **WNetAddConnection** function. If you can pass a handle to a window that the provider of network resources can use as an owner window for dialog boxes, call the **WNetAddConnection3** function instead.

```
DWORD WNetAddConnection2(
  LPNETRESOURCE lpNetResource,  // connection details
  LPCTSTR lpPassword,           // password
  LPCTSTR lpUsername,           // user name
  DWORD dwFlags                 // connection options
);
```

Parameters

lpNetResource

[in] Pointer to a **NETRESOURCE** structure that specifies details of the proposed connection, such as information about the network resource, the local device, and the network resource provider.

You must specify the following members of the **NETRESOURCE** structure.

Member	Description
dwType	Specifies the type of network resource to connect to. If the **lpLocalName** member points to a nonempty string, this member can be equal to RESOURCETYPE_DISK or RESOURCETYPE_PRINT. If **lpLocalName** is NULL, or if it points to an empty string, **dwType** can be equal to RESOURCETYPE_DISK, RESOURCETYPE_PRINT, or RESOURCETYPE_ANY.
lpLocalName	Points to a null-terminated string that specifies the name of a local device to redirect, such as "F:" or "LPT1". The string is treated in a case-insensitive manner. If the string is empty, or if **lpLocalName** is NULL, the function makes a connection to the network resource without redirecting a local device.
lpRemoteName	Points to a null-terminated string that specifies the network resource to connect to. The string can be up to MAX_PATH characters in length, and must follow the network provider's naming conventions.

(continued)

(continued)

Member	Description
lpProvider	Points to a null-terminated string that specifies the network provider to connect to. If **lpProvider** is NULL, or if it points to an empty string, the operating system attempts to determine the correct provider by parsing the string pointed to by the **lpRemoteName** member. If this member is not NULL, the operating system attempts to make a connection only to the named network provider. You should set this member only if you know the network provider you want to use. Otherwise, let the operating system determine which provider the network name maps to.

The **WNetAddConnection2** function ignores the other members of the **NETRESOURCE** structure.

lpPassword

[in] Pointer to a constant null-terminated string that specifies a password to be used in making the network connection.

If *lpPassword* is NULL, the function uses the current default password associated with the user specified by the *lpUserName* parameter.

If *lpPassword* points to an empty string, the function does not use a password.

Windows 95/98: This parameter must be NULL or an empty string.

lpUsername

[in] Pointer to a constant null-terminated string that specifies a user name for making the connection.

If *lpUserName* is NULL, the function uses the default user name. (The user context for the process provides the default user name.)

The *lpUserName* parameter is specified when users want to connect to a network resource for which they have been assigned a user name or account other than the default user name or account.

The user-name string represents a security context. It may be specific to a network provider.

Windows 95/98: This parameter must be NULL or an empty string.

dwFlags

[in] Specifies a **DWORD** value that describes connection options. The following value is currently defined.

Value	Meaning
CONNECT_UPDATE_PROFILE	The network resource connection should be remembered. If this bit flag is set, the operating system automatically attempts to restore the connection when the user logs on. The operating system remembers only successful connections that redirect local devices. It does not remember connections that are unsuccessful or deviceless connections. (A deviceless connection occurs when the **lpLocalName** member is NULL or points to an empty string.) If this bit flag is clear, the operating system does not automatically restore the connection at logon.

Return Values

If the function succeeds, the return value is NO_ERROR.

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

Value	Meaning
ERROR_ACCESS_DENIED	Access to the network resource was denied.
ERROR_ALREADY_ASSIGNED	The local device specified by the **lpLocalName** member is already connected to a network resource.
ERROR_BAD_DEV_TYPE	The type of local device and the type of network resource do not match.
ERROR_BAD_DEVICE	The value specified by **lpLocalName** is invalid.
ERROR_BAD_NET_NAME	The value specified by the **lpRemoteName** member is not acceptable to any network resource provider, either because the resource name is invalid, or because the named resource cannot be located.
ERROR_BAD_PROFILE	The user profile is in an incorrect format.
ERROR_BAD_PROVIDER	The value specified by the **lpProvider** member does not match any provider.
ERROR_BUSY	The router or provider is busy, possibly initializing. The caller should retry.

(continued)

(continued)

Value	Meaning
ERROR_CANCELLED	The attempt to make the connection was cancelled by the user through a dialog box from one of the network resource providers, or by a called resource.
ERROR_CANNOT_OPEN_PROFILE	The system is unable to open the user profile to process persistent connections.
ERROR_DEVICE_ALREADY_REMEMBERED	An entry for the device specified by **lpLocalName** is already in the user profile.
ERROR_EXTENDED_ERROR	A network-specific error occurred. Call the **WNetGetLastError** function to obtain a description of the error.
ERROR_INVALID_PASSWORD	The specified password is invalid.
ERROR_NO_NET_OR_BAD_PATH	The operation cannot be performed because a network component is not started or because a specified name cannot be used.
ERROR_NO_NETWORK	The network is unavailable.

Remarks

For a code sample that illustrates how to make a connection to a network resource using the **WNetAddConnection2** function, see *Adding a Network Connection*. For an example that demonstrates how to use the Error! Bookmark not defined. function to connect a drive letter to a remote share on a server, see *Assigning a Drive to a Share*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetAddConnection3**, **WNetCancelConnection2**, **WNetGetConnection**, **NETRESOURCE**

WNetAddConnection3

The **WNetAddConnection3** function makes a connection to a network resource. The function can redirect a local device to the network resource.

The **WNetAddConnection3** function is similar to the **WNetAddConnection2** function. The main difference is that **WNetAddConnection3** has an additional parameter, a handle to a window that the provider of network resources can use as an owner window for dialog boxes. The **WNetAddConnection2** function and the **WNetAddConnection3** function supersede the **WNetAddConnection** function.

```
DWORD WNetAddConnection3(
  HWND hwndOwner,                 // owner window
  LPNETRESOURCE lpNetResource,    // connection details
  LPTSTR lpPassword,              // password string
  LPTSTR lpUserName,              // user name string
  DWORD dwFlags                   // connection options
);
```

Parameters

hwndOwner

[in] Specifies the handle to a window that the provider of network resources can use as an owner window for dialog boxes.

The *hwndOwner* parameter can be NULL. If it is, a call to **WNetAddConnection3** is equivalent to calling the **WNetAddConnection2** function.

lpNetResource

[in] Pointer to a **NETRESOURCE** structure that specifies details of the proposed connection, such as information about the network resource, the local device, and the network resource provider.

You must specify the following members of the **NETRESOURCE** structure.

Member	Description
dwType	Specifies the type of network resource to connect to. If the **lpLocalName** member points to a nonempty string, this member can be equal to RESOURCETYPE_DISK or RESOURCETYPE_PRINT. If **lpLocalName** is NULL, or if it points to an empty string, **dwType** can be equal to RESOURCETYPE_DISK, RESOURCETYPE_PRINT, or RESOURCETYPE_ANY.
lpLocalName	Points to a null-terminated string that specifies the name of a local device to redirect, such as "F:" or "LPT1". The string is treated in a case-insensitive manner. If the string is empty or if **lpLocalName** is NULL, the function makes a connection to the network resource without redirecting a local device.

(continued)

(continued)

Member	Description
lpRemoteName	Points to a null-terminated string that specifies the network resource to connect to. The string can be up to MAX_PATH characters in length, and must follow the network provider's naming conventions.
lpProvider	Points to a null-terminated string that specifies the network provider to connect to. If **lpProvider** is NULL, or if it points to an empty string, the operating system attempts to determine the correct provider by parsing the string pointed to by the **lpRemoteName** member. If this member is not NULL, the operating system attempts to make a connection only to the named network provider. You should set this member only if you know which network provider you want to use. Otherwise, let the operating system determine which network provider the network name maps to.

The **WNetAddConnection3** function ignores the other members of the **NETRESOURCE** structure.

lpPassword

[in] Pointer to a null-terminated string that specifies a password to be used in making the network connection.

If *lpPassword* is NULL, the function uses the current default password associated with the user specified by the *lpUserName* parameter.

If *lpPassword* points to an empty string, the function does not use a password.

Windows 95/98: This parameter must be NULL or an empty string.

lpUserName

[in] Pointer to a null-terminated string that specifies a user name for making the connection.

If *lpUserName* is NULL, the function uses the default user name. (The user context for the process provides the default user name.)

The *lpUserName* parameter is specified when users want to connect to a network resource for which they have been assigned a user name or account other than the default user name or account.

The user-name string represents a security context. It may be specific to a network provider.

Windows 95/98: This parameter must be NULL or an empty string.

dwFlags

[in] Specifies a **DWORD** value that describes connection options. The following value is currently defined.

Value	Meaning
CONNECT_UPDATE_PROFILE	The network resource connection should be remembered. If this bit flag is set, the operating system automatically attempts to restore the connection when the user logs on. The operating system remembers only successful connections that redirect local devices. It does not remember connections that are unsuccessful or deviceless connections. (A deviceless connection occurs when the **lpLocalName** member is NULL or when it points to an empty string.) If this bit flag is clear, the operating system does not automatically restore the connection at logon.

Return Values

If the function succeeds, the return value is NO_ERROR.

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

Value	Meaning
ERROR_ACCESS_DENIED	Access to the network resource was denied.
ERROR_ALREADY_ASSIGNED	The local device specified by the **lpLocalName** member is already connected to a network resource.
ERROR_BAD_DEV_TYPE	The type of local device and the type of network resource do not match.
ERROR_BAD_DEVICE	The value specified by **lpLocalName** is invalid.
ERROR_BAD_NET_NAME	The value specified by the **lpRemoteName** member is not acceptable to any network resource provider, either because the resource name is invalid, or because the named resource cannot be located.
ERROR_BAD_PROFILE	The user profile is in an incorrect format.
ERROR_BAD_PROVIDER	The value specified by the **lpProvider** member does not match any provider.
ERROR_BUSY	The router or provider is busy, possibly initializing. The caller should retry.
ERROR_CANCELLED	The attempt to make the connection was cancelled by the user through a dialog box from one of the network resource providers, or by a called resource.

(continued)

(continued)

Value	Meaning
ERROR_CANNOT_OPEN_PROFILE	The system is unable to open the user profile to process persistent connections.
ERROR_DEVICE_ALREADY_REMEMBERED	An entry for the device specified by the **lpLocalName** member is already in the user profile.
ERROR_EXTENDED_ERROR	A network-specific error occurred. Call the **WNetGetLastError** function to obtain a description of the error.
ERROR_INVALID_PASSWORD	The specified password is invalid.
ERROR_NO_NET_OR_BAD_PATH	The operation cannot be performed because a network component is not started or because a specified name cannot be used.
ERROR_NO_NETWORK	The network is unavailable.

The **WNetUseConnection** function is similar to the **WNetAddConnection3** function. The main difference is that **WNetUseConnection** can automatically select an unused local device to redirect to the network resource.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **NETRESOURCE**, **WNetAddConnection2**, **WNetCancelConnection2**, **WNetUseConnection**, **WNetGetConnection**

WNetCancelConnection

The **WNetCancelConnection** function cancels an existing network connection.

The **WNetCancelConnection** function is provided for compatibility with 16-bit versions of Windows. Win32-based applications should call the **WNetCancelConnection2** function.

```
DWORD WNetCancelConnection(
  LPCTSTR lpName,  // resource name
  BOOL fForce      // unconditional disconnect option
);
```

Parameters

lpName

[in] Pointer to a constant null-terminated string that specifies the name of either the redirected local device or the remote network resource to disconnect from.

When this parameter specifies a redirected local device, the function cancels only the specified device redirection. If the parameter specifies a remote network resource, only the connections to remote networks without devices are canceled.

fForce

[in] Specifies whether or not the disconnection should occur if there are open files or jobs on the connection. If this parameter is FALSE, the function fails if there are open files or jobs.

Return Values

If the function succeeds, the return value is NO_ERROR.

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

Value	Meaning
ERROR_BAD_PROFILE	The user profile is in an incorrect format.
ERROR_CANNOT_OPEN_PROFILE	The system is unable to open the user profile to process persistent connections.
ERROR_DEVICE_IN_USE	The device is in use by an active process and cannot be disconnected.
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call the **WNetGetLastError** function.
ERROR_NOT_CONNECTED	The name specified by the *lpName* parameter is not a redirected device, or the system is not currently connected to the device specified by the parameter.
ERROR_OPEN_FILES	There are open files, and the *fForce* parameter is FALSE.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetAddConnection**, **WNetAddConnection2**, **WNetCancelConnection2**, **WNetGetConnection**

WNetCancelConnection2

The **WNetCancelConnection2** function cancels an existing network connection. You can also call the function to remove remembered network connections that are not currently connected.

The **WNetCancelConnection2** function supersedes the **WNetCancelConnection** function.

```
DWORD WNetCancelConnection2(
  LPCTSTR lpName,  // resource name
  DWORD dwFlags,   // connection type
  BOOL fForce      // unconditional disconnect option
);
```

Parameters

lpName

[in] Pointer to a constant null-terminated string that specifies the name of either the redirected local device or the remote network resource to disconnect from.

If this parameter specifies a redirected local resource, the function cancels only the specified redirection; otherwise, the function cancels all connections to the remote network resource.

dwFlags

[in] Specifies a **DWORD** value that describes the connection type. The following values are defined:

Value	Meaning
0	The system does not update information about the connection. If the connection was marked as persistent in the registry, the system continues to restore the connection at the next logon. If the connection was not marked as persistent, the function ignores the setting of the CONNECT_UPDATE_PROFILE flag.
CONNECT_UPDATE_ PROFILE	The system updates the user profile with the information that the connection is no longer a persistent one. The system will not restore this connection during subsequent logon operations. (Disconnecting resources using remote names has no effect on persistent connections.)

fForce

[in] Specifies a Boolean value that indicates whether the disconnection should occur if there are open files or jobs on the connection. If this parameter is FALSE, the function fails if there are open files or jobs.

Return Values

If the function succeeds, the return value is NO_ERROR.

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

Value	Meaning
ERROR_BAD_PROFILE	The user profile is in an incorrect format.
ERROR_CANNOT_OPEN_PROFILE	The system is unable to open the user profile to process persistent connections.
ERROR_DEVICE_IN_USE	The device is in use by an active process and cannot be disconnected.
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call the **WNetGetLastError** function.
ERROR_NOT_CONNECTED	The name specified by the *lpName* parameter is not a redirected device, or the system is not currently connected to the device specified by the parameter.
ERROR_OPEN_FILES	There are open files, and the *fForce* parameter is FALSE.

Remarks

For a code sample that illustrates how to cancel a connection to a network resource with a call to the **WNetCancelConnection2** function, see *Canceling a Network Connection*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetAddConnection2**, **WNetAddConnection3**, **WNetGetConnection**

WNetCloseEnum

The **WNetCloseEnum** function ends a network resource enumeration started by a call to the **WNetOpenEnum** function.

```
DWORD WNetCloseEnum(
  HANDLE hEnum   // handle to enumeration
);
```

Parameters

hEnum

[in] Specifies a handle that identifies an enumeration instance. This handle must be returned by the **WNetOpenEnum** function.

Return Values

If the function succeeds, the return value is NO ERROR.

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

Value	Meaning
ERROR_NO_NETWORK	The network is unavailable. (This condition is tested before the handle specified in the *hEnum* parameter is tested for validity.)
ERROR_INVALID_HANDLE	The *hEnum* parameter does not specifiy a valid handle.
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call the **WNetGetLastError** function.

Remarks

For a code sample that illustrates an application-defined function that enumerates all the resources on a network, see *Enumerating Network Resources*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetEnumResource**, **WNetOpenEnum**

WNetConnectionDialog

The **WNetConnectionDialog** function starts a general browsing dialog box for connecting to network resources. The function requires a handle to the owner window for the dialog box.

```
DWORD WNetConnectionDialog(
  HWND hwnd,      // handle to window owning dialog box
  DWORD dwType    // resource type
);
```

Parameters

hwnd

[in] Specifies a handle to the owner window for the dialog box.

dwType

[in] Specifies the resource type to allow connections to. This parameter can be the following value.

Value	Meaning
RESOURCETYPE_DISK	Connections to disk resources.

Return Values

If the function succeeds, the return value is NO_ERROR. If the user cancels the dialog box, the function returns –1.

If the function fails, the return value can be one of the following error codes:

Value	Meaning
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call the **WNetGetLastError** function.
ERROR_INVALID_PASSWORD	The specified password is invalid.
ERROR_NO_NETWORK	The network is unavailable.
ERROR_NOT_ENOUGH_MEMORY	There is insufficient memory to start the dialog box.

Remarks

If the user clicks **OK** in the dialog box, the requested network connection will have been made when the **WNetConnectionDialog** function returns.

If the function attempts to make a connection and the network provider returns the message ERROR_INVALID_PASSWORD, the system prompts the user to enter a password. The system uses the new password in another attempt to make the connection.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetAddConnection3**, **WNetCancelConnection2**, **WNetDisconnectDialog**

WNetConnectionDialog1

The **WNetConnectionDialog1** function brings up a general browsing dialog for connecting to network resources. The function requires a **CONNECTDLGSTRUCT** to establish the dialog box parameters.

```
DWORD WNetConnectionDialog1(
  LPCONNECTDLGSTRUCT lpConnDlgStruct // data buffer
);
```

Parameters

lpConnDlgStruct

[in/out] Pointer to a **CONNECTDLGSTRUCT** structure. The structure establishes the browsing dialog parameters.

Return Values

If the user cancels the dialog box, the function returns –1. If the function is successful, it returns NO_ERROR. Also, if the call is successful, the **dwDevNum** member of the **CONNECTDLGSTRUCT** structure contains the number of the connected device.

Typically this dialog returns an error only if the user cannot enter a dialog session. This is because errors that occur after a dialog session are reported to the user directly. If the function fails, the return value can be one of the following error codes.

Value	Meaning
ERROR_INVALID_PARAMETER	Both the CONNDLG_RO_PATH and the CONNDLG_USE_MRU dialog box options are set. (Dialog box options are specified by the **dwFlags** member of the **CONNECTDLGSTRUCT** structure.) -or- Both the CONNDLG_PERSIST and the CONNDLG_NOT_PERSIST dialog box options are set. -or- The CONNDLG_RO_PATH dialog box option is set and the **lpRemoteName** member of the **NETRESOURCE** structure does not point to a remote network. (The **CONNECTDLGSTRUCT** structure points to a **NETRESOURCE** structure.)
ERROR_BAD_DEV_TYPE	The **dwType** member of the **NETRESOURCE** structure is *not* set to RESOURCETYPE_DISK.
ERROR_BUSY	The network provider is busy (possibly initializing). The caller should retry.
ERROR_NO_NETWORK	The network is unavailable.
ERROR_NOT_ENOUGH_MEMORY	There is insufficient memory to display the dialog box.
ERROR_EXTENDED_ERROR	A network-specific error occurred. Call **WNetGetLastError** to obtain a description of the error.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **CONNECTDLGSTRUCT**, **NETRESOURCE**, **WNetConnectionDialog**, **WNetDisconnectDialog**

WNetDisconnectDialog

The **WNetDisconnectDialog** function starts a general browsing dialog box for disconnecting from network resources. The function requires a handle to the owner window for the dialog box.

```
DWORD WNetDisconnectDialog(
  HWND hwnd,      // handle to window owning dialog box
  DWORD dwType    // resource type
);
```

Parameters

hwnd
[in] Specifies a handle to the owner window for the dialog box.

dwType
[in] Specifies the resource type to disconnect from. This parameter can have the following value.

Value	Meaning
RESOURCETYPE_DISK	Disconnects from disk resources.

Return Values

If the function succeeds, the return value is NO_ERROR. If the user cancels the dialog box, the return value is –1.

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

Value	Meaning
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call the **WNetGetLastError** function.
ERROR_NO_NETWORK	The network is unavailable.
ERROR_NOT_ENOUGH_MEMORY	There is insufficient memory to start the dialog box.

Remarks

If the user chooses **OK** in the dialog box, the requested network disconnection will have been made when the **WNetDisconnectDialog** function returns.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetAddConnection2**, **WNetCancelConnection2**, **WNetConnectionDialog**, **WNetConnectionDialog1**

WNetDisconnectDialog1

The **WNetDisconnectDialog1** function attempts to disconnect a network resource. If the underlying network returns ERROR_OPEN_FILES, the function prompts the user for confirmation. If there is any error, the function informs the user. The function requires a **DISCDLGSTRUCT** to specify the parameters for the disconnect attempt.

```
DWORD WNetDisconnectDialog1(
  LPDISCDLGSTRUCT lpConnDlgStruct  // data buffer
);
```

Parameters

lpConnDlgStruct
[in] Pointer to a **DISCDLGSTRUCT** structure. The structure specifies the behavior for the disconnect attempt.

Return Values

If the function succeeds, the return value is NO_ERROR. If the user cancels the dialog box, the return value is –1.

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

Value	Meaning
ERROR_CANCELLED	When the system prompted the user for a decision about disconnecting, the user elected not to disconnect.
ERROR_OPEN_FILES	Unable to disconnect because the user is actively using the connection.
ERROR_BUSY	The network provider is busy (possibly initializing). The caller should retry.

(continued)

(continued)

Value	Meaning
ERROR_NO_NETWORK	The network is unavailable.
ERROR_NOT_ENOUGH_MEMORY	There is insufficient memory to start the dialog box.
ERROR_EXTENDED_ERROR	A network-specific error occurred. Call the **WNetGetLastError** function to obtain a description of the error.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetConnectionDialog**, **WNetConnectionDialog1**, **DISCDLGSTRUCT**, **WNetDisconnectDialog**

WNetEnumResource

The **WNetEnumResource** function continues an enumeration of network resources that was started by a call to the **WNetOpenEnum** function.

```
DWORD WNetEnumResource(
    HANDLE hEnum,           // handle to enumeration
    LPDWORD lpcCount,       // entries to list
    LPVOID lpBuffer,        // buffer
    LPDWORD lpBufferSize    // buffer size
);
```

Parameters

hEnum

[in] Specifies a handle that identifies an enumeration instance. This handle must be returned by the **WNetOpenEnum** function.

lpcCount

[in/out] Pointer to a variable specifying the number of entries requested. If the number requested is –1, the function returns as many entries as possible.

If the function succeeds, on return the variable pointed to by this parameter contains the number of entries actually read.

lpBuffer

[out] Pointer to the buffer that receives the enumeration results. The results are returned as an array of **NETRESOURCE** structures. Note that the buffer you allocate must be large enough to hold the structures, plus the strings to which their members point. For more information, see the following Remarks section.

The buffer is valid until the next call using the handle specified by the *hEnum* parameter. The order of **NETRESOURCE** structures in the array is not predictable.

lpBufferSize

[in/out] Pointer to a variable that specifies the size, in bytes, of the *lpBuffer* parameter. If the buffer is too small to receive even one entry, this parameter receives the required size of the buffer.

Return Values

If the function succeeds, the return value is one of the following values.

Value	Meaning
NO_ERROR	The enumeration succeeded, and the buffer contains the requested data. The calling application can continue to call **WNetEnumResource** to complete the enumeration.
ERROR_NO_MORE_ITEMS	There are no more entries. The buffer contents are undefined.

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

Value	Meaning
ERROR_MORE_DATA	More entries are available with subsequent calls.
ERROR_INVALID_HANDLE	The handle specified by the *hEnum* parameter is not valid.
ERROR_NO_NETWORK	The network is unavailable. (This condition is tested before *hEnum* is tested for validity.)
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call the **WNetGetLastError** function.

Remarks

The **WNetEnumResource** function does not enumerate users connected to a share; you can call the **NetConnectionEnum** function to accomplish this task. To enumerate hidden shares, call the **NetShareEnum** function.

An application cannot set the *lpBuffer* parameter to NULL and retrieve the required buffer size from the *lpBufferSize* parameter. Instead, the application should allocate a buffer of a reasonable size—16 kilobytes (K) is typical—and use the value of *lpBufferSize* for error detection.

For a code sample that illustrates an application-defined function that enumerates all the resources on a network, see *Enumerating Network Resources*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **NETRESOURCE**, **WNetCloseEnum**, **WNetOpenEnum**

WNetGetConnection

The **WNetGetConnection** function retrieves the name of the network resource associated with a local device.

```
DWORD WNetGetConnection(
    LPCTSTR lpLocalName,   // local name
    LPTSTR lpRemoteName,   // buffer for remote name
    LPDWORD lpnLength      // buffer size
);
```

Parameters

lpLocalName
[in] Pointer to a constant null-terminated string that specifies the name of the local device to get the network name for.

lpRemoteName
[out] Pointer to a buffer that receives the null-terminated remote name used to make the connection.

lpnLength
[in/out] Pointer to a variable that specifies the size, in characters, of the buffer pointed to by the *lpRemoteName* parameter. If the function fails because the buffer is not large enough, this parameter returns the required buffer size.

Return Values

If the function succeeds, the return value is NO_ERROR.

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

Value	Meaning
ERROR_BAD_DEVICE	The string pointed to by the *lpLocalName* parameter is invalid.
ERROR_NOT_CONNECTED	The device specified by *lpLocalName* is not a redirected device. For more information, see the following Remarks section.
ERROR_MORE_DATA	The buffer is too small. The *lpnLength* parameter points to a variable that contains the required buffer size. More entries are available with subsequent calls.
ERROR_CONNECTION_UNAVAIL	The device is not currently connected, but it is a persistent connection. For more information, see the following Remarks section.
ERROR_NO_NETWORK	The network is unavailable.
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call the **WNetGetLastError** function.
ERROR_NO_NET_OR_BAD_PATH	None of the providers recognize the local name as having a connection. However, the network is not available for at least one provider to whom the connection may belong.

Remarks

Windows NT/Windows 2000: If the network connection was made using the Microsoft LAN Manager network, and the calling application is running in a different logon session than the application that made the connection, a call to the **WNetGetConnection** function for the associated local device will fail. The function fails with ERROR_NOT_CONNECTED or ERROR_CONNECTION_UNAVAIL. This is because a connection made using Microsoft LAN Manager is visible only to applications running in the same logon session as the application that made the connection. (To prevent the call to **WNetGetConnection** from failing it is not sufficient for the application to be running in the user account that created the connection.)

For a code sample that illustrates how to use the **WNetGetConnection** function to retrieve the name of the network resource associated with a local device, see *Retrieving the Connection Name*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetAddConnection2**, **WNetAddConnection3**, **WNetGetUser**

WNetGetLastError

The **WNetGetLastError** function retrieves the most recent extended error code set by a WNet function. The network provider reported this error code; it will not generally be one of the errors included in the SDK header file WinError.h.

```
DWORD WNetGetLastError(
    LPDWORD lpError,        // error code
    LPTSTR lpErrorBuf,      // error description buffer
    DWORD nErrorBufSize,    // size of description buffer
    LPTSTR lpNameBuf,       // buffer for provider name
    DWORD nNameBufSize      // size of provider name buffer
);
```

Parameters

lpError
[out] Pointer to the variable that receives the error code reported by the network provider. The error code is specific to the network provider.

lpErrorBuf
[out] Pointer to the buffer that receives the null-terminated string describing the error.

nErrorBufSize
[in] Specifies the size, in characters, of the buffer pointed to by the *lpErrorBuf* parameter. If the buffer is too small for the error string, the string is truncated but still null-terminated. A buffer of at least 256 characters is recommended.

lpNameBuf
[out] Pointer to the buffer that receives the null-terminated string identifying the network provider that raised the error.

nNameBufSize
[in] Specifies the size, in characters, of the buffer pointed to by the *lpNameBuf* parameter. If the buffer is too small for the error string, the string is truncated but still null-terminated.

Return Values

If the function succeeds, and it obtains the last error that the network provider reported, the return value is NO_ERROR.

If the caller supplies an invalid buffer, the return value is ERROR_INVALID_ADDRESS.

Remarks

The **WNetGetLastError** function retrieves errors that are specific to a network provider. You can call **WNetGetLastError** when a WNet function returns ERROR_EXTENDED_ERROR.

Like the **GetLastError** function, **WNetGetLastError** returns extended error information, which is maintained on a per-thread basis. Unlike **GetLastError**, the **WNetGetLastError** function can return a string for reporting errors that are not described by any existing error code in WinError.h.

For more information about using an application-defined error handler that calls the **WNetGetLastError** function, see *Retrieving Network Errors*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*

WNetGetNetworkInformation

The **WNetGetNetworkInformation** function returns extended information about a specific network provider whose name was returned by a previous network enumeration.

```
DWORD WNetGetNetworkInformation(
    LPCTSTR lpProvider,                // provider name
    LPNETINFOSTRUCT lpNetInfoStruct    // network information
);
```

Parameters

lpProvider

[in] Pointer to a constant null-terminated string that contains the name of the network provider for which information is required.

lpNetInfoStruct
[out] Pointer to a **NETINFOSTRUCT** structure. The structure describes characteristics of the network.

Return Values

If the function succeeds, the return value is NO_ERROR.

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

Value	Meaning
ERROR_BAD_PROVIDER	The *lpProvider* parameter does not match any running network provider.
ERROR_BAD_VALUE	The **cbStructure** member of the **NETINFOSTRUCT** structure does not contain a valid structure size.

Remarks

For a code sample that illustrates an application-defined function that enumerates all the resources on a network, see *Enumerating Network Resources*.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetGetProviderName**, **NETINFOSTRUCT**, **NETRESOURCE**, **WNetOpenEnum**, **WNetEnumResource**

WNetGetProviderName

The **WNetGetProviderName** function obtains the provider name for a specific type of network.

```
DWORD WNetGetProviderName(
  DWORD dwNetType,        // network type
  LPTSTR lpProviderName, // provider name buffer
  LPDWORD lpBufferSize   // size of buffer
);
```

Parameters

dwNetType

[in] Specifies the network type that is unique to the network. If two networks claim the same type, the function returns the name of the provider loaded first. Only the high word of the network type is used. If a network reports a subtype in the low word, it is ignored.

You can find a complete list of network types in the header file WinNetwk.h.

lpProviderName

[out] Pointer to a buffer in which to return the network provider name.

lpBufferSize

[in/out] Specifies the size, in characters, of the buffer passed to the function. If the return value is ERROR_MORE_DATA, *lpBufferSize* returns the buffer size required (in characters) to hold the provider name.

Windows 95/98: The size of the buffer is in bytes, not characters. Also, the buffer must be at least 1 byte long.

Return Values

If the function succeeds, the return value is NO_ERROR.

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

Value	Meaning
ERROR_MORE_DATA	The buffer is too small to hold the network provider name.
ERROR_NO_NETWORK	The network is unavailable.
ERROR_INVALID_ADDRESS	The *lpProviderName* parameter or the *lpBufferSize* parameter is invalid.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetGetResourceInformation**, **WNetGetNetworkInformation**, **WNetGetUniversalName**

WNetGetResourceInformation

When provided with a remote path to a network resource, the **WNetGetResourceInformation** function identifies the network provider that owns the resource and obtains information about the type of the resource. The function is typically used in conjunction with the **WNetGetResourceParent** function to parse and interpret a network path typed in by a user.

```
DWORD WNetGetResourceInformation (
  LPNETRESOURCE lpNetResource, // network resource
  LPVOID lpBuffer,             // information buffer
  LPDWORD lpcbBuffer,          // size of information
                               // buffer
  LPTSTR *lplpSystem           // string
);
```

Parameters

lpNetResource

[in] Pointer to a **NETRESOURCE** structure that specifies the network resource for which information is required.

The **lpRemoteName** member of the structure should specify the remote path name of the resource, typically one typed in by a user. The **lpProvider** and **dwType** members should also be filled in if known, because this operation can be memory intensive, especially if you do not specify the **dwType** member. If you do not know the values for these members, you should set them to NULL. All other members of the **NETRESOURCE** structure are ignored.

lpBuffer

[out] Pointer to the buffer to receive the result. On successful return, the first portion of the buffer is a **NETRESOURCE** structure representing that portion of the input resource path that is accessed through the WNet functions, rather than through system functions specific to the input resource type.

For example, if the input remote resource path is \\server\share\dir1\dir2, then the output **NETRESOURCE** structure contains information about the resource \\server\share. The \dir1\dir2 portion of the path is accessed through the file I/O functions. The **lpRemoteName**, **lpProvider**, **dwType**, **dwDisplayType**, and **dwUsage** members of **NETRESOURCE** are returned, with all other members set to NULL.

The **lpRemoteName** member is returned in the same syntax as the one returned from an enumeration by the **WNetEnumResource** function. This allows the caller to perform a string comparison to determine whether the resource passed to **WNetGetResourceInformation** is the same as the resource returned by a separate call to **WNetEnumResource**.

lpcbBuffer
[in/out] Pointer to a location that, on entry, specifies the size, in bytes, of the buffer pointed to by *lpBuffer*. The buffer you allocate must be large enough to hold the **NETRESOURCE** structure, plus the strings to which its members point. If the buffer is too small for the result, this location receives the required buffer size, and the function returns ERROR_MORE_DATA.

lplpSystem
[out] If the function returns successfully, this parameter points to a string in the output buffer that specifies the part of the resource that is accessed through system functions. (This applies only to functions specific to the resource type rather than the WNet functions.)

For example, if the input remote resource name is \\server\share\dir1\dir2, the **lpRemoteName** member of the output **NETRESOURCE** structure points to \\server\share. Also, the *lplpSystem* parameter points to \dir1\dir2. Both strings are stored in the buffer pointed to by the *lpBuffer* 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_BAD_NET_NAME	The input **lpRemoteName** member is not an existing network resource for any network.
ERROR_BAD_DEV_TYPE	The input **dwType** member does not match the type of resource specified by the **lpRemoteName** member.
ERROR_EXTENDED_ERROR	A network-specific error occurred. Call **WNetGetLastError** to obtain a description of the error.
ERROR_MORE_DATA	The buffer pointed to by the *lpBuffer* parameter is too small.
ERROR_NO_NETWORK	The network is unavailable.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetGetResourceParent**, **WNetGetProviderName**, **WNetGetNetworkInformation** **WNetGetUniversalName**

WNetGetResourceParent

The **WNetGetResourceParent** function returns the parent of a network resource in the network browse hierarchy. Browsing begins at the location of the specified network resource.

Call the **WNetGetResourceInformation** and **WNetGetResourceParent** functions to move up the network hierarchy. Call the **WNetOpenEnum** function to move down the hierarchy.

```
DWORD WNetGetResourceParent(
  LPNETRESOURCE lpNetResource,  // network resource
  LPVOID lpBuffer,              // buffer
  LPDWORD lpcbBuffer            // buffer size
);
```

Parameters

lpNetResource

[in] Pointer to a **NETRESOURCE** structure that specifies the network resource for which the parent name is required.

Specify the members of the input **NETRESOURCE** structure as follows. The caller typically knows the values to provide for the **lpProvider** and **dwType** members after previous calls to **WNetGetResourceInformation** or **WNetGetResourceParent**.

Member	Description
dwType	This member should be filled in if known; otherwise, it should be set to NULL.
lpRemoteName	This member should specify the remote name of the network resource whose parent is required.
lpProvider	This member should specify the network provider that owns the resource. This member is required; otherwise, the function could produce incorrect results.

All other members of the **NETRESOURCE** structure are ignored.

lpBuffer

[out] Pointer to a buffer to receive a single **NETRESOURCE** structure that represents the parent resource. The function returns the **lpRemoteName**, **lpProvider**, **dwType**, **dwDisplayType**, and **dwUsage** members of the structure; all other members are set to NULL.

The **lpRemoteName** member points to the remote name for the parent resource. This name uses the same syntax as the one returned from an enumeration by the **WNetEnumResource** function. The caller can perform a string comparison to determine whether the **WNetGetResourceParent** resource is the same as that returned by **WNetEnumResource**. If the input resource has no parent on any of the networks, the **lpRemoteName** member is returned as NULL.

The presence of the RESOURCEUSAGE_CONNECTABLE bit in the **dwUsage** member indicates that you can connect to the parent resource, but only when it is available on the network.

lpcbBuffer

[in/out] Pointer to a location that, on entry, specifies the size, in bytes, of the buffer pointed to by the *lpBuffer* parameter. If the buffer is too small to hold the result, this location receives the required buffer size, and the function returns ERROR_MORE_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	Meaning
ERROR_ACCESS_DENIED	The user is authenticated to the network, but does not have sufficient permissions (access rights) to perform the operation.
ERROR_BAD_NET_NAME	The input **lpRemoteName** member is not an existing network resource for any network.
ERROR_BAD_PROVIDER	The input **lpProvider** member does not match any installed network provider.
ERROR_MORE_DATA	The buffer pointed to by the *lpBuffer* parameter is too small.
ERROR_NOT_AUTHENTICATED	The caller does not have the necessary permissions to obtain the name of the parent.

Remarks

The **WNetGetResourceParent** function is typically used in conjunction with the **WNetGetResourceInformation** function to parse and interpret a network path typed in by a user.

Unlike the **WNetGetResourceInformation** function, if the resource includes a parent in its syntax, the **WNetGetResourceParent** function returns the parent, whether or not the resource actually exists. **WNetGetResourceParent** should typically be used only by applications that display network resources to the user in a hierarchical fashion.

The Windows Explorer and the **File Open** dialog box are two well-known examples of this type of application. Note that no assumptions should be made about the type of resource that will be returned.

You can call the **WNetEnumResource**, **WNetGetResourceInformation**, or **WNetGetResourceParent** function to return information from the **NETRESOURCE** structure. You can also construct network resource information using the members of the **NETRESOURCE** structure.

An example of an inappropriate use of **WNetGetResourceParent** is to determine the name of the domain to which a specified server belongs. The function may happen to return the correct domain name for some networks in which domains appear directly above servers in the browse hierarchy.The function will return incorrect results for other networks.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetGetResourceInformation**, **WNetGetProviderName**, **WNetGetNetworkInformation**, **WNetGetUniversalName**

WNetGetUniversalName

The **WNetGetUniversalName** function takes a drive-based path for a network resource and returns an information structure that contains a more universal form of the name.

```
DWORD WNetGetUniversalName(
  LPCTSTR lpLocalPath,  // path for network resource
  DWORD dwInfoLevel,    // level of information
  LPVOID lpBuffer,      // name buffer
  LPDWORD lpBufferSize  // size of buffer
);
```

Parameters

lpLocalPath
[in] Pointer to a constant null-terminated string that is a drive-based path for a network resource.

For example, if drive H has been mapped to a network drive share, and the network resource of interest is a file named SAMPLE.DOC in the directory \WIN32\EXAMPLES on that share, the drive-based path is H:\WIN32\EXAMPLES\SAMPLE.DOC.

dwInfoLevel

[in] Specifies the type of structure that the function stores in the buffer pointed to by the *lpBuffer* parameter. This parameter can be one of the following values.

Value	Meaning
UNIVERSAL_NAME_INFO_ LEVEL	The function stores a **UNIVERSAL_NAME_INFO** structure in the buffer.
REMOTE_NAME_INFO_ LEVEL	The function stores a **REMOTE_NAME_INFO** structure in the buffer.

The **UNIVERSAL_NAME_INFO** structure points to a Universal Naming Convention (UNC) name string.

The **REMOTE_NAME_INFO** structure points to a UNC name string and two additional connection information strings. For more information, see the following Remarks section.

lpBuffer

[out] Pointer to a buffer that receives the structure specified by the *dwInfoLevel* parameter.

lpBufferSize

[in/out] Pointer to a variable that specifies the size, in bytes, of the buffer pointed to by the *lpBuffer* parameter.

If the function succeeds, it sets the variable pointed to by *lpBufferSize* to the number of bytes stored in the buffer. If the function fails because the buffer is too small, this location receives the required buffer size, and the function returns ERROR_MORE_DATA.

Return Values

If the function succeeds, the return value is NO_ERROR.

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

Value	Meaning
ERROR_BAD_DEVICE	The string pointed to by the *lpLocalPath* parameter is invalid.
ERROR_CONNECTION_UNAVAIL	There is no current connection to the remote device, but there is a remembered (persistent) connection to it.

(continued)

(continued)

Value	Meaning
ERROR_EXTENDED_ERROR	A network-specific error occurred. Use the **WNetGetLastError** function to obtain a description of the error.
ERROR_MORE_DATA	The buffer pointed to by the *lpBuffer* parameter is too small. The function sets the variable pointed to by the *lpBufferSize* parameter to the required buffer size. More entries are available with subsequent calls.
ERROR_NOT_SUPPORTED	The *dwInfoLevel* parameter is set to UNIVERSAL_NAME_INFO_LEVEL, but the network provider does not support UNC names. (None of the network providers support this function.)
ERROR_NO_NET_OR_BAD_PATH	None of the network providers recognize the local name as having a connection. However, the network is not available for at least one provider to whom the connection may belong.
ERROR_NO_NETWORK	The network is unavailable.
ERROR_NOT_CONNECTED	The device specified by the *lpLocalPath* parameter is not redirected.

Remarks

A universal form of a local drive-based path identifies a network resource in an unambiguous, computer-independent manner. The name can then be passed to processes on other computers, allowing those processes to obtain access to the resource.

The **WNetGetUniversalName** function currently supports one universal name form: universal naming convention (UNC) names, which look like the following:

```
\\servername\sharename\path\file
```

Using the example from the preceding description of the *lpLocalPath* parameter, if the shared network drive is on a server named COOLSERVER, and the share name is HOTSHARE, the UNC name for the network resource whose drive-based name is H:\WIN32\EXAMPLES\SAMPLE.DOC would be

```
\\coolserver\hotshare\win32\examples\sample.doc
```

The **UNIVERSAL_NAME_INFO** structure contains a pointer to a UNC name string. The **REMOTE_NAME_INFO** structure also contains a pointer to a UNC name string as well as pointers to two other useful strings. For example, a process can pass the **REMOTE_NAME_INFO** structure's **lpszConnectionInfo** member to the **WNetAddConnection2** function to connect a local device to the network resource.

Then the process can append the string pointed to by the **lpszRemainingPath** member to the local device string. The resulting string can be passed to Win32 functions that require a drive-based path.

For an example that demonstrates how to use the **WNetGetUniversalName** function to determine the location of a share on a redirected drive, see *Determining the Location of a Share*.

! Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **REMOTE_NAME_INFO**, **UNIVERSAL_NAME_INFO**, **WNetAddConnection2**

WNetGetUser

The **WNetGetUser** function retrieves the current default user name, or the user name used to establish a network connection.

```
DWORD WNetGetUser(
  LPCTSTR lpName,      // device or resource name
  LPTSTR lpUserName,   // name buffer
  LPDWORD lpnLength    // buffer size
);
```

Parameters

lpName

[in] Pointer to a constant null-terminated string that specifies either the name of a local device that has been redirected to a network resource, or the remote name of a network resource to which a connection has been made without redirecting a local device.

If this parameter is NULL, the system returns the name of the current user for the process.

lpUserName

[out] Pointer to a buffer that receives the null-terminated user name.

lpnLength
[in/out] Pointer to a variable that specifies the size, in characters, of the buffer pointed to by the *lpUserName* parameter. If the call fails because the buffer is not large enough, this variable contains the required buffer size.

Return Values

If the function succeeds, the return value is NO_ERROR.

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

Value	Meaning
ERROR_NOT_CONNECTED	The device specified by the *lpName* parameter is not a redirected device or a connected network name.
ERROR_MORE_DATA	More entries are available with subsequent calls.
ERROR_NO_NETWORK	The network is unavailable.
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call the **WNetGetLastError** function.
ERROR_NO_NET_OR_BAD_PATH	None of the providers recognize the local name as having a connection. However, the network is not available for at least one provider to whom the connection may belong.

Remarks

For a code sample that illustrates how to use the **WNetGetUser** function to retrieve the name of the user associated with a local device, see *Retrieving the User Name*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetGetConnection**

WNetOpenEnum

The **WNetOpenEnum** function starts an enumeration of network resources or existing connections. You can continue the enumeration by calling the **WNetEnumResource** function:

```
DWORD WNetOpenEnum(
  DWORD dwScope,                   // scope of enumeration
  DWORD dwType,                    // resource types to list
  DWORD dwUsage,                   // resource usage to list
  LPNETRESOURCE lpNetResource,     // resource structure
  LPHANDLE lphEnum                 // enumeration handle buffer
);
```

Parameters

dwScope

[in] Specifies the scope of the enumeration. This parameter can be one of the following values.

Value	Meaning
RESOURCE_CONNECTED	Enumerate all currently connected resources. The function ignores the *dwUsage* parameter. For more information, see the following Remarks section.
RESOURCE_CONTEXT	Enumerate only resources in the network context of the caller. Specify this value for a *Network Neighborhood* view. The function ignores the *dwUsage* parameter.
RESOURCE_GLOBALNET	Enumerate all resources on the network.
RESOURCE_REMEMBERED	Enumerate all remembered (persistent) connections. The function ignores the *dwUsage* parameter.

dwType

[in] Specifies the resource types to enumerate. This parameter can be a combination of the following values.

Value	Meaning
RESOURCETYPE_ANY	All resources. This value cannot be combined with RESOURCETYPE_DISK or RESOURCETYPE_PRINT.
RESOURCETYPE_DISK	All disk resources.
RESOURCETYPE_PRINT	All print resources.

If a network provider cannot distinguish between print and disk resources, it can enumerate all resources.

dwUsage

[in] Specifies the resource usage type to enumerate. This parameter can be a combination of the following values.

Value	Meaning
0	All resources.
RESOURCEUSAGE_ CONNECTABLE	All connectable resources.
RESOURCEUSAGE_CONTAINER	All container resources.
RESOURCEUSAGE_ATTACHED	Setting this value forces **WNetOpenEnum** to fail if the user is not authenticated. The function fails even if the network allows enumeration without authentication.
RESOURCEUSAGE_ALL	Setting this value is equivalent to setting RESOURCEUSAGE_CONNECTABLE, RESOURCEUSAGE_CONTAINER, and RESOURCEUSAGE_ATTACHED.

This parameter is ignored unless the *dwScope* parameter is equal to RESOURCE_GLOBALNET. For more information, see the following Remarks section.

lpNetResource

[in] Pointer to a **NETRESOURCE** structure that specifies the container to enumerate. If the *dwScope* parameter is not RESOURCE_GLOBALNET, this parameter must be NULL.

If this parameter is NULL, the root of the network is assumed. (The system organizes a network as a hierarchy; the root is the topmost container in the network.)

If this parameter is not NULL, it must point to a **NETRESOURCE** structure. This structure can be filled in by the application or it can be returned by a call to the **WNetEnumResource** function. The **NETRESOURCE** structure must specify a container resource; that is, the RESOURCEUSAGE_CONTAINER value must be specified in the *dwUsage* parameter.

To enumerate all network resources, an application can begin the enumeration by calling **WNetOpenEnum** with the *lpNetResource* parameter set to NULL, and then use the returned handle to call **WNetEnumResource** to enumerate resources. If one of the resources in the **NETRESOURCE** array returned by the **WNetEnumResource** function is a container resource, you can call **WNetOpenEnum** to open the resource for further enumeration.

lphEnum

[out] Pointer to an enumeration handle that can be used in a subsequent call to **WNetEnumResource**.

Return Values

If the function succeeds, the return value is NO_ERROR.

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

Value	Meaning
ERROR_NOT_CONTAINER	The *lpNetResource* parameter does not point to a container.
ERROR_INVALID_PARAMETER	Either the *dwScope* or the *dwType* parameter is invalid, or there is an invalid combination of parameters.
ERROR_NO_NETWORK	The network is unavailable.
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call the **WNetGetLastError** function.

Remarks

Windows NT/Windows 2000: If the *dwScope* parameter is equal to RESOURCE_CONNECTED, a network connection made using the Microsoft LAN Manager network is omitted from the enumeration if the connection was made by an application running in a different logon session than the application calling the **WNetOpenEnum** function. This is because connections made using Microsoft LAN Manager are visible only to applications running in the same logon session as the application that made the connection. (To include the connection in the enumeration, it is not sufficient for the application to be running in the user account that created the connection.)

The exact interpretation of RESOURCE_CONTEXT in the *dwScope* parameter depends on the networks installed on the machine.

The **WNetOpenEnum** function is used to begin enumeration of the resources in a single container. The following examples show the hierarchical structure of a Microsoft LAN Manager network and a Novell Netware network and identify the containers:

```
LanMan (container, in this case the provider)
  ACCOUNTING (container, in this case the domain)
    \\ACCTSPAY (container, in this case the server)
      PAYFILES (disk)
      LASERJET (print)

Netware (container, in this case the provider)
  MARKETING (container, in this case the server)
    SYS (disk, first one on any Netware server)
    ANOTHERVOLUME (disk)
    LASERJET (print)
```

For a code sample that illustrates an application-defined function that enumerates all the resources on a network, see *Enumerating Network Resources*.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **NETRESOURCE**, **WNetCloseEnum**, **WNetEnumResource**

WNetUseConnection

The **WNetUseConnection** function makes a connection to a network resource. The function can redirect a local device to a network resource.

The **WNetUseConnection** function is similar to the **WNetAddConnection3** function. The main difference is that **WNetUseConnection** can automatically select an unused local device to redirect to the network resource.

Windows NT/2000: The parameter order is as follows.

```
DWORD WNetUseConnection(
  HWND hwndOwner,                // owner window
  LPNETRESOURCE lpNetResource,   // connection details
  LPCTSTR lpUserID,              // user
  LPCTSTR lpPassword,            // password
  DWORD dwFlags,                 // connection options
  LPTSTR lpAccessName,           // buffer for system requests
  LPDWORD lpBufferSize,          // buffer size
  LPDWORD lpResult               // receives connection information
);
```

Windows 95/98: The *lpUserID* and *lpPassword* parameters are in reverse order from the order used on Windows NT/Windows 2000. Therefore, the parameter order is as follows.

```
DWORD WNetUseConnection(
  HWND hwndOwner,
  LPNETRESOURCE lpNetResource,
  LPCTSTR lpPassword,
  LPCTSTR lpUserID,
  DWORD dwFlags,
  LPTSTR lpAccessName,
    LPDWORD lpBufferSize,
    LPDWORD lpResult
);
```

Parameters

hwndOwner

[in] Specifies the handle to a window that the provider of network resources can use as an owner window for dialog boxes. Use this parameter if you set the CONNECT_INTERACTIVE value in the *dwFlags* parameter.

lpNetResource

[in] Pointer to a **NETRESOURCE** structure that specifies details of the proposed connection. The structure contains information about the network resource, the local device, and the network resource provider.

You must specify the following members of the **NETRESOURCE** structure.

Member	Description
dwType	Specifies the type of resource to connect to. It is most efficient to specify a resource type in this member, such as RESOURCETYPE_DISK or RESOURCETYPE_PRINT. However, if the **lpLocalName** member is NULL, or if it points to an empty string and CONNECT_REDIRECT is *not* set, **dwType** can be RESOURCETYPE_ANY. This method works only if the function does not automatically choose a device to redirect to the network resource.
lpLocalName	Pointer to a null-terminated string that specifies the name of a local device to be redirected, such as "F:" or "LPT1". The string is treated in a case-insensitive manner. If the string is empty, or if **lpLocalName** is NULL, a connection to the network occurs without redirection. If the CONNECT_REDIRECT value is set in the *dwFlags* parameter, or if the network requires a redirected local device, the function chooses a local device to redirect and returns the name of the device in the *lpAccessName* parameter.
lpRemoteName	Pointer to a null-terminated string that specifies the network resource to connect to. The string can be up to MAX_PATH characters in length, and it must follow the network provider's naming conventions.

(continued)

(continued)

Member	Description
lpProvider	Pointer to a null-terminated string that specifies the network provider to connect to. If **lpProvider** is NULL, or if it points to an empty string, the operating system attempts to determine the correct provider by parsing the string pointed to by the **lpRemoteName** member. If this member is *not* NULL, the operating system attempts to make a connection only to the named network provider. You should set this member only if you know the network provider you want to use. Otherwise, let the operating system determine which provider the network name maps to.

The **WNetUseConnection** function ignores the other members of the **NETRESOURCE** structure. For more information, see the descriptions following for the *dwFlags* parameter.

lpUserID

[in] Pointer to a constant null-terminated string that specifies a user name for making the connection.

If *lpUserID* is NULL, the function uses the default user name. (The user context for the process provides the default user name.)

The *lpUserID* parameter is specified when users want to connect to a network resource for which they have been assigned a user name or account other than the default user name or account.

The user-name string represents a security context. It may be specific to a network provider.

lpPassword

[in] Pointer to a constant null-terminated string that specifies a password to be used in making the network connection.

If *lpPassword* is NULL, the function uses the current default password associated with the user specified by *lpUserID.*

If *lpPassword* points to an empty string, the function does not use a password.

If the connection fails because of an invalid password and the CONNECT_INTERACTIVE value is set in the *dwFlags* parameter, the function displays a dialog box asking the user to type the password.

dwFlags

[in] Specifies a **DWORD** value that contains a set of bit flags describing the connection. This parameter can be any combination of the following values.

Value	Meaning
CONNECT_INTERACTIVE	If this flag is set, the operating system may interact with the user for authentication purposes.
CONNECT_PROMPT	This flag instructs the system not to use any default settings for user names or passwords without offering the user the opportunity to supply an alternative. This flag is ignored unless CONNECT_INTERACTIVE is also set.
CONNECT_REDIRECT	This flag forces the redirection of a local device when making the connection. If the **lpLocalName** member of **NETRESOURCE** specifies a local device to redirect, this flag has no effect, because the operating system still attempts to redirect the specified device. When the operating system automatically chooses a local device, the *lpAccessName* parameter must point to a return buffer and the **dwType** member must not be equal to RESOURCETYPE_ANY. If this flag is *not* set, a local device is automatically chosen for redirection only if the network requires a local device to be redirected.
CONNECT_UPDATE_PROFILE	This flag instructs the operating system to store the network resource connection. If this bit flag is set, the operating system automatically attempts to restore the connection when the user logs on. The system remembers only successful connections that redirect local devices. It does not remember connections that are unsuccessful or deviceless connections. (A deviceless connection occurs when **lpLocalName** is NULL or when it points to an empty string.) If this bit flag is clear, the operating system does not automatically restore the connection at logon.

lpAccessName

[out] Pointer to a buffer that receives system requests on the connection.

If the **lpLocalName** member of the **NETRESOURCE** structure specifies a local device, this buffer is optional, and will have the local device name copied into it. If **lpLocalName** does not specify a device and the network requires a local device redirection, or if the CONNECT_REDIRECT value is set, this buffer is required and the name of the redirected local device is returned here.

Otherwise, the name copied into the buffer is that of a remote resource. If specified, this buffer must be at least as large as the string pointed to by the **lpRemoteName** member.

lpBufferSize

[in/out] Pointer to a variable that specifies the size, in characters, of the *lpAccessName* buffer. If the call fails because the buffer is not large enough, the function returns the required buffer size in this location.

lpResult

[out] Pointer to a variable that receives additional information about the connection. This parameter can be the following value.

Value	Meaning
CONNECT_LOCALDRIVE	If this flag is set, the connection was made using a local device redirection. If the *lpAccessName* parameter points to a buffer, the local device name is copied to the buffer.

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	Access to the network resource was denied.
ERROR_ALREADY_ASSIGNED	The local device specified by the **lpLocalName** member is already connected to a network resource.
ERROR_BAD_DEVICE	The value specified by **lpLocalName** is invalid.
ERROR_BAD_NET_NAME	The value specified by the **lpRemoteName** member is not acceptable to any network resource provider because the resource name is invalid, or because the named resource cannot be located.
ERROR_BAD_PROVIDER	The value specified by the **lpProvider** member does not match any provider.

Value	Meaning
ERROR_CANCELLED	The attempt to make the connection was canceled by the user through a dialog box from one of the network resource providers, or by a called resource.
ERROR_EXTENDED_ERROR	A network-specific error occurred. To obtain a description of the error, call the **WNetGetLastError** function.
ERROR_INVALID_ADDRESS	The caller passed in a pointer to a buffer that could not be accessed.
ERROR_INVALID_PARAMETER	This error is a result of one of the following conditions: 1. The **lpRemoteName** member is NULL. In addition, *lpAccessName* is not NULL, but *lpBufferSize* is either NULL or points to zero. 2. The **dwType** member is neither RESOURCETYPE_DISK nor RESOURCETYPE_PRINT. In addition, either CONNECT_REDIRECT is set in *dwFlags* and **lpLocalName** is NULL, or the connection is to a network that requires the redirecting of a local device.
ERROR_INVALID_PASSWORD	The specified password is invalid and the CONNECT_INTERACTIVE flag is *not* set.
ERROR_MORE_DATA	The *lpAccessName* buffer is too small. If a local device is redirected, the buffer needs to be large enough to contain the local device name. Otherwise, the buffer needs to be large enough to contain either the string pointed to by **lpRemoteName**, or the name of the connectable resource whose alias is pointed to by **lpRemoteName**. If this error is returned, then no connection has been made.
ERROR_NO_MORE_ITEMS	The operating system cannot automatically choose a local redirection because all the valid local devices are in use.
ERROR_NO_NET_OR_BAD_PATH	The operation could not be completed, either because a network component is not started, or because the specified resource name is not recognized.
ERROR_NO_NETWORK	The network is unavailable.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Library: Use Mpr.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Windows Networking (WNet) Overview, *Windows Networking Functions*, **WNetAddConnection2**, **WNetAddConnection3**, **WnetCancelConnection**, **WNetGetConnection**

Windows Networking Structures

The following structures are used in Windows networking:

CONNECTDLGSTRUCT
DISCDLGSTRUCT
NETCONNECTINFOSTRUCT
NETINFOSTRUCT
NETRESOURCE
REMOTE_NAME_INFO
UNIVERSAL_NAME_INFO

CONNECTDLGSTRUCT

The **CONNECTDLGSTRUCT** structure is used by the **WNetConnectionDialog1** function to establish browsing dialog box parameters.

```
typedef struct {
    DWORD cbStructure;
    HWND hwndOwner;
    LPNETRESOURCE lpConnRes;
    DWORD dwFlags;
    DWORD dwDevNum;
} CONNECTDLGSTRUCT, *LPCONNECTDLGSTRUCT;
```

Members

cbStructure
Specifies, in bytes, the size of the **CONNECTDLGSTRUCT** structure. The caller must supply this value.

hwndOwner
Specifies a handle to the owner window for the dialog box.

lpConnRes

Pointer to a **NETRESOURCE** structure.

If the **lpRemoteName** member of **NETRESOURCE** is specified, it will be entered into the path field of the dialog box. With the exception of the **dwType** member, all other members of the **NETRESOURCE** structure must be set to NULL. The **dwType** member must be equal to RESOURCETYPE_DISK.
Windows NT/2000: The system does not support the RESOURCETYPE_PRINT flag for browsing and connecting to print resources.

dwFlags

Specifies a **DWORD** value that contains a set of bit flags describing variations in the dialog box display. This member can be a combination of the following values.

Value	Meaning
SidTypeUser	The account is a user account.
CONNDLG_RO_PATH	Display a read-only path instead of allowing the user to type in a path. This flag should be set only if the **lpRemoteName** member of the **NETRESOURCE** structure pointed to by *lpConnRes* is *not* NULL (or an empty string), and the CONNDLG_USE_MRU flag is *not* set.
CONNDLG_CONN_POINT	Internal flag. Do not use.
CONNDLG_USE_MRU	Enter the most recently used paths into the combination box. Set this value to simulate the **WNetConnectionDialog** function.
CONNDLG_HIDE_BOX	Show the check box allowing the user to restore the connection at logon.
CONNDLG_PERSIST	Restore the connection at logon.
CONNDLG_NOT_PERSIST	Do not restore the connection at logon.

For more information, see the following *Remarks* section.

dwDevNum

If the call to the **WNetConnectionDialog1** function is successful, this member returns the number of the connected device. The value is 1 for A:, 2 for B:, 3 for C:, and so on. If the user made a deviceless connection, the value is –1.

Remarks

If neither the CONNDLG_RO_PATH nor the CONNDLG_USE_MRU flag is set, and the **lpRemoteName** member of the **NETRESOURCE** structure does not specify a remote path, the request defaults to the CONNDLG_RO_PATH dialog display type.

The CONNDLG_PERSIST and CONNDLG_NOT_PERSIST values cannot both be set. If neither is set, the dialog box defaults to the last option that was selected in this dialog box for the particular type of device connection.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Unicode: Declared as Unicode and ANSI structures.

See Also

Windows Networking (WNet) Overview, *Windows Networking Structures*, **NETRESOURCE**, **WNetConnectionDialog1**

DISCDLGSTRUCT

The **DISCDLGSTRUCT** structure is used in the **WNetDisconnectDialog1** function. The structure contains required information for the disconnect attempt.

```
typedef struct _DISCDLGSTRUCT{
    DWORD cbStructure;
    HWND hwndOwner;
    LPTSTR lpLocalName;
    LPTSTR lpRemoteName;
    DWORD dwFlags;
} DISCDLGSTRUCT, *LPDISCDLGSTRUCT;
```

Members

cbStructure
Specifies the size, in bytes, of the **DISCDLGSTRUCT** structure. The caller must supply this value.

hwndOwner
Specifies a handle to the owner window of the dialog box.

lpLocalName
Pointer to a null-terminated character string that specifies the local device name that is redirected to the network resource, such as "F:" or "LPT1".

lpRemoteName
Pointer to a null-terminated character string that specifies the name of the network resource to disconnect. This member can be NULL if the **lpLocalName** member is specified. When **lpLocalName** is specified, the connection to the network resource redirected from **lpLocalName** is disconnected.

dwFlags

Specifies a **DWORD** value that contains a set of bit flags describing the connection. This member can be a combination of the following values.

Value	Meaning
DISC_UPDATE_PROFILE	If this value is set, the specified connection is no longer a persistent one (automatically restored every time the user logs on). This flag is valid only if the **lpLocalName** member specifies a local device.
DISC_NO_FORCE	If this value is *not* set, the system applies force when attempting to disconnect from the network resource. This situation typically occurs when the user has files open over the connection. This value means that the user will be informed if there are open files on the connection, and asked if he or she still wants to disconnect. If the user wants to proceed, the disconnect procedure re-attempts with additional force.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.
Unicode: Declared as Unicode and ANSI structures.

See Also

Windows Networking (WNet) Overview, *Windows Networking Structures*, **WNetDisconnectDialog1**

NETCONNECTINFOSTRUCT

The **NETCONNECTINFOSTRUCT** structure contains information about the expected performance of a connection used to access a network resource. The structure is used in the **MultinetGetConnectionPerformance** function.

```
typedef struct _NETCONNECTINFOSTRUCT{
    DWORD cbStructure;
    DWORD dwFlags;
    DWORD dwSpeed;
    DWORD dwDelay;
    DWORD dwOptDataSize;
} NETCONNECTINFOSTRUCT, *LPNETCONNECTINFOSTRUCT;
```

Members

cbStructure

Specifies the size, in bytes, of the **NETCONNECTINFOSTRUCT** structure. The caller must supply this value.

dwFlags

Specifies a **DWORD** value that contains a set of bit flags describing the connection. This member can be one or more of the following values.

Value	Meaning
WNCON_FORNETCARD	In the absence of information about the actual connection, the information returned applies to the performance of the network card. If this flag is *not* set, information is being returned for the current connection with the resource, with any routing degradation taken into consideration.
WNCON_NOTROUTED	The connection is not being routed. If this flag is *not* set, the connection may be going through routers that limit performance. Consequently, if WNCON_FORNETCARD is set, actual performance may be much less than the information returned.
WNCON_SLOWLINK	The connection is over a medium that is typically slow (for example, over a modem using a normal quality phone line). You should not set the WNCON_SLOWLINK bit if the **dwSpeed** member is set to a nonzero value.
WNCON_DYNAMIC	Some of the information returned is calculated dynamically, so reissuing this request may return different (and more current) information.

dwSpeed

Specifies a **DWORD** value that contains the speed of the media to the network resource in units of 100 bits per second (bps).

For example, a 1200 baud point-to-point link returns 12. A value of zero indicates that no information is available. A value of one indicates that the actual value is greater than the maximum that can be represented by the member.

dwDelay

Specifies a **DWORD** value that contains the one-way delay time, in milliseconds, that the network introduces when sending information. (The delay is the time between when the network begins sending data and the time that the data starts being received.) This delay is in addition to any latency incorporated in the calculation of the **dwSpeed** member; therefore the value of this member is zero for most resources.

A value of zero indicates that no information is available. A value of one indicates that the actual value is greater than the maximum that can be represented by the member.

dwOptDataSize

Specifies a **DWORD** value that contains a recommendation for the size of data, in bytes, that an application should use when making a single request to the network resource.

For example, for a disk network resource, this value might be 2048 or 512 when writing a block of data. A value of zero indicates that no information is available.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Header: Declared in Npapi.h.

See Also

Windows Networking (WNet) Overview, *Windows Networking Structures*, **MultinetGetConnectionPerformance**

NETINFOSTRUCT

The **NETINFOSTRUCT** structure is used in the **WNetGetNetworkInformation** function. The structure contains information describing the network, such as the version of the network provider software and the network's current status.

```
typedef struct _NETINFOSTRUCT{
    DWORD cbStructure;
    DWORD dwProviderVersion;
    DWORD dwStatus;
    DWORD dwCharacteristics;
    ULONG_PTR dwHandle;
    WORD wNetType;
    DWORD dwPrinters;
    DWORD dwDrives;
} NETINFOSTRUCT, *LPNETINFOSTRUCT;
```

Members

cbStructure

Specifies the size, in bytes, of the **NETINFOSTRUCT** structure. The caller must supply this value to indicate the size of the structure passed in. Upon return, it has the size of the structure filled in.

dwProviderVersion

Specifies a **DWORD** value that contains the version number of the network provider software.

dwStatus

Specifies a **DWORD** value that contains the current status of the network provider software. This member can be one of the following values.

Value	Meaning
NO_ERROR	The network is running.
ERROR_NO_NETWORK	The network is unavailable.
ERROR_BUSY	The network is not currently able to service requests, but it should become available shortly. (This value typically indicates that the network is starting up.)

dwCharacteristics

Specifies a **DWORD** value that contains characteristics of the network provider software.

Windows NT/2000: This value is always set to zero.

Windows 95/98: This member can be one or more of the following values.

Value	Meaning
NETINFO_DLL16	The network provider is running as a 16-bit Windows network driver.
NETINFO_DISKRED	The network provider requires a redirected local disk drive device to access server file systems.
NETINFO_PRINTERRED	The network provider requires a redirected local printer port to access server file systems.

dwHandle

Specifies an instance handle for the network provider or for the 16-bit Windows network driver.

wNetType

Specifies a network type unique to the running network. This value associates resources with a specific network when the resources are persistent or stored in links. You can find a complete list of network types in the header file WinNetwk.h.

dwPrinters

Specifies a **DWORD** value that contains a set of bit flags indicating the valid print numbers for redirecting local printer devices, with the low-order bit corresponding to LPT1.

Windows 95/98: This value is always set to –1.

dwDrives

Specifies a **DWORD** value that contains a set of bit flags indicating the valid local disk devices for redirecting disk drives, with the low-order bit corresponding to A: .

Windows 95/98: This value is always set to –1.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnetwk.h.

See Also

Windows Networking (WNet) Overview, *Windows Networking Structures*, **WNetGetNetworkInformation**

NETRESOURCE

The **NETRESOURCE** structure contains information about a network resource. The structure is returned during enumeration of network resources. **NETRESOURCE** is also specified when making or querying a network connection with calls to various Windows Networking functions.

```
typedef struct _NETRESOURCE {
    DWORD  dwScope;
    DWORD  dwType;
    DWORD  dwDisplayType;
    DWORD  dwUsage;
    LPTSTR lpLocalName;
    LPTSTR lpRemoteName;
    LPTSTR lpComment;
    LPTSTR lpProvider;
} NETRESOURCE;
```

Members

dwScope

Specifies a **DWORD** value that contains the scope of the enumeration. This member can be one of the following values.

Value	Meaning
RESOURCE_CONNECTED	Enumerate currently connected resources. The **dwUsage** member cannot be specified.
RESOURCE_GLOBALNET	Enumerate all resources on the network. The **dwUsage** member is specified.
RESOURCE_REMEMBERED	Enumerate remembered (persistent) connections. The **dwUsage** member cannot be specified.

dwType

Specifies a **DWORD** value that contains a set of bit flags identifying the type of resource. This member can be one of the following values.

Value	Meaning
RESOURCETYPE_ANY	All resources
RESOURCETYPE_DISK	Disk resources
RESOURCETYPE_PRINT	Print resources

The **WNetEnumResource** function can also return the value RESOURCETYPE_UNKNOWN if a resource is neither a disk nor a print resource.

dwDisplayType

Specifies a **DWORD** value that indicates how the network object should be displayed in a network browsing user interface. This member can be one of the following values.

Value	Meaning
RESOURCEDISPLAYTYPE_ DOMAIN	The object should be displayed as a domain.
RESOURCEDISPLAYTYPE_ SERVER	The object should be displayed as a server.
RESOURCEDISPLAYTYPE_ SHARE	The object should be displayed as a share.
RESOURCEDISPLAYTYPE_ GENERIC	The method used to display the object does not matter.

dwUsage

Specifies a **DWORD** value that contains a set of bit flags describing how the resource can be used.

Note that this member can be specified only if the **dwScope** member is equal to RESOURCE_GLOBALNET. This member can be one of the following values.

Value	Meaning
RESOURCEUSAGE_ CONNECTABLE	The resource is a connectable resource; the name pointed to by the **lpRemoteName** member can be passed to the **WNetAddConnection** function to make a network connection.
RESOURCEUSAGE_ CONTAINER	The resource is a container resource; the name pointed to by the **lpRemoteName** member can be passed to the **WNetOpenEnum** function to enumerate the resources in the container.

lpLocalName

If the **dwScope** member is equal to RESOURCE_CONNECTED or RESOURCE_REMEMBERED, this member is a pointer to a null-terminated character string that specifies the name of a local device. This member is NULL if the connection does not use a device.

lpRemoteName
If the entry is a network resource, this member is a pointer to a null-terminated character string that specifies the remote network name.

If the entry is a current or persistent connection, **lpRemoteName** points to the network name associated with the name pointed to by the **lpLocalName** member.

The string can be MAX_PATH characters in length, and it must follow the network provider's naming conventions.

lpComment
Pointer to a null-terminated character string that contains a comment supplied by the network provider.

lpProvider
Pointer to a null-terminated character string that contains the name of the provider that owns the resource. This member can be NULL if the provider name is unknown. To retrieve the provider name, you can call the **WNetGetProviderName** function.

Remarks

For more information about setting the values of the **dwType**, **lpLocalName**, **lpRemoteName**, and **lpProvider** members, see ***MultinetGetConnectionPerformance***, ***WNetAddConnection2***, ***WNetAddConnection3***, ***WNetGetResourceInformation***, ***WNetGetResourceParent***, and ***WNetUseConnection***.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Header: Declared in Npapi.h.

See Also

Windows Networking (WNet) Overview, *Windows Networking Structures*, **MultinetGetConnectionPerformance**, **WNetAddConnection2**, **WNetAddConnection3**, **WNetUseConnection**, **WNetCloseEnum**, **WNetEnumResource**, **WNetGetProviderName**, **WNetGetResourceInformation**, **WNetGetResourceParent**, **WNetOpenEnum**

REMOTE_NAME_INFO

The **REMOTE_NAME_INFO** structure contains path and name information for a network resource. The structure contains a member that points to a Universal Naming Convention (UNC) name string for the resource, and two members that point to additional network connection information strings.

```
typedef struct _REMOTE_NAME_INFO {
    LPTSTR  lpUniversalName;
    LPTSTR  lpConnectionName;
    LPTSTR  lpRemainingPath;
} REMOTE_NAME_INFO;
```

Members

lpUniversalName
Pointer to the null-terminated UNC name string that identifies a network resource.

lpConnectionName
Pointer to a null-terminated string that is the name of a network connection. For more information, see the following Remarks section.

lpRemainingPath
Pointer to a null-terminated name string. For more information, see the following *Remarks* section.

Remarks

The **REMOTE_NAME_INFO** structure contains a pointer to a Universal Naming Convention (UNC) name string. A UNC path identifies a network resource in an unambiguous, computer-independent manner. You can pass the path to processes on other computers, allowing those processes to obtain access to the network resource.

UNC names look like this:

```
\\servername\sharename\path\file
```

In addition, if you pass the value of the **lpConnectionName** member to the **WNetAddConnection2** function, it enables you to connect a local device to a network resource. You can do this by passing the value of **lpConnectionName** in the **lpRemoteName** member of the **NETRESOURCE** structure pointed to by the function's *lpNetResource* parameter.

If you append the string pointed to by the **lpRemainingPath** member of **REMOTE_NAME_INFO** to the local device string, you can pass the resulting string to Win32 functions that require a drive-based path.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Header: Declared in Npapi.h.

See Also

Windows Networking (WNet) Overview, *Windows Networking Structures*, **WNetAddConnection2**, **NETRESOURCE**, **UNIVERSAL_NAME_INFO**

UNIVERSAL_NAME_INFO

The **UNIVERSAL_NAME_INFO** structure contains a pointer to a Universal Naming Convention (UNC) name string for a network resource.

```
typedef struct _UNIVERSAL_NAME_INFO {
    LPTSTR  lpUniversalName;
} UNIVERSAL_NAME_INFO;
```

Members

lpUniversalName

Pointer to the null-terminated UNC name string that identifies a network resource.

Remarks

A UNC path identifies a network resource in an unambiguous, computer-independent manner. You can pass the path to processes on other computers, allowing those processes to obtain access to the network resource.

Universal Naming Convention (UNC) names look like this:

```
\\servername\sharename\path\file
```

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Header: Declared in Npapi.h.

See Also

Windows Networking (WNet) Overview, *Windows Networking Structures*, **WNetGetUniversalName**, **REMOTE_NAME_INFO**

Glossary

This final part of Volume 3 of the Networking Services Library includes a glossary of RPC terms and a comprehensive programming element index, both of which have been designed to make your network programming 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.

A

ADSP *See* AppleTalk Data Stream Protocol.

aliasing In RPC, having two or more pointers to the same data object.

AppleTalk Data Stream Protocol (ADSP) A network protocol for interprocess communication between Apple Macintosh computers and other platforms.

attribute Any keyword of the Interface Definition Language that describes a property of a data type or remote procedure call.

B

bind In RPC, the process through which a logical connection is established between a client and a server to direct a remote procedure call to that server.

binding A relationship between a client and a server established over a specific protocol sequence to a specific host system and endpoint. Also used as a short form of binding handle.

binding handle A data structure that represents the logical connection between a client and a server.

binding vector An RPC data structure that contains a list of binding handles over which a server application can receive remote procedure calls.

C

CDS *See* Cell Directory Service.

Cell Directory Service (CDS) The name-service provider for the Open Software Foundation's Distributed Computing Environment.

client stub MIDL-generated C-language source code. It contains all the functions necessary for the client application to make remote procedure calls using the model of a traditional function call in a standalone application. The client stub is responsible for marshaling input parameters and unmarshaling output parameters. *See also* server stub, proxy stub.

conformant array In RPC, an array whose size is determined at run time by another parameter, structure field, or expression.

connection-oriented Describes a communications protocol or transport that provides a virtual circuit through which data packets are received in the same order as they were transmitted. If the connection between computers fails, the application is notified. TCP and SPX are examples of connection-oriented protocols. *See also* datagram.

connectionless *See* datagram.

context rundown A server notification that results from an unexpected termination of the binding between client and server applications.

D

datagram Describes a communications protocol or transport in which data packets are routed independently. They may follow different routes and arrive in a different order from which they were sent. UDP and IPX are examples of transport layer–datagram protocols. *See also* connection-oriented.

DCE *See* Distributed Computing Environment.

discriminant A variable that specifies the data types that can be stored in a union.

discriminated union (or **variant record**) A union that includes a discriminator as part of the data structure so that the currently valid data type is transmitted along with the union. *See also* encapsulated union, nonencapsulated union.

Distributed Computing Environment (DCE) The Open Software Foundation's specification for a set of integrated services, including remote procedure calls, distributed file systems, and security services. The OSF-DCE RPC standard is the basis for Microsoft RPC.

dynamic endpoint An endpoint (network specific–server address) that is requested and assigned at run time. *See also* well-known endpoint.

dynamic identity tracking Dynamic identity tracking specifies that the RPC run-time library will use the credentials of the calling thread, rather than the binding handle, for authentication each time the client calls a remote procedure. *See also* static identity tracking.

E

embedded pointer A pointer embedded in a parameter that is a data structure such as an array, structure, or union. *See also* top-level pointer.

encoding services MIDL-generated stub routines that provide support for data encoding and decoding (also known as *pickling* or *serialization*). Allow programmers to control buffers containing data to be marshaled and unmarshaled. *See also* type serialization, procedure serialization.

endpoint A network-specific address of a server process for remote procedure calls. The actual name of the endpoint depends on the protocol sequence being used. *See also* dynamic endpoint and well-known endpoint.

endpoint mapper Also, endpoint-mapping service. Part of the RPC subsystem (RPCSS) that allows the run-time library to dynamically assign and resolve endpoints. *See also* endpoint.

encapsulated union A MIDL construct that allows unions to be passed as part of a remote procedure call by embedding the union in a structure in which the discriminant is the first field of the structure, and the union is the second (and final) field of the structure. The IDL keyword **switch** specifies that a union is encapsulated. *See also* nonencapsulated union.

entry point vector (EPV) An array of pointers to functions that implement the operations specified in the interface. Each element in the array corresponds to a function defined in the IDL file. Entry-point vectors allow distributed applications to support more than one implementation of the functions defined in the IDL file.

EPV *See* entry point vector.

F

firewall A control system that prevents unauthorized users from gaining access to a local network that is connected to the Internet.

full pointer In RPC, a pointer that has all the capabilities normally associated with pointers in C/C++. Specifically, a full pointer can be null and can be aliased to another pointer parameter. The **[ptr]** attribute designates a full pointer. *See also* unique pointer and reference pointer.

fullsic A principal name in Full Subject Issuer Chain format as defined in RFC1779.

fully bound handle A binding handle that includes endpoint information. *See also* partially bound handle.

I

idempotent In RPC, describes a remote procedure call that does not change a state and returns the same information each time it is called with the same input parameters.

IDL *See* Interface Definition Language.

input parameter In a remote procedure call, a parameter containing data that the client application is transmitting to the server application. The **in** attribute designates an input parameter.

Interface Definition Language (IDL) The OSF-DCE standard language for specifying the interface for remote procedure calls. *See also* MIDL.

Internet Protocol (IP) A connectionless network-layer communications protocol. *See also* datagram.

Internetwork Packet Exchange (IPX) A Novell NetWare communication protocol that uses datagram sockets to route information packets over local area networks and wide area networks.

intranet A private network that uses Internet products and technologies (for example, web servers), but is not available to external Internet users.

IP *See* Internet Protocol.

IPX *See* Internetwork Packet Exchange

L

local remote procedure call (LRPC) In RPC, describes a remote procedure call to another process on the same computer as the calling process.

Locator The Microsoft Windows NT/ Windows 2000 name-service provider. *See also* name service.

LRPC *See* local remote procedure call.

M

major version number *See* version number.

manager In RPC, a set of server routines that implements the interface operations.

marshaling The process through which operation parameters are packaged into NDR format so that they may be transmitted across process boundaries.

memory leak Allocated memory that is no longer in use, but that has not been freed.

Message Queue Information Service (MQIS) A Microsoft SQL Server database that stores information for Microsoft Message Queuing. *See also* Microsoft Message Queuing (MSMQ).

Microsoft Interface Definition Language (MIDL) The Microsoft implementation and extension of OSF-DCE Interface Definition Language.

Microsoft Message Queuing (MSMQ) A group of Microsoft services that allow users to communicate across networks and systems regardless of the current state of the communicating applications and systems.

MIDL *See* Microsoft Interface Definition Language.

minor version number *See* version number.

MSMQ *See* Microsoft Message Queuing.

MQIS *See* Message Queue Information Service.

msstd A principal name in Microsoft-standard form.

N

name service A service that maps names to objects and stores the name/object pairs in a database. For example, the RPC name service maps a logical name to a binding handle so client applications can refer to that logical name, rather than a protocol sequence and network address. *See also* name service–interface daemon (NSID), Client Directory Service (CDS), Locator.

NCA *See* Network Computing Architecture.

NetBEUI *See* NetBIOS Extended User Interface.

NetBIOS *See* Network Basic Input/Output System.

Network Computing Architecture (NCA) A collection of guidelines for distributed computing. The RPC communication protocols follow these guidelines.

Name Service Independent (NSI) A standard for API functions that allows a distributed application to access RPC name-service database elements through various name-service providers, such as OSF-DCE Cell Directory Service or Microsoft Locator. *See also* name service–interface daemon (NSID).

name service–interface daemon (NSID) A service that provides an interface between Microsoft Locator and the OSF-DCE Cell Directory Service name service databases for RPC name-service functions.

named pipe A connection-oriented protocol, based on Server Message Blocks (SMBs) and NetBIOS, used for communication between a server process and one or more client processes.

NDR *See* Network Data Representation.

NetBIOS Extended User Interface (NetBEUI) The LAN Manager native transport protocol and network device driver. *See also* NetBIOS.

Network Basic Input/Output System (NetBIOS) A software interface between the Microsoft MS-DOS operating system, the I/O bus, and a local area network.

Network Data Representation (NDR) A standard format used during network transmission that is independent of the data-type format on any particular computer architecture. Transmitted data includes information that specifies its NDR format.

network address An address that identifies a server on a network.

nonencapsulated union A discriminated union that is less restrictive than an encapsulated union in that the discriminant and the union are not tightly bound. If the union is a parameter, the discriminant is another parameter; if the union is a structure field, the discriminant is another structure field. The IDL keywords **[switch_is]** and **[switch_type]** identify the discriminant and its type. *See also* encapsulated union.

nonidempotent In RPC, indicates that a remote procedure call cannot be executed more than once because it will return a different value or change a state.

NSI *See* Name Service Independent.

O

Object Description Language (ODL) A subset of MIDL attributes, keywords, statements, and directives used to define type libraries for OLE Automation applications.

ODL *See* Object Description Language.

open array In RPC, an array that is both conformant and varying; that is, both its size and range of transmitted elements are determined at run time by other parameters, structures, or expressions.

Open Software Foundation (OSF) A consortium of companies, formed to define the distributed computing environment (DCE).

orphaned memory Memory on a client previously refererred to by a pointer parameter that has been reset to null by the server.

OSF *See* Open Software Foundation.

output parameter In a remote procedure call, a parameter containing data that the server application is transmitting to the client application. The **[out]** attribute designates an output parameter.

P

partially bound handle A binding handle that does not include endpoint information. *See also* fully bound handle.

pickling *See* serialization.

pipe An IDL-type constructor that supports transmission of an open-ended stream of data between client and server applications.

primary enterprise controller The master Message Queue Information Service (MQIS) database for a network. *See also* Message Queue Information Service (MQIS).

primary site controller A Message Queue Information Service (MQIS) database for a particular site within an enterprise. *See also* Message Queue Information Service (MQIS).

procedure serialization Data serialization that uses a MIDL-generated serialization stub to accomplish the encoding and decoding of one or more types with a single procedure call. Procedure serialization is accomplished by applying the **[encode]** and **[decode]** attributes to a function prototype in the ACF file. *See also* type serialization.

protocol sequence A character string that represents a valid combination of an RPC protocol, a network layer protocol, and a transport layer protocol. For example, the protocol sequence NCACN_IP_TCP describes an NCA connection over an Internet Protocol (IP) with a Transmission Control Protocol (TCP) as transport.

proxy stub MIDL-generated C or C++ language–source code that contains all the functions necessary for a custom OLE interface.

Q

queue An ordered list of tasks to be performed or messages to be transmitted.

queue manager In Microsoft Message Queuing (MSMQ), a service running on a client computer that manages messages for that client.

R

RPC object Server instances or other resources, such as devices, databases, and queues, that are operated on and managed by RPC-server applications. Each object is uniquely identified by one or more object UUIDs.

RPC Subsystem (RPCSS) A Windows NT/ Windows 2000 subsystem that includes a variety of RPC and OLE services, including the endpoint mapper, OLE Service Control Manager (SCM), and the DCOM Object Resolver. Do not confuse this with the RPC-specific memory allocator package, RpcSs.

reference pointer In RPC, the simplest pointer type. A reference pointer always points to valid storage and that storage does not change (although the contents may change). A reference pointer cannot be aliased. The **[ref]** attribute designates a reference pointer. *See also* unique pointer and full pointer.

RPCSS *See* RPC Subsystem.

S

Sequenced Packet Protocol (SPP) Banyan Vines connection-oriented communication protocol for routing information packets over local area networks.

Sequenced Packet Exchange (SPX) A Novell NetWare connection-oriented communication protocol for routing information packets over local area networks and wide area networks.

serialization In RPC, the process of marshaling data to (encoding) and unmarshaling data from (decoding) buffers that you control. This is in contrast to traditional RPC usage, where the stubs and the RPC run time control the marshaling buffers. Also called pickling. *See also* procedure serialization, type serialization.

server stub MIDL-generated C-language source code that contains all the functions necessary for the server application to handle remote requests using local procedure calls. *See also* client stub.

session In RPC, an established relationship between a client application and a server application. *See also* bind, binding handle.

SPP *See* Sequenced Packet Protocol.

SPX *See* Sequenced Packet Exchange.

static callback function A remote procedure that is part of the client side of a distributed application, that a server can call to obtain information from the client. The **[callback]** attribute designates a static callback function.

static identity tracking Static identity tracking specifies that the RPC run-time uses the security credentials in the client's binding handle for all RPC calls. *See also* dynamic identity tracking.

string binding A character string that consists of the object UUID, protocol sequence, network address, endpoint, and endpoint options, all of which can be used to create a binding handle to the specified server.

strong typing Compiler enforcement of strict control over data types. In MIDL and RPC, strong typing is used to ensure that data is interpreted consistently by different computers in a distributed environment.

T

TCP *See* Transmission Control Protocol.

top-level pointer A pointer that is specified as the name of a parameter in a function prototype. *See also* embedded pointer.

Transmission Control Protocol (TCP) A connection-oriented network transport layered on top of the Internet Protocol (IP).

tunneling A TCP/IP protocol for transmitting data from a content-server application to a broadcast router.

type serialization Data serialization that uses MIDL-generated routines to size, encode, and decode objects of a specified type. The client application calls these routines to serialize the data. Type serialization is accomplished by applying the **[encode]** and **[decode]** attributes to a single data type, or to an interface, in the ACF file. *See also* procedure serialization.

U

UDP *See* User Datagram Protocol.

unbind In RPC, to terminate the logical connection between a client and server.

unique pointer In RPC, a pointer that can be null or point to existing data, and whose value can change during a remote procedure call. A unique pointer cannot be aliased. The **[unique]** attribute designates a unique pointer. *See also* full pointer, reference pointer.

Universal Unique Identifier (UUID) Also, Global Unique Identifier (GUID). A 128-bit value used in cross-process communication to identify entities such as client and server interfaces, manager entry-point vectors, and RPC objects. *See also* uuidgen.

unmarshaling The process of unpackaging parameters that have been sent across process boundaries.

User Datagram Protocol (UDP) A network transport that uses connectionless datagram sockets and is layered on top of the Internet Protocol (IP).

UUID *See* Universal Unique Identifer.

uuidgen A utility program, provided with the Win32 SDK, that uses a time value and your machine's network card ID to generate UUIDs that are guaranteed to be unique.

V

varying array In RPC, an array whose range of transmitted elements is determined at run time by another parameter, structure, or expression. *See also* conformant array and open array.

version number In RPC, two numbers, separated by a decimal point, that identify the version of an interface. To be compatible, the major version number (the number to the left of the decimal point) must be the same for both client and server, and the minor version number of the server must be greater than or equal to the minor version number of the client.

W

well-known endpoint An endpoint that does not change. Well-known endpoint information is stored as part of the binding handle, or within the name service–database server entry. *See also* dynamic endpoint.

INDEX

Networking Services Programming Elements – Alphabetical Listing

Locators are arranged by Volume Number followed by Page Number.

A

accept .... Vol. 1, 133
AcceptEx .... Vol. 1, 135
ACTION_HEADER .... Vol. 2, 147
ADAPTER_STATUS .... Vol. 2, 148
AddInterface .... Vol. 5, 266
AddIPAddress .... Vol. 2, 239
ADDRESS_LIST_DESCRIPTOR ..... Vol. 1, 835
AFPROTOCOLS .... Vol. 1, 377
AsnAny .... Vol. 2, 336
AsnCounter64 .... Vol. 2, 338
AsnObjectIdentifier .... Vol. 2, 339
AsnOctetString .... Vol. 2, 339
Authentication-Level Constants .... Vol. 3, 330
Authentication-Service Constants .... Vol. 3, 331
Authorization-Service Constants ...... Vol. 3, 332

B

bind .... Vol. 1, 139
Binding Option Constants .... Vol. 3, 333
Binding Time-out Constants .... Vol. 3, 333
BLOB .... Vol. 1, 378
BlockConvertServicesToStatic .... Vol. 5, 316
BlockDeleteStaticServices .... Vol. 5, 317

C

cbpAdmitRsvpMsg .... Vol. 1, 860
cbpGetRsvpObjects .... Vol. 1, 861
Change Notification Flags .... Vol. 5, 505
ClAddFlowComplete .... Vol. 1, 830
ClDeleteFlowComplete .... Vol. 1, 831
ClModifyFlowComplete .... Vol. 1, 831
ClNotifyHandler .... Vol. 1, 832
CloseServiceEnumerationHandle .... Vol. 5, 318
closesocket .... Vol. 1, 142
connect .... Vol. 1, 145
ConnectClient .... Vol. 5, 268
CONNECTDLGSTRUCT .... Vol. 3, 656
CreateIpForwardEntry .... Vol. 2, 240
CreateIpNetEntry .... Vol. 2, 242
CreateProxyArpEntry .... Vol. 2, 242
CreateServiceEnumerationHandle ... Vol. 5, 319
CreateStaticService .... Vol. 5, 320
CSADDR_INFO .... Vol. 1, 378

D

DCE_C_ERROR_STRING_LEN ...... Vol. 3, 336
DceErrorInqText .... Vol. 3, 349
DeleteInterface .... Vol. 5, 269
DeleteIPAddress .... Vol. 2, 243
DeleteIpForwardEntry .... Vol. 2, 244
DeleteIpNetEntry .... Vol. 2, 245
DeleteProxyArpEntry .... Vol. 2, 245
DeleteStaticService .... Vol. 5, 321
DemandDialRequest .... Vol. 5, 306
DhcpCApiCleanup .... Vol. 2, 74
DhcpCApiInitialize .... Vol. 2, 74
DhcpDeRegisterParamChange .... Vol. 2, 80
DhcpRegisterParamChange .... Vol. 2, 78
DhcpRequestParams .... Vol. 2, 75
DhcpUndoRequestParams .... Vol. 2, 77
DISCDLGSTRUCT .... Vol. 3, 658
DisconnectClient .... Vol. 5, 270
DnsAcquireContextHandle .... Vol. 2, 49
DnsExtractRecordsFromMessage ...... Vol. 2, 50
DnsFreeRecordList .... Vol. 2, 51
DnsModifyRecordsInSet .... Vol. 2, 51
DnsNameCompare .... Vol. 2, 53
DnsQuery .... Vol. 2, 61
DnsQueryConfig .... Vol. 2, 63
DnsRecordCompare .... Vol. 2, 55
DnsRecordCopyEx .... Vol. 2, 55
DnsRecordSetCompare .... Vol. 2, 56
DnsRecordSetCopyEx .... Vol. 2, 57
DnsRecordSetDetach .... Vol. 2, 58
DnsReleaseContextHandle .... Vol. 2, 54
DnsReplaceRecordSet .... Vol. 2, 59
DnsValidateName .... Vol. 2, 64
DnsWriteQuestionToBuffer .... Vol. 2, 67
DoUpdateRoutes .... Vol. 5, 271
DoUpdateServices .... Vol. 5, 271

E

EnumerateGetNextService .... Vol. 5, 322
Enumeration Flags .... Vol. 5, 505
ENUMERATION_BUFFER .... Vol. 1, 835
EnumProtocols .... Vol. 1, 149

F

fd_set .... Vol. 1, 380
FIND_NAME_BUFFER .... Vol. 2, 151
FIND_NAME_HEADER.................... Vol. 2, 152
FIXED_INFO .... Vol. 2, 277
FLOWSPEC .... Vol. 1, 380
FLOWSPEC .... Vol. 1, 791
FlushIpNetTable .... Vol. 2, 246

G

GetAcceptExSockaddrs .... Vol. 1, 153
GetAdapterIndex .... Vol. 2, 247
GetAdaptersInfo .... Vol. 2, 248
GetAddressByName......................... Vol. 1, 154
GetBestInterface .... Vol. 2, 249
GetBestRoute .... Vol. 2, 250
GetEventMessage .... Vol. 5, 272
GetFirstOrderedService .... Vol. 5, 323
GetFriendlyIfIndex .... Vol. 2, 251
GetGlobalInfo .... Vol. 5, 274
gethostbyaddr.................................. Vol. 1, 159
gethostbyname .... Vol. 1, 160
gethostname.................................... Vol. 1, 162
GetIcmpStatistics .... Vol. 2, 252
GetIfEntry .... Vol. 2, 252
GetIfTable... Vol. 2, 253
GetInterfaceInfo............................... Vol. 2, 254
GetInterfaceInfo............................... Vol. 5, 275
GetIpAddrTable .... Vol. 2, 255
GetIpForwardTable .... Vol. 2, 256
GetIpNetTable .... Vol. 2, 257
GetIpStatistics .... Vol. 2, 258
GetMfeStatus .... Vol. 5, 277
GetNameByType............................... Vol. 1, 163
GetNeighbors .... Vol. 5, 278
GetNetworkParams .... Vol. 2, 258
GetNextOrderedService................... Vol. 5, 324
GetNumberOfInterfaces .... Vol. 2, 260
getpeername .... Vol. 1, 164
GetPerAdapterInfo .... Vol. 2, 260
getprotobyname .... Vol. 1, 165
getprotobynumber .... Vol. 1, 167
GetRTTAndHopCount........................ Vol. 2, 262
getservbyname................................. Vol. 1, 168
getservbyport................................... Vol. 1, 169
GetService .... Vol. 1, 171
GetServiceCount .... Vol. 5, 325
getsockname.................................... Vol. 1, 175
getsockopt.. Vol. 1, 176
GetTcpStatistics............................... Vol. 2, 263
GetTcpTable .... Vol. 2, 263
GetTypeByName .... Vol. 1, 185
GetUdpStatistics .... Vol. 2, 264
GetUdpTable..................................... Vol. 2, 265
GetUniDirectionalAdapterInfo........... Vol. 2, 266
GLOBAL_FILTER .... Vol. 5, 262
GUARANTEE..................................... Vol. 1, 413
GUID .... Vol. 1, 848
GUID .... Vol. 3, 295

H

hostent .... Vol. 1, 381
htonl .... Vol. 1, 186
htons .... Vol. 1, 187

I

IEAPProviderConfig.......................... Vol. 4, 426
IEAPProviderConfig::
  RouterInvokeConfigUI .... Vol. 4, 430
IEAPProviderConfig::
  RouterInvokeCredentialsUI............ Vol. 4, 432
IEAPProviderConfig::
  ServerInvokeConfigUI.................... Vol. 4, 429
IEAPProviderConfig::Initialize........... Vol. 4, 426
IEAPProviderConfig::Uninitialize ...... Vol. 4, 428
in_addr .... Vol. 1, 381
inet_addr .... Vol. 1, 187
inet_ntoa .... Vol. 1, 189
Interface Registration Flags.............. Vol. 3, 336
InterfaceStatus.................................. Vol. 5, 280
ioctlsocket .... Vol. 1, 190
IP Info Types for Router
  Information Blocks .... Vol. 5, 183
IP_ADAPTER_BINDING_INFO........ Vol. 5, 149
IP_ADAPTER_INDEX_MAP............. Vol. 2, 278
IP_ADAPTER_INFO......................... Vol. 2, 279
IP_INTERFACE_INFO .... Vol. 2, 280
IP_LOCAL_BINDING........................ Vol. 5, 150
IP_NETWORK .... Vol. 5, 352
IP_NEXT_HOP_ADDRESS.............. Vol. 5, 352
IP_PATTERN.................................... Vol. 1, 842
IP_PER_ADAPTER_INFO .... Vol. 2, 281
IP_SPECIFIC_DATA .... Vol. 5, 353
IP_UNIDIRECTIONAL_ADAPTER_
  ADDRESS...................................... Vol. 2, 282
IPNG_ADDRESS................................ Vol. 2, 88
IpReleaseAddress .... Vol. 2, 267

IpRenewAddress .... Vol. 2, 268
IPX Info Types for Router
Information Blocks .... Vol. 5, 184
IPX_ADAPTER_BINDING_INFO .... Vol. 5, 151
IPX_ADDRESS_DATA .... Vol. 1, 670
IPX_IF_INFO .... Vol. 5, 181
IPX_NETNUM_DATA .... Vol. 1, 672
IPX_NETWORK .... Vol. 5, 355
IPX_NEXT_HOP_ADDRESS .... Vol. 5, 355
IPX_SERVER_ENTRY .... Vol. 5, 327
IPX_SERVICE .... Vol. 5, 328
IPX_SPECIFIC_DATA .... Vol. 5, 356
IPX_SPXCONNSTATUS_DATA .... Vol. 1, 673
IPX_STATIC_SERVICE_INFO .... Vol. 5, 181
IPXWAN_IF_INFO .... Vol. 5, 182
ISensLogon .... Vol. 2, 212
ISensLogon::DisplayLock .... Vol. 2, 216
ISensLogon::DisplayUnLock .... Vol. 2, 217
ISensLogon::Logoff .... Vol. 2, 214
ISensLogon::Logon .... Vol. 2, 213
ISensLogon::StartScreenSaver .... Vol. 2, 218
ISensLogon::StartShell .... Vol. 2, 215
ISensLogon::StopScreenSaver .... Vol. 2, 219
ISensNetwork .... Vol. 2, 220
ISensNetwork::
ConnectionMadeNoQOCInfo .... Vol. 2, 222
ISensNetwork::
DestinationReachable .... Vol. 2, 225
ISensNetwork::
DestinationReachable
NoQOCInfo .... Vol. 2, 226
ISensNetwork::ConnectionLost .... Vol. 2, 223
ISensNetwork::ConnectionMade .... Vol. 2, 221
ISensOnNow .... Vol. 2, 228
ISensOnNow::BatteryLow .... Vol. 2, 231
ISensOnNow::OnACPower .... Vol. 2, 229
ISensOnNow::OnBatteryPower .... Vol. 2, 230
IsService .... Vol. 5, 326
ISyncMgrEnumItems .... Vol. 2, 166
ISyncMgrRegister .... Vol. 2, 193
ISyncMgrRegister::
GetHandlerRegistrationInfo .... Vol. 2, 195
ISyncMgrRegister::
RegisterSyncMgrHandler .... Vol. 2, 194
ISyncMgrRegister::
UnregisterSyncMgrHandler .... Vol. 2, 194
ISyncMgrSynchronize .... Vol. 2, 168
ISyncMgrSynchronize::
EnumSyncMgrItems .... Vol. 2, 171
ISyncMgrSynchronize::
GetHandlerInfo .... Vol. 2, 170
ISyncMgrSynchronize::
GetItemObject .... Vol. 2, 172
ISyncMgrSynchronize::
PrepareForSync .... Vol. 2, 175
ISyncMgrSynchronize::
SetItemStatus .... Vol. 2, 178
ISyncMgrSynchronize::
SetProgressCallback .... Vol. 2, 174
ISyncMgrSynchronize::
ShowProperties .... Vol. 2, 173
ISyncMgrSynchronize::
Synchronize .... Vol. 2, 176
ISyncMgrSynchronize::Initialize .... Vol. 2, 169
ISyncMgrSynchronize::ShowError .... Vol. 2, 179
ISyncMgrSynchronizeCallback .... Vol. 2, 180
ISyncMgrSynchronizeCallback::
DeleteLogError .... Vol. 2, 189
ISyncMgrSynchronizeCallback::
EnableModeless .... Vol. 2, 186
ISyncMgrSynchronizeCallback::
EstablishConnection .... Vol. 2, 190
ISyncMgrSynchronizeCallback::
LogError .... Vol. 2, 187
ISyncMgrSynchronizeCallback::
PrepareForSyncCompleted .... Vol. 2, 184
ISyncMgrSynchronizeCallback::
Progress .... Vol. 2, 182
ISyncMgrSynchronizeCallback::
ShowErrorCompleted .... Vol. 2, 188
ISyncMgrSynchronizeCallback::
ShowPropertiesCompleted .... Vol. 2, 183
ISyncMgrSynchronizeCallback::
SynchronizeCompleted .... Vol. 2, 185
ISyncMgrSynchronizeInvoke .... Vol. 2, 191
ISyncMgrSynchronizeInvoke::
UpdateAll .... Vol. 2, 192
ISyncMgrSynchronizeInvoke::
UpdateItems .... Vol. 2, 191

L

LANA_ENUM .... Vol. 2, 152
linger .... Vol. 1, 382
listen .... Vol. 1, 192
LPM_AdmitRsvpMsg .... Vol. 1, 863
LPM_CommitResv .... Vol. 1, 866
LPM_Deinitialize .... Vol. 1, 867
LPM_DeleteState .... Vol. 1, 868
LPM_GetRsvpObjects .... Vol. 1, 870
LPM_Initialize .... Vol. 1, 872
Lpm_IpAddressTable .... Vol. 1, 874
LPMIPTABLE .... Vol. 1, 875

M

MACYIELDCALLBACK .... Vol. 3, 575
MCAST_CLIENT_UID .... Vol. 2, 89
MCAST_LEASE_REQUEST .... Vol. 2, 90

MCAST_LEASE_RESPONSE .... Vol. 2, 92
MCAST_SCOPE_CTX .... Vol. 2, 89
MCAST_SCOPE_ENTRY .... Vol. 2, 90
McastApiCleanup .... Vol. 2, 82
McastApiStartup .... Vol. 2, 82
McastEnumerateScopes .... Vol. 2, 83
McastGenUID .... Vol. 2, 85
McastReleaseAddress .... Vol. 2, 87
McastRenewAddress .... Vol. 2, 86
McastRequestAddress .... Vol. 2, 85
MesBufferHandleReset .... Vol. 3, 350
MesDecodeBufferHandleCreate ...... Vol. 3, 351
MesDecodeIncrementalHandle
  Create .... Vol. 3, 353
MesEncodeDynBufferHandle
  Create .... Vol. 3, 354
MesEncodeFixedBufferHandle
  Create .... Vol. 3, 355
MesEncodeIncrementalHandle
  Create .... Vol. 3, 356
MesHandleFree .... Vol. 3, 357
MesIncrementalHandleReset .... Vol. 3, 358
MesInqProcEncodingId .... Vol. 3, 359
MESSAGE .... Vol. 5, 297
MGM_ENUM_TYPES .... Vol. 5, 564
MGM_IF_ENTRY .... Vol. 5, 561
MgmAddGroupMembershipEntry ..... Vol. 5, 524
MgmDeleteGroupMembership
  Entry .... Vol. 5, 526
MgmDeRegisterMProtocol .... Vol. 5, 527
MgmGetFirstMfe .... Vol. 5, 528
MgmGetFirstMfeStats .... Vol. 5, 530
MgmGetMfe .... Vol. 5, 531
MgmGetMfeStats .... Vol. 5, 533
MgmGetNextMfe .... Vol. 5, 534
MgmGetNextMfeStats .... Vol. 5, 536
MgmGetProtocolOnInterface .... Vol. 5, 537
MgmGroupEnumerationEnd .... Vol. 5, 539
MgmGroupEnumerationGetNext ...... Vol. 5, 539
MgmGroupEnumerationStart .... Vol. 5, 541
MgmRegisterMProtocol .... Vol. 5, 542
MgmReleaseInterfaceOwnership ..... Vol. 5, 543
MgmSetMfe .... Vol. 5, 545
MgmTakeInterfaceOwnership .... Vol. 5, 545
MIB_BEST_IF .... Vol. 5, 202
MIB_ICMP .... Vol. 5, 203
MIB_IFNUMBER .... Vol. 5, 203
MIB_IFROW .... Vol. 5, 204
MIB_IFSTATUS .... Vol. 5, 206
MIB_IFTABLE .... Vol. 5, 207
MIB_IPADDRROW .... Vol. 5, 207
MIB_IPADDRTABLE .... Vol. 5, 208
MIB_IPFORWARDNUMBER .... Vol. 5, 209
MIB_IPFORWARDROW .... Vol. 5, 210
MIB_IPFORWARDTABLE .... Vol. 5, 212
MIB_IPMCAST_GLOBAL .... Vol. 5, 212
MIB_IPMCAST_IF_ENTRY .... Vol. 5, 213
MIB_IPMCAST_IF_TABLE .... Vol. 5, 214
MIB_IPMCAST_MFE .... Vol. 5, 214
MIB_IPMCAST_MFE_STATS .... Vol. 5, 216
MIB_IPMCAST_OIF .... Vol. 5, 218
MIB_IPMCAST_OIF_STATS .... Vol. 5, 219
MIB_IPNETROW .... Vol. 5, 220
MIB_IPNETTABLE .... Vol. 5, 221
MIB_IPSTATS .... Vol. 5, 222
MIB_MFE_STATS_TABLE .... Vol. 5, 224
MIB_MFE_TABLE .... Vol. 5, 224
MIB_OPAQUE_INFO .... Vol. 5, 225
MIB_OPAQUE_QUERY .... Vol. 5, 225
MIB_PROXYARP .... Vol. 5, 226
MIB_TCPROW .... Vol. 5, 227
MIB_TCPSTATS .... Vol. 5, 228
MIB_TCPTABLE .... Vol. 5, 230
MIB_UDPROW .... Vol. 5, 230
MIB_UDPSTATS .... Vol. 5, 231
MIB_UDPTABLE .... Vol. 5, 232
MibCreate .... Vol. 5, 281
MibDelete .... Vol. 5, 282
MibEntryCreate .... Vol. 5, 307
MibEntryDelete .... Vol. 5, 308
MibEntryGet .... Vol. 5, 309
MibEntryGetFirst .... Vol. 5, 311
MibEntryGetNext .... Vol. 5, 312
MibEntrySet .... Vol. 5, 313
MibGet .... Vol. 5, 283
MibGetFirst .... Vol. 5, 284
MibGetNext .... Vol. 5, 285
MibGetTrapInfo .... Vol. 5, 286
MIBICMPINFO .... Vol. 5, 232
MIBICMPSTATS .... Vol. 5, 233
MibSet .... Vol. 5, 287
MibSetTrapInfo .... Vol. 5, 288
MPR_CREDENTIALSEX_0 .... Vol. 5, 152
MPR_IFTRANSPORT_0 .... Vol. 5, 152
MPR_INTERFACE_0 .... Vol. 5, 153
MPR_INTERFACE_1 .... Vol. 5, 154
MPR_INTERFACE_2 .... Vol. 5, 156
MPR_ROUTING_
  CHARACTERISTICS .... Vol. 5, 297
MPR_SERVER_0 .... Vol. 5, 166
MPR_SERVICE_
  CHARACTERISTICS .... Vol. 5, 301
MPR_TRANSPORT_0 .... Vol. 5, 167
MprAdminAcceptNewConnection ..... Vol. 4, 341
MprAdminAcceptNewConnection2 ... Vol. 4, 342
MprAdminAcceptNewLink .... Vol. 4, 343
MprAdminBufferFree .... Vol. 5, 70
MprAdminConnectionClearStats ...... Vol. 4, 329
MprAdminConnectionEnum .... Vol. 4, 330
MprAdminConnectionGetInfo .... Vol. 4, 332

MprAdminConnectionHangup
  Notification....................................Vol. 4, 344
MprAdminConnectionHangup
  Notification2..................................Vol. 4, 345
MprAdminDeregisterConnection
  Notification......................................Vol. 5, 71
MprAdminGetErrorString....................Vol. 5, 72
MprAdminGetIpAddressForUser......Vol. 4, 346
MprAdminGetPDCServer.................Vol. 4, 349
MprAdminInterfaceConnect ....Vol. 5, 73
MprAdminInterfaceCreate ....Vol. 5, 75
MprAdminInterfaceDelete...................Vol. 5, 76
MprAdminInterfaceDisconnect...........Vol. 5, 77
MprAdminInterfaceEnum....................Vol. 5, 78
MprAdminInterfaceGetCredentials.....Vol. 5, 80
MprAdminInterfaceGetCredentials
  Ex ....Vol. 5, 82
MprAdminInterfaceGetHandle............Vol. 5, 83
MprAdminInterfaceGetInfo ....Vol. 5, 84
MprAdminInterfaceQueryUpdate
  Result ....Vol. 5, 86
MprAdminInterfaceSetCredentials .....Vol. 5, 87
MprAdminInterfaceSetCredentials
  Ex ....Vol. 5, 89
MprAdminInterfaceSetInfo..................Vol. 5, 90
MprAdminInterfaceTransport
  GetInfo..Vol. 5, 93
MprAdminInterfaceTransport
  Remove ....Vol. 5, 94
MprAdminInterfaceTransport
  SetInfo ....Vol. 5, 95
MprAdminInterfaceTransportAdd.......Vol. 5, 91
MprAdminInterfaceUpdate
  PhonebookInfo ....Vol. 5, 97
MprAdminInterfaceUpdateRoutes......Vol. 5, 98
MprAdminIsServiceRunning.............Vol. 5, 100
MprAdminLinkHangupNotification....Vol. 4, 347
MprAdminMIBBufferFree.................Vol. 5, 188
MprAdminMIBEntryCreate ....Vol. 5, 188
MprAdminMIBEntryDelete................Vol. 5, 190
MprAdminMIBEntryGet ....Vol. 5, 191
MprAdminMIBEntryGetFirst ....Vol. 5, 193
MprAdminMIBEntryGetNext.............Vol. 5, 195
MprAdminMIBEntrySet.....................Vol. 5, 196
MprAdminMIBGetTrapInfo ....Vol. 5, 198
MprAdminMIBServerConnect ....Vol. 5, 199
MprAdminMIBServerDisconnect......Vol. 5, 200
MprAdminMIBSetTrapInfo................Vol. 5, 200
MprAdminPortClearStats..................Vol. 4, 334
MprAdminPortDisconnect.................Vol. 4, 335
MprAdminPortEnum.........................Vol. 4, 336
MprAdminPortGetInfo.......................Vol. 4, 338
MprAdminPortReset.........................Vol. 4, 339
MprAdminRegisterConnection
  Notification....................................Vol. 5, 100
MprAdminReleaseIpAddress............Vol. 4, 348
MprAdminSendUserMessage...........Vol. 4, 351
MprAdminServerConnect ....Vol. 5, 102
MprAdminServerDisconnect.............Vol. 5, 102
MprAdminServerGetInfo...................Vol. 5, 103
MprAdminTransportCreate ....Vol. 5, 104
MprAdminTransportGetInfo ....Vol. 5, 106
MprAdminTransportSetInfo...............Vol. 5, 108
MprAdminUserGetInfo......................Vol. 4, 352
MprAdminUserSetInfo ....Vol. 4, 353
MprConfigBufferFree ....Vol. 5, 110
MprConfigGetFriendlyName.............Vol. 5, 110
MprConfigGetGuidName ....Vol. 5, 112
MprConfigInterfaceCreate ....Vol. 5, 114
MprConfigInterfaceDelete.................Vol. 5, 115
MprConfigInterfaceEnum..................Vol. 5, 116
MprConfigInterfaceGetHandle..........Vol. 5, 118
MprConfigInterfaceGetInfo ....Vol. 5, 119
MprConfigInterfaceSetInfo................Vol. 5, 121
MprConfigInterfaceTransport
  Enum..Vol. 5, 124
MprConfigInterfaceTransport
  GetHandle.....................................Vol. 5, 126
MprConfigInterfaceTransport
  GetInfo ....Vol. 5, 128
MprConfigInterfaceTransport
  Remove...Vol. 5, 130
MprConfigInterfaceTransport
  SetInfo..Vol. 5, 131
MprConfigInterfaceTransportAdd .....Vol. 5, 122
MprConfigServerBackup...................Vol. 5, 133
MprConfigServerConnect.................Vol. 5, 134
MprConfigServerDisconnect............Vol. 5, 135
MprConfigServerGetInfo...................Vol. 5, 136
MprConfigServerInstall ....Vol. 5, 113
MprConfigServerRestore..................Vol. 5, 137
MprConfigTransportCreate...............Vol. 5, 138
MprConfigTransportDelete ....Vol. 5, 140
MprConfigTransportEnum ....Vol. 5, 141
MprConfigTransportGetHandle ....Vol. 5, 143
MprConfigTransportGetInfo..............Vol. 5, 144
MprConfigTransportSetInfo ....Vol. 5, 147
MprInfoBlockAdd ....Vol. 5, 170
MprInfoBlockFind.............................Vol. 5, 172
MprInfoBlockQuerySize....................Vol. 5, 173
MprInfoBlockRemove ....Vol. 5, 174
MprInfoBlockSet...............................Vol. 5, 175
MprInfoCreate..................................Vol. 5, 176
MprInfoDelete ....Vol. 5, 177
MprInfoDuplicate..............................Vol. 5, 178
MprInfoRemoveAll ....Vol. 5, 179
MultinetGetConnection
  Performance ....Vol. 3, 609

N

NAME_BUFFER........Vol. 2, 153
NCB........Vol. 2, 154
NDR_USER_MARSHAL_INFO........Vol. 3, 296
NdrGetUserMarshalInfo........Vol. 3, 360
Netbios........Vol. 2, 145
NETCONNECTINFOSTRUCT........Vol. 3, 659
NETINFOSTRUCT........Vol. 3, 661
NETRESOURCE........Vol. 3, 663
Next Hop Flags........Vol. 5, 503
NotifyAddrChange........Vol. 2, 268
NotifyRouteChange........Vol. 2, 269
NS_SERVICE_INFO........Vol. 1, 383
NSPCleanup........Vol. 1, 497
NSPGetServiceClassInfo........Vol. 1, 498
NSPInstallServiceClass........Vol. 1, 499
NSPLookupServiceBegin........Vol. 1, 500
NSPLookupServiceEnd........Vol. 1, 504
NSPLookupServiceNext........Vol. 1, 505
NSPRemoveServiceClass........Vol. 1, 509
NSPSetService........Vol. 1, 510
NSPStartup........Vol. 1, 513
ntohl........Vol. 1, 194
ntohs........Vol. 1, 195

O

ORASADFunc........Vol. 4, 103

P

PALLOCMEM........Vol. 1, 876
PF_FILTER_DESCRIPTOR........Vol. 5, 256
PF_FILTER_STATS........Vol. 5, 257
PF_INTERFACE_STATS........Vol. 5, 258
PF_LATEBIND_INFO........Vol. 5, 260
PfAddFiltersToInterface........Vol. 5, 239
PfAddGlobalFilterToInterface........Vol. 5, 241
PFADDRESSTYPE........Vol. 5, 262
PfBindInterfaceToIndex........Vol. 5, 241
PfBindInterfaceToIPAddress........Vol. 5, 242
PfCreateInterface........Vol. 5, 243
PfDeleteInterface........Vol. 5, 245
PfDeleteLog........Vol. 5, 246
PFFORWARD_ACTION........Vol. 5, 263
PFFRAMETYPE........Vol. 5, 264
PfGetInterfaceStatistics........Vol. 5, 246
PFLOGFRAME........Vol. 5, 260
PfMakeLog........Vol. 5, 248
PfRebindFilters........Vol. 5, 249
PFREEMEM........Vol. 1, 876
PfRemoveFilterHandles........Vol. 5, 250
PfRemoveFiltersFromInterface........Vol. 5, 250
PfRemoveGlobalFilterFrom
nterface........Vol. 5, 252
PfSetLogBuffer........Vol. 5, 252
PfTestPacket........Vol. 5, 253
PfUnBindInterface........Vol. 5, 255
PMGM_CREATION_ALERT_
CALLBACK........Vol. 5, 547
PMGM_DISABLE_IGMP_
CALLBACK........Vol. 5, 549
PMGM_ENABLE_IGMP_
CALLBACK........Vol. 5, 549
PMGM_JOIN_ALERT_
CALLBACK........Vol. 5, 550
PMGM_LOCAL_JOIN_
CALLBACK........Vol. 5, 552
PMGM_LOCAL_LEAVE_
CALLBACK........Vol. 5, 554
PMGM_PRUNE_ALERT_
CALLBACK........Vol. 5, 555
PMGM_RPF_CALLBACK........Vol. 5, 558
PMGM_WRONG_IF_CALLBACK........Vol. 5, 560
Portability Macros........Vol. 3, 583
PPP_ATCP_INFO........Vol. 4, 355
PPP_CCP_INFO........Vol. 4, 356
PPP_EAP_ACTION........Vol. 4, 414
PPP_EAP_INFO........Vol. 4, 403
PPP_EAP_INPUT........Vol. 4, 404
PPP_EAP_OUTPUT........Vol. 4, 409
PPP_EAP_PACKET........Vol. 4, 412
PPP_INFO........Vol. 4, 358
PPP_INFO_2........Vol. 4, 358
PPP_IPCP_INFO........Vol. 4, 359
PPP_IPCP_INFO2........Vol. 4, 360
PPP_IPXCP_INFO........Vol. 4, 361
PPP_LCP_INFO........Vol. 4, 362
PPP_NBFCP_INFO........Vol. 4, 364
Protection Level Constants........Vol. 3, 337
Protocol Identifiers........Vol. 5, 235
Protocol Sequence Constants........Vol. 3, 338
PROTOCOL_INFO........Vol. 1, 384
PROTOCOL_SPECIFIC_DATA........Vol. 5, 357
protoent........Vol. 1, 387
PROTSEQ........Vol. 3, 317
PS_ADAPTER_STATS........Vol. 1, 851
PS_COMPONENT_STATS........Vol. 1, 850
PS_CONFORMER_STATS........Vol. 1, 853
PS_DRRSEQ_STATS........Vol. 1, 854
PS_FLOW_STATS........Vol. 1, 852
PS_SHAPER_STATS........Vol. 1, 853

Q

QOCINFO........Vol. 2, 209
QOS........Vol. 1, 388
QOS........Vol. 1, 797

QOS_DIFFSERV_RULE .... Vol. 1, 844
QOS_OBJECT_DESTADDR .... Vol. 1, 800
QOS_OBJECT_DIFFSERV .... Vol. 1, 858
QOS_OBJECT_DS_CLASS .... Vol. 1, 857
QOS_OBJECT_HDR .... Vol. 1, 799
QOS_OBJECT_SD_MODE .... Vol. 1, 801
QOS_OBJECT_SHAPING_RATE ... Vol. 1, 802
QOS_OBJECT_TRAFFIC_CLASS .. Vol. 1, 856
QueryPower..................................... Vol. 5, 289

R

RADIUS_ACTION .... Vol. 2, 112
RADIUS_ATTRIBUTE...................... Vol. 2, 110
RADIUS_ATTRIBUTE_TYPE .... Vol. 2, 112
RADIUS_AUTHENTICATION_
PROVIDER.................................. Vol. 2, 120
RADIUS_DATA_TYPE...................... Vol. 2, 121
RadiusExtensionInit.......................... Vol. 2, 107
RadiusExtensionProcess .... Vol. 2, 108
RadiusExtensionProcessEx .... Vol. 2, 109
RadiusExtensionTerm....................... Vol. 2, 107
RAS_AUTH_ATTRIBUTE .... Vol. 4, 413
RAS_AUTH_ATTRIBUTE_TYPE..... Vol. 4, 415
RAS_CONNECTION_0................... Vol. 4, 365
RAS_CONNECTION_1................... Vol. 4, 367
RAS_CONNECTION_2................... Vol. 4, 368
RAS_HARDWARE_CONDITION..... Vol. 4, 375
RAS_PARAMETERS .... Vol. 4, 293
RAS_PARAMS_FORMAT .... Vol. 4, 314
RAS_PARAMS_VALUE.................. Vol. 4, 312
RAS_PORT_0 .... Vol. 4, 294
RAS_PORT_0 .... Vol. 4, 369
RAS_PORT_1 .... Vol. 4, 297
RAS_PORT_1 .... Vol. 4, 370
RAS_PORT_CONDITION................ Vol. 4, 376
RAS_PORT_STATISTICS .... Vol. 4, 298
RAS_PPP_ATCP_RESULT............. Vol. 4, 302
RAS_PPP_IPCP_RESULT .... Vol. 4, 303
RAS_PPP_IPXCP_RESULT............ Vol. 4, 303
RAS_PPP_NBFCP_RESULT .... Vol. 4, 304
RAS_PPP_PROJECTION_
RESULT .... Vol. 4, 305
RAS_SECURITY_INFO .... Vol. 4, 306
RAS_SERVER_0 .... Vol. 4, 307
RAS_STATS .... Vol. 4, 308
RAS_USER_0 .... Vol. 4, 310
RAS_USER_0 .... Vol. 4, 372
RAS_USER_1 .... Vol. 4, 373
RASADFunc .... Vol. 4, 105
RasAdminAcceptNewConnection .... Vol. 4, 277
RasAdminConnectionHangup
Notification.................................... Vol. 4, 279
RasAdminFreeBuffer........................ Vol. 4, 265
RasAdminGetErrorString.................. Vol. 4, 266
RasAdminGetIpAddressForUser ...... Vol. 4, 281
RasAdminGetUserAccountServer .... Vol. 4, 267
RasAdminPortClearStatistics............ Vol. 4, 269
RasAdminPortDisconnect................. Vol. 4, 270
RasAdminPortEnum .... Vol. 4, 271
RasAdminPortGetInfo....................... Vol. 4, 272
RasAdminReleaseIpAddress............ Vol. 4, 282
RasAdminServerGetInfo................... Vol. 4, 274
RasAdminUserGetInfo...................... Vol. 4, 275
RasAdminUserSetInfo .... Vol. 4, 276
RASADPARAMS .... Vol. 4, 205
RASAMB.. Vol. 4, 206
RASAUTODIALENTRY .... Vol. 4, 207
RasClearConnectionStatistics .... Vol. 4, 107
RasClearLinkStatistics...................... Vol. 4, 107
RASCONN .... Vol. 4, 208
RasConnectionNotification .... Vol. 4, 109
RASCONNSTATE .... Vol. 4, 258
RASCONNSTATUS.......................... Vol. 4, 210
RasCreatePhonebookEntry .... Vol. 4, 110
RASCREDENTIALS .... Vol. 4, 211
RASCTRYINFO .... Vol. 4, 212
RasCustomDeleteEntryNotify .... Vol. 4, 111
RasCustomDial .... Vol. 4, 112
RasCustomDialDlg .... Vol. 4, 114
RasCustomEntryDlg .... Vol. 4, 116
RasCustomHangUp.......................... Vol. 4, 118
RasCustomScriptExecute................. Vol. 4, 197
RasDeleteEntry................................ Vol. 4, 119
RASDEVINFO.................................. Vol. 4, 214
RasDial .... Vol. 4, 120
RasDialDlg....................................... Vol. 4, 123
RASDIALDLG .... Vol. 4, 215
RASDIALEXTENSIONS .... Vol. 4, 217
RasDialFunc .... Vol. 4, 125
RasDialFunc1 .... Vol. 4, 127
RasDialFunc2 .... Vol. 4, 129
RASDIALPARAMS .... Vol. 4, 219
RasEapBegin .... Vol. 4, 389
RasEapEnd...................................... Vol. 4, 391
RasEapFreeMemory......................... Vol. 4, 391
RasEapGetIdentity........................... Vol. 4, 392
RasEapGetInfo .... Vol. 4, 395
RASEAPINFO................................... Vol. 4, 222
RasEapInitialize .... Vol. 4, 396
RasEapInvokeConfigUI .... Vol. 4, 397
RasEapInvokeInteractiveUI .... Vol. 4, 399
RasEapMakeMessage...................... Vol. 4, 401
RASEAPUSERIDENTITY................. Vol. 4, 222
RasEditPhonebookEntry................... Vol. 4, 131
RASENTRY .... Vol. 4, 223
RasEntryDlg..................................... Vol. 4, 133
RASENTRYDLG .... Vol. 4, 236
RASENTRYNAME............................ Vol. 4, 238
RasEnumAutodialAddresses .... Vol. 4, 135

RasEnumConnections....................Vol. 4, 136
RasEnumDevices....................Vol. 4, 137
RasEnumEntries....................Vol. 4, 139
RasFreeBuffer....................Vol. 4, 199
RasFreeEapUserIdentity....................Vol. 4, 142
RasGetAutodialAddress....................Vol. 4, 143
RasGetAutodialEnable....................Vol. 4, 144
RasGetAutodialParam....................Vol. 4, 145
RasGetBuffer....................Vol. 4, 198
RasGetConnectionStatistics....................Vol. 4, 147
RasGetConnectStatus....................Vol. 4, 148
RasGetCountryInfo....................Vol. 4, 149
RasGetCredentials....................Vol. 4, 151
RasGetCustomAuthData....................Vol. 4, 153
RasGetEapUserData....................Vol. 4, 155
RasGetEapUserIdentity....................Vol. 4, 156
RasGetEntryDialParams....................Vol. 4, 158
RasGetEntryProperties....................Vol. 4, 160
RasGetErrorString....................Vol. 4, 162
RasGetLinkStatistics....................Vol. 4, 164
RasGetProjectionInfo....................Vol. 4, 165
RasGetSubEntryHandle....................Vol. 4, 167
RasGetSubEntryProperties....................Vol. 4, 168
RasHangUp....................Vol. 4, 170
RasInvokeEapUI....................Vol. 4, 171
RASIPADDR....................Vol. 4, 239
RasMonitorDlg....................Vol. 4, 173
RASMONITORDLG....................Vol. 4, 240
RASNOUSER....................Vol. 4, 241
RASPBDLG....................Vol. 4, 243
RasPBDlgFunc....................Vol. 4, 174
RasPhonebookDlg....................Vol. 4, 176
RASPPPCCP....................Vol. 4, 245
RASPPPIP....................Vol. 4, 247
RASPPPIPX....................Vol. 4, 251
RASPPPLCP....................Vol. 4, 248
RASPPPNBF....................Vol. 4, 252
RASPROJECTION....................Vol. 4, 263
RasReceiveBuffer....................Vol. 4, 201
RasRenameEntry....................Vol. 4, 178
RasRetrieveBuffer....................Vol. 4, 203
RasSecurityDialogBegin....................Vol. 4, 284
RasSecurityDialogComplete....................Vol. 4, 286
RasSecurityDialogEnd....................Vol. 4, 287
RasSecurityDialogGetInfo....................Vol. 4, 288
RasSecurityDialogReceive....................Vol. 4, 289
RasSecurityDialogSend....................Vol. 4, 291
RasSendBuffer....................Vol. 4, 200
RasSetAutodialAddress....................Vol. 4, 179
RasSetAutodialEnable....................Vol. 4, 181
RasSetAutodialParam....................Vol. 4, 182
RasSetCredentials....................Vol. 4, 184
RasSetCustomAuthData....................Vol. 4, 186
RasSetEapUserData....................Vol. 4, 187
RasSetEntryDialParams....................Vol. 4, 189
RasSetEntryProperties....................Vol. 4, 191
RasSetSubEntryProperties....................Vol. 4, 193
RASSLIP....................Vol. 4, 253
RASSUBENTRY....................Vol. 4, 254
RasValidateEntryName....................Vol. 4, 195
recv....................Vol. 1, 196
recvfrom....................Vol. 1, 199
RegisterProtocol....................Vol. 5, 290
REMOTE_NAME_INFO....................Vol. 3, 665
Route Flags....................Vol. 5, 501
ROUTER_CONNECTION_STATE...Vol. 5, 167
ROUTER_INTERFACE_TYPE....................Vol. 5, 168
Routing Table Query Flags....................Vol. 5, 504
ROUTING_PROTOCOL_CONFIG...Vol. 5, 562
RPC_ASYNC_EVENT....................Vol. 3, 315
RPC_ASYNC_STATE....................Vol. 3, 298
RPC_AUTH_IDENTITY_HANDLE...Vol. 3, 318
RPC_AUTH_KEY_RETRIEVAL_
FN....................Vol. 3, 576
RPC_AUTHZ_HANDLE....................Vol. 3, 319
RPC_BINDING_HANDLE....................Vol. 3, 319
RPC_BINDING_VECTOR....................Vol. 3, 301
RPC_CLIENT_INTERFACE....................Vol. 3, 302
RPC_DISPATCH_TABLE....................Vol. 3, 302
RPC_EP_INQ_HANDLE....................Vol. 3, 320
RPC_IF_CALLBACK_FN....................Vol. 3, 577
RPC_IF_HANDLE....................Vol. 3, 321
RPC_IF_ID....................Vol. 3, 303
RPC_IF_ID_VECTOR....................Vol. 3, 304
RPC_MGMT_AUTHORIZATION_
FN....................Vol. 3, 577
RPC_MGR_EPV....................Vol. 3, 321
RPC_NOTIFICATION_TYPES....................Vol. 3, 315
RPC_NS_HANDLE....................Vol. 3, 322
RPC_OBJECT_INQ_FN....................Vol. 3, 579
RPC_POLICY....................Vol. 3, 304
RPC_PROTSEQ_VECTOR....................Vol. 3, 308
RPC_SECURITY_QOS....................Vol. 3, 308
RPC_STATS_VECTOR....................Vol. 3, 310
RPC_STATUS....................Vol. 3, 323
RpcAbnormalTermination....................Vol. 3, 362
RpcAsyncAbortCall....................Vol. 3, 362
RpcAsyncCancelCall....................Vol. 3, 363
RpcAsyncCompleteCall....................Vol. 3, 365
RpcAsyncGetCallHandle....................Vol. 3, 585
RpcAsyncGetCallStatus....................Vol. 3, 366
RpcAsyncInitializeHandle....................Vol. 3, 367
RpcAsyncRegisterInfo....................Vol. 3, 368
RpcBindingCopy....................Vol. 3, 369
RpcBindingFree....................Vol. 3, 370
RpcBindingFromStringBinding....................Vol. 3, 372
RpcBindingInqAuthClient....................Vol. 3, 373
RpcBindingInqAuthClientEx....................Vol. 3, 375
RpcBindingInqAuthInfo....................Vol. 3, 377
RpcBindingInqAuthInfoEx....................Vol. 3, 380

RpcBindingInqObject........................Vol. 3, 382
RpcBindingInqOption........................Vol. 3, 383
RpcBindingReset............................Vol. 3, 384
RpcBindingServerFromClient.................Vol. 3, 385
RpcBindingSetAuthInfo......................Vol. 3, 387
RpcBindingSetAuthInfoEx....................Vol. 3, 389
RpcBindingSetObject........................Vol. 3, 391
RpcBindingSetOption........................Vol. 3, 392
RpcBindingToStringBinding..................Vol. 3, 394
RpcBindingVectorFree.......................Vol. 3, 395
RpcCancelThread............................Vol. 3, 396
RpcCancelThreadEx..........................Vol. 3, 397
RpcCertGeneratePrincipalName...............Vol. 3, 398
RpcEndExcept...............................Vol. 3, 586
RpcEndFinally..............................Vol. 3, 586
RpcEpRegister..............................Vol. 3, 399
RpcEpRegisterNoReplace.....................Vol. 3, 401
RpcEpResolveBinding........................Vol. 3, 404
RpcEpUnregister............................Vol. 3, 405
RpcExcept..................................Vol. 3, 587
RpcExceptionCode...........................Vol. 3, 407
RpcFinally.................................Vol. 3, 588
RpcIfIdVectorFree..........................Vol. 3, 407
RpcIfInqId.................................Vol. 3, 408
RpcImpersonateClient.......................Vol. 3, 409
RpcMacSetYieldInfo.........................Vol. 3, 410
RpcMgmtEnableIdleCleanup...................Vol. 3, 411
RpcMgmtEpEltInqBegin.......................Vol. 3, 412
RpcMgmtEpEltInqDone........................Vol. 3, 415
RpcMgmtEpEltInqNext........................Vol. 3, 416
RpcMgmtEpUnregister........................Vol. 3, 417
RpcMgmtInqComTimeout.......................Vol. 3, 418
RpcMgmtInqDefaultProtectLevel..............Vol. 3, 419
RpcMgmtInqIfIds............................Vol. 3, 421
RpcMgmtInqServerPrincName..................Vol. 3, 422
RpcMgmtInqStats............................Vol. 3, 423
RpcMgmtIsServerListening...................Vol. 3, 425
RpcMgmtSetAuthorizationFn..................Vol. 3, 426
RpcMgmtSetCancelTimeout....................Vol. 3, 427
RpcMgmtSetComTimeout.......................Vol. 3, 428
RpcMgmtSetServerStackSize..................Vol. 3, 429
RpcMgmtStatsVectorFree.....................Vol. 3, 430
RpcMgmtStopServerListening.................Vol. 3, 431
RpcMgmtWaitServerListen....................Vol. 3, 432
RpcNetworkInqProtseqs......................Vol. 3, 433
RpcNetworkIsProtseqValid...................Vol. 3, 434
RPCNOTIFICATION_ROUTINE....................Vol. 3, 579
RpcNsBindingExport.........................Vol. 3, 435
RpcNsBindingExportPnP......................Vol. 3, 438
RpcNsBindingImportBegin....................Vol. 3, 440
RpcNsBindingImportDone.....................Vol. 3, 442
RpcNsBindingImportNext.....................Vol. 3, 443
RpcNsBindingInqEntryName...................Vol. 3, 445
RpcNsBindingLookupBegin....................Vol. 3, 446
RpcNsBindingLookupDone.....................Vol. 3, 449
RpcNsBindingLookupNext.....................Vol. 3, 450
RpcNsBindingSelect.........................Vol. 3, 452
RpcNsBindingUnexport.......................Vol. 3, 453
RpcNsBindingUnexportPnP....................Vol. 3, 456
RpcNsEntryExpandName.......................Vol. 3, 457
RpcNsEntryObjectInqBegin...................Vol. 3, 458
RpcNsEntryObjectInqDone....................Vol. 3, 460
RpcNsEntryObjectInqNext....................Vol. 3, 461
RpcNsGroupDelete...........................Vol. 3, 462
RpcNsGroupMbrAdd...........................Vol. 3, 463
RpcNsGroupMbrInqBegin......................Vol. 3, 465
RpcNsGroupMbrInqDone.......................Vol. 3, 466
RpcNsGroupMbrInqNext.......................Vol. 3, 467
RpcNsGroupMbrRemove........................Vol. 3, 468
RpcNsMgmtBindingUnexport...................Vol. 3, 470
RpcNsMgmtEntryCreate.......................Vol. 3, 473
RpcNsMgmtEntryDelete.......................Vol. 3, 474
RpcNsMgmtEntryInqIfIds.....................Vol. 3, 475
RpcNsMgmtHandleSetExpAge...................Vol. 3, 476
RpcNsMgmtInqExpAge.........................Vol. 3, 478
RpcNsMgmtSetExpAge.........................Vol. 3, 480
RpcNsProfileDelete.........................Vol. 3, 481
RpcNsProfileEltAdd.........................Vol. 3, 482
RpcNsProfileEltInqBegin....................Vol. 3, 484
RpcNsProfileEltInqDone.....................Vol. 3, 488
RpcNsProfileEltInqNext.....................Vol. 3, 488
RpcNsProfileEltRemove......................Vol. 3, 490
RpcObjectInqType...........................Vol. 3, 492
RpcObjectSetInqFn..........................Vol. 3, 493
RpcObjectSetType...........................Vol. 3, 494
RpcProtseqVectorFree.......................Vol. 3, 496
RpcRaiseException..........................Vol. 3, 497
RpcRevertToSelf............................Vol. 3, 501
RpcRevertToSelfEx..........................Vol. 3, 502
RpcServerInqBindings.......................Vol. 3, 503
RpcServerInqDefaultPrincName...............Vol. 3, 504
RpcServerInqIf.............................Vol. 3, 505
RpcServerListen............................Vol. 3, 506
RpcServerRegisterAuthInfo..................Vol. 3, 508
RpcServerRegisterIf........................Vol. 3, 511
RpcServerRegisterIf2.......................Vol. 3, 512
RpcServerRegisterIfEx......................Vol. 3, 514
RpcServerTestCancel........................Vol. 3, 516
RpcServerUnregisterIf......................Vol. 3, 517
RpcServerUseAllProtseqs....................Vol. 3, 519
RpcServerUseAllProtseqsEx..................Vol. 3, 521
RpcServerUseAllProtseqsIf..................Vol. 3, 523
RpcServerUseAllProtseqsIfEx................Vol. 3, 524
RpcServerUseProtseq........................Vol. 3, 526
RpcServerUseProtseqEp......................Vol. 3, 530
RpcServerUseProtseqEpEx....................Vol. 3, 532
RpcServerUseProtseqEx......................Vol. 3, 528
RpcServerUseProtseqIf......................Vol. 3, 534
RpcServerUseProtseqIfEx....................Vol. 3, 536
RpcSmAllocate..............................Vol. 3, 538

RpcSmClientFree .... Vol. 3, 539
RpcSmDestroyClientContext .... Vol. 3, 540
RpcSmDisableAllocate .... Vol. 3, 541
RpcSmEnableAllocate .... Vol. 3, 542
RpcSmFree .... Vol. 3, 543
RpcSmGetThreadHandle .... Vol. 3, 544
RpcSmSetClientAllocFree .... Vol. 3, 545
RpcSmSetThreadHandle .... Vol. 3, 546
RpcSmSwapClientAllocFree .... Vol. 3, 547
RpcSsAllocate .... Vol. 3, 548
RpcSsDestroyClientContext .... Vol. 3, 549
RpcSsDisableAllocate .... Vol. 3, 550
RpcSsDontSerializeContext .... Vol. 3, 550
RpcSsEnableAllocate .... Vol. 3, 551
RpcSsFree .... Vol. 3, 552
RpcSsGetThreadHandle .... Vol. 3, 553
RpcSsSetClientAllocFree .... Vol. 3, 554
RpcSsSetThreadHandle .... Vol. 3, 555
RpcSsSwapClientAllocFree .... Vol. 3, 556
RpcStringBindingCompose .... Vol. 3, 558
RpcStringBindingParse .... Vol. 3, 559
RpcStringFree .... Vol. 3, 561
RpcTestCancel .... Vol. 3, 562
RpcTryExcept .... Vol. 3, 590
RpcTryFinally .... Vol. 3, 590
RpcWinSetYieldInfo .... Vol. 3, 563
RpcWinSetYieldTimeout .... Vol. 3, 566
RSVP_ADSPEC .... Vol. 1, 802
RSVP_RESERVE_INFO .... Vol. 1, 803
RSVP_STATUS_INFO .... Vol. 1, 805
RTM_DEST_INFO .... Vol. 5, 480
RTM_ENTITY_EXPORT_
  METHOD .... Vol. 5, 477
RTM_ENTITY_EXPORT_
  METHODS .... Vol. 5, 481
RTM_ENTITY_ID .... Vol. 5, 482
RTM_ENTITY_INFO .... Vol. 5, 483
RTM_ENTITY_METHOD_
  OUTPUT .... Vol. 5, 484
RTM_ENTITY_METHOD_INPUT .... Vol. 5, 483
RTM_EVENT_CALLBACK .... Vol. 5, 478
RTM_EVENT_TYPE .... Vol. 5, 506
RTM_IP_ROUTE .... Vol. 5, 357
RTM_IPV4_GET_ADDR_AND_
  LEN .... Vol. 5, 492
RTM_IPV4_GET_ADDR_AND_
  MASK .... Vol. 5, 493
RTM_IPV4_LEN_FROM_MASK .... Vol. 5, 494
RTM_IPV4_MAKE_NET_
  ADDRESS .... Vol. 5, 495
RTM_IPV4_MASK_FROM_LEN .... Vol. 5, 496
RTM_IPV4_SET_ADDR_AND_
  LEN .... Vol. 5, 497
RTM_IPV4_SET_ADDR_AND_
  MASK .... Vol. 5, 498
RTM_IPX_ROUTE .... Vol. 5, 358
RTM_NET_ADDRESS .... Vol. 5, 485
RTM_NEXTHOP_INFO .... Vol. 5, 486
RTM_NEXTHOP_LIST .... Vol. 5, 487
RTM_PREF_INFO .... Vol. 5, 488
RTM_REGN_PROFILE .... Vol. 5, 488
RTM_ROUTE_INFO .... Vol. 5, 489
RTM_SIZE_OF_DEST_INFO .... Vol. 5, 499
RTM_SIZE_OF_ROUTE_INFO .... Vol. 5, 500
RtmAddNextHop .... Vol. 5, 405
RtmAddRoute .... Vol. 5, 335
RtmAddRouteToDest .... Vol. 5, 406
RtmBlockDeleteRoutes .... Vol. 5, 347
RtmBlockMethods .... Vol. 5, 409
RtmCloseEnumerationHandle .... Vol. 5, 346
RtmCreateDestEnum .... Vol. 5, 410
RtmCreateEnumerationHandle .... Vol. 5, 343
RtmCreateNextHopEnum .... Vol. 5, 413
RtmCreateRouteEnum .... Vol. 5, 414
RtmCreateRouteList .... Vol. 5, 417
RtmCreateRouteListEnum .... Vol. 5, 418
RtmDeleteEnumHandle .... Vol. 5, 419
RtmDeleteNextHop .... Vol. 5, 420
RtmDeleteRoute .... Vol. 5, 338
RtmDeleteRouteList .... Vol. 5, 421
RtmDeleteRouteToDest .... Vol. 5, 422
RtmDequeueRouteChange
  Message .... Vol. 5, 333
RtmDeregisterClient .... Vol. 5, 332
RtmDeregisterEntity .... Vol. 5, 423
RtmDeregisterFromChange
  Notification .... Vol. 5, 424
RtmEnumerateGetNextRoute .... Vol. 5, 345
RtmFindNextHop .... Vol. 5, 425
RtmGetChangedDests .... Vol. 5, 426
RtmGetChangeStatus .... Vol. 5, 428
RtmGetDestInfo .... Vol. 5, 429
RtmGetEntityInfo .... Vol. 5, 430
RtmGetEntityMethods .... Vol. 5, 431
RtmGetEnumDests .... Vol. 5, 432
RtmGetEnumNextHops .... Vol. 5, 434
RtmGetEnumRoutes .... Vol. 5, 435
RtmGetExactMatchDestination .... Vol. 5, 436
RtmGetExactMatchRoute .... Vol. 5, 438
RtmGetFirstRoute .... Vol. 5, 348
RtmGetLessSpecificDestination .... Vol. 5, 440
RtmGetListEnumRoutes .... Vol. 5, 441
RtmGetMostSpecificDestination .... Vol. 5, 443
RtmGetNetworkCount .... Vol. 5, 341
RtmGetNextHopInfo .... Vol. 5, 444
RtmGetNextHopPointer .... Vol. 5, 445
RtmGetNextRoute .... Vol. 5, 350
RtmGetOpaqueInformation
  Pointer .... Vol. 5, 446
RtmGetRegisteredEntities .... Vol. 5, 447

RtmGetRouteAge .... Vol. 5, 342
RtmGetRouteInfo .... Vol. 5, 449
RtmGetRoutePointer .... Vol. 5, 450
RtmHoldDestination .... Vol. 5, 451
RtmIgnoreChangedDests .... Vol. 5, 452
RtmInsertInRouteList .... Vol. 5, 453
RtmInvokeMethod .... Vol. 5, 454
RtmIsBestRoute .... Vol. 5, 455
RtmIsMarkedForChange
  Notification .... Vol. 5, 456
RtmIsRoute .... Vol. 5, 340
RtmLockDestination .... Vol. 5, 457
RtmLockNextHop .... Vol. 5, 459
RtmLockRoute .... Vol. 5, 460
RtmMarkDestForChange
  Notification .... Vol. 5, 461
RtmReferenceHandles .... Vol. 5, 463
RtmRegisterClient .... Vol. 5, 331
RtmRegisterEntity .... Vol. 5, 464
RtmRegisterForChange
  Notification .... Vol. 5, 466
RtmReleaseChangedDests .... Vol. 5, 467
RtmReleaseDestInfo .... Vol. 5, 469
RtmReleaseDests .... Vol. 5, 469
RtmReleaseEntities .... Vol. 5, 471
RtmReleaseEntityInfo .... Vol. 5, 471
RtmReleaseNextHopInfo .... Vol. 5, 472
RtmReleaseNextHops .... Vol. 5, 473
RtmReleaseRouteInfo .... Vol. 5, 474
RtmReleaseRoutes .... Vol. 5, 475
RtmUpdateAndUnlockRoute .... Vol. 5, 476

S

SEC_WINNT_AUTH_IDENTITY ...... Vol. 3, 312
SECURITY_MESSAGE .... Vol. 4, 311
select .... Vol. 1, 202
send .... Vol. 1, 206
SendARP .... Vol. 2, 270
sendto .... Vol. 1, 209
SENS_QOCINFO .... Vol. 2, 227
servent .... Vol. 1, 388
SERVICE_ADDRESS .... Vol. 1, 389
SERVICE_ADDRESSES .... Vol. 1, 390
SERVICE_INFO .... Vol. 1, 390
SERVICE_TYPE_INFO_ABS .... Vol. 1, 393
SERVICE_TYPE_VALUE_ABS .... Vol. 1, 394
SESSION_BUFFER .... Vol. 2, 160
SESSION_HEADER .... Vol. 2, 162
SetGlobalInfo .... Vol. 5, 291
SetIfEntry .... Vol. 2, 271
SetInterfaceInfo .... Vol. 5, 292
SetInterfaceReceiveType .... Vol. 5, 314
SetIpForwardEntry .... Vol. 2, 272
SetIpNetEntry .... Vol. 2, 273
SetIpStatistics .... Vol. 2, 274
SetIpTTL .... Vol. 2, 275
SetPower .... Vol. 5, 293
SetService .... Vol. 1, 212
setsockopt .... Vol. 1, 215
SetTcpEntry .... Vol. 2, 276
shutdown .... Vol. 1, 223
smiCNTR64 .... Vol. 2, 458
smiOCTETS .... Vol. 2, 459
smiOID .... Vol. 2, 460
smiVALUE .... Vol. 2, 461
smiVENDORINFO .... Vol. 2, 464
SNMPAPI_CALLBACK .... Vol. 2, 375
SnmpCancelMsg .... Vol. 2, 376
SnmpCleanup .... Vol. 2, 378
SnmpClose .... Vol. 2, 379
SnmpContextToStr .... Vol. 2, 380
SnmpCountVbl .... Vol. 2, 382
SnmpCreatePdu .... Vol. 2, 383
SnmpCreateSession .... Vol. 2, 385
SnmpCreateVbl .... Vol. 2, 388
SnmpDecodeMsg .... Vol. 2, 390
SnmpDeleteVb .... Vol. 2, 392
SnmpDuplicatePdu .... Vol. 2, 394
SnmpDuplicateVbl .... Vol. 2, 395
SnmpEncodeMsg .... Vol. 2, 396
SnmpEntityToStr .... Vol. 2, 398
SnmpExtensionClose .... Vol. 2, 290
SnmpExtensionInit .... Vol. 2, 291
SnmpExtensionInitEx .... Vol. 2, 293
SnmpExtensionMonitor .... Vol. 2, 294
SnmpExtensionQuery .... Vol. 2, 295
SnmpExtensionQueryEx .... Vol. 2, 298
SnmpExtensionTrap .... Vol. 2, 302
SnmpFreeContext .... Vol. 2, 399
SnmpFreeDescriptor .... Vol. 2, 401
SnmpFreeEntity .... Vol. 2, 402
SnmpFreePdu .... Vol. 2, 403
SnmpFreeVbl .... Vol. 2, 404
SnmpGetLastError .... Vol. 2, 406
SnmpGetPduData .... Vol. 2, 407
SnmpGetRetransmitMode .... Vol. 2, 411
SnmpGetRetry .... Vol. 2, 412
SnmpGetTimeout .... Vol. 2, 414
SnmpGetTranslateMode .... Vol. 2, 416
SnmpGetVb .... Vol. 2, 417
SnmpGetVendorInfo .... Vol. 2, 420
SnmpListen .... Vol. 2, 421
SnmpMgrClose .... Vol. 2, 304
SnmpMgrGetTrap .... Vol. 2, 305
SnmpMgrOidToStr .... Vol. 2, 307
SnmpMgrOpen .... Vol. 2, 308
SnmpMgrRequest .... Vol. 2, 309
SnmpMgrStrToOid .... Vol. 2, 311
SnmpMgrTrapListen .... Vol. 2, 312

SnmpOidCompare............................Vol. 2, 423
SnmpOidCopy.................................Vol. 2, 425
SnmpOidToStr.................................Vol. 2, 427
SnmpOpen......................................Vol. 2, 428
SnmpRecvMsg.................................Vol. 2, 430
SnmpRegister..................................Vol. 2, 433
SnmpSendMsg.................................Vol. 2, 436
SnmpSetPduData............................Vol. 2, 438
SnmpSetPort...................................Vol. 2, 440
SnmpSetRetransmitMode................Vol. 2, 442
SnmpSetRetry.................................Vol. 2, 444
SnmpSetTimeout.............................Vol. 2, 445
SnmpSetTranslateMode..................Vol. 2, 446
SnmpSetVb.....................................Vol. 2, 448
SnmpStartup...................................Vol. 2, 450
SnmpStrToContext..........................Vol. 2, 453
SnmpStrToEntity.............................Vol. 2, 455
SnmpStrToOid.................................Vol. 2, 456
SnmpSvcGetUptime........................Vol. 2, 314
SnmpSvcSetLogLevel......................Vol. 2, 315
SnmpSvcSetLogType.......................Vol. 2, 316
SnmpUtilAsnAnyCpy........................Vol. 2, 317
SnmpUtilAsnAnyFree.......................Vol. 2, 317
SnmpUtilDbgPrint............................Vol. 2, 318
SnmpUtilIdsToA..............................Vol. 2, 319
SnmpUtilMemAlloc..........................Vol. 2, 321
SnmpUtilMemFree...........................Vol. 2, 321
SnmpUtilMemReAlloc......................Vol. 2, 322
SnmpUtilOctetsCmp........................Vol. 2, 323
SnmpUtilOctetsCpy.........................Vol. 2, 324
SnmpUtilOctetsFree........................Vol. 2, 325
SnmpUtilOctetsNCmp......................Vol. 2, 325
SnmpUtilOidAppend........................Vol. 2, 326
SnmpUtilOidCmp.............................Vol. 2, 327
SnmpUtilOidCpy..............................Vol. 2, 328
SnmpUtilOidFree.............................Vol. 2, 329
SnmpUtilOidNCmp...........................Vol. 2, 330
SnmpUtilOidToA..............................Vol. 2, 331
SnmpUtilPrintAsnAny.......................Vol. 2, 331
SnmpUtilPrintOid.............................Vol. 2, 332
SnmpUtilVarBindCpy........................Vol. 2, 333
SnmpUtilVarBindFree.......................Vol. 2, 335
SnmpUtilVarBindListCpy..................Vol. 2, 334
SnmpUtilVarBindListFree.................Vol. 2, 335
SnmpVarBind..................................Vol. 2, 340
SnmpVarBindList.............................Vol. 2, 341
sockaddr...Vol. 1, 396
SOCKADDR_IRDA...........................Vol. 1, 397
socket...Vol. 1, 225
SOCKET_ADDRESS.......................Vol. 1, 397
SOURCE_GROUP_ENTRY............Vol. 5, 563
StartComplete.................................Vol. 5, 293
StartProtocol...................................Vol. 5, 294
StopProtocol....................................Vol. 5, 295
String Binding..................................Vol. 3, 324
String UUID......................................Vol. 3, 329
SUPPORT_FUNCTIONS.................Vol. 5, 305
SYNCMGRFLAG...............................Vol. 2, 196
SYNCMGRHANDLERFLAGS...........Vol. 2, 197
SYNCMGRHANDLERINFO..............Vol. 2, 201
SYNCMGRINVOKEFLAGS..............Vol. 2, 200
SYNCMGRITEM...............................Vol. 2, 203
SYNCMGRITEMFLAGS...................Vol. 2, 199
SYNCMGRLOGERRORINFO...........Vol. 2, 202
SYNCMGRLOGLEVEL.....................Vol. 2, 199
SYNCMGRPROGRESSITEM...........Vol. 2, 201
SYNCMGRSTATUS.........................Vol. 2, 198

T

TC_GEN_FILTER.............................Vol. 1, 845
TC_GEN_FLOW...............................Vol. 1, 846
TC_IFC_DESCRIPTOR.....................Vol. 1, 847
TcAddFilter.......................................Vol. 1, 807
TcAddFlow..Vol. 1, 809
TcCloseInterface...............................Vol. 1, 811
TcDeleteFilter....................................Vol. 1, 812
TcDeleteFlow.....................................Vol. 1, 813
TcDeregisterClient............................Vol. 1, 814
TcEnumerateFlows............................Vol. 1, 815
TcEnumerateInterfaces.....................Vol. 1, 817
TcGetFlowName................................Vol. 1, 819
TCI_CLIENT_FUNC_LIST.................Vol. 1, 847
TcModifyFlow....................................Vol. 1, 820
TcOpenInterface................................Vol. 1, 822
TcQueryFlow......................................Vol. 1, 823
TcQueryInterface...............................Vol. 1, 824
TcRegisterClient................................Vol. 1, 826
TcSetFlow..Vol. 1, 827
TcSetInterface...................................Vol. 1, 828
The ProviderSpecific Buffer..............Vol. 1, 799
timeval...Vol. 1, 398
TraceDeregister.................................Vol. 4, 438
TraceDump..Vol. 4, 438
TraceDumpEx....................................Vol. 4, 440
TracePrintf...Vol. 4, 441
TracePrintfEx.....................................Vol. 4, 442
TracePuts..Vol. 4, 444
TracePutsEx......................................Vol. 4, 445
TraceRegister....................................Vol. 4, 446
TraceRegisterEx................................Vol. 4, 447
TraceVprintf.......................................Vol. 4, 449
TraceVprintfEx...................................Vol. 4, 450
TRANSMIT_FILE_BUFFERS............Vol. 1, 399
TransmitFile.......................................Vol. 1, 228
Transport Identifiers..........................Vol. 5, 235

U

UnbindInterface .... Vol. 5, 296
UNIVERSAL_NAME_INFO .... Vol. 3, 667
UPDATE_COMPLETE_
  MESSAGE .... Vol. 5, 303
UUID .... Vol. 3, 313
UUID_VECTOR .... Vol. 3, 314
UuidCompare .... Vol. 3, 567
UuidCreate .... Vol. 3, 568
UuidCreateNil .... Vol. 3, 570
UuidCreateSequential .... Vol. 3, 569
UuidEqual .... Vol. 3, 570
UuidFromString .... Vol. 3, 571
UuidHash .... Vol. 3, 572
UuidIsNil .... Vol. 3, 573
UuidToString .... Vol. 3, 574

V

ValidateRoute .... Vol. 5, 315
View Flags .... Vol. 5, 501

W

WM_RASDIALEVENT .... Vol. 4, 257
WNetAddConnection .... Vol. 3, 611
WNetAddConnection2 .... Vol. 3, 613
WNetAddConnection3 .... Vol. 3, 616
WNetCancelConnection .... Vol. 3, 620
WNetCancelConnection2 .... Vol. 3, 622
WNetCloseEnum .... Vol. 3, 624
WNetConnectionDialog .... Vol. 3, 625
WNetConnectionDialog1 .... Vol. 3, 626
WNetDisconnectDialog .... Vol. 3, 628
WNetDisconnectDialog1 .... Vol. 3, 629
WNetEnumResource .... Vol. 3, 630
WNetGetConnection .... Vol. 3, 632
WNetGetLastError .... Vol. 3, 634
WNetGetNetworkInformation .... Vol. 3, 635
WNetGetProviderName .... Vol. 3, 636
WNetGetResourceInformation .... Vol. 3, 638
WNetGetResourceParent .... Vol. 3, 640
WNetGetUniversalName .... Vol. 3, 642
WNetGetUser .... Vol. 3, 645
WNetOpenEnum .... Vol. 3, 647
WNetUseConnection .... Vol. 3, 650
WPUCloseEvent .... Vol. 1, 515
WPUCloseSocketHandle .... Vol. 1, 515
WPUCloseThread .... Vol. 1, 516
WPUCompleteOverlapped
  Request .... Vol. 1, 517
WPUCreateEvent .... Vol. 1, 520
WPUCreateSocketHandle .... Vol. 1, 521
WPUFDIsSet .... Vol. 1, 523
WPUGetProviderPath .... Vol. 1, 524
WPUGetQOSTemplate .... Vol. 1, 783
WPUModifyIFSHandle .... Vol. 1, 525
WPUOpenCurrentThread .... Vol. 1, 527
WPUPostMessage .... Vol. 1, 528
WPUQueryBlockingCallback .... Vol. 1, 529
WPUQuerySocketHandleContext ..... Vol. 1, 530
WPUQueueApc .... Vol. 1, 531
WPUResetEvent .... Vol. 1, 533
WPUSetEvent .... Vol. 1, 534
WSAAccept .... Vol. 1, 231
WSAAddressToString .... Vol. 1, 235
WSAAsyncGetHostByAddr .... Vol. 1, 236
WSAAsyncGetHostByName .... Vol. 1, 239
WSAAsyncGetProtoByName .... Vol. 1, 242
WSAAsyncGetProtoByNumber .... Vol. 1, 245
WSAAsyncGetServByName .... Vol. 1, 248
WSAAsyncGetServByPort .... Vol. 1, 251
WSAAsyncSelect .... Vol. 1, 254
WSABUF .... Vol. 1, 399
WSACancelAsyncRequest .... Vol. 1, 263
WSACancelBlockingCall .... Vol. 1, 265
WSACleanup .... Vol. 1, 265
WSACloseEvent .... Vol. 1, 267
WSAConnect .... Vol. 1, 268
WSACreateEvent .... Vol. 1, 272
WSADATA .... Vol. 1, 400
WSADuplicateSocket .... Vol. 1, 273
WSAECOMPARATOR .... Vol. 1, 413
WSAEnumNameSpaceProviders ..... Vol. 1, 276
WSAEnumNetworkEvents .... Vol. 1, 277
WSAEnumProtocols .... Vol. 1, 279
WSAEventSelect .... Vol. 1, 281
WSAGetLastError .... Vol. 1, 287
WSAGetOverlappedResult .... Vol. 1, 288
WSAGetQOSByName .... Vol. 1, 290
WSAGetQOSByName .... Vol. 1, 784
WSAGetServiceClassInfo .... Vol. 1, 292
WSAGetServiceClassNameBy
  ClassId .... Vol. 1, 293
WSAHtonl .... Vol. 1, 294
WSAHtons .... Vol. 1, 295
WSAInstallServiceClass .... Vol. 1, 296
WSAIoctl .... Vol. 1, 297
WSAIsBlocking .... Vol. 1, 308
WSAJoinLeaf .... Vol. 1, 309
WSALookupServiceBegin .... Vol. 1, 313
WSALookupServiceEnd .... Vol. 1, 317
WSALookupServiceNext .... Vol. 1, 318
WSANAMESPACE_INFO .... Vol. 1, 401
WSANETWORKEVENTS .... Vol. 1, 402
WSANtohl .... Vol. 1, 322
WSANtohs .... Vol. 1, 323
WSAOVERLAPPED .... Vol. 1, 403

WSAPROTOCOL_INFO .... Vol. 1, 404
WSAPROTOCOLCHAIN.................. Vol. 1, 408
WSAProviderConfigChange............. Vol. 1, 324
WSAQUERYSET .... Vol. 1, 409
WSARecv .... Vol. 1, 326
WSARecvDisconnect........................ Vol. 1, 332
WSARecvEx....................................... Vol. 1, 334
WSARecvFrom................................... Vol. 1, 337
WSARemoveServiceClass............... Vol. 1, 343
WSAResetEvent................................ Vol. 1, 344
WSASend.. Vol. 1, 345
WSASendDisconnect........................ Vol. 1, 350
WSASendTo...................................... Vol. 1, 352
WSASERVICECLASSINFO............. Vol. 1, 411
WSASetBlockingHook....................... Vol. 1, 357
WSASetEvent.................................... Vol. 1, 358
WSASetLastError.............................. Vol. 1, 359
WSASetService................................. Vol. 1, 360
WSASocket .... Vol. 1, 363
WSAStartup....................................... Vol. 1, 367
WSAStringToAddress .... Vol. 1, 371
WSATHREADID................................. Vol. 1, 412
WSAUnhookBlockingHook............... Vol. 1, 372
WSAWaitForMultipleEvents .... Vol. 1, 373
WSCDeinstallProvider....................... Vol. 1, 535
WSCEnableNSProvider .... Vol. 1, 536
WSCEnumProtocols.......................... Vol. 1, 537
WSCGetProviderPath .... Vol. 1, 539
WSCInstallNameSpace..................... Vol. 1, 540
WSCInstallProvider .... Vol. 1, 541
WSCInstallQOSTemplate .... Vol. 1, 786
WSCRemoveQOSTemplate .... Vol. 1, 788
WSCUnInstallNameSpace .... Vol. 1, 543
WSCWriteProviderOrder.................. Vol. 1, 543
WSPAccept .... Vol. 1, 545
WSPAddressToString.......................Vol. 1, 549
WSPAsyncSelect.............................Vol. 1, 550
WSPBind..Vol. 1, 558
WSPCancelBlockingCall..................Vol. 1, 560
WSPCleanup ....Vol. 1, 562
WSPCloseSocket ....Vol. 1, 564
WSPConnect ....Vol. 1, 566
WSPDuplicateSocket.......................Vol. 1, 570
WSPEnumNetworkEvents...............Vol. 1, 573
WSPEventSelect ....Vol. 1, 576
WSPGetOverlappedResult..............Vol. 1, 581
WSPGetPeerName...........................Vol. 1, 584
WSPGetQOSByName......................Vol. 1, 585
WSPGetQOSByName......................Vol. 1, 789
WSPGetSockName ....Vol. 1, 586
WSPGetSockOpt..............................Vol. 1, 588
WSPIoctl ....Vol. 1, 593
WSPJoinLeaf...................................Vol. 1, 604
WSPListen ....Vol. 1, 608
WSPRecv..Vol. 1, 610
WSPRecvDisconnect.......................Vol. 1, 617
WSPRecvFrom ....Vol. 1, 618
WSPSelect.......................................Vol. 1, 624
WSPSend ....Vol. 1, 628
WSPSendDisconnect ....Vol. 1, 633
WSPSendTo ....Vol. 1, 634
WSPSetSockOpt ....Vol. 1, 640
WSPShutdown.................................Vol. 1, 644
WSPSocket......................................Vol. 1, 645
WSPStartup ....Vol. 1, 649
WSPStringToAddress.......................Vol. 1, 654

Y

YieldFunctionName ....Vol. 3, 580

RPC and Windows® Networking

Networking Services

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.*

RPC and Windows Networking

This volume includes concise reference information about remote procedure calls (RPC) and Windows Networking features and functions. RPC is a powerful technology that simplifies distributed client/server development by managing most of the details of network protocols and communication so you can focus on creating your application instead of on the details of the network. Windows Networking lets you implement networking capabilities in your application without worrying about any particular network provider or physical network implementation.

Part No. 097-0002785

Microsoft