



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.isevinger.com](http://www.isevinger.com)



# **Windows® Sockets and QOS**

BASED ON  
**msdn™** library

**Microsoft®**



**David Iseminger**  
Series Editor

# **Windows® Sockets and QOS**



**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](http://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. 097-0002783

# 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

<b>Acknowledgements</b> .....	<b>iii</b>
-------------------------------	------------

## Part 1

<b>Chapter 1: Getting Around in the Networking Services Library</b> .....	<b>1</b>
How the Networking Services Library Is Structured.....	2
How the Networking Services Library Is Designed .....	3
<b>Chapter 2: What's In This Volume?</b> .....	<b>5</b>
Winsock.....	6
Quality of Service .....	7
<b>Chapter 3: Using Microsoft Reference Resources</b> .....	<b>9</b>
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
<b>Chapter 4: Finding the Developer Resources You Need</b> .....	<b>31</b>
Developer Support.....	31
Online Resources .....	33
Internet Standards.....	34
Learning Products .....	35
Conferences .....	37
Other Resources .....	37

**Chapter 5: Writing Great IrDA Applications (with Winsock) ..... 39**

- What Is an Ad-Hoc Networking-Enabled Application? ..... 39
- What Is IrDA? ..... 40
- What Is IrDA-C (Previously Known as IrBus)? ..... 40
- What Is Unique about IrDA? ..... 41
- IrDA Core Protocols and Services ..... 41
  - Serial IrDA (SIR) Physical Layer (115 Kb/s) ..... 41
  - Fast IrDA (FIR) Physical Layer (4 Mb/s) ..... 41
  - IrLAP Data Link Layer ..... 42
  - IrLMP and TinyTP ..... 42
  - IrCOMM ..... 43
    - IrCOMM Modes ..... 44
  - No IrCOMM Virtual Serial Ports on Windows 2000 ..... 45
  - Windows 2000 Support for IrCOMM Through Winsock ..... 45
- IrDA and the Windows Sockets API ..... 46
  - Talking to Non-Windows Devices ..... 46
  - Application Addressing ..... 46
  - Data Transfer and Connection Close ..... 47
- IrDA and Winsock Reference ..... 49
  - WSAStartup ..... 49
  - af\_irda.h ..... 49
  - socket ..... 49
  - SOCKADDR\_IRDA Structure ..... 50
  - bind ..... 50
  - listen ..... 51
  - accept ..... 51
  - send and recv ..... 52
  - closesocket ..... 52
  - getsockopt(, IRLMP\_ENUMDEVICES,,) and connect() ..... 52
  - IAS ..... 54
  - IrCOMM Client ..... 57
- Windows 2000 IrDA Architecture ..... 60
  - IrDA Hardware Drivers ..... 60
  - Windows 2000 Multiple-Adapter Support ..... 61

---

## Part 2

<b>Chapter 6: Winsock 2 API Overview .....</b>	<b>63</b>
Welcome to Windows Sockets 2 .....	63
Using the Windows Sockets 2 API Document .....	63
Overview of Windows Sockets 2 .....	63
Windows Sockets 2 Features.....	64
Conventions for New Functions .....	65
Microsoft Extensions and the Windows Sockets 2 API.....	65
Socket Handles for Windows Sockets 2 .....	65
New Concepts, Additions, and Changes for Windows Sockets 2 .....	66
Windows Sockets 2 Architecture.....	66
Simultaneous Access to Multiple Transport Protocols.....	66
Backward Compatibility for Windows Sockets 1.1 Applications.....	67
Making Transport Protocols Available to Windows Sockets .....	69
Layered Protocols and Protocol Chains.....	69
Using Multiple Protocols.....	70
Multiple Provider Restrictions on Select.....	71
Function Extension Mechanism .....	72
Debug and Trace Facilities.....	72
Name Resolution .....	73
Overlapped I/O and Event Objects .....	73
Event Objects .....	74
Receiving Completion Indications .....	75
Asynchronous Notification Using Event Objects .....	76
Flow Specification Quality of Service .....	77
QOS Templates.....	77
Default Values .....	77
Socket Groups.....	77
Shared Sockets .....	77
Enhanced Functionality During Connection Setup and Teardown .....	78
Extended Byte-Order Conversion Routines .....	79
Support for Scatter/Gather I/O in the API.....	79
Protocol-Independent Multicast and Multipoint .....	79
Summary of New Socket Options .....	80
Summary of New Socket ioctl Opcodes.....	81
Summary of New Functions .....	82
Windows Sockets Programming Considerations .....	85
Deviation from Berkeley Sockets .....	85



---

Socket Data Type .....	85
Select and FD_* .....	86
Error Codes—errno, h_errno and WSAGetLastError .....	86
Pointers .....	87
Renamed Functions .....	87
Maximum Number of Sockets Supported .....	87
Include Files .....	88
Return Values on Function Failure .....	88
Service Provided Raw Sockets .....	88
Byte Ordering .....	89
Windows Sockets Compatibility Issues .....	89
Default State for a Socket's Overlapped Attribute .....	90
Windows Sockets 1.1 Blocking Routines and EINPROGRESS .....	90
Graceful Shutdown, Linger Options, and Socket Closure .....	92
Protocol-Independent Out-of-Band Data .....	93
Summary of Windows Sockets 2 Functions .....	96
Socket Functions .....	96
Microsoft Windows-Specific Extension Functions .....	97
Registration and Name Resolution .....	99
Protocol-Independent Name Resolution .....	100
Name Resolution Model .....	100
Summary of Name Resolution Functions .....	103
Name Resolution Data Structures .....	105
Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 API .....	108
Basic Approach for GetXbyY in the API .....	109
getprotobyname and getprotobynumber Functions in the API .....	109
getservbyname and getservbyport Functions in the API .....	109
gethostbyname Function in the API .....	110
gethostbyaddr Function in the API .....	110
gethostname Function in the API .....	111
Multipoint and Multicast Semantics .....	111
Multipoint Taxonomy .....	111
Windows Sockets 2 Interface Elements for Multipoint and Multicast .....	112
Attributes in WSAPROTOCOL_INFO Structure .....	113
Flag Bits for WSASocket .....	113
SIO_MULTIPPOINT_LOOPBACK Command Code for WSALocctl .....	114
SIO_MULTICAST_SCOPE Command Code for WSALocctl .....	114
Semantics for Joining Multipoint Leaves .....	114
Using WSAJoinLeaf .....	115

---

Semantic Differences Between Multipoint Sockets and Regular Sockets .....	116
How Existing Multipoint Protocols Support These Extensions .....	117
IP Multicast .....	117
ATM Point to Multipoint .....	118
Additional Windows Socket Information .....	119
Windows Sockets 2 API Header File—Winsock2.h .....	119
Socket Options Specific to Microsoft Service Providers .....	119
Socket Option for Windows NT 4.0 Only .....	119
Socket Option for Windows NT 4.0 and Windows 95 .....	120
Additional Documentation .....	121
<b>Chapter 7: Error Codes in the Winsock API .....</b>	<b>123</b>
Error Codes .....	123
<b>Chapter 8: Winsock 2 Functions .....</b>	<b>133</b>
Windows Sockets 2 Functions .....	133
<b>Chapter 9: Winsock 2 Structures and Enumerations .....</b>	<b>377</b>
Windows Sockets Structures in the API .....	377
Windows Sockets Enumeration in the API .....	413
<b>Chapter 10: Winsock 2 SPI Overview .....</b>	<b>415</b>
Welcome to Windows Sockets 2 SPI .....	415
Using the SPI Document .....	415
Overview of the Windows Sockets 2 SPI .....	415
Windows Sockets 2 SPI Features .....	416
Microsoft Extensions and the Windows Sockets 2 SPI .....	417
Socket Handles for the Windows Sockets 2 SPI .....	417
Windows Sockets 2 Architectural Overview .....	418
Windows Sockets 2 as a WOSA Component .....	418
Windows Sockets 2 DLLs .....	419
Function Interface Model .....	419
Naming Conventions .....	420
Windows Sockets 2 Service Providers .....	420
Transport Service Providers .....	420
Namespace Service Providers .....	422
Windows Sockets 2 Identifiers .....	424
Data Transport Providers .....	424
Transport Division of Responsibilities Between DLL and Service Providers .....	424
Transport Mapping Between API and SPI Functions .....	426

Function Extension Mechanism in the SPI .....	427
Transport Configuration and Installation .....	428
Name Resolution Providers .....	429
Name Resolution Model for the SPI .....	429
Name Resolution Division of Responsibilities Between DLL and Service Providers .....	432
Name Resolution Mapping Between API and SPI Functions .....	433
Name Resolution Configuration and Installation .....	433
Windows Sockets 2 Transport Provider Requirements .....	434
Service Provider Activation .....	434
Initialization .....	434
Cleanup .....	436
Error Reporting and Parameter Validation .....	436
Byte Ordering Assumptions .....	436
Socket Creation and Descriptor Management .....	437
Descriptor Allocation .....	437
Socket Attribute Flags and Modes .....	438
Closing Sockets .....	438
Blocking Operations .....	438
Pseudo vs. True Blocking .....	439
Blocking Hook .....	439
Canceling Blocking Operations .....	440
Event Objects in the Windows Sockets 2 SPI .....	440
Creating Event Objects .....	440
Using Event Objects .....	441
Destroying Event Objects .....	441
Notification of Network Events .....	441
Selects .....	442
Windows Messages .....	442
Event Object Signaling .....	442
Socket Groups in the Windows Sockets 2 SPI .....	442
Socket Group Operations .....	442
Required Socket Grouping Behavior .....	442
Recommended Socket Grouping Behavior .....	442
Quality of Service in the Windows Sockets 2 SPI .....	443
Socket Connections on Connection-Oriented Protocols .....	443
Binding to a Local Address .....	443
Protocol Basics: Listen, Connect, Accept .....	443
Determining Local and Remote Names .....	444

Enhanced Functionality at Connect Time .....	444
Connection Shutdown .....	445
Socket Connections on Connectionless Protocols.....	448
Connecting to a Default Peer .....	448
Reconnecting and Disconnecting.....	448
Using Sendto While Connected .....	448
Socket I/O .....	448
Blocking Input/Output .....	449
Nonblocking Input/Output.....	449
Overlapped Input/Output .....	449
Support for Scatter/Gather Input/Output in the SPI .....	453
Out-of-Band Data in the SPI .....	453
Shared Sockets in the SPI .....	455
Multiple Handles to a Single Socket.....	456
Reference Counting .....	456
Precedence Guidelines .....	457
Protocol-Independent Multicast and Multipoint in the SPI .....	457
Multipoint Taxonomy and Glossary .....	458
Multipoint Attributes in the WSAPROTOCOL_INFOW Structure.....	459
Multipoint Socket Attributes.....	459
SIO_MULTIPOINT_LOOPBACK loctl.....	460
SIO_MULTICAST_SCOPE loctl .....	460
SPI Semantics for Joining Multipoint Leaves.....	460
Using WSPJoinLeaf .....	461
Semantic Differences Between Multipoint Sockets and Regular Sockets in the SPI.....	462
Socket Options and IOCTLs.....	463
Summary of Socket loctl Opcodes.....	465
Summary of SPI Functions.....	466
Generic Data Transport Functions .....	466
Upcalls Exposed by Windows Sockets 2 DLL .....	468
Installation and Configuration Functions .....	471
Name Resolution Service Provider Requirements.....	471
Summary of Namespace Provider Functions.....	471
Namespace Provider Configuration and Installation.....	472
Namespace Provider Initialization and Cleanup .....	472
Service Installation in the Windows Sockets 2 SPI.....	472
Service Query.....	473
Helper Functions in the SPI .....	473

Name Resolution Data Structures in the SPI.....	474
Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 SPI.....	477
Basic Approach for getXbyY in the SPI .....	478
getprotobyname and getprotobynumber Functions in the SPI .....	478
getservbyname and getservbyport Functions in the SPI .....	478
gethostbyname Function in the SPI.....	479
gethostbyaddr Function in the SPI.....	479
gethostname Function in the SPI.....	480
Sample Code for a Service Provider .....	480
Additional Windows Sockets 2 SPI Concerns .....	495
Service Provider Ordering.....	495
Windows Sockets SPI Header File - Ws2spi.h .....	496
<b>Chapter 11: Winsock 2 SPI Reference .....</b>	<b>497</b>
Winsock 2 SPI Reference .....	497
<b>Chapter 12: Winsock 2 Protocol-Specific Annex .....</b>	<b>657</b>
Using the Annex.....	657
Overview of Windows Sockets 2.....	657
Microsoft Extensions and Windows Sockets 2 .....	658
Socket Handles for Windows Sockets 2 .....	658
TCP/IP.....	658
TCP/IP Introduction.....	658
TCP/IP Overview .....	659
TCP/IP Data Structures .....	659
TCP/IP Controls .....	660
UNIX loctls .....	660
TCP/IP Socket Options.....	660
TCP/IP Function Details.....	663
Multicast.....	663
TCP/IP Raw Sockets .....	663
IPv6 Support .....	664
Text Representation of IPv6 Addresses .....	665
TCP/IP Header File.....	666
IPX/SPX .....	666
IPX/SPX Introduction .....	666
IPX/SPX Overview .....	666
AF_IPX Address Family.....	667
IPX Family of Protocol Identifiers.....	667
Broadcast to Local Network.....	668

---

All Routes Broadcast.....	668
Directed Broadcast.....	668
About Media Packet Size .....	669
How Packet Size Affects Protocols .....	669
IPX/SPX Data Structures .....	670
IPX/SPX Controls.....	674
NSPROTO_IPX Socket Options .....	675
DECnet.....	676
DECnet Overview.....	676
DNPROTO_NSP Protocol Family .....	677
AF_DECnet Address Families .....	677
SOCK_SEQPACKET Socket Type .....	678
DECnet Data Structures.....	678
Manifest Constants (Winsock2.h) .....	678
Manifest Constants (Ws2dnet.h).....	678
Data Structures (Ws2dnet.h).....	678
DECnet Function Details.....	680
Connections Using Accept/WSAAccept/WSPAccept .....	680
Structure Information for Bind/WSPBind.....	682
Connections Using Connect/WSAConnect/WSPConnect .....	682
Addressing with GetPeerName/WSPGetPeerName .....	684
Receiving Local Name with getsockname/WSPGetSockName.....	684
Using Getsockopt/WSPGetSockOpt .....	685
Using Socket/WSASocket/WSPSocket.....	686
DECnet Out-of-Band Data .....	686
DECnet-Specific Extended Functions Identifiers .....	686
dnet_addr .....	687
dnet_eof .....	687
dnet_getacc.....	688
dnet_getalias.....	688
dnet_htoa .....	688
dnet_ntoa .....	689
getnodeadd .....	689
getnodebyaddr .....	689
getnodebyname.....	690
getnodename .....	690
DECnet Header File .....	691
Open Systems Interconnection (OSI).....	691
OSI Introduction .....	691



International Organization for Standardization (IOS).....	691
OSI Expedited Data .....	692
ISO Qualified Data .....	692
ISO Reset .....	692
OSI Quality of Service.....	692
Option Profiles .....	692
Address Format .....	693
OSI Data Structures.....	693
OSI Controls.....	693
Ioctl.....	694
Socket Options.....	694
OSI Function Specifics.....	695
Quality of Service.....	695
OSI Header File .....	695
ATM-Specific Extensions .....	695
ATM Introduction.....	695
ATM Overview.....	696
ATM Data Structures .....	696
Using the ATM_ADDRESS Structure .....	698
ATM_BLLI Structure and Associated Manifest Constants.....	699
ATM_BHLI Structure and Associated Manifest Constants .....	701
ATM Controls .....	701
ATM Function Specifics .....	702
ATM-Specific Quality of Service Extension .....	702
AAL Parameters.....	703
ATM Traffic Descriptor .....	704
Broadband Bearer Capability.....	705
Broadband High Layer Information.....	706
Broadband Lower Layer Information .....	706
Called Party Number.....	707
Called Party Subaddress .....	707
Calling Party Number.....	707
Calling Party Subaddress .....	708
Quality of Service Parameter .....	708
Transit Network Selection.....	708
Cause.....	709
ATM Header File.....	711

---

Other Windows Sockets 2 Considerations .....	711
Secure Sockets Layer (SSL) .....	711
RSVP .....	711
<b>Chapter 13: QOS Overview .....</b>	<b>713</b>
QOS Documentation Structure.....	713
Determining Which Discussion Is for You .....	714
Additional Information on QOS.....	715
About Quality of Service .....	715
Introduction to QOS.....	715
Quality of Service Defined.....	716
Windows 2000 Quality of Service Defined .....	716
What QOS Solves .....	716
How Windows 2000 QOS Works .....	718
Windows 98 QOS Notes .....	718
QOS Header Files .....	720
QOS Components .....	720
Application-Driven QOS Components.....	721
Network-Driven QOS Components .....	724
Policy-Driven QOS Components.....	726
RSVP and QOS.....	729
<b>Chapter 14: QOS Programming.....</b>	<b>731</b>
Basic QOS Operations .....	731
QOS-Enabling Your Application.....	731
Opening a QOS-Enabled Socket .....	732
Invoking the RSVP SP .....	732
Providing the RSVP SP with QOS-specific Parameters .....	733
Receiving QOS-Enabled Data .....	734
Sending QOS-Enabled Data .....	735
Closing the QOS Connection .....	736
QOS Templates.....	736
Enumerating Available QOS Templates .....	737
Applying a QOS Template .....	737
Installing a QOS Template .....	738
Removing a QOS Template .....	738
Built-in QOS Templates.....	738
RSVP SP Error Codes .....	739
Error Codes .....	739
Error Values .....	740

Service Types .....	749
Primary Service Types.....	750
BEST EFFORT .....	750
CONTROLLED LOAD.....	750
GUARANTEED .....	750
QUALITATIVE.....	751
Secondary Service Types .....	751
SERVICETYPE_NOTRAFFIC .....	751
SERVICETYPE_GENERAL_INFORMATION .....	752
SERVICETYPE_NOCHANGE .....	752
SERVICE_NO_TRAFFIC_CONTROL.....	752
SERVICE_NO_QOS_SIGNALING .....	753
Using Service Types .....	753
Directional Implications of Service Types .....	753
Examples of Setting the Service Type .....	754
Using the ProviderSpecific Buffer .....	755
Structure of the ProviderSpecific Buffer.....	755
Use of the ProviderSpecific Buffer as a Receiver.....	755
Use of the ProviderSpecific Buffer as a Sender .....	755
Understanding Traffic Control .....	756
How the RSVP SP Invokes TC .....	756
Using SIO_CHK_QOS .....	757
Disabling Traffic Control.....	758
QOS Events .....	758
Listening for FD_QOS Events .....	759
Using WSAEventSelect or WSAAsyncSelect.....	759
Using Overlapped WSALocatl(SIO_GET_QOS) .....	759
QOS Event Codes .....	760
RSVP SP and RSVP .....	761
Basic RSVP Operations.....	761
Invoking RSVP .....	761
Using the RSVP_RESERVE_INFO Object.....	762
Confirming RSVP Reservations.....	762
Disabling RSVP Signaling .....	763
RSVP Reservation Styles .....	763
Base RSVP Reservation Styles.....	763
Default RSVP Filter Style Settings.....	764
Overriding Default RSVP Filter Style Settings.....	765
Mapping RSVP SP Parameters to RSVP .....	766

RSVP PATH and RESV Messages.....	767
Tspec, FlowSpec, and Adspec.....	769
Mapping QOS Call Sequences to RSVP .....	771
Sending Applications.....	771
Receiving Applications .....	775
Receiver Reservation Semantics .....	778
Using WSAConnect to Join Unicast RSVP Sessions .....	778
Using WSAJoinLeaf to Join Multicast RSVP Sessions.....	780
Use of Sendto and WSASendTo by Multicast Senders .....	781
Using WSALocctl(SIO_SET_QOS) During RSVP Sessions .....	781
<b>Chapter 15: QOS API Reference.....</b>	<b>783</b>
QOS Functions .....	783
QOS Structures .....	791
QOS Objects .....	798
<b>Chapter 16: Traffic Control API Reference.....</b>	<b>807</b>
Traffic Control Functions .....	807
Entry Points Exposed by Clients of the Traffic Control Interface .....	829
Traffic Control Structures .....	834
Traffic Control Objects.....	855
<b>Chapter 17: Local Policy Module API Reference .....</b>	<b>859</b>
LPM Functions.....	859
LPM Structures.....	875

### Part 3

<b>Index: Networking Services Programming Elements – Alphabetical Listing.....</b>	<b>879</b>
--	------------



## 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, presharpener toolset to build compelling Windows applications.

---

## CHAPTER 2

# What's In This Volume?

Volume 1 of the *Networking Services Developer's Reference Library* focuses on what many developers think of first when network programming comes to mind: Winsock. Quality of Service (QOS) is implemented through calls to Winsock functions, so it only makes sense to keep these two technologies together in one volume—enabling you to read about a given Winsock function when learning about QOS, then to flip back to the Winsock section and get the details on that particular Winsock function. If you've read through the first chapter in any of the WPRS Libraries (including this one), you'll know that my primary objective is to ensure that these volumes provide you with the information you need in as convenient and useful a way as possible. Keeping QOS with Winsock (in the same volume) is one way I've tried to achieve that objective in this library. I've also structured other volumes in this library with similar goals toward cohesiveness and cross-technology referencing.

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

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

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

Part 2 of this volume is broken into two sections:

- Windows Sockets 2
- Quality of Service (QOS)

This distinction between Windows Sockets 2 (Winsock 2) and QOS is achieved by the focus of the chapters, rather than the introduction of additional partitioning (such as dividing it into Part 2 and Part 3). This ensures consistency throughout the volumes in this library and in WPRS libraries in general. I've ensured that the chapter names clearly identify whether the contents focus on Winsock or QOS.

## Winsock

The first collection of chapters in Part 2 describes Windows Sockets 2 (Winsock) and enables application programmers to create advanced Internet, intranet, and other network-capable applications that transmit application data across the wire, independent of the network protocol being used. With Winsock, application programmers are provided access to advanced Windows networking capabilities such as multicast and QOS. Since Windows Sockets 2 is a continuation of the previous Windows Sockets programming standard, Windows Sockets 2 applications have the added feature of backward compatibility with Windows Sockets 1.1 applications.

Windows Sockets 2 uses the sockets paradigm that was first popularized by Berkeley Software Distribution (BSD) UNIX. It was later adapted for Microsoft Windows in Windows Sockets 1.1.

Because Windows Sockets 2 is an interface and not a protocol, it is capable of discovering and utilizing the communications capabilities of any number of underlying transport protocols.

The first chapters in Part 2 provide a complete treatment of the following:

- Windows Sockets API
- Windows Sockets SPI
- Windows Sockets Annex

## Quality of Service

QOS is an industry-wide initiative to enable more efficient use of the network. The IETF has provided many documents in the form of Internet Drafts and RFCs that outline such capabilities, including those provided by the Intserv, Diffserv, ISSLL, and RAP IETF working groups, among others. The goal of QOS is to provide preferential treatment to certain subsets of data, enabling such data to traverse the traditionally best-effort Internet or intranet with higher quality transmission service.

QOS in Microsoft Windows 2000 is a collection of components that enable such differentiation, preferential treatment, and management of higher quality data transmissions across the network. The collection of QOS components included in Windows 2000 constitutes the Microsoft Corporation implementation of the IETF vision of QOS.

The collection of QOS chapters in this volume covers the following information:

- QOS Overview
- Programming QOS
- QOS Functions
- Traffic Control (TC) Reference
- Local Policy Module (LPM) Reference





---

## 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:

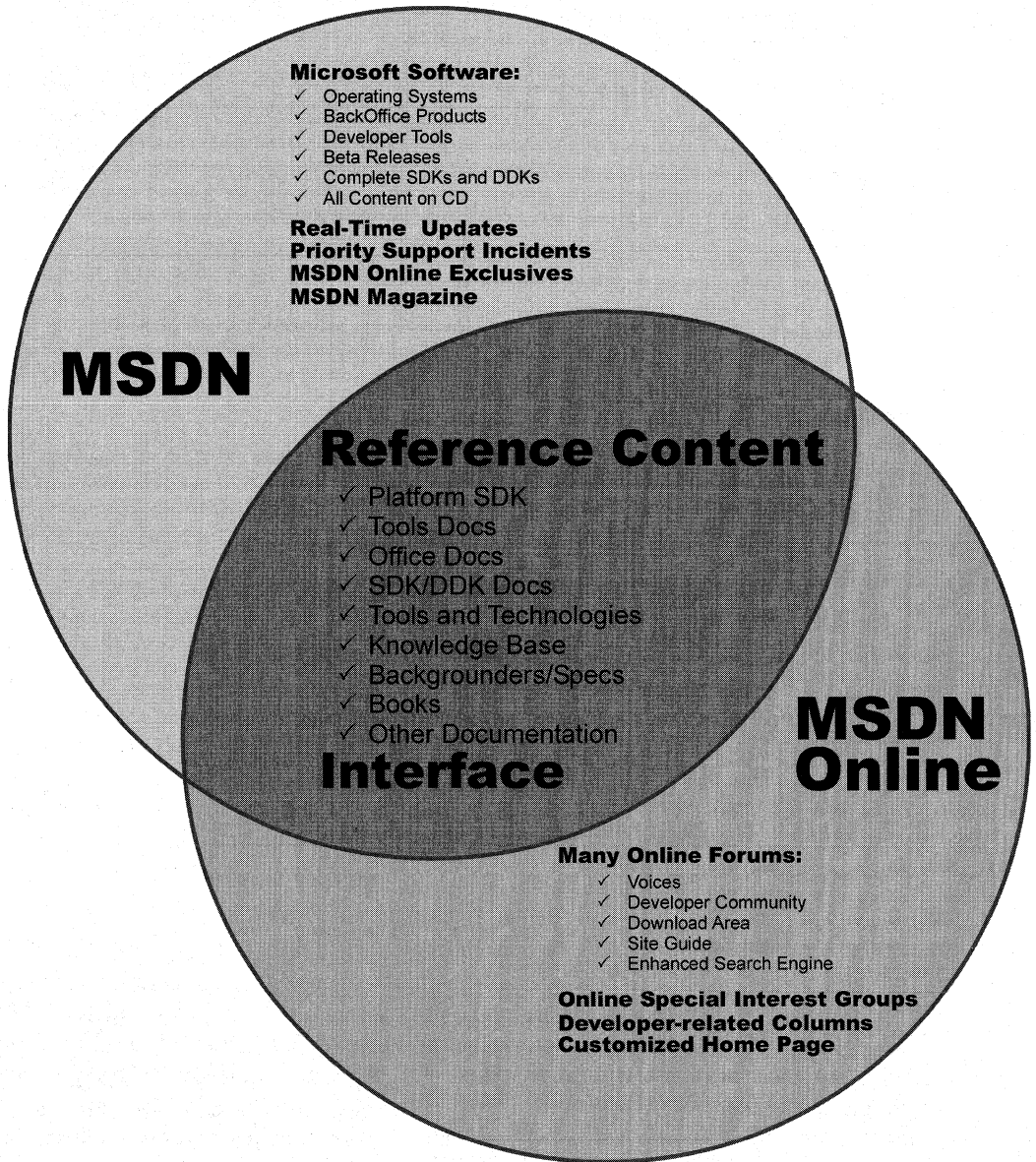
- MSDN provides reference content *and* the latest Microsoft product software, all shipped to its subscribers on CD or DVD.
- MSDN Online provides reference content *and* a development community forum, and is available only over the Internet.

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](http://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](http://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 Reference Library* on their desk and the MSDN Universal subscription on their desktop, and still come out \$50 ahead. That's the kind of math even accountants can like.

## Using MSDN

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

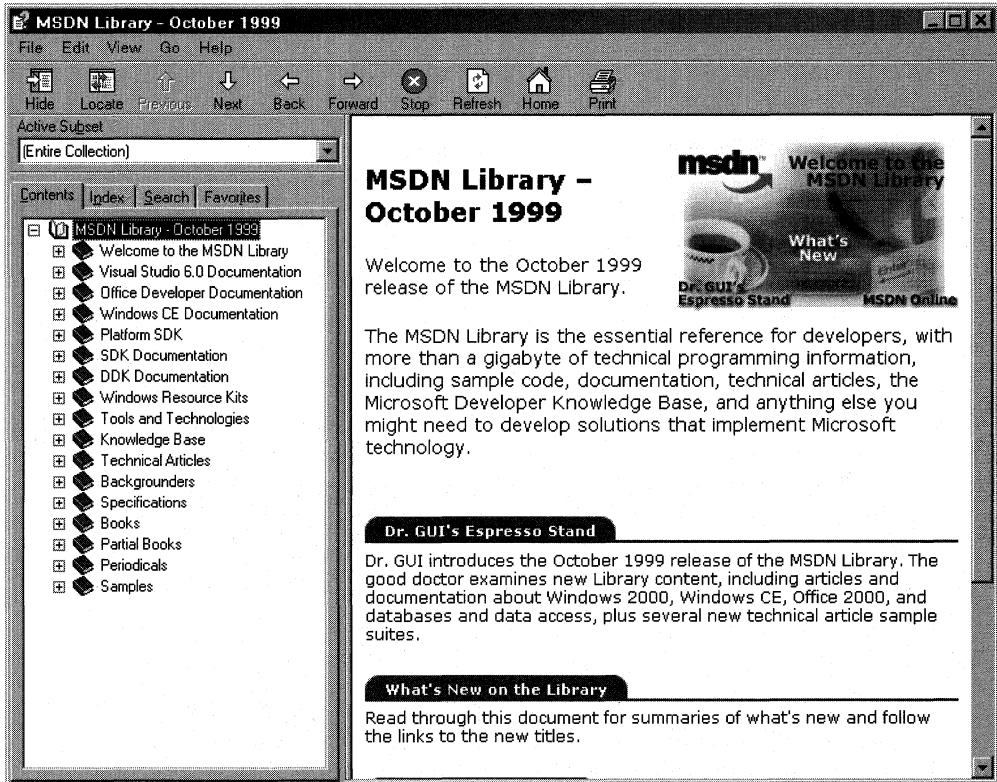
To those who have used MSDN, the interface shown in Figure 3-2 is likely familiar; it's the navigational front-end to MSDN reference material.

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



## Navigating MSDN

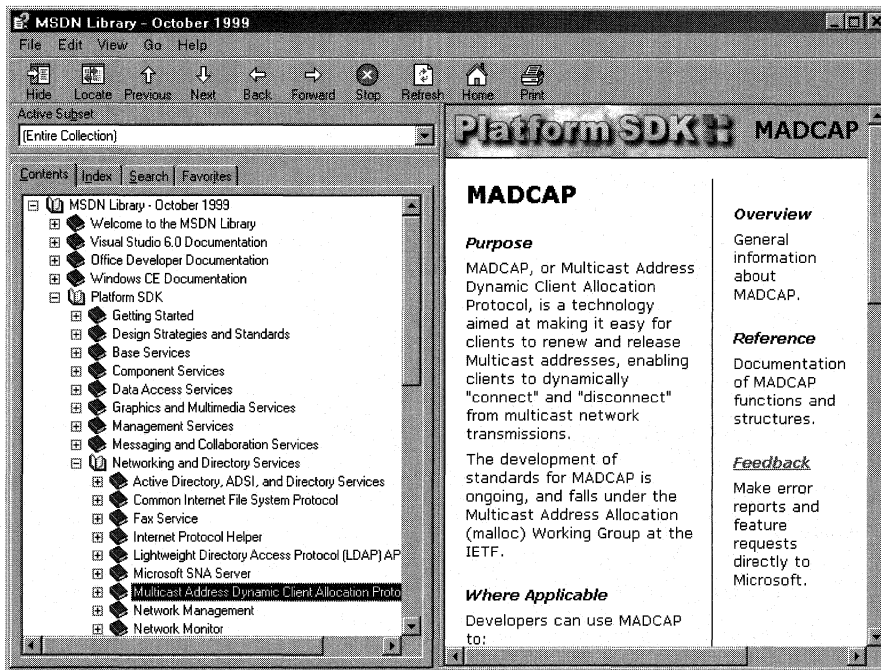
One of the primary features of MSDN—and to many, its primary drawback—is the sheer volume of information it contains, over 1.1GB and growing. The creators of MSDN likely realized this, though, and have taken steps to assuage the problem. Most of those steps relate to enabling developers to selectively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



**Figure 3-2: The MSDN interface.**

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

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



**Figure 3-3: Basic navigation through MSDN.**

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

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

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

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

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

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

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

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

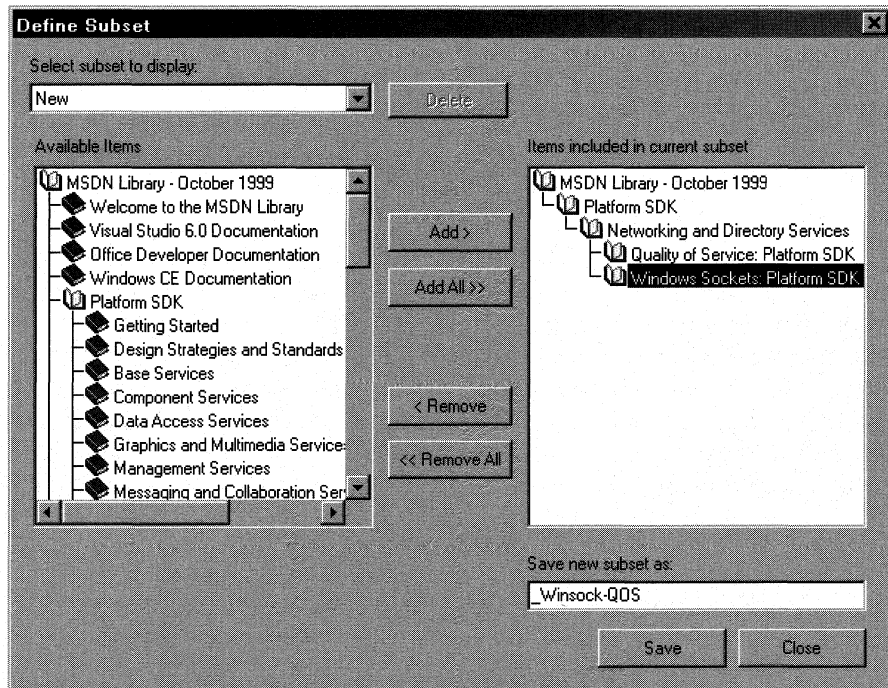
Once you have defined a subset, it becomes available in MSDN just like the predefined subsets, and filters the information available in the four Navigation Tabs, just like the predefined subsets do.

## Quick Tips

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

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

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



**Figure 3-4: The Define Subsets window.**

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

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

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

## Using MSDN Online

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

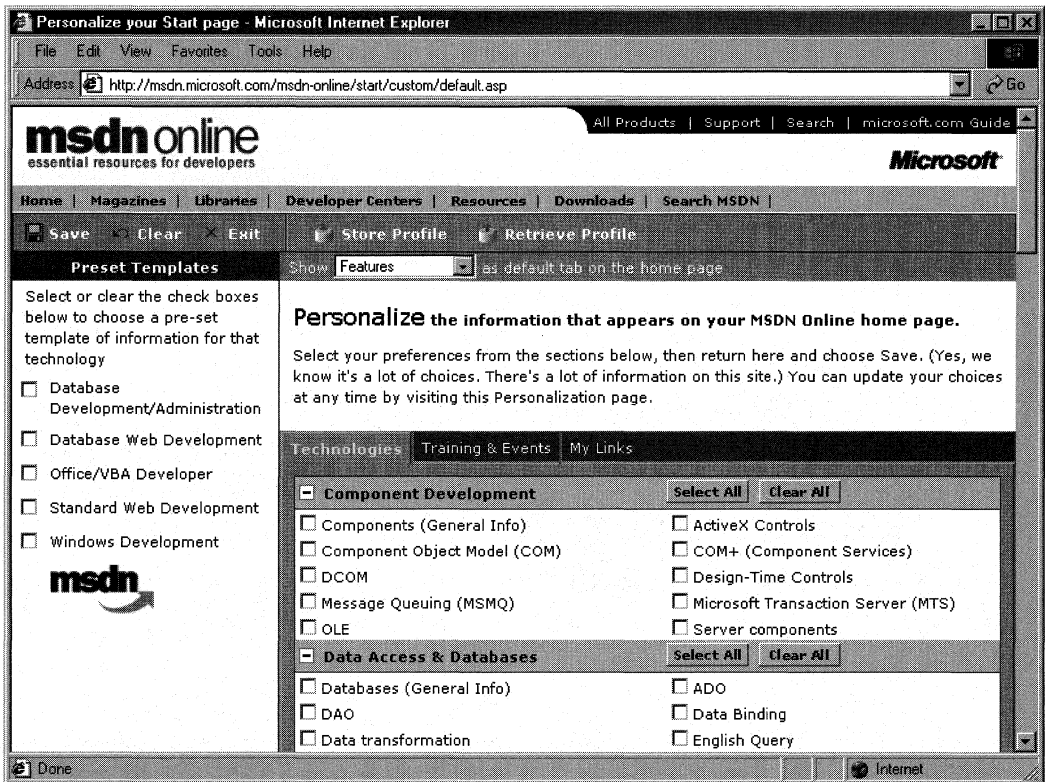
If you've used MSN's home page before ([www.msn.com](http://www.msn.com)), you're familiar with the fact that you can customize the page to your liking; choose from an assortment of available national news, computer news, local news, local weather, stock quotes, and other collections of information or 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](http://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](http://msdn.microsoft.com/msdn-online/start/custom). However you get there, the page you'll see is shown in Figure 3-5.

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

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

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



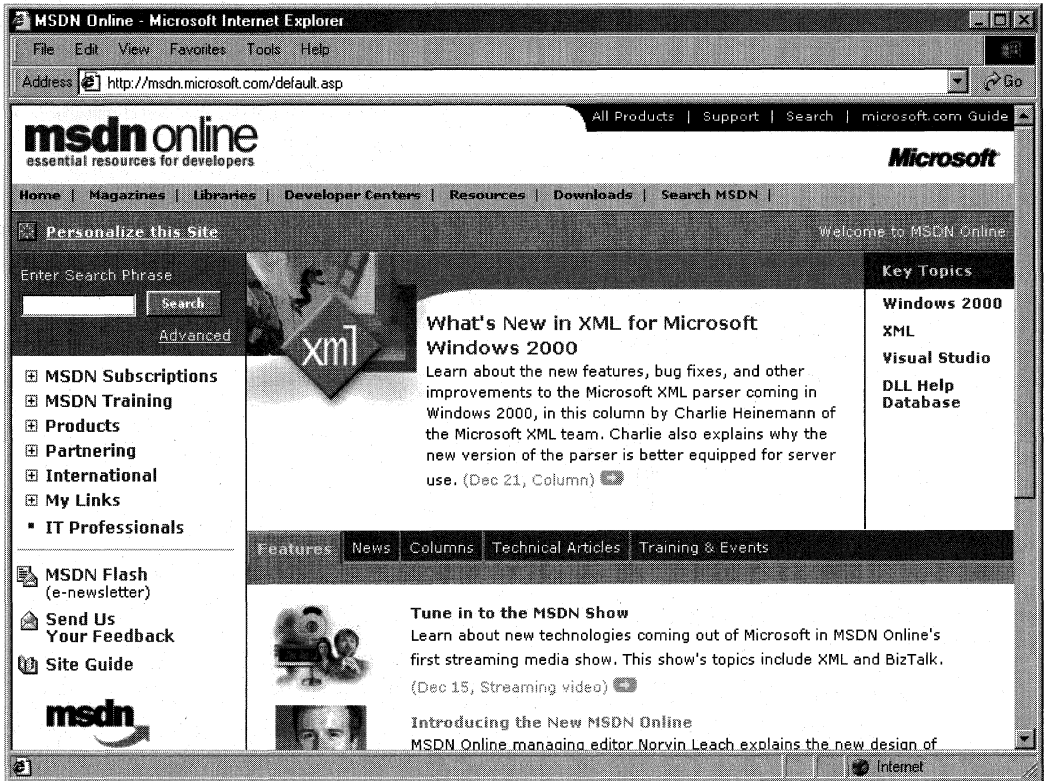
**Figure 3-5: The MSDN Online Personalize Page.**

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

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

## Navigating MSDN Online

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



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

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

- |                   |             |
|-------------------|-------------|
| Home              | 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](http://msdn.microsoft.com/resources/magazines.asp). The Magazines home page is shown in Figure 3-7.

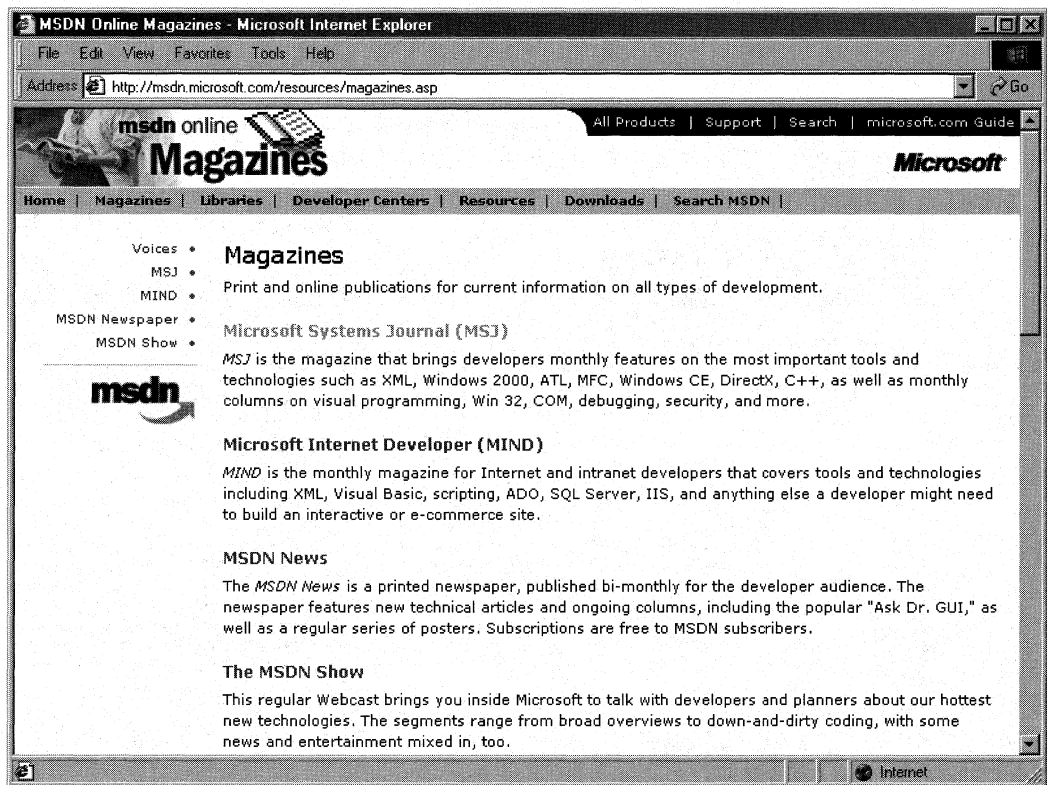


Figure 3-7: The Magazines Home Page.

For those of you familiar with the **Voices** feature section that formerly found its home on the MSDN Online navigation banner, don't worry; all content formerly in the Voices section is included the Magazines section as a subsite (or menu item, if you prefer) of the Magazines site. For those of you who aren't familiar with the Voices subsite, you'll



find a bunch of different articles or “voices” there, each of which adds its own particular twist on the issues that face developers. Both application and Web developers can get their fill of magazine-like articles from the sizable list of different articles available (and frequently refreshed) in the Voices subsite. With the combination of columns and online developer magazines offered in the Magazines section, you’re sure to find plenty of interesting insights.

**Libraries** is where the reference material available on MSDN Online lives. The Libraries site is divided into two sections: Library and Web Workshop. This distinction divides the reference material between Windows application development and Web development. Choosing Library from the Libraries menu takes you to a page through which you can navigate in traditional MSDN fashion, and gain access to traditional MSDN reference material. The Library home page can be linked to directly at [msdn.microsoft.com/library](http://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](http://msdn.microsoft.com/workshop).

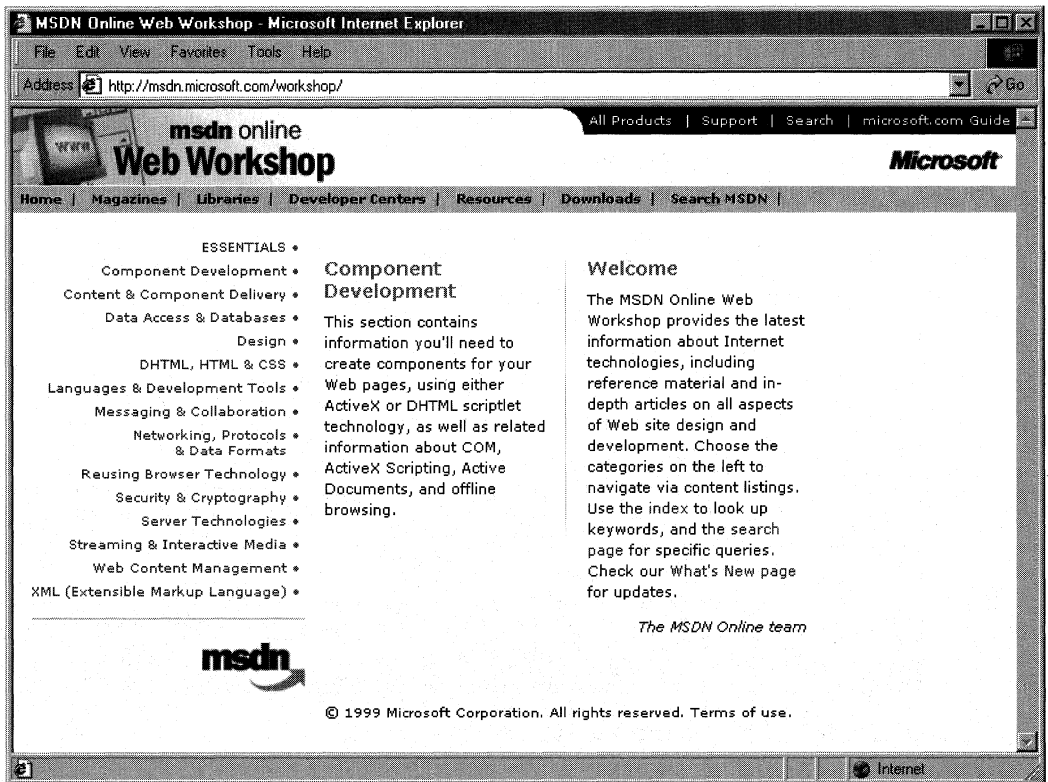
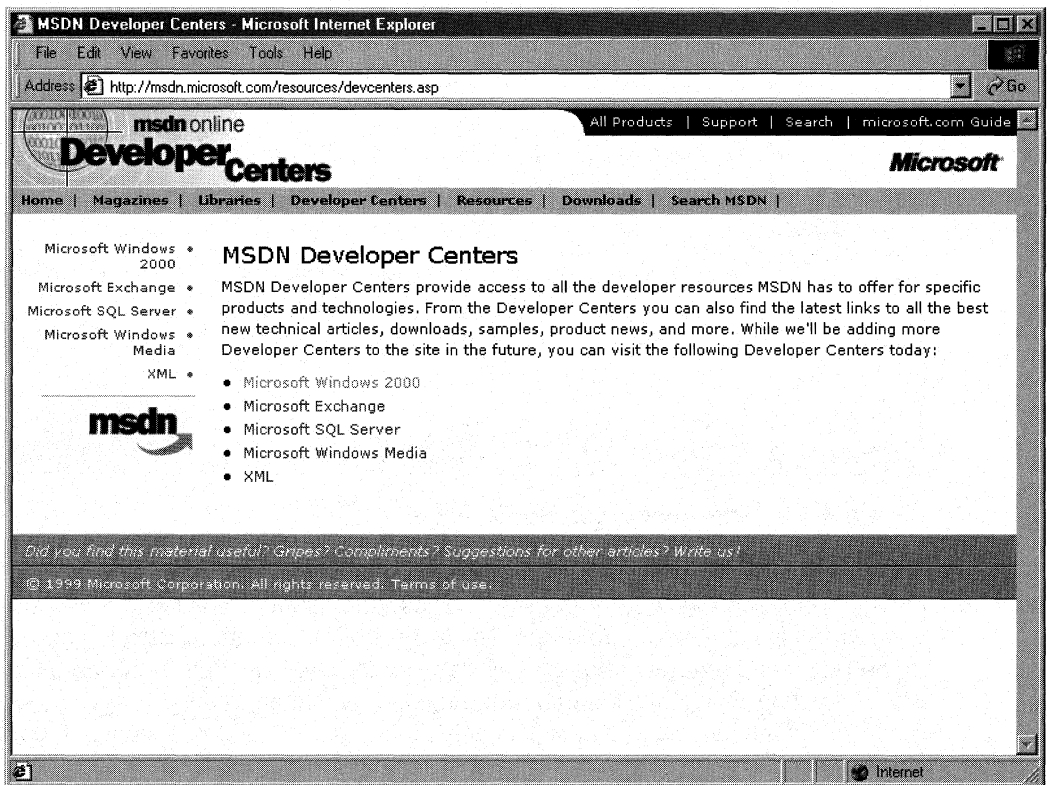


Figure 3-8: The Web Workshop Home Page.

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

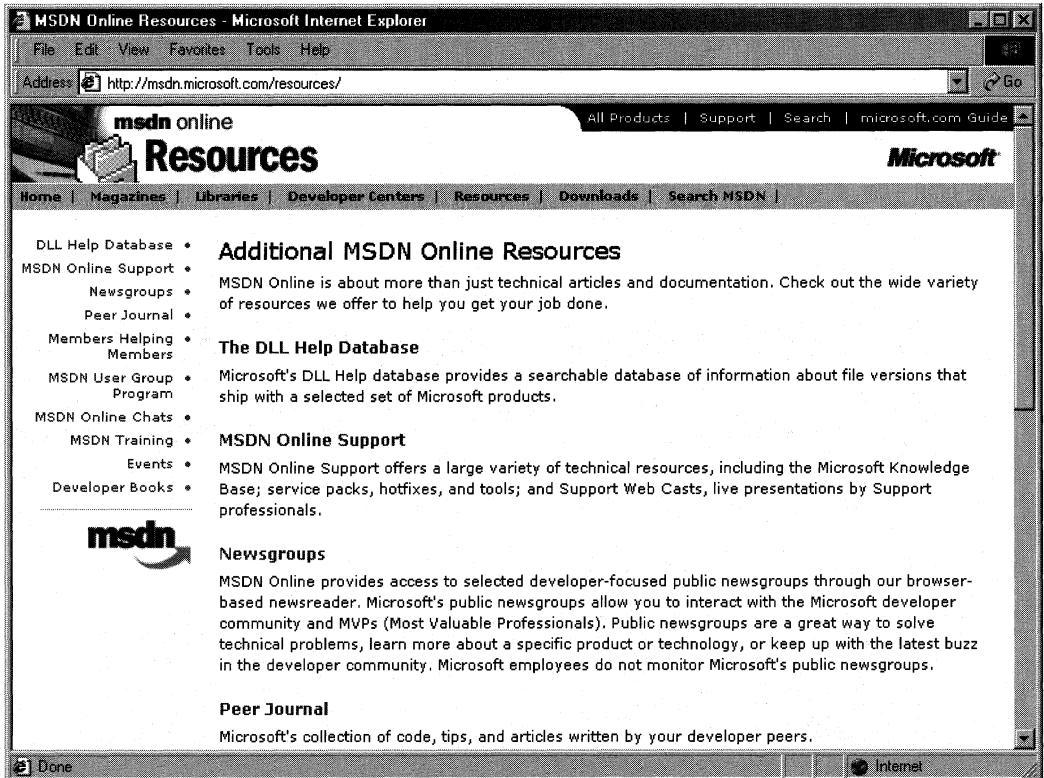
- Microsoft Windows 2000
- Microsoft Exchange
- Microsoft SQL Server
- Microsoft Windows Media
- XML

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



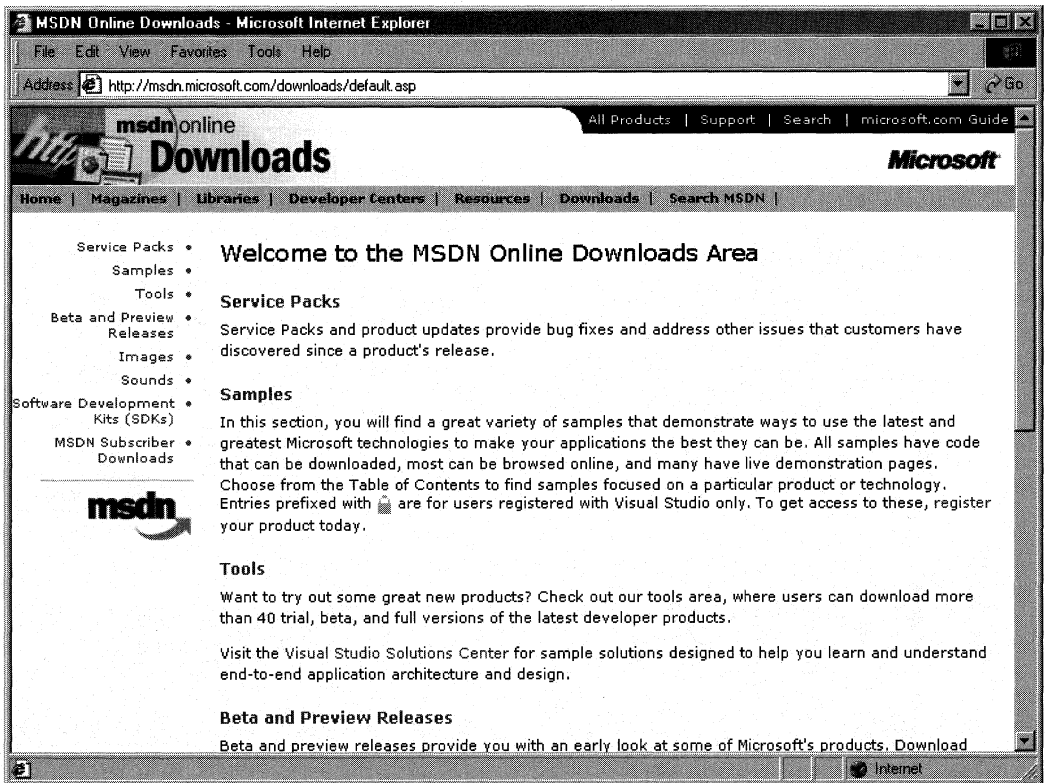
**Figure 3-9: The Developer Centers Home Page.**

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



**Figure 3-10: The Resources Home Page.**

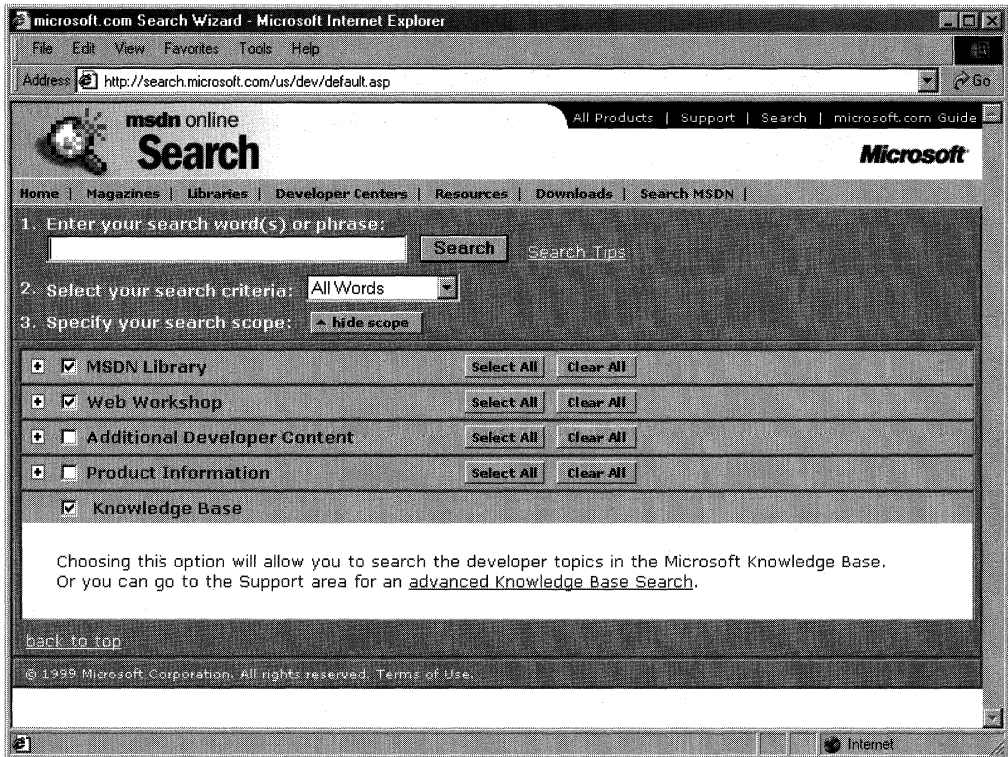
The **Downloads** site is where developers can find all sorts of useable items fit to be downloaded, such as tools, samples, images, and sounds. The Downloads site is also where MSDN subscribers go to get their subscription content updated over the Internet to the latest and greatest releases, as described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



**Figure 3-11: The Downloads Home Page.**

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

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



**Figure 3-12: The Search MSDN Home Page.**

The MSDN Online Member Community home page can be directly reached at [msdn.microsoft.com/community](http://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](http://msdn.microsoft.com/siteguide).

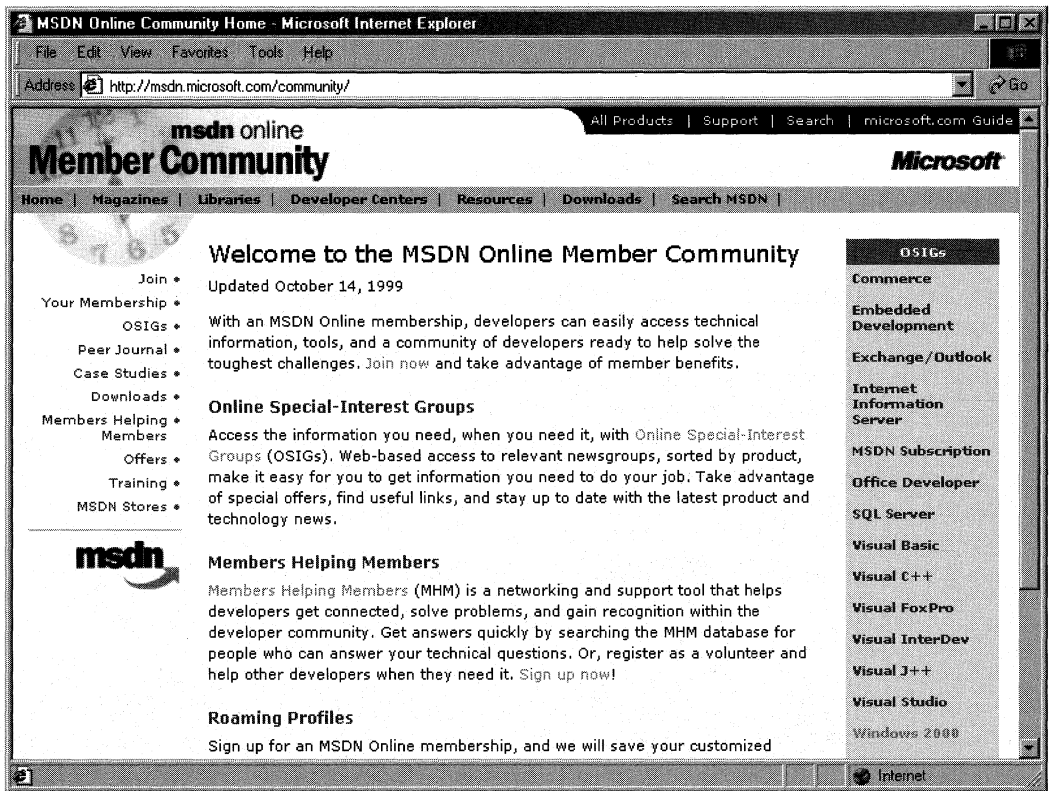


Figure 3-13: The MSDN Online Member Community Home Page.

## MSDN Online Registered Users

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

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

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

## The Windows Programming Reference Series

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

The WPRS currently includes the following libraries:

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

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

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

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

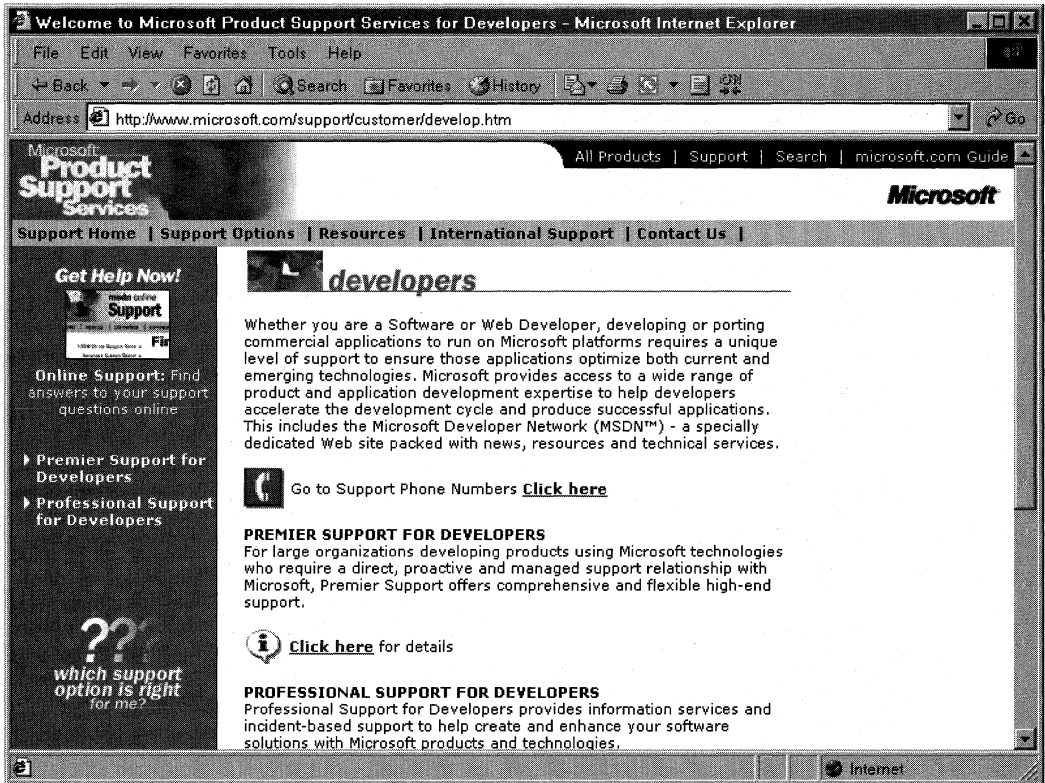
Microsoft provides developer resources through a number of different media, channels, and approaches. The extensiveness of Microsoft's resource offerings mirrors the fact that many are appropriate under various circumstances. For example, you wouldn't go to a conference to find the answer to a specific 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](http://www.microsoft.com/support/customer/develop.htm).

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





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

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

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

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

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

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

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

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

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

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

# Writing Great IrDA Applications (with Winsock)

This chapter provides overview and programming information about infrared technology, which is standardized by the Infrared Data Association (IrDA), a non-profit organization based in Walnut Creek, California. This chapter discusses how to use IrDA technology with Microsoft Windows applications (and therefore, with Winsock). IrDA is becoming increasingly popular as the digital revolution takes its networking capabilities into the wireless world, so IrDA information seemed especially useful for the *Networking Services Developer's Reference Library*.

A great resource for all things related to infrared networking technology is [www.irda.org](http://www.irda.org). Another great resource for finding out about how Microsoft uses (and exposes) IrDA technology can be found at [www.microsoft.com/hwdev/infrared](http://www.microsoft.com/hwdev/infrared).

## What Is an Ad-Hoc Networking-Enabled Application?

Imagine the following Windows application: two notebook computers are placed beside each other. A computer icon appears on both desktops with the name of the peer computer below it. Open one of the icons to display a folder with the contents of the peer computer's desktop. Drag and drop between your desktop and the open folder to move files between the two computers.

Imagine that the only configuration this application required was a checkbox for the user to enable or disable it. Imagine that several similar applications could be running at the same time without interfering with each other.

Imagine that this application could run on millions of existing notebook computers at transfer speeds of up to 4 Mb/s. Imagine that instances of the application, regardless of the speed of the underlying hardware, would work with all other instances at a common fastest speed.

Imagine that the other notebook computer in this example was a digital still camera, a handheld personal computer, a data capture device, or an electronic commerce device.

As a bonus, assume that the two computers did not need to be cabled together.

This application is possible today under Windows 2000, Windows 98, and Windows CE. The underlying technology is based on inexpensive, widely available, short-range infrared transceivers that adhere to the IrDA standards. IrDA standards also enable communications between non-Windows devices and Windows applications; these standards are freely available from the IrDA website at [www.irda.org](http://www.irda.org).



## What Is IrDA?

IrDA is an international organization that creates and promotes interoperable, low cost, infrared data interconnection standards that support a walk-up, point-to-point user model.

IrDA is a protocol suite designed to support transmission of data between two devices over short-range point-to-point infrared at speeds between 9.6Kbps and 4Mbps.

IrDA is that small semi-transparent red window that you may have wondered about on your notebook computer.

Over the last three years, the members of the IrDA have been very successful at getting IrDA hardware deployed in a large number of new notebook computers. One of the reasons for this has been the simplicity and low cost of IrDA hardware. Unfortunately, until recently, the hardware has not been available for applications programmers to use because of a lack of suitable protocol drivers.

Microsoft Windows CE 1.0 was the first Windows operating system to provide built-in IrDA support. Windows 2000 and Windows 98 now also include support for the same IrDA programming APIs that have enabled file sharing applications and games on Windows CE.

IrDA implementations are becoming available on several popular embedded operation systems.

SIR and FIR IrDA hardware can easily be added to a desktop computer by attaching a dongle to a serial port (SIR) or by adding a card and a dongle (FIR). In the future, USB-attached FIR IrDA devices will be available.

## What Is IrDA-C (Previously Known as IrBus)?

IrDA-C is a standard from the IrDA organization that is intended to support low speed wireless PC peripheral devices such as keyboards and joysticks. IrDA-C is a low speed (75 Kb/s shared among several devices), low latency, unreliable, long distance (20 feet), wide angle, multiple device protocol. IrDA-C is a PC-peripheral style bus protocol, and is not exposed to the application developer. In this chapter, the term "IrDA" refers to the IrDA1.1 protocols, which are also known as IrDA-D.

IrDA-C does not interoperate with IrDA-D. The IrDA-C specification includes a mechanism that could allow IrDA-C and IrDA-D to share hardware, but this mechanism is incompatible with the IrDA protocols in Windows 2000 and Windows CE because of performance enhancements in these implementations of the IrDA-D protocols. IrDA-C and IrDA-D can still exist in the same device, but must be physically separate.

## What Is Unique about IrDA?

IrDA is uniquely suited to ad-hoc point-to-point networking:

**IrDA is a great non-cable.** Mismatched connectors and wiring are impossible with IrDA. Speed and configuration parameters are transparently negotiated at connect time and a common set is used for the connection. IrDA at 4 Mb/s is compatible with 9.6 Kb/s IrDA. Additionally, the IrDA connector is completely sealed, inexpensive, and available from multiple vendors.

**Common user-space APIs.** The combination of IrDA and Windows Sockets presents the application programmer with a powerful yet simple Win32 user-space API that exposes multiple, fully error-corrected data streams. Serial and parallel ports are the only other point-to-point technologies that have a commonly available user space API. IrDA defines rich functionality that does not exist with serial and parallel cables. IrDA borrows from the very successful client/server connection and programming model defined by the TCP/IP family of protocols and the Winsock APIs.

**Open protocols support non-Windows devices.** Winsock exposes the IrDA TinyTP protocol to the application writer. A non-Windows device that implements the TinyTP protocol will be able to easily exchange data with Windows applications.

IrDA and Winsock support the implementation of easy to use, zero configuration, always works data sharing applications—ad-hoc point-to-point networking.

## IrDA Core Protocols and Services

The core IrDA services are similar to those exposed by the popular TCP protocol. Applications running on two different machines are able to easily open multiple reliable connections between themselves for the purpose of sending and receiving data. Like TCP, client applications connect to a server application by specifying a device (TCP *host*) address and an application (TCP *port*) address.

### Serial IrDA (SIR) Physical Layer (115 Kb/s)

The SIR specification defines a short-range infrared asynchronous serial transmission mode with one start bit, eight data bits and one stop bit. The maximum data rate is 115.2Kbps (half duplex). The primary benefit of this scheme is that existing serial hardware can be used very cheaply. This is one of the reasons for the widespread availability of IrDA.

### Fast IrDA (FIR) Physical Layer (4 Mb/s)

The FIR specification defines short-range low power operation at 4Mbps (half duplex). All FIR devices are also required to support SIR operation.

## IrLAP Data Link Layer

Data rates are negotiated and changed during IrLAP connection establishment. This is completely transparent to the application.

IrLAP defines two protocols. The first is a discovery protocol that is used directly by an application to learn about currently visible devices, their nicknames and device (MAC) addresses. Discovery is an area where IrDA differs from other common bus or networking protocols. It is not expected that clients have advance knowledge of the device addresses of servers—they simply ask for a list of who is visible and then select one.

The information returned from a discovery is a list of device MAC addresses, human readable device nicknames, and a bitmask of hints that suggest the nature of peer devices. MAC addresses are randomly generated 32-bit values.

The IrLAP data link protocol is based on the widely implemented HDLC data link protocol. IrLAP provides a simple reliable connection between two devices. If the HDLC connection is broken for any reason, an error is quickly reported back to applications.

## IrLMP and TinyTP

The IrLMP and TinyTP layers add multiple session support to IrLAP. In theory, this support exists to allow multiple applications to have multiple concurrent connections active. While this operation is fully supported, often only one application can be active at once. IrLMP and TinyTP still provide significant value in that they allow multiple applications to be listening for incoming connections without interfering with each other. A single application might also choose to open a control connection and a data connection at the same time. This is made possible by the IrLMP and TinyTP layers.

IrLMP and TinyTP also add per-connection flow control to the flow control provided by the single IrLAP connection. This allows an application to offer data in large blocks to the IrDA stack and allow the stack to send it at optimal speeds. It is not necessary for the application to be concerned about lost data or flow control.

Winsock exposes the TinyTP service interface to application programs.

IrLMP includes a directory services protocol, Information Advertising Service (IAS), that runs directly on top of IrLMP. IAS is commonly used to map an ASCII service name to a LSAP-SEL. LSAP-SEL is a protocol element used to select one from possibly many applications running on the server. The service name is a friendlier abstraction that is exposed to applications. Non-Windows devices must be aware of the conventions used by Windows. These are described below.

IrLMP defines a mode of service called exclusive mode. In this mode, only a single IrLMP connection is supported and the flow control features of IrLAP are used. TinyTP is not used. Exclusive mode is used by the IrLPT protocol that talks to IrLPT printers. (See Figure 5-1.)

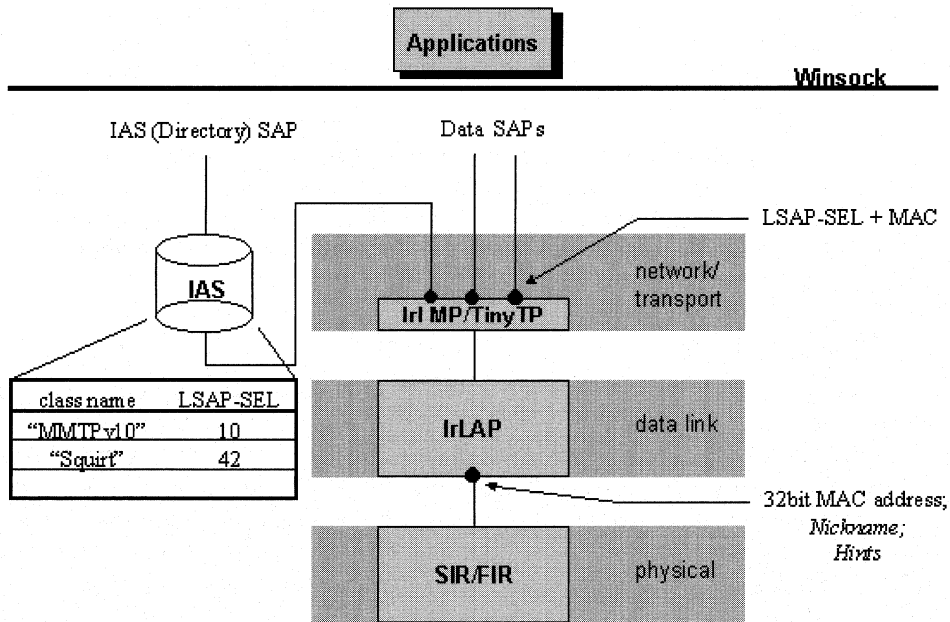


Figure 5-1: Core IrDA Protocols.

## IrCOMM

IrCOMM was the most frequently used programming interface for IrDA on Windows 95. Windows 98 continues to support IrCOMM, but Winsock is the strategic interface for IrDA on all Windows platforms. There are several reasons behind this decision.

IrCOMM is a family of protocols that run on top of the core IrDA protocol suite. IrCOMM supports the emulation of a peer device connected via a serial or parallel cable. This emulation is from the perspective of applications that are accessing serial and parallel ports through the operating system API.

An IrCOMM implementation generally takes the form of a system installable serial port driver. Rather than talking to real serial hardware, it uses the services exported by the core IrDA protocols.

The basic mechanism of IrCOMM serial-connected device emulation is to convert RS-232 serial line state change requests generated by one application into protocol messages that are communicated to the peer application through the native serial API.

In the case of modems, some of line state change protocol messages correspond to established conventions for relaying state changes on the analog side of the modem back to the computer (drop DCD to signal a line carrier drop). Other line state changes are used to stop the computer from overrunning buffers in the modem (CTS, RTS).

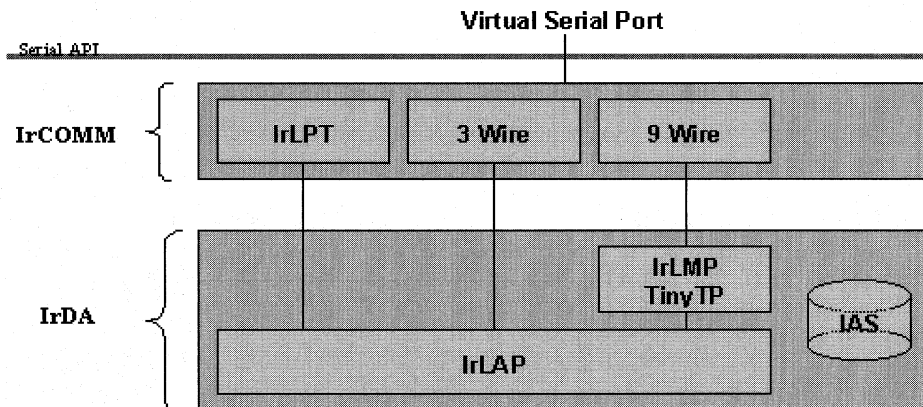
When two systems are connected together via an RS-232 cable (no modems), ad-hoc application conventions are used to map line state changes into bi-directional flow control and application specific signaling. The Windows 95 Direct Cable Connect feature is an example of such an application.

The use of line state changes to implement flow control is largely a legacy application to application concept. The underlying IrDA protocols fully support flow control between systems; even an IrDA modem could dispense with lead state changes and simply use the underlying IrDA protocol flow control mechanisms to stop the computer from sending more data.

## IrCOMM Modes

IrCOMM is actually a family of protocols. Only 9 Wire Mode supports the propagation of line state change information. The IrLPT protocol is used to talk to printers. 3 Wire Raw Mode and IrLPT run the core IrDA stack in exclusive mode, which precludes other applications from using the stack at the same time. 3 Wire Cooked Mode uses the services of IrLMP and TinyTP, does not preclude other applications from using the stack, and does not propagate line state change information. 9 Wire Mode is like 3 Wire Cooked Mode but also supports line state change messages.

In order to achieve IrCOMM communication, a common mode must be negotiated at IrCOMM connection setup time.



**Figure 5-2: IrCOMM Internal Architecture.**

The philosophy behind IrCOMM was to support IrDA modems and legacy applications built on the serial API. In practice, because the serial API was the most commonly available IrDA user space API, new IrDA applications were designed around IrCOMM.

IrCOMM runs on top of a reliable protocol layer, but session establishment and release services are not exposed through the serial API. Additionally, the underlying IrDA connection can be broken and re-established without this information being communicated to the application through the serial API. The result of this is that IrCOMM is not a reliable protocol in practice, and IrCOMM applications must be prepared to add yet another layer of reliability.

## No IrCOMM Virtual Serial Ports on Windows 2000

Windows 2000 does not expose IrCOMM virtual serial ports. There are several reasons for this.

Many customers have difficulty with the concept of virtual ports. This is especially confusing when the SIR IrDA hardware itself may need to be configured on a real serial port, and then the customer must further configure their application to use a virtual serial port.

Multiple applications cannot share a virtual serial port. This is particularly troublesome if, for example, an IrCOMM-based application opens the single virtual serial port and holds it open until system shutdown. An example of this would be an IrTran-P file transfer application running as a background service. No other IrDA application or driver will be able to run on that system, even though the underlying IrDA protocols provide support to allow multiple applications to be waiting for incoming connections, and allow clients to select a target application at connect time through established protocol mechanisms.

Windows 2000's support for multiple concurrent adapters and IrDA connections to different devices cannot be well supported under an API and protocol that assumes only a single device connection.

The complexity and various modes of IrCOMM make real world interoperability a very tough problem.

## Windows 2000 Support for IrCOMM Through Winsock

In order to support certain legacy IrDA devices, including IrTran-P based cameras, Windows 2000 implements a subset of the IrCOMM protocol, but exposes this protocol through the Winsock API rather than through the serial API. In particular, only 9 Wire Mode IrCOMM is supported, and line state change information is not supported. Windows 2000 only advertises that it supports the IrCOMM 9 Wire Mode. Devices must be able to initiate the IrDA connection, and not to expect Windows to initiate the connection as a side-effect of discovering new devices. IrCOMM through Winsock programming details are found later in this chapter.

The fundamental limitation of the IrCOMM protocol, that you cannot have multiple servers listening for incoming connections, is still exposed in this implementation. If one application is listening for incoming IrCOMM connections, another application trying to do so will get an error from Winsock. For this reason it is recommended that all new applications either avoid IrCOMM or support multiple mode concurrently.

## IrDA and the Windows Sockets API

Good Winsock applications are critical to the easy-to-use, zero-configuration, always-works IrDA vision. IrDA and Winsock present a unique opportunity for applications that can fill the ever-increasing need for simple data exchange between Windows and non-Windows devices.

### Talking to Non-Windows Devices

The Winsock programming API adds only a basic application-level naming convention (supported through IAS) to the IrDA specified protocol suite. A device that implements the core IrDA protocol and adheres to the simple naming conventions should be able to interoperate easily with Windows.

Devices are free to implement the required functionality of a Winsock client or server, or both. Client and server refer only to who initiates the connection. Once a connection is established, data can be reliably exchanged in both directions. Since server side functionality requires slightly more IrDA stack functionality, Windows will often be the server. It is up to the application writer to choose the trigger that initiates the IrDA connection. The connection can be driven from a user initiated action, or can be the result of discovering a device in a discovery polling loop. The application programmer is completely unconstrained as to client/server roles and connection establishment.

Devices that are battery constrained should refrain from continuous polling to drive connection initiation.

Either side can close the connection, although this is generally coordinated by application level protocols. A receiver will receive all data that has been sent to Winsock by the peer application before it receives notification of the peer connection closure. If the connection closes abortively, both sides of the connection will receive an error through Winsock. It is always possible to tell the difference between graceful closes and abortive closes.

Well-designed servers can often support multiple incoming connections concurrently, although there is no requirement that simple servers implement this.

### Application Addressing

Application level addressing is the ability of a client application to request a connection to a particular server application (actually to a particular socket, or communications endpoint). Multiple server applications can be waiting concurrently for incoming connections without interfering with one another.

The actual protocol, IrLMP, directly supports the concept of application level addressing through an 8-bit protocol field called an LSAP-SEL.

Since LSAP-SELs are only 8 bits wide and no authority coordinates this space, using a well-known LSAP-SEL to identify an application is not recommended. Windows uses the GetValueByClass IAS service to map an ASCII service name to an LSAP-SEL at connect time. This means that a server can register itself with an ASCII name, and a client can connect to this server by using the same name.

Non-Windows devices can connect to a Windows server by performing an IAS GetValueByClass query on the attribute IrDA:TinyTP:LsapSel with the desired service name as the class. The result of this query will be an LSAP-SEL, which the device can then use to initiate a TinyTP connection. The correct server will receive the incoming connection. The actual LSAP-SELs used in this scheme may change every time a server is restarted.

Non-Windows devices can support an inbound connection from a Windows system by supporting the same query initiated by the Windows side.

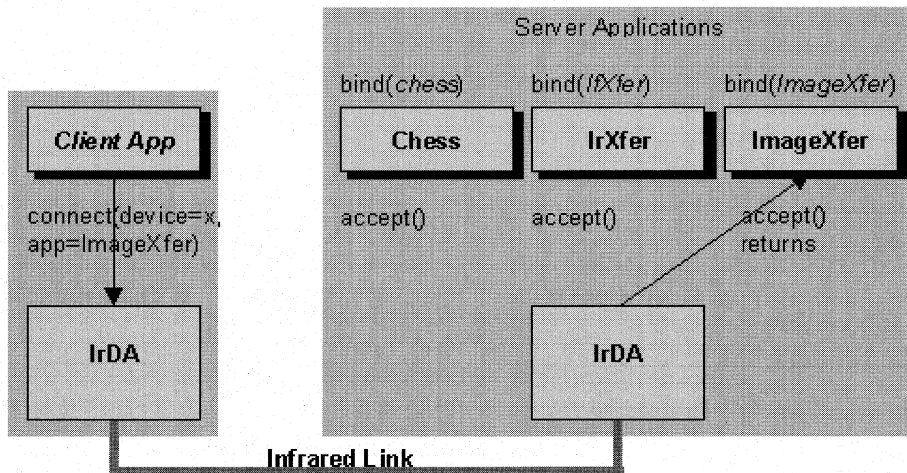


Figure 5-3: Application Addressing.

## Data Transfer and Connection Close

Once a connection is established, Winsock `send()` and `recv()` calls translate into TinyTP sends and receives. Even though IrLAP itself is half-duplex, the application is not aware of this. `Send()` and `recv()` can be called at the same time, on the same connection, on two different threads.

The stack manages TinyTP credits on behalf of the application. When the peer stops issuing TinyTP credits, the sender will block in the `send()` call. A non-Windows device is required to issue TinyTP credits as it is able to consume new data. Windows will stop issuing credits when the receiver stops calling `recv()` to consume data.



A Winsock application can send a large buffer of data on a `send()` call and the stack will segment it as required. Applications will get substantially higher performance if they pass in at least 8 kB of data on a single `send()` call.

IrDA through Winsock supports the `SOCK_STREAM` data stream semantics, which means that any notion of message boundaries is not preserved. Applications commonly add a length field to the head of messages to pass this information to a peer.

When one side of a connection is closed with the `closesocket()` call, all data that was previously sent will be delivered to the peer. When the peer has consumed all data, the next `recv()` call it issues will return with a length of zero, indicating a normal socket close. Any error that prevents data from being sent correctly will result in a Winsock error being returned to the application.

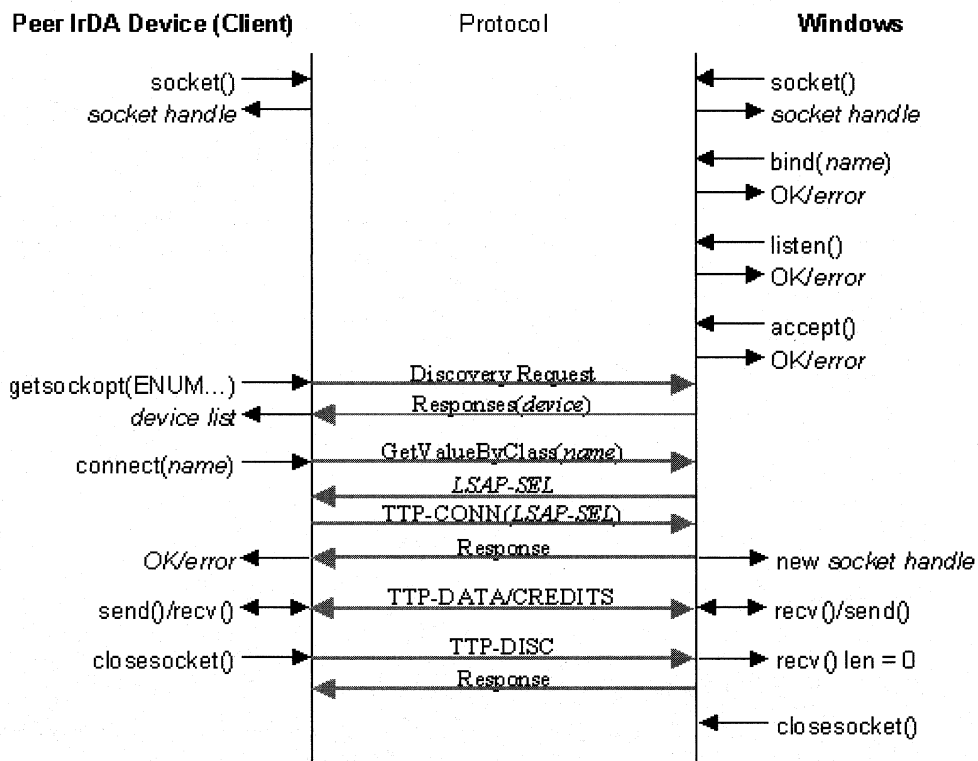


Figure 5-4: Winsock/IrDA Protocol Mapping.

## IrDA and Winsock Reference

This section explains how common Windows Sockets 2 functions are used when working with IrDA development. Note that this section is not a replacement of these functions' programmatic reference information found in Part 2 of this volume; these entries merely serve to specifically address IrDA-specific applications of these applications. For complete reference information on Windows Sockets 2 functions, see *Chapter 8*.

### WSAStartup

Call the **WSAStartup** function before making any other IrDA function calls:

```
WORD      WSAVerReq = MAKEWORD(1,1);
WSADATA   WSADATA;

if (WSAStartup(WSAVerReq, &WSADATA) != 0)
{
    // wrong winsock dlls?
}
```

### af\_irda.h

This header file must be included by Windows Sockets applications to support IrDA. There are several incompatible versions of `af_irda.h` that have been distributed with Windows CE and Windows 95 Ir3.0 DDKs and SDKs. A common `af_irda.h` that supports all three platforms is available with the Windows 2000 IrDA DDK. This file may continue to evolve as the APIs of the systems grow closer together.

In order to compile for one of the target platforms, one of the following must be defined:

```
_WIN32_WINNT
_WIN32_WCE
_WIN32_WINDOWS
```

### socket

The Windows Sockets **socket** function is used to create a connection endpoint of type **SOCKET**. This is nothing more than an application anchor for future references to a connection. The connection is not yet established. Both clients and servers begin all communication by opening a socket:

```
SOCKET ServSock;

if ((ServSock = socket(AF_IRDA, SOCK_STREAM, 0)) == INVALID_SOCKET)
{
    // WSAGetLastError()
}
```

## SOCKADDR\_IRDA Structure

The following **SOCKADDR** structure is used for AF\_IRDA sockets:

```
typedef struct _SOCKADDR_IRDA
{
    u_short    irdaAddressFamily;
    u_char    irdaDeviceID[4];
    char      irdaServiceName[25];
} SOCKADDR_IRDA, *PSOCKADDR_IRDA, FAR *LPSOCKADDR_IRDA;
```

The **irdaAddressFamily** member is always AF\_IRDA.

Server applications use the **irdaServiceName** member to specify their well-known service name in a **bind** function call. The **irdaDeviceID** member is ignored by the server application.

Client applications fill in all members. The **irdaDeviceID** member is filled in with the device address of the device that the client wishes to use a **connect** function call to connect to. This address is returned from a previous discovery operation initiated by a **getsockopt** function call with the IRLMP\_ENUMDEVICES option. The **irdaServiceName** member is initialized to the well-known value that the server specified in its **bind** function call.

It is an error for an IrDA application to issue a **connect** function call after issuing a **bind** call.

## bind

The **bind** function is used by server applications to register that they wish to receive incoming connections that are addressed to a specified service name on the specified socket. The **bind** function associates a server socket with an application level address:

```
SOCKADDR_IRDA    ServSockAddr = { AF_IRDA, 0, 0, 0, 0, "SampleIrDAService" };
int              SizeOfSockAddr;

if (bind(ServSock, (const struct sockaddr *) &ServSockAddr,
        sizeof(SOCKADDR_IRDA)) == SOCKET_ERROR)
{
    // WSAGetLastError()
}
```

A Windows Sockets **bind** function call causes the stack to generate a new local LSAP-SEL and to add it to the IAS database associated with the service name supplied in the **SOCKADDR\_IRDA** structure.

## listen

The **listen** function is used by server applications to place the stack into a mode where it will receive incoming connections on that socket. The **listen** function does not block. The *backlog* parameter tells the stack how many inbound connections to accept on behalf of the application before the application is able to further process (**accept**) these connections:

```
if (listen(ServSock, 2) == SOCKET_ERROR)
{
    // WSAGetLastError()
}
```

## accept

Once a server application has put its server socket into listen mode, it then calls the **accept** function and blocks until an incoming connection is received. An unusual characteristic of the **accept** function is that it returns a new socket. The reason for this is that the server application may wish to continue accepting new inbound connections. To support this, the server typically creates a new thread to handle the new connection, and then blocks again on another **accept** call. There is no requirement that a simple server support multiple connections, and it is free to ignore the old listening socket until it is done with the newly created socket. The **SOCKADDR\_IRDA** structure passed to the **accept** function is filled in with the peer's addresses, and can usually be ignored:

```
SOCKET NewSock;

while(1)
{
    sizeofSockAddr = sizeof(SOCKADDR_IRDA);

    if ((NewSock = accept(ServSock, (struct sockaddr *) &PeerSockAddr,
        &sizeofSockAddr)) == INVALID_SOCKET)
    {
        // WSAGetLastError()
        // exit
    }

    // NewSock is a connected socket
    // create a new thread and pass it NewSock, return to
    // accept() on main or use NewSock here until done, then close it
}
```

## send and recv

These function calls are used to transfer data. A **recv** function call blocks until there is data available, and a **send** function call normally does not block. The **send** function can block if the peer is not receiving data (has stopped calling **recv**). The **send** call unblocks when the peer resumes **recvs**. A **recv** of length zero has the special meaning that the client has performed a graceful close on the socket. The application can assume that it has received all data that was sent by the peer. If any unrecoverable protocol error occurs during the connection, **send** or **recv** returns an error code and the connection is aborted:

```
int  BytesRead, BytesSent;
char Buffer[4096];

// recv() example
if ((BytesRead = recv(Sock, Buffer, sizeof(Buffer), 0)) == SOCKET_ERROR)
{
    // WSAGetLastError()
}
if (BytesRead == 0)
{
    // peer has closed the connection and I have all the data
    // close the socket now
}

// send() example
if ((BytesSent = send(Sock, Buffer, sizeof(Buffer), 0)) == SOCKET_ERROR)
{
    // WSAGetLastError()
}
// check that BytesSent == sizeof(Buffer)
```

## closesocket

The **closesocket** function initiates a graceful close on the connection and releases the socket handle:

```
if (closesocket(Sock) == SOCKET_ERROR)
{
    // WSAGetLastError()
}
```

## getsockopt(, IRLMP\_ENUMDEVICES,,) and connect()

This is the function used to perform a discovery. Before a connection can be initiated, a device address must be obtained by doing a discovery operation. An extension to the Windows Sockets **getsockopt** function call returns a list of all currently visible IrDA devices. A device address returned from this list is copied into a **SOCKADDR\_IRDA** structure to be used by the **connect** function call.



Discovery can be run in one of two ways. Performing a **getsockopt** function call with the **IRLMP\_ENUMDEVICES** option will cause a single discovery to be run on each idle adapter. The list of discovered devices and cached devices (on active adapters) is returned immediately. The following code demonstrates this:

```
SOCKADDR_IRDA DestSockAddr = { AF_IRDA, 0, 0, 0, 0, "SampleIrDAService" };

#define DEVICE_LIST_LEN 10

unsigned char DevListBuff[sizeof(DEVICELIST) -
    sizeof(IRDA_DEVICE_INFO) +
    (sizeof(IRDA_DEVICE_INFO) * DEVICE_LIST_LEN)];
int DevListLen = sizeof(DevListBuff);
PDEVICELIST pDevList = (PDEVICELIST) &DevListBuff;

pDevList->numDevice = 0;

// Sock is not in connected state
if (getsockopt(Sock, SOL_IRLMP, IRLMP_ENUMDEVICES,
    (char *) pDevList, &DevListLen) == SOCKET_ERROR)
{
    // WSAGetLastError
}

if (pDevList->numDevice == 0)
{
    // no devices discovered or cached
    // not a bad idea to run a couple of times
}
else
{
    // one per discovered device
    for (i = 0; i < (int) pDevList->numDevice; i++)
    {
        // typedef struct _IRDA_DEVICE_INFO
        // {
        //     u_char irdaDeviceID[4];
        //     char irdaDeviceName[22];
        //     u_char irdaDeviceHints1;
        //     u_char irdaDeviceHints2;
        //     u_char irdaCharSet;
        // } _IRDA_DEVICE_INFO;

        // pDevList->Device[i]. see _IRDA_DEVICE_INFO for fields
    }
}
```

(continued)

(continued)

```

    // display the device names and let the user select one
    }
}

// assume the user selected the first device [0]
memcpy(&DestSockAddr.irdaDeviceID[0], &pDevList->Device[0].irdaDeviceID[0], 4);

if (connect(Sock, (const struct sockaddr *) &DestSockAddr,
            sizeof(SOCKADDR_IRDA)) == SOCKET_ERROR)
{
    // WSAGetLastError
}

```

It is also possible to run a lazy discovery—the application will not be notified until the discovered device list changes from the last discovery run by the stack.

## IAS

Limited access to the IAS database is available through Winsock, but this is not normally used by applications. It exists to support connections to non-Windows devices that are not compliant with the Winsock/IrDA conventions.

This structure is used with the IRLMP\_IAS\_SET **setsockopt** option to manage the local IAS database:

```

typedef struct _IAS_SET
{
    char    irdaClassName[IAS_MAX_CLASSNAME];
    char    irdaAttribName[IAS_MAX_ATTRIBNAME];
    u_long  irdaAttribType;
    union
    {
        {
            LONG irdaAttribInt;
            struct
            {
                {
                    u_short Len;
                    u_char  OctetSeq[IAS_MAX_OCTET_STRING];
                } irdaAttribOctetSeq;
                struct
                {
                    u_char Len;
                    u_char CharSet;
                    u_char  UserStr[IAS_MAX_USER_STRING];
                } irdaAttribUserStr;
            } irdaAttribute;
        }
    }
} IAS_SET, *PIAS_SET, FAR *LPIAS_SET;

```

This structure is used with the IRLMP\_IAS\_QUERY **getsockopt** option to query a peer's IAS database:

```
typedef struct _WINDOWS_IAS_QUERY
{
    u_char irdaDeviceID[4];
    char irdaClassName[IAS_MAX_CLASSNAME];
    char irdaAttribName[IAS_MAX_ATTRIBNAME];
    u_long irdaAttribType;
    union
    {
        {
            LONG irdaAttribInt;
            struct
            {
                {
                    u_long Len;
                    u_char OctetSeq[IAS_MAX_OCTET_STRING];
                } irdaAttribOctetSeq;
            } irdaAttribOctetSeq;
            struct
            {
                {
                    u_long Len;
                    u_long CharSet;
                    u_char UsrStr[IAS_MAX_USER_STRING];
                } irdaAttribUsrStr;
            } irdaAttribute;
        }
    } IAS_QUERY, *PIAS_QUERY, FAR *LPIAS_QUERY;
```

The following code shows the steps necessary to build a server that listens for incoming IrCOMM connections:

```
#define IAS_SET_ATTRIB_MAX_LEN 32

// buffer for IAS set
BYTE IASSetBuff[sizeof(IAS_SET) - 3 + IAS_SET_ATTRIB_MAX_LEN];
Int IASSetLen = sizeof(IASSetBuff);
PIAS_SET pIASSet = (PIAS_SET) &IASSetBuff;

// for the setsockopt call to enable 9 wire IrCOMM
int Enable9WireMode = 1;

// server sockaddr with IrCOMM name
SOCKADDR_IRDA ServSockAddr = { AF_IRDA, 0, 0, 0, 0, "IrDA:IrCOMM" };
SOCKADDR_IRDA PeerSockAddr;
int sizeofSockAddr;

SOCKET ServSock;
```

(continued)



*(continued)*

```

SOCKET      NewSock;

if ((ServSock = socket(AF_IRDA, SOCK_STREAM, 0)) == INVALID_SOCKET)
{
    // WSAGetLastError
}

// add IrCOMM IAS attributes for 3 wire cooked and 9 wire raw.
// see IrCOMM spec
memcpy(&piASSet->irdaClassName[0], "IrDA:IrCOMM", 12);
memcpy(&piASSet->irdaAttribName[0], "Parameters", 11);

piASSet->irdaAttribType          = IAS_ATTRIB_OCTETSEQ;
piASSet->irdaAttribute.irdaAttribOctetSeq.Len = 6;

memcpy(&piASSet->irdaAttribute.irdaAttribOctetSeq.OctetSeq[0],
      "\000\001\006\001\001\001", 6);

if (setsockopt(ServSock, SOL_IRLMP, IRLMP_IAS_SET, (const char *) piASSet,
IASSetLen)
    == SOCKET_ERROR)
{
    // WSAGetLastError
}

// enable 9wire mode before bind()
if (setsockopt(ServSock, SOL_IRLMP, IRLMP_9WIRE_MODE, (const char *)
&Enable9WireMode,
      sizeof(int)) == SOCKET_ERROR)
{
    // WSAGetLastError
}

if (bind(ServSock, (const struct sockaddr *) &ServSockAddr,
sizeof(SOCKADDR_IRDA))
    == SOCKET_ERROR)
{
    // WSAGetLastError
}

// nothing special for IrCOMM from now on...
if (listen(ServSock, SERV_BACKLOG) == SOCKET_ERROR)
{
    // WSAGetLastError
}

```

## IrCOMM Client

The following code shows the steps necessary to build a client that connects via 9 Wire IrCOMM:

```
#define DEVICE_LIST_LEN 5

// discovery buffer
BYTE      DevListBuff[sizeof(DEVICELIST) - sizeof(IRDA_DEVICE_INFO) +
                      (sizeof(IRDA_DEVICE_INFO) * DEVICE_LIST_LEN)];
int       DevListLen = sizeof(DevListBuff);
PDEVICELIST pDevList = (PDEVICELIST) &DevListBuff;
int       DevNum;

#define IAS_QUERY_ATTRIB_MAX_LEN 32

// buffer for IAS query
BYTE      IASQueryBuff[sizeof(IAS_QUERY) - 3 + IAS_QUERY_ATTRIB_MAX_LEN];
int       IASQueryLen = sizeof(IASQueryBuff);
PIAS_QUERY pIASQuery = (PIAS_QUERY) &IASQueryBuff;

// for searching through peers IAS response
BOOL      Found = FALSE;
UCHAR     *pPI, *pPL, *pPV;

// for the setsockopt call to enable 9 wire IrCOMM
int       Enable9WireMode = 1;

SOCKADDR_IRDA DstAddrIR = { AF_IRDA, 0, 0, 0, 0, "IrDA:IrCOMM" };

if ((pConn->Sock = socket(AF_IRDA, SOCK_STREAM, 0)) == INVALID_SOCKET)
{
    // WSAGetLastError
}

// search for the peer device
pDevList->numDevice = 0;
if (getsockopt(pConn->Sock, SOL_IRLMP, IRLMP_ENUMDEVICES, (CHAR *) pDevList,
&DevListLen)
    == SOCKET_ERROR)
{
    // WSAGetLastError
}
```

*(continued)*



(continued)

```
// if (pDevList->numDevice == 0)
{
    // no devices found, tell the user
}

// assume first device, we should have a common dialog here
memcpy(&DstAddrIR.irdaDeviceID[0], &pDevList->Device[0].irdaDeviceID[0], 4);

// query the peer to check for 9wire IrCOMM support
memcpy(&pIASQuery->irdaDeviceID[0], &pDevList->Device[0].irdaDeviceID[0], 4);

// IrCOMM IAS attributes
memcpy(&pIASQuery->irdaClassName[0], "IrDA:IrCOMM", 12);
memcpy(&pIASQuery->irdaAttribName[0], "Parameters", 11);

if (getsockopt(pConn->Sock, SOL_IRLMP, IRLMP_IAS_QUERY, (char *) pIASQuery,
    &IASQueryLen) == SOCKET_ERROR)
{
    // WSAGetLastError
}

if (pIASQuery->irdaAttribType != IAS_ATTRIB_OCTETSEQ)
{
    // peer's IAS database entry for IrCOMM is bad
    // error
}

if (pIASQuery->irdaAttribute.irdaAttribOctetSeq.Len < 3)
{
    // peer's IAS database entry for IrCOMM is bad
    // error
}

// search for the PI value 0x00 and check 9 wire, see IrCOMM spec.
pPI = pIASQuery->irdaAttribute.irdaAttribOctetSeq.OctetSeq;
pPL = pPI + 1;
pPV = pPI + 2;

while (1)
{
    if (*pPI == 0 && (*pPV & 0x04))
    {
        Found = TRUE;
        break;
    }
}
```

```
    }

    if (pPL + *pPL >= pIASQuery->irdaAttribute.irdaAttribOctetSeq.OctetSeq +
        pIASQuery->irdaAttribute.irdaAttribOctetSeq.Len)
    {
        break;
    }

    pPI = pPL + *pPL;
    pPL = pPI + 1;
    pPV = pPI + 2;
}

if (! Found)
{
    // peer doesn't support 9 wire mode
    // error
}

// enable 9wire mode before connect()
if (setsockopt(ServSock, SOL_IRLMP, IRLMP_9WIRE_MODE, (const char *)
&Enable9WireMode,
    sizeof(int)) == SOCKET_ERROR)
{
    // WSAGetLastError
}

// nothing special for IrCOMM from now on...
if (connect(pConn->Sock, (const struct sockaddr *) &DstAddrIR,
    sizeof(SOCKADDR_IRDA))
    == SOCKET_ERROR)
{
    // WSAGetLastError
}
```

# Windows 2000 IrDA Architecture

Windows 2000 provides some unique support for IrDA.

## IrDA Hardware Drivers

SIR UART based serial adapters are supported by the Windows 2000 component IrSIR.SYS. IrSIR uses the services of the Windows NT serial driver SERIAL.SYS or a SERIAL.SYS-compatible serial driver to communicate with the IrDA hardware. Built-in SIR hardware should expose itself through the system BIOS as Plug and Play Id PNP0510 or PNP0511.

FIR IrDA hardware must be exposed as an NDIS4.0 miniport driver. FIR drivers can expose as many NDIS adapters as the driver can support. Each adapter is a unique IrDA transceiver that can support a unique instance of IrLAP. FIR hardware should have a unique Plug and Play ID and have an associated vendor-supplied driver. FIR hardware that is also compatible with SIR can also expose an alias Plug and Play ID of PNP0510 or PNP0511 to allow SIR-only operation using IrSIR.

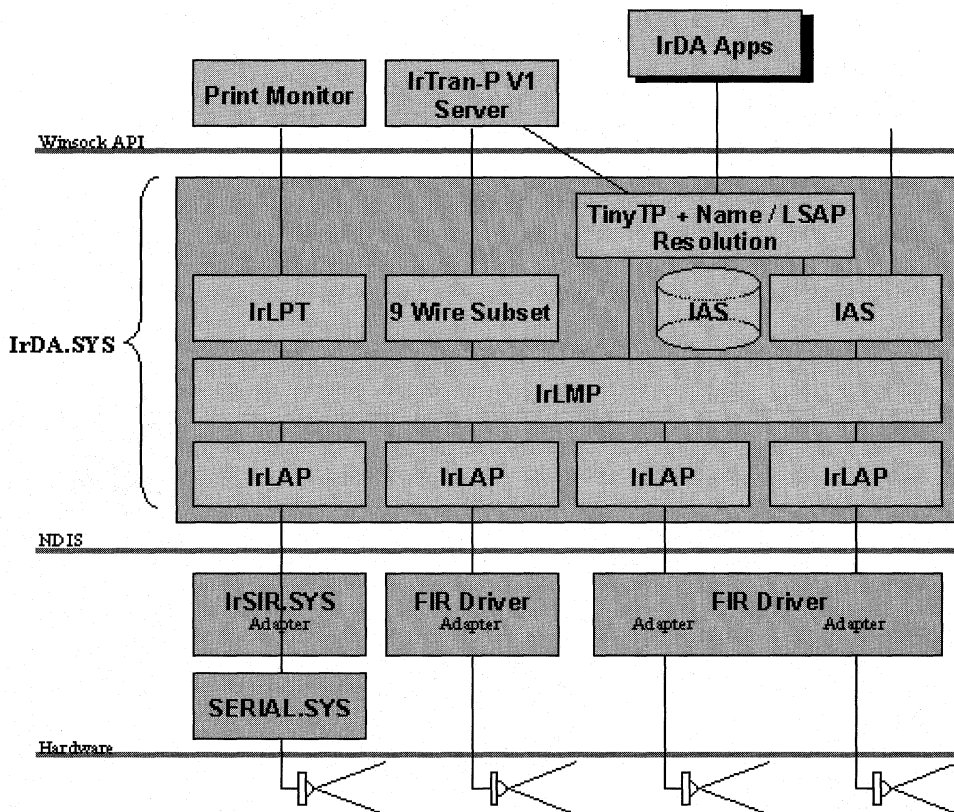


Figure 5-5: Windows 2000 IrDA Architecture.

## Windows 2000 Multiple-Adapter Support

The Windows 2000 IrDA stack supports concurrent operation of several NDIS4.0 FIR/SIR miniport adapters. This support allows a single server to support multiple inbound connections in a way that is transparent to both client and server applications.

An adapter is defined as the hardware/software needed to support a single IrLAP connection.

Since IrDA is not a routable protocol, multiple-adapters support is limited to connections to a single server by multiple clients. Peer devices cannot talk to each other through the server.

Each adapter and IrLAP instance will have a unique IrDA MAC address (DeviceID).

Discovery operations are run on every idle adapter in sequence. A global list of discovered devices is returned. Cached discovery information is maintained on a per-IrLAP instance and returned for each adapter that has a connection active.

The Windows 2000 IrDA stack maintains a mapping between device address and last seen adapter. When a connection is requested to a peer device, the stack routes the connection to the correct adapter. Incoming connections are delivered to a single listening transport endpoint. The listening client will not receive any per-adapter information. The mapping between the new connection and the adapter is maintained by the IrDA stack.



---

## CHAPTER 6

# Winsock 2 API Overview

## Welcome to Windows Sockets 2

This chapter describes the Windows Sockets 2 Application Programming Interface (API). It consists, primarily, of information from the Windows Sockets 2 API specification, but also includes additional information. The information in this document is not presented in exactly the same way as in the specification.

## Using the Windows Sockets 2 API Document

This document provides the online material needed to create a Windows Sockets application for Microsoft Windows® operating systems, using the Microsoft implementation of Windows Sockets 2. It is intended as a reference tool and outlines the functions in the Windows Sockets API.

You should be familiar with Win32® programming concepts to make the best use of this document. Thus, you may want to refer to other references that provide a more systematic guide to writing Windows Sockets applications.

---

**Note** This documentation is intended for application developers. If you are developing a transport or service provider, see the Service Provider Documentation in Chapters 10 and 11 of this volume.

---

## Overview of Windows Sockets 2

Windows Sockets 2 uses the sockets paradigm that was first popularized by Berkeley Software Distribution (BSD) UNIX. It was later adapted for Microsoft Windows in Windows Sockets 1.1.

One of the primary goals of Windows Sockets 2 has been to provide a protocol-independent interface fully capable of supporting emerging networking capabilities, such as real-time multimedia communications.

Windows Sockets 2 is an interface, *not* a protocol. As an interface, it is used to discover and utilize the communications capabilities of any number of underlying transport protocols. Because it is not a protocol, it does not in any way affect the *bits on the wire*, and does not need to be utilized on both ends of a communications link.

Windows Sockets programming previously centered around TCP/IP. Some of the programming practices that worked with TCP/IP do not work with every protocol. As a result, the Windows Sockets 2 API added new functions where necessary to handle several protocols.



Windows Sockets 2 has changed its architecture to provide easier access to multiple transport protocols. Following the Windows Open System Architecture (WOSA) model, Windows Sockets 2 now defines a standard service provider interface (SPI) between the application programming interface (API), with its functions exported from WS2\_32.dll and the protocol stacks. Consequently, Windows Sockets 2 support is not limited to TCP/IP protocol stacks as is the case for Windows Sockets 1.1. For more information, see *Windows Sockets 2 Architecture*.

There are new challenges in developing Windows Sockets 2 applications. When sockets only supported TCP/IP, a developer could create an application that supported only two socket types: connectionless and connection-oriented. Connectionless protocols used SOCK\_DGRAM sockets and connection-oriented protocols used SOCK\_STREAM sockets. Now, these are just two of the many new socket types. Additionally, developers can no longer rely on socket type to describe all the essential attributes of a transport protocol.

## Windows Sockets 2 Features

Windows Sockets 2 extends functionality in a number of areas.

Features	Description
Access to protocols other than TCP/IP	Windows Sockets 2 allows an application to use the familiar socket interface to achieve simultaneous access to a number of installed transport protocols.
Overlapped I/O with scatter/gather	Windows Sockets 2 incorporates the overlapped paradigm for socket I/O and incorporates scatter/gather capabilities as well, following the model established in Win32 environments.
Protocol-independent name resolution facilities	Windows Sockets 2 includes a standardized set of functions for querying and working with the myriad name resolution domains that exist today (for example DNS, SAP, and X.500).
Protocol-independent multicast and multipoint	Windows Sockets 2 applications discover what type of multipoint or multicast capabilities a transport provides and use these facilities in a generic manner.
Quality of service (QOS)	Windows Sockets 2 establishes conventions that applications use to negotiate required service levels for parameters such as bandwidth and latency. Other QOS-related enhancements include mechanisms for network-specific QOS extensions.
Other frequently requested extensions	Windows Sockets 2 incorporates shared sockets and conditional acceptance; exchange of user data at connection setup/teardown time; and protocol-specific extension mechanisms.

## Conventions for New Functions

Windows Sockets 2, with its expanded scope, takes the socket paradigm beyond the original design. As a result, a number of new functions have been added. These have been assigned names that begin with WSA. In all but a few instances, these new functions are expanded versions of existing functions from BSD sockets.

The new functions are described in the reference section of the document, following the conventions of the Platform SDK. The new functions are also listed in *Summary of New Functions*.

## Microsoft Extensions and the Windows Sockets 2 API

The Windows Sockets 2 specification defines an extension mechanism that exposes advanced transport functionality to application programs. For more information, see Function Extension Mechanism.

The following Microsoft-specific extensions were added to Windows Sockets 1.1. They are also available in Windows Sockets 2.

**AcceptEx**  
**GetAcceptExSockaddrs**  
**TransmitFile**  
**WSARecvEx**

These functions are not exported from the `Ws2_32.dll`; they are exported from `Mswsock.dll`.

An application written to use the Microsoft-specific extensions to Windows Sockets does not run correctly over a Windows Sockets service provider that does not support those extensions.

## Socket Handles for Windows Sockets 2

A socket handle can optionally be a file handle in Windows Sockets 2. It is possible to use socket handles with **ReadFile**, **WriteFile**, **ReadFileEx**, **WriteFileEx**, **DuplicateHandle**, and other Win32 functions. Not all transport service providers support this option. For an application to run over the widest possible number of service providers, it should not assume that socket handles are file handles.

Windows Sockets 2 has expanded certain functions that transfer data between sockets using handles. The functions offer advantages specific to sockets for transferring data and include **WSARecv**, **WSASend**, and **WSADuplicateSocket**.

## New Concepts, Additions, and Changes for Windows Sockets 2

This section summarizes Windows Sockets 2 and describes the major changes and additions it contains. Windows Sockets 2 differs from Windows Sockets 1.1 in several ways, particularly in architecture. The new architecture, discussed in Windows Sockets 2 Architecture, provides the foundation for many of the new concepts that have been incorporated into Windows Sockets 2.

An overview of the additions and changes in Windows Sockets 2 follows the discussion of the new architecture.

Many of the functions in Windows Sockets 2 are the same as in the other versions of sockets. However, there are several new functions, which are summarized in Summary of New Functions. For detailed information on how to use a specific function or feature, refer to the Reference section.

### Windows Sockets 2 Architecture

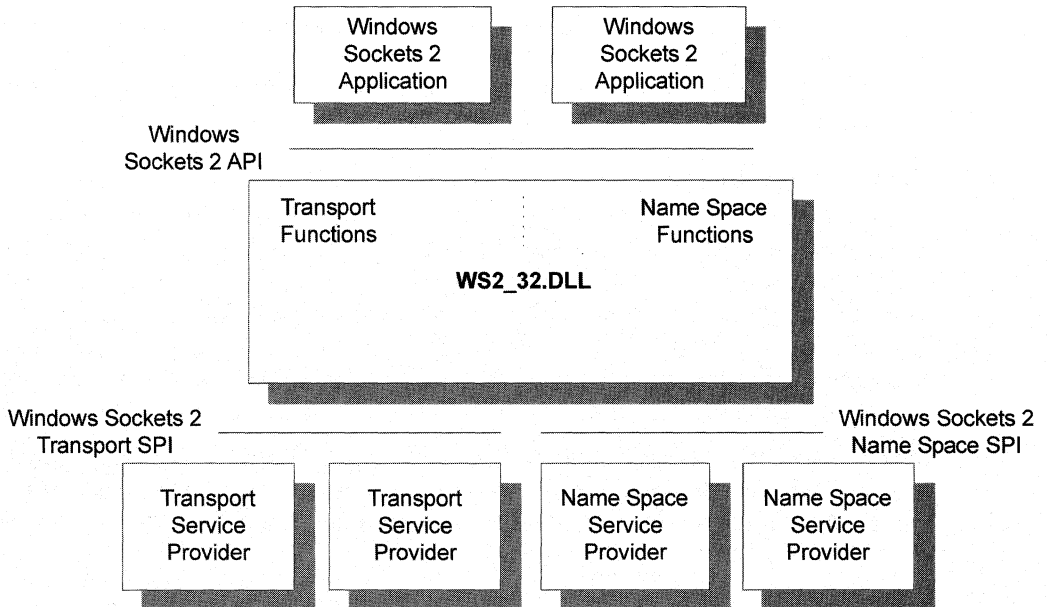
A number of Windows Sockets 2 features required substantial change in the Windows Sockets architecture. The resulting architecture is considerably different from previous versions, but the benefits are numerous. Foremost among these is Simultaneous Access to Multiple Transport Protocols, explained in detail in the following section.

Other features include the adoption of protocol-independent name resolution facilities, provisions for layered protocols and protocol chains, and a different mechanism for Windows Sockets service providers to offer extended, provider-specific functionality.

### Simultaneous Access to Multiple Transport Protocols

In order to provide simultaneous access to multiple transport protocols, the architecture has changed for Windows Sockets 2. With Windows Sockets 1.1, the vendor of the TCP/IP protocol stack supplies the DLL that implements the Windows Sockets interface. The interface between the Windows Sockets DLL and the underlying stack was both unique and proprietary. Windows Sockets 2 changes this model. It defines a standard Service Provider Interface (SPI) between the Windows Sockets DLL and protocol stacks. In this way, a single Windows Sockets DLL can simultaneously access multiple stacks from different vendors. Furthermore, Windows Sockets 2 support is not limited to TCP/IP protocol stacks as it is in Windows Sockets 1.1.

The Windows Open System Architecture (WOSA)-compliant Windows Sockets 2 architecture is shown in Figure 6-1.



**Figure 6-1: Windows Sockets 2 Architecture.**

With the Windows Sockets 2 architecture, it is not necessary or desirable, for stack vendors to supply their own implementation of WS2\_32.dll, since a single WS2\_32.dll must work across all stacks. The WS2\_32.dll and compatibility shims should be viewed in the same way as an operating system component.

## Backward Compatibility for Windows Sockets 1.1 Applications

Windows Sockets 2 is backward compatible with Windows Sockets 1.1 on two levels: source and binary. This maximizes interoperability between Windows Sockets applications of any version and Windows Sockets implementations of any version. It also minimizes problems for users of Windows Sockets applications, network stacks, and service providers. Current Windows Sockets 1.1-compliant applications operate on a Windows Sockets 2 implementation without modification of any kind, as long as at least one TCP/IP service provider is properly installed.

### Source Code Compatibility

Source code compatibility in Windows Sockets 2 means, with few exceptions, that all the Windows Sockets 1.1 functions are preserved in Windows Sockets 2. Windows Sockets 1.1 applications that use blocking hooks should be modified since blocking hooks are no longer supported in Windows Sockets 2. (For more information, see Windows Sockets 1.1 Blocking Routines and EINPROGRESS.)

Existing Windows Sockets 1.1 application source code can easily be moved to the Windows Sockets 2 system by including the new header file, Winsock2.h, and performing a straightforward relink with the appropriate Windows Sockets 2 libraries.

Application developers are encouraged to view this as the first step in a full transition to Windows Sockets 2 because there are numerous ways in which a Windows Sockets 1.1 application can be improved by exploring and using the new functionality in Windows Sockets 2.

### Binary Compatibility

A major design goal for Windows Sockets 2 was to enable existing Windows Sockets 1.1 applications to work, unchanged at a binary level, with Windows Sockets 2. Since Windows Sockets 1.1 applications are TCP/IP-based, binary compatibility implies that TCP/IP-based Windows Sockets 2 Transport and Name Resolution Service Providers are present in the Windows Sockets 2 system. In order to enable Windows Sockets 1.1 applications in this scenario, the Windows Sockets 2 system has an additional *shim* component supplied with it: a Version 1.1-compliant Winsock.dll.

Installation guidelines for Windows Sockets 2 ensure there is no negative impact to existing Windows Sockets-based applications on an end-user system with the introduction of any Windows Sockets 2 components. (See Figure 6-2.)

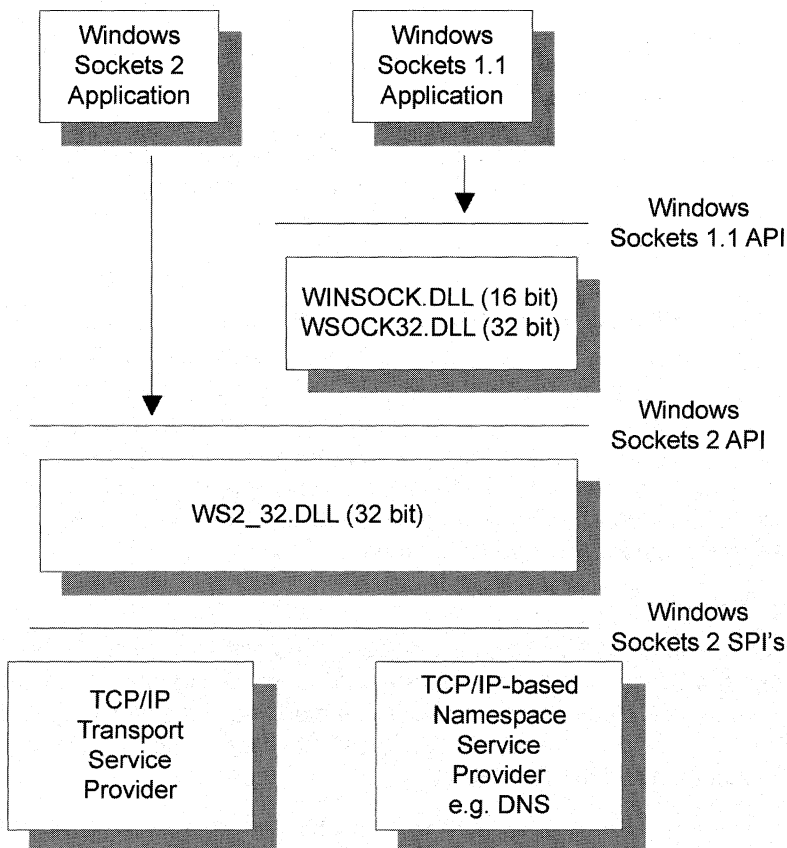


Figure 6-2: Windows Sockets 1.1 Compatibility Architecture.

---

**Important** To obtain information about the underlying TCP/IP stack, Windows Sockets 1.1 applications currently use certain members of the **WSADATA** structure (obtained through a call to **WSAStartup**). These members include: **IMAXSOCKETS**, **IMAXUDPDG**, and **LPVENDORINFO**.

While Windows Sockets 2 applications ignore these values (since they cannot uniformly apply to all available protocol stacks), safe values are supplied to avoid breaking Windows Sockets 1.1 applications.

---

## Making Transport Protocols Available to Windows Sockets

A transport protocol must be properly installed on the system and registered with Windows Sockets to be accessible to an application. The `Ws2_32.dll` exports a set of functions to facilitate the registration process. This includes creating a new registration and removing an existing one.

When new registrations are created, the caller (that is, the stack vendor's installation script) supplies one or more filled in **WSAPROTOCOL\_INFO** structures containing a complete set of information about the protocol. (See the Welcome To Windows Sockets 2 SPI for information on how this is accomplished.) Any transport stack installed in this manner is referred to as a Windows Sockets service provider.

The Windows Sockets 2 SDK includes a small Windows applet, `Sporder.exe`, that allows the user to view and modify the order in which service providers are enumerated. By using this `Sporder.exe`, a user can manually establish a particular TCP/IP protocol stack as the default TCP/IP provider if more than one such stack is present.

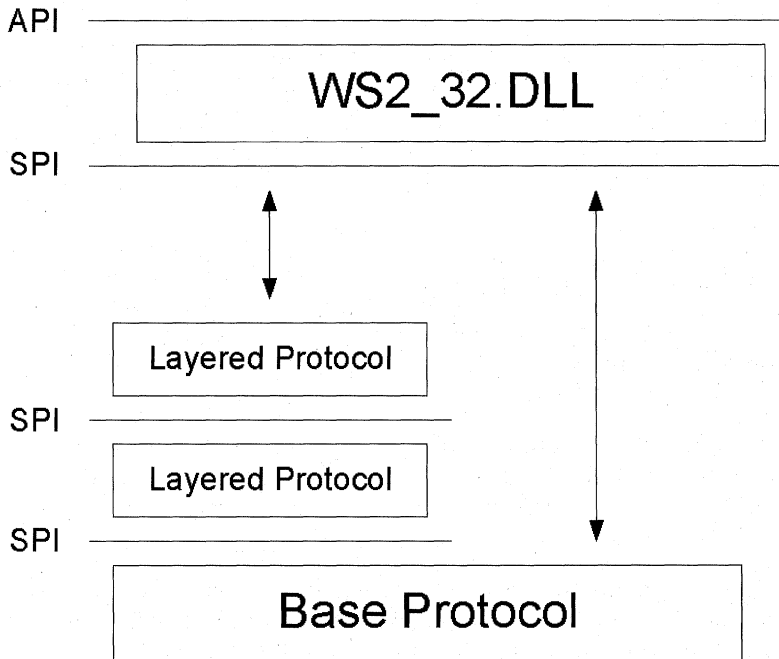
The `Sporder.exe` applet exports functions from `Sporder.dll` to reorder the service providers. As a result, installation applications can use the interface of `Sporder.dll` to programmatically reorder service providers to suit their needs.

## Layered Protocols and Protocol Chains

Windows Sockets 2 incorporates the concept of a layered protocol. A layered protocol is one that implements only higher-level communications functions while relying on an underlying transport stack for the actual exchange of data with a remote endpoint. An example of this type of layered protocol is a security layer that adds a protocol to the socket connection process in order to perform authentication and establish an encryption scheme. Such a security protocol generally requires the services of an underlying and reliable transport protocol such as TCP or SPX.

The term *base protocol* refers to a protocol, such as TCP or SPX, that is fully capable of performing data communications with a remote endpoint. A *layered protocol* is a protocol that cannot stand alone, while a *protocol chain* is one or more layered protocols strung together and anchored by a base protocol.

You can create a protocol chain if you design the layered protocols to support the Windows Sockets 2 SPI at both their upper and lower edges. A special **WSAPROTOCOL\_INFO** structure refers to the protocol chain as a whole and describes the explicit order in which the layered protocols are joined. This is shown in Figure 6-3. Since only base protocols and protocol chains are directly usable by applications, they are the only ones listed when the installed protocols are enumerated with the **WSAEnumProtocols** function.



**Figure 6-3: Layered Protocol Architecture.**

## Using Multiple Protocols

An application uses the **WSAEnumProtocols** function to determine which transport protocols and protocol chains are present, and to obtain information about each as contained in the associated **WSAPROTOCOL\_INFO** structure.

In most instances, there is a single **WSAPROTOCOL\_INFO** structure for each protocol or protocol chain. However, some protocols exhibit multiple behaviors. For example, the SPX protocol is message oriented (that is, the sender's message boundaries are preserved by the network), but the receiving socket can ignore these message boundaries and treat them as a byte stream. Thus, two different **WSAPROTOCOL\_INFO** structure entries could exist for SPX—one for each behavior.

In Windows Sockets 2, several new address family, socket type, and protocol values appear. Windows Sockets 1.1 supported a single address family (AF\_INET) comprising a small number of well-known socket types and protocol identifiers. Windows Sockets 2 retains the existing address family, socket type, and protocol identifiers for compatibility reasons, but it also supports new transport protocols with new media types.

A Windows Sockets 2 clearinghouse allows protocol stack vendors to obtain unique identifiers for new address families, socket types, and protocols. FTP and World Wide Web servers supply current identifier/value mappings and use email to request allocation of new ones. This is the World Wide Web URL for the Windows Sockets 2 Identifier Clearinghouse:

```
http://www.stardust.com/winsock/
```

New, unique identifiers are not necessarily well known, but this should not pose a problem. Applications that need to be protocol-independent are encouraged to select a protocol on the basis of its suitability rather than the values assigned to their *socket\_type* or *protocol* parameters. Protocol suitability is indicated by the communications attributes, such as message-versus-byte stream, and reliable-versus-unreliable, that are contained in the protocol **WSAPROTOCOL\_INFO** structure. Selecting protocols on the basis of suitability as opposed to well-known protocol names and socket types lets protocol-independent applications take advantage of new transport protocols and their associated media types, as they become available.

The server half of a client/server application benefits by establishing listening sockets on all suitable transport protocols. Then, the client can establish its connection using any suitable protocol. For example, this would let a client application be unmodified whether it was running on a desktop system connected through LAN or on a laptop using a wireless network.

## Multiple Provider Restrictions on Select

The **select** function is used to determine the status of one or more sockets in a set. For each socket, the caller can request information on read, write, or error status. A set of sockets is indicated by an **FD\_SET** structure.

Windows Sockets 2 allows an application to use more than one service provider, but the **select** function is limited to a set of sockets associated with a single service provider. This does not in any way restrict an application from having multiple sockets open through multiple providers.

There are two ways to determine the status of a set of sockets that spans more than one service provider:

- Using the **WSAWaitForMultipleEvents** or **WSAEventSelect** functions when blocking semantics are employed
- Using the **WSAAsyncSelect** function when nonblocking operations are employed.



When an application needs to use blocking semantics on a set of sockets that spans multiple providers, **WSAWaitForMultipleEvents** is recommended. The application can also use the **WSAEventSelect** function, which allows the FD\_XXX network events (see **WSAEventSelect**) to associate with an event object and be handled from within the event object paradigm (described in *Overlapped I/O and Event Objects*).

The **WSAAsyncSelect** function is recommended when nonblocking operations are preferred. This function is not restricted to a single provider because it takes a socket descriptor as an input parameter.

## Function Extension Mechanism

The Windows Sockets .dll, `Ws2_32.dll`, is no longer supplied by each individual stack vendor. As a result, it is no longer possible for a stack vendor to offer extended functionality by just adding entry points to the `Ws2_32.dll`. To overcome this limitation, Windows Sockets 2 takes advantage of the new **WSAIoctl** function to accommodate service providers who want to offer provider-specific functionality extensions. This mechanism assumes, of course, that an application is aware of a particular extension and understands both the semantics and syntax involved. Such information would typically be supplied by the service provider vendor.

In order to invoke an extension function, the application must first ask for a pointer to the desired function. This is done through the **WSAIoctl** function using the `SIO_GET_EXTENSION_FUNCTION_POINTER` command code. The input buffer to the **WSAIoctl** function contains an identifier for the desired extension function while the output buffer contains the function pointer itself. The application can then invoke the extension function directly without passing through the `Ws2_32.dll`.

The identifiers assigned to extension functions are globally unique identifiers (GUIDs) that are allocated by service provider vendors. Vendors who create extension functions are urged to publish full details about the function including the syntax of the function prototype. This makes it possible for common and popular extension functions to be offered by more than one service provider vendor. An application can obtain the function pointer and use the function without needing to know anything about the particular service provider that implements the function.

## Debug and Trace Facilities

Windows Sockets 2 application developers need to isolate bugs in:

- The application.
- The `Ws2_32.dll` or one of the compatibility shim .dlls.
- The service provider. Windows Sockets 2 addresses this need through a specially devised version of the `Ws2_32.dll` and a separate debug/trace .dll. This combination allows all procedure calls across the Windows Sockets 2 API or SPI to be monitored and, to some extent, controlled.

Developers can use this mechanism to trace procedure calls, procedure returns, parameter values, and return values. Parameter values and return values can be altered on procedure call or procedure return. If desired, a procedure call can be prevented or redirected. With access to this level of information and control, a developer can isolate any problem in the application, `Ws2_32.dll`, or service provider.

The Windows Sockets 2 SDK includes the debug `Ws2_32.dll`, a sample debug/trace `.dll`, and a document containing a detailed description of the components. The sample debug/trace `.dll` is provided in both source and object form. Developers are free to use the source to develop versions of the debug/trace `.dll` that meet their specific needs.

## Name Resolution

Windows Sockets 2 includes provisions for standardizing the way applications access and use the various network name resolution services. Windows Sockets 2 applications do not need to be aware of the widely differing interfaces associated with name services such as DNS, NIS, X.500, SAP, and others. An introduction to this topic and the details of the functions are currently located in *Protocol-Independent Name Resolution*.

## Overlapped I/O and Event Objects

Windows Sockets 2 introduces overlapped I/O and requires that all transport providers support this capability. Overlapped I/O follows the model established in Win32 and can be performed only on sockets created through the **WSASocket** function with the `WSA_FLAG_OVERLAPPED` flag set or sockets created through the **socket** function.

---

**Note** Creating a socket with the overlapped attribute has no impact on whether a socket is currently in blocking or nonblocking mode. Sockets created with the overlapped attribute can be used to perform overlapped I/O—doing so does not change the blocking mode of a socket. Since overlapped I/O operations do not block, the blocking mode of a socket is irrelevant for these operations.

---

For receiving, applications use the **WSARecv** or **WSARecvFrom** functions to supply buffers into which data is to be received. If one or more buffers are posted prior to the time when data has been received by the network, that data could be placed in the user's buffers immediately as it arrives. Thus, it can avoid the copy operation that would otherwise occur at the time the **recv** or **recvfrom** function is invoked. If data is already present when receive buffers are posted, it is copied immediately into the user's buffers.

If data arrives when no receive buffers have been posted by the application, the network resorts to the familiar synchronous style of operation. That is, the incoming data is buffered internally until the application issues a receive call and thereby supplies a buffer into which the data can be copied. An exception to this is when the application uses **setsockopt** to set the size of the receive buffer to zero. In this instance, reliable protocols would only allow data to be received when application buffers had been posted and data on unreliable protocols would be lost.

On the sending side, applications use **WSASend** or **WSASendTo** to supply pointers to filled buffers and then agree not to disturb the buffers in any way until the network has consumed the buffer's contents.

Overlapped send and receive calls return immediately. A return value of zero indicates that the I/O operation was completed immediately and that the corresponding completion indication already occurred. That is, the associated event object has been signaled, or a completion routine has been queued and will be executed when the calling thread gets into the alertable wait state.

A return value of `SOCKET_ERROR` coupled with an error code of `WSA_IO_PENDING` indicates that the overlapped operation has been successfully initiated and that a subsequent indication will be provided when send buffers have been consumed or when a receive operation has been completed. However, for sockets that are byte-stream style, the completion indication occurs whenever the incoming data is exhausted, regardless of whether the buffers are full. Any other error code indicates that the overlapped operation was not successfully initiated and that no completion indication will be forthcoming.

Both send and receive operations can be overlapped. The receive functions can be invoked several times to post receive buffers in preparation for incoming data, and the send functions can be invoked several times to queue multiple buffers to send. While the application can rely upon a series of overlapped send buffers being sent in the order supplied, the corresponding completion indications might occur in a different order. Likewise, on the receiving side, buffers can be filled in the order they are supplied, but the completion indications might occur in a different order.

Canceling individual overlapped operations pending on a given socket is impossible. However, the **closesocket** function can be called to close the socket and eventually discontinue all pending operations.

The deferred completion feature of overlapped I/O is also available for **WSAIoctl**, which is an enhanced version of **ioctlsocket**.

## Event Objects

Introducing overlapped I/O requires a mechanism for applications to unambiguously associate send and receive requests with their subsequent completion indications. In Windows Sockets 2, this is accomplished with event objects that are modeled after Win32 events. Windows Sockets event objects are fairly simple constructs that can be created and closed, set and cleared, and waited upon and polled. Their prime utility is the ability of an application to block and wait until one or more event objects become set.

Applications use **WSACreateEvent** to obtain an event object handle that can then be supplied as a required parameter to the overlapped versions of send and receive calls (**WSASend**, **WSASendTo**, **WSARecv**, **WSARecvFrom**). The event object, which is cleared when first created, is set by the transport providers when the associated overlapped I/O operation has completed (either successfully or with errors). Each event object created by **WSACreateEvent** should have a matching **WSACloseEvent** to destroy it.

Event objects are also used in **WSAEventSelect** to associate one or more FD\_XXX network events with an event object. This is described in *Asynchronous Notification Using Event Objects*.

In 32-bit environments, event object–related functions, including **WSACreateEvent**, **WSACloseEvent**, **WSASetEvent**, **WSAResetEvent**, and **WSAWaitForMultipleEvents** are directly mapped to the corresponding native Win32 functions, using the same function name, but without the WSA prefix.

## Receiving Completion Indications

Several options are available for receiving completion indications, thus providing applications with appropriate levels of flexibility. These include: waiting (or blocking) on event objects, polling event objects, and socket I/O completion routines.

### Blocking and Waiting for Completion Indication

Applications can block while waiting for one or more event objects to become set using the **WSAWaitForMultipleEvents** function. In Win32 implementations, the process or thread truly blocks. Since Windows Sockets 2 event objects are implemented as Win32 events, the native Win32 function, **WaitForMultipleObjects** can also be used for this purpose. This is especially useful if the thread needs to wait on both socket and non-socket events.

### Polling for Completion Indication

Applications that prefer not to block can use the **WSAGetOverlappedResult** function to poll for the completion status associated with any particular event object. This function indicates whether or not the overlapped operation has completed, and if completed, arranges for the **WSAGetLastError** function to retrieve the error status of the overlapped operation.

### Using Socket I/O Completion Routines

The functions used to initiate overlapped I/O (**WSASend**, **WSASendTo**, **WSARecv**, **WSARecvFrom**) all take *lpCompletionRoutine* as an optional input parameter. This is a pointer to an application-specific function that is called after a successfully initiated overlapped I/O operation completes (successfully or otherwise). The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. That is, the completion routine is not invoked until the thread is in an alertable wait state, such as when the function **WSAWaitForMultipleEvents** is invoked with the **FA\_ALERTABLE** flag set. An application that uses the completion routine option for a particular overlapped I/O request may not use the wait option of **WSAGetOverlappedResult** for that same overlapped I/O request.

The transports allow an application to invoke send and receive operations from within the context of the socket I/O completion routine and guarantee that, for a given socket, I/O completion routines will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

## Summary of Overlapped Completion Indication Mechanisms

The particular overlapped I/O completion indication to be used for a given overlapped operation is determined by whether the application supplies a pointer to a completion function, whether a **WSAOVERLAPPED** structure is referenced, and by the value of the **hEvent** member within the **WSAOVERLAPPED** structure (if supplied). The following table summarizes the completion semantics for an overlapped socket and shows the various combinations of **lpOverlapped**, **hEvent**, and **lpCompletionRoutine**:

<b>lpOverlapped</b>	<b>hEvent</b>	<b>lpCompletionRoutine</b>	<b>Completion Indication</b>
NULL	Not applicable	Ignored	Operation completes synchronously. It behaves as if it were a non-overlapped socket.
!NULL	NULL	NULL	Operation completes overlapped, but there is no Windows Sockets 2-supported completion mechanism. The completion port mechanism (if supported) can be used in this case. Otherwise, there is no completion notification.
!NULL	!NULL	NULL	Operation completes overlapped, notification by signaling event object.
!NULL	Ignored	!NULL	Operation completes overlapped, notification by scheduling completion routine.

## Asynchronous Notification Using Event Objects

The **WSAEventSelect** and **WSAEnumNetworkEvents** functions are provided to accommodate applications such as daemons and services that have no user interface (and hence do not use Windows handles). The **WSAEventSelect** function behaves exactly like the **WSAAsyncSelect** function. However, instead of causing a Windows message to be sent on the occurrence of an FD\_XXX network event (for example, FD\_READ and FD\_WRITE), an application-designated event object is set.

Also, the fact that a particular FD\_XXX network event has occurred is remembered by the service provider. The application can call **WSAEnumNetworkEvents** to have the current contents of the network event memory copied to an application-supplied buffer and to have the network event memory automatically cleared. If needed, the application can also designate a particular event object that is cleared along with the network event memory.

## Flow Specification Quality of Service

Quality of Service is implemented in Windows 2000 through various Windows 2000 QOS components. For details and implementation guidelines, see the separate section under the Networking Services node of the Platform SDK titled Quality of Service.

### QOS Templates

For details about QOS templates, see the chapters later in this volume that address QOS, or see the Platform SDK section titled Quality of Service.

### Default Values

For details and implementation guidelines about Quality of Service, the FLOWSPEC structure, and FLOWSPEC's default values, see the separate section under the Networking Services node of the Platform SDK titled Quality of Service. The FLOWSPEC structure is defined in the Quality of Service section's reference section.

## Socket Groups

All use of Socket Groups is reserved.

## Shared Sockets

The **WSADuplicateSocket** function is introduced to enable socket sharing across processes. A source process calls **WSADuplicateSocket** to obtain a special **WSAPROTOCOL\_INFO** structure for a target process identifier. It uses some interprocess communications (IPC) mechanism to pass the contents of this structure to a target process. The target process then uses the **WSAPROTOCOL\_INFO** structure in a call to **WSPSocket**. The socket descriptor returned by this function will be an additional socket descriptor to an underlying socket which thus becomes shared. Sockets can be shared among threads in a given process without using the **WSADuplicateSocket** function because a socket descriptor is valid in all threads of a process.

The two (or more) descriptors that reference a shared socket can be used independently as far as I/O is concerned. However, the Windows Sockets interface does not implement any type of access control, so the processes must coordinate any operations on a shared socket. A typical example of sharing sockets is to use one process for creating sockets and establishing connections. This process then hands off sockets to other processes that are responsible for information exchange.

The **WSADuplicateSocket** function creates socket descriptors and not the underlying socket. As a result, all the states associated with a socket are held in common across all the descriptors. For example, a **setsockopt** operation performed using one descriptor is subsequently visible using a **getsockopt** from any or all descriptors. A process can call **closesocket** on a duplicated socket and the descriptor will become deallocated. The underlying socket, however, remains open until **closesocket** is called with the last remaining descriptor.

Notification on shared sockets is subject to the usual constraints of the **WSAAsyncSelect** and **WSAEventSelect** functions. Issuing either of these calls using any of the shared descriptors cancels any previous event registration for the socket, regardless of which descriptor was used to make that registration. Thus, for example, it would not be possible to have process A receive FD\_READ events and process B receive FD\_WRITE events. For situations when such tight coordination is required, it is suggested that developers use threads instead of separate processes.

## Enhanced Functionality During Connection Setup and Teardown

The **WSAAccept** function lets an application obtain caller information such as caller identifier and QOS before deciding whether to accept an incoming connection request. This is done with a callback to an application-supplied condition function.

User-to-user data specified by parameters in the **WSAConnect** function and the condition function of **WSAAccept** can be transferred to the peer during connection establishment, provided this feature is supported by the service provider.

It is also possible (for protocols that support this) to exchange user data between the endpoints at connection teardown time. The end that initiates the teardown can call the **WSASendDisconnect** function to indicate that no more data be sent and to initiate the connection teardown sequence. For certain protocols, part of teardown is the delivery of disconnect data from the teardown initiator. After receiving notice that the remote end has initiated teardown (typically by the FD\_CLOSE indication), the **WSARecvDisconnect** function can be called to receive the disconnect data, if any.

To illustrate how disconnect data can be used, consider the following scenario. The client half of a client/server application is responsible for terminating a socket connection. Coincident with the termination, it provides (using disconnect data) the total number of transactions it processed with the server. The server in turn responds with the cumulative total of transactions that it has processed with all clients. The sequence of calls and indications might occur as follows.

### Client side

(1) Invoke **WSASendDisconnect** to conclude session and supply transaction total.

### Server side

(2) Get FD\_CLOSE, **recv** with a return value of zero, or WSAEDISCON error return from **WSARecv** indicating graceful shutdown in progress.

(3) Invoke **WSARecvDisconnect** to get client's transaction total.

**Client side****Server side**

- 
- (6) Receive `FD_CLOSE` indication.
- (7) Invoke **WSARecvDisconnect** to receive and store cumulative grand total of transactions.
- (8) Invoke **closesocket**

- (4) Compute cumulative grand total of all transactions.
- (5) Invoke **WSASendDisconnect** to transmit grand total.
- (5a) Invoke **closesocket**.

---

**Note** Step (5a) must follow step (5), but has no timing relationship with step (6), (7), or (8).

---

## Extended Byte-Order Conversion Routines

Windows Sockets 2 does not assume that the network byte order for all protocols is the same. A set of conversion routines is supplied for converting 16-bit and 32-bit quantities to and from network byte order. These routines take as an input parameter the socket handle that has a **WSAPROTOCOL\_INFO** structure associated with it. The **NetworkByteOrder** member of the **WSAPROTOCOL\_INFO** structure specifies the desired network byte order (currently either big-endian or little-endian).

## Support for Scatter/Gather I/O in the API

The **WSASend**, **WSASendTo**, **WSARecv**, and **WSARecvFrom** functions all take an array of application buffers as input parameters and can be used for scatter/gather (or vectored) I/O. This can be very useful in instances where portions of each message being transmitted consist of one or more fixed-length header components in addition to message body. Such header components need not be concatenated by the application into a single contiguous buffer prior to sending. Likewise on receiving, the header components can be automatically split off into separate buffers, leaving the message body *pure*.

When receiving into multiple buffers, completion occurs as data arrives from the network, regardless of whether all the supplied buffers are utilized.

## Protocol-Independent Multicast and Multipoint

Windows Sockets 2 provides a generic method for utilizing the multipoint and multicast capabilities of transports. This generic method implements these features just as it allows the basic data transport capabilities of numerous transport protocols to be accessed. The term multipoint is used hereafter to refer to both multicast and multipoint communications.



Current multipoint implementations (for example, IP multicast, ST-II, T.120, and ATM UNI) vary widely. How nodes join a multipoint session, whether a particular node is designated as a central or root node, and whether data is exchanged between all nodes or only between a root node and the various leaf nodes differ among implementations. The **WSAPROTOCOL\_INFO** structure for Windows Sockets 2 is used to declare the various multipoint attributes of a protocol. By examining these attributes, the programmer knows what conventions to follow with the applicable Windows Sockets 2 functions to set up, utilize, and tear down multipoint sessions.

Following is a summary of the features of Windows Sockets 2 that support multipoint.

- Two-attribute bits in the **WSAPROTOCOL\_INFO** structure.
- Four flags defined for the *dwFlags* parameter of the **WSASocket** function.
- One function, **WSAJoinLeaf**, for adding leaf nodes into a multipoint session
- Two **WSAIoctl** command codes for controlling multipoint loopback and establishing the scope for multicast transmissions. (The latter corresponds to the IP multicast time-to-live or TTL parameter.)

---

**Note** The inclusion of these multipoint features in Windows Sockets 2 does not preclude an application from using an existing protocol-dependent interface, such as the Deering socket options for IP multicast.

---

See *Multipoint and Multicast Semantics* for detailed information on how the various multipoint schemes are characterized and how the applicable features of Windows Sockets 2 are utilized.

## Summary of New Socket Options

The new socket options for Windows Sockets 2 are summarized in the following table. See **getsockopt** and **setsockopt** for detailed information on these options. The other new protocol-specific socket options can be found in the Protocol-specific Annex (a separate document included with the Platform SDK).

Value	Type	Meaning	Default	Note
SO_GROUP_ID	GROUP	Reserved.	NULL	Get only
SO_GROUP_PRIORITY	int	Reserved.	0	
SO_MAX_MSG_SIZE	int	Maximum outbound (send) size of a message for message-oriented socket types. There is no provision for finding out the maximum inbound message size. Has no meaning for stream-oriented sockets.	Implementation dependent	Get only

Value	Type	Meaning	Default	Note
SO_PROTOCOL_INFO	structure WSAPROTO COL_INFO	Description of protocol information for protocol that is bound to this socket.	Protocol dependent	Get only
PVD_CONFIG	char FAR *	An opaque data structure object containing configuration information of the service provider.	Implementation dependent	

## Summary of New Socket ioctl Opcodes

The new socket ioctl opcodes for Windows Sockets 2 are summarized in the following table. See **WSAIoctl** for detailed information on these opcodes. The **WSAIoctl** function also supports all the ioctl opcodes specified in **ioctlsocket**. The other new protocol-specific ioctl opcodes can be found in the Protocol-specific Annex (a separate document included with the Platform SDK).

Opcode	Input Type	Output Type	Meaning
SIO_ASSOCIATE_HANDLE	Companion API dependent	<not used>	Associate the socket with the specified handle of a companion interface.
SIO_ENABLE_CIRCULAR_QUEUEING	<not used>	<not used>	Circular queuing is enabled.
SIO_FIND_ROUTE	Structure <b>SOCKADDR</b>	<not used>	Request the route to the specified address to be discovered.
SIO_FLUSH	<not used>	<not used>	Discard current contents of the sending queue.
SIO_GET_BROADCAST_ADDRESS	<not used>	Structure <b>SOCKADDR</b>	Retrieve the protocol-specific broadcast address to be used in <b>sendto/WSASendTo</b> .
SIO_GET_QOS	<not used>	QOS	Retrieve current flow specification(s) for the socket.
SIO_GET_GROUP_QOS	<not used>	QOS	Reserved.
SIO_MULTIPOINT_LOOPBACK	BOOL	<not used>	Control whether data sent in a multipoint session will also be received by the same socket on the local host.

(continued)

*(continued)*

<b>Opcode</b>	<b>Input Type</b>	<b>Output Type</b>	<b>Meaning</b>
SIO_MULTICAST_SCOPE	int	<not used>	Specify the scope over which multicast transmissions will occur.
SIO_SET_QOS	QOS	<not used>	Establish new flow specification(s) for the socket.
SIO_SET_GROUP_QOS	QOS	<not used>	Reserved.
SIO_TRANSLATE_HANDLE	int	Companion API dependent	Obtain a corresponding handle for socket <i>s</i> that is valid in the context of a companion interface.
SIO_ROUTING_INTERFACE_QUERY	<b>SOCKADDR</b>	<b>SOCKADDR</b>	Obtain the address of local interface which should be used to send to the specified address.
SIO_ROUTING_INTERFACE_CHANGE	<b>SOCKADDR</b>	<not used>	Request notification of changes in information reported through SIO_ROUTING_INTERFACE_QUERY for the specified address.
SIO_ADDRESS_LIST_QUERY	<not used>	SOCKET_ADDRESS_LIST	Obtain the list of addresses to which application can bind.
SIO_ADDRESS_LIST_CHANGE	<not used>	<not used>	Request notification of changes in information reported through SIO_ADDRESS_LIST_QUERY

## Summary of New Functions

The new API functions for Windows Sockets 2 are summarized in the table on the following page.

## Data Transport Functions

Function	Description
<b>WSAAccept<sup>1</sup></b>	An extended version of <b>accept</b> which allows for conditional acceptance.
<b>WSACloseEvent</b>	Destroys an event object.
<b>WSAConnect<sup>1</sup></b>	An extended version of <b>connect</b> which allows for exchange of connect data and QOS specification.
<b>WSACreateEvent</b>	Creates an event object.
<b>WSADuplicateSocket</b>	Creates a new socket descriptor for a shared socket.
<b>WSAEnumNetworkEvents</b>	Discovers occurrences of network events.
<b>WSAEnumProtocols</b>	Retrieves information about each available protocol.
<b>WSAEventSelect</b>	Associates network events with an event object.
<b>WSAGetOverlappedResult</b>	Gets completion status of overlapped operation.
<b>WSAGetQOSByName</b>	Supplies QOS parameters based on a well-known service name.
<b>WSAHtonl</b>	Extended version of <b>htonl</b> .
<b>WSAHtons</b>	Extended version of <b>htons</b> .
<b>WSAIoctl<sup>1</sup></b>	Overlapped-capable version of <b>ioctlsocket</b> .
<b>WSAJoinLeaf<sup>1</sup></b>	Joins a leaf node into a multipoint session.
<b>WSANTohl</b>	Extended version of <b>ntohl</b> .
<b>WSANTohs</b>	Extended version of <b>ntohs</b> .
<b>WSAProviderConfigChange</b>	Receive notifications of service providers being installed/removed.
<b>WSARecv<sup>1</sup></b>	An extended version of <b>recv</b> which accommodates scatter/gather I/O, overlapped sockets, and provides the <i>flags</i> parameter as IN OUT.
<b>WSARecvDisconnect</b>	Terminates reception on a socket and retrieves the disconnect data, if the socket is connection-oriented.
<b>WSARecvFrom<sup>1</sup></b>	An extended version of <b>recvfrom</b> which accommodates scatter/gather I/O, overlapped sockets, and provides the <i>flags</i> parameter as IN OUT.
<b>WSAResetEvent</b>	Resets an event object.
<b>WSASend<sup>1</sup></b>	An extended version of <b>send</b> which accommodates scatter/gather I/O and overlapped sockets.
<b>WSASendDisconnect</b>	Initiates termination of a socket connection and optionally sends disconnect data.

(continued)

(continued)

Function	Description
<b>WSASendTo</b> <sup>1</sup>	An extended version of <b>sendto</b> which accommodates scatter/gather I/O and overlapped sockets.
<b>WSASetEvent</b>	Sets an event object.
<b>WSASocket</b>	An extended version of <b>socket</b> which takes a <b>WSAPROTOCOL_INFO</b> structure as input and allows overlapped sockets to be created.
<b>WSAWaitForMultipleEvents</b> <sup>1</sup>	Blocks on multiple event objects.

### Name Registration and Resolution Functions

Function	Description
<b>WSAAddressToString</b>	Converts an address structure into a human-readable numeric string.
<b>WSAEnumNameSpaceProviders</b>	Retrieves the list of available Name Registration and Resolution service providers.
<b>WSAGetServiceClassInfo</b>	Retrieves all of the class-specific information pertaining to a service class.
<b>WSAGetServiceClassNameByClassId</b>	Returns the name of the service associated with the given type.
<b>WSAInstallServiceClass</b>	Creates a new new service class type and stores its class-specific information.
<b>WSALookupServiceBegin</b>	Initiates a client query to retrieve name information as constrained by a <b>WSAQUERYSET</b> data structure.
<b>WSALookupServiceEnd</b>	Finishes a client query started by <b>WSALookupServiceBegin</b> and frees resources associated with the query.
<b>WSALookupServiceNext</b>	Retrieves the next unit of name information from a client query initiated by <b>WSALookupServiceBegin</b> .
<b>WSARemoveServiceClass</b>	Permanently removes a service class type.
<b>WSASetService</b>	Registers or removes from the registry a service instance within one or more namespaces.
<b>WSAStringToAddress</b>	Converts a human-readable numeric string to a socket address structure suitable for passing to Windows Sockets routines.

<sup>1</sup> The routine can block if acting on a blocking socket.

# Windows Sockets Programming Considerations

This section provides programmers with important information on a number of topics. It is especially pertinent to those who are porting socket applications from UNIX®-based environments or who are upgrading their Windows Sockets 1.1 applications to Windows Sockets 2.

## Deviation from Berkeley Sockets

There are a few limited instances where Windows Sockets has had to divert from strict adherence to the Berkeley conventions, usually due to implementation difficulties in the Microsoft® Windows environment.

### Socket Data Type

A new data type, **SOCKET**, has been defined. This is needed because a Windows Sockets application cannot assume that socket descriptors are equivalent to file descriptors as they are in UNIX. Furthermore, in UNIX, all handles, including socket handles, are small, non-negative integers, and some applications make assumptions that this will be true. Windows Sockets handles have no restrictions, other than that the value `INVALID_SOCKET` is not a valid socket. Socket handles may take any value in the range 0 to `INVALID_SOCKET-1`.

Because the **SOCKET** type is unsigned, compiling existing source code from, for example, a UNIX environment may lead to compiler warnings about signed/unsigned data type mismatches.

This means, for example, that checking for errors when the **socket** and **accept** routines return should not be done by comparing the return value with `-1`, or seeing if the value is negative (both common, and legal, approaches in BSD). Instead, an application should use the manifest constant `INVALID_SOCKET` as defined in `Winsock.h`. For example:

#### Typical BSD Style

```
s = socket(...);
if (s == -1) /* or s < 0 */
    {...}
```

#### Preferred Style

```
s = socket(...);
if (s == INVALID_SOCKET)
    {...}
```

## Select and FD\_\*

Because a socket is no longer represented by the UNIX-style small non-negative integer, the implementation of the **select** function was changed in Windows Sockets. Each set of sockets is still represented by the **FD\_SET** type, but instead of being stored as a bitmask the set is implemented as an array of sockets. To avoid potential problems, applications must adhere to the use of the **FD\_XXX** macros to set, initialize, clear, and check the **FD\_SET** structures.

## Error Codes—**errno**, **h\_errno** and **WSAGetLastError**

Error codes set by Windows Sockets are *not* made available through the *errno* variable. Additionally, for the **getXbyY** class of functions, error codes are *not* made available through the *h\_errno* variable. Instead, error codes are accessed by using the **WSAGetLastError** function. This function is provided in Windows Sockets as a precursor (and eventually an alias) for the Win32 function **GetLastError**. This is intended to provide a reliable way for a thread in a multithreaded process to obtain per-thread error information.

For compatibility with BSD, an application may choose to include a line of the form:

```
#define errno WSAGetLastError
```

This allows networking code which was written to use the global *errno* to work correctly in a single-threaded environment. There are, obviously, some drawbacks. If a source file includes code which inspects *errno* for both socket and nonsocket functions, this mechanism cannot be used. Furthermore, it is not possible for an application to assign a new value to *errno*. (In Windows Sockets the function **WSASetLastError** may be used for this purpose.)

### Typical BSD Style

```
r = recv(...);
if (r == -1
    && errno == EWOULDBLOCK)
    {...}
```

### Preferred Style

```
r = recv(...);
if (r == -1 /* (but see below) */
    && WSAGetLastError == EWOULDBLOCK)
    {...}
```

Although error constants consistent with Berkeley Sockets 4.3 are provided for compatibility purposes, applications should, where possible, use the WSA error code definitions. This is because error codes returned by certain Windows Sockets routines fall into the standard range of error codes as defined by Microsoft® C®. A better version of the preceding source code fragment is shown on the following page.

```
r = recv(...);
if (r == -1 /* (but see below) */
    && WSAGetLastError == WSAEWOULDBLOCK)
    {...}
```

This specification defines a recommended set of error codes, and lists the possible errors that can be returned as a result of each function. It may be the case in some implementations that other Windows Sockets error codes are returned in addition to those listed, and applications should be prepared to handle errors other than those enumerated under each function description. However Windows Sockets does not return any value that is not enumerated in the table of legal Windows Sockets errors given in the section *Error Codes*.

## Pointers

All pointers used by applications with Windows Sockets should be FAR although this is only relevant to 16-bit applications and meaningless in a 32-bit. To facilitate this, data type definitions such as **LPHOSTENT** are provided.

## Renamed Functions

In two cases it was necessary to rename functions that are used in Berkeley Sockets in order to avoid clashes with other Win32 API functions.

### Close and Closesocket

Sockets are represented by standard file descriptors in Berkeley Sockets, so the **close** function can be used to close sockets as well as regular files. While nothing in the Windows Sockets prevents an implementation from using regular file handles to identify sockets, nothing requires it either. Sockets must be closed by using the **closesocket** routine. Using the **close** routine to close a socket is incorrect and the effects of doing so are undefined by this specification.

### ioctl and ioctlsocket/WSAIoctl

Various C language run-time systems use the **IOCTL** routine for purposes unrelated to Windows Sockets. As a consequence, the **ioctlsocket** function and the **WSAIoctl** function were defined to handle socket functions that were performed by **IOCTL** and **fcntl** in the Berkeley Software Distribution.

## Maximum Number of Sockets Supported

The maximum number of sockets supported by a particular Windows Sockets service provider is implementation specific. An application should make no assumptions about the availability of a certain number of sockets. For more information on this topic see **WSAStartup**.

The maximum number of sockets that an application can actually use is independent of the number of sockets supported by a particular implementation. The maximum number of sockets that a Windows Sockets application can use is determined at compile time by



the manifest constant `FD_SETSIZE`. This value is used in constructing the `FD_SET` structures used in `select`. The default value in `Winsock2.h` is 64. If an application is designed to be capable of working with more than 64 sockets, the implementer should define the manifest `FD_SETSIZE` in every source file before including `Winsock2.h`. One way of doing this may be to include the definition within the compiler options in the makefile. For example, you could add “`-DFD_SETSIZE=128`” as an option to the compiler command line for Microsoft C. It must be emphasized that defining `FD_SETSIZE` as a particular value has no effect on the actual number of sockets provided by a Windows Sockets service provider.

## Include Files

A number of standard Berkeley include files are supported for ease of porting existing source code based on Berkeley sockets. However, these Berkeley header files merely include the `Winsock2.h` include file, and it is therefore sufficient (and recommended) that Windows Sockets application source files just include `Winsock2.h`.

## Return Values on Function Failure

The manifest constant `SOCKET_ERROR` is provided for checking function failure. Although use of this constant is not mandatory, it is recommended. The following example illustrates the use of the `SOCKET_ERROR` constant.

### Typical BSD Style

```
r = recv(...);
if (r == -1 /* or r < 0 */
    && errno == EWOULDBLOCK)
    {...}
```

### Preferred Style

```
r = recv(...);
if (r == SOCKET_ERROR
    && WSAGetLastError == WSAEWOULDBLOCK)
    {...}
```

## Service Provided Raw Sockets

The Windows Sockets specification does not mandate that a Windows Sockets service provider support raw sockets, that is, sockets of type `SOCK_RAW`. However, service providers are encouraged to supply raw socket support. A Windows Sockets-compliant application that wishes to use raw sockets should attempt to open the socket with the `socket` call, and if it fails either attempt to use another socket type or indicate the failure to the user.

## Byte Ordering

Care must always be taken to account for any differences between the byte ordering used by Intel® architecture and the byte ordering used on the wire by individual transport protocols. Any reference to an address or port number passed to or from a Windows Sockets routine must be in the network order for the protocol being utilized. In the case of IP, this includes the IP address and port parameters of a **SOCKADDR\_IN** structure (but not the *sin\_family* parameter).

Consider an application that normally contacts a server on the TCP port corresponding to the time service, but provides a mechanism for the user to specify an alternative port. The port number returned by **getservbyname** is already in network order, which is the format required for constructing an address so that no translation is required. However, if the user elects to use a different port, entered as an integer, the application must convert this from host to TCP/IP network order (using the **WSAhtons** function) before using it to construct an address. Conversely, if the application were to display the number of the port within an address (returned by **getpeername** for example), the port number must be converted from network to host order (using **WSAntohs**) before it can be displayed.

Since the Intel and Internet byte orders are different, the conversions described in the preceding are unavoidable. Application writers are cautioned that they should use the standard conversion functions provided as part of Windows Sockets rather than writing their own conversion code since future implementations of Windows Sockets are likely to run on systems for which the host order is identical to the network byte order. Only applications that use the standard conversion functions are likely to be portable.

## Windows Sockets Compatibility Issues

Windows Sockets 2 continues to support all of the Windows Sockets 1.1 semantics and function calls except for those dealing with psuedo-blocking. Since Windows Sockets 2 runs only in 32-bit, preemptively scheduled environments, there is no need to implement the psuedo-blocking found in Windows Sockets 1.1. This means that the **WSAEINPROGRESS** error code will never be indicated and that the following Windows Sockets 1.1 functions are not available to Windows Sockets 2 applications:

- **WSACancelBlockingCall**
- **WSAIsBlocking**
- **WSASetBlockingHook**
- **WSAUnhookBlockingHook**

Windows Sockets 1.1 programs that are written to utilize psuedo-blocking will continue to operate correctly since they link to either **Winsock.dll** or **Wsock32.dll**. Both continue to support the complete set of Windows Sockets 1.1 functions. In order for programs to become Windows Sockets 2 applications, some code modification must occur. In most cases, the judicious use of threads can be substituted to accommodate processing that was being accomplished with a blocking hook function.

## Default State for a Socket's Overlapped Attribute

The **socket** function created sockets with the overlapped attribute set by default in the first Wsock32.dll, the 32-bit version of Windows Sockets 1.1. In order to insure backward compatibility with currently deployed Wsock32.dll implementations, this will continue to be the case for Windows Sockets 2 as well. That is, in Windows Sockets 2 sockets created with the **socket** function will have the overlapped attribute. However, in order to be more compatible with the rest of the Win32 API, sockets created with **WSA Socket** will not, default, have the overlapped attribute. This attribute will only be applied if the WSA\_FLAG\_OVERLAPPED bit is set.

## Windows Sockets 1.1 Blocking Routines and EINPROGRESS

One major issue in porting applications from a Berkeley sockets environment to a Windows environment involves blocking; that is, invoking a function that does not return until the associated operation is completed. A problem arises when the operation takes an arbitrarily long time to complete: an example is a **recv**, which might block until data has been received from the peer system. The default behavior within the Berkeley sockets model is for a socket to operate in blocking mode unless the programmer explicitly requests that operations be treated as nonblocking. Windows Sockets 1.1 environments could not assume preemptive scheduling. Therefore, it was strongly recommended that programmers use the nonblocking (asynchronous) operations if at all possible with Windows Sockets 1.1. Because this was not always possible, the pseudo-blocking facilities described in the following were provided.

---

**Note** Windows Sockets 2 only runs on preemptive 32-bit operating systems where deadlocks are not a problem. Programming practices recommended for Windows Sockets 1.1 are not necessary in Windows Sockets 2.

---

Even on a blocking socket, some functions—**bind**, **getsockopt**, and **getpeername** for example—complete immediately. There is no difference between a blocking and a nonblocking operation for those functions. Other operations, such as **recv**, can complete immediately or take an arbitrary time to complete, depending on various transport conditions. When applied to a blocking socket, these operations are referred to as blocking operations. All routines that can block are listed with an asterisk in the preceding and following tables.

With 16-bit Windows Sockets 1.1, a blocking operation that cannot complete immediately is handled by pseudo-blocking as follows.

The service provider initiates the operation, then enters a loop in which it dispatches any Windows messages (yielding the processor to another thread, if necessary), and then checks for the completion of the Windows Sockets function. If the function has completed, or if **WSACancelBlockingCall** has been invoked, the blocking function completes with an appropriate result.

A service provider must allow installation of a blocking hook function that does not process messages in order to avoid the possibility of re-entrant messages while a blocking operation is outstanding. The simplest such blocking hook function would return **FALSE**. If a Windows Sockets DLL depends on messages for internal operation, it can execute **PeekMessage(hMyWnd...)** before executing the application blocking hook so that it can get its messages without affecting the rest of the system.

In a 16-bit Windows Sockets 1.1 environment, if a Windows message is received for a process for which a blocking operation is in progress, there is a risk that the application will attempt to issue another Windows Sockets call. Because of the difficulty in managing this condition safely, Windows Sockets 1.1 does not support such application behavior. An application is not permitted to make more than one nested Windows Sockets function call. Only one outstanding function call is allowed for a particular task. The only exceptions are two functions that are provided to assist the programmer in this situation: **WSAIsBlocking** and **WSACancelBlockingCall**.

The **WSAIsBlocking** function can be called at any time to determine whether or not a blocking Windows Sockets 1.1 call is in progress. Similarly, the **WSACancelBlockingCall** function can be called at any time to cancel an in-progress blocking call. Any other nesting of Windows Sockets functions fails with the error "WSAEINPROGRESS".

It should be emphasized that this restriction applies to both blocking and nonblocking operations. For Windows Sockets 2 applications that negotiate version 2.0 or higher at the time of calling **WSAStartup**, no restriction on the nesting of operations exists. Operations can become nested under rare circumstances, such as during a **WSAAccept** conditional-acceptance callback, or if a service provider in turn invokes a Windows Sockets 2 function.

Although this mechanism is sufficient for simple applications, it cannot support the complex message-dispatching requirements of more advanced applications (for example, those using the MDI model). For such applications, the Windows Sockets API includes the function **WSASetBlockingHook**, which allows the application to specify a special routine which can be called instead of the default message dispatch routine described in the preceding.

The Windows Sockets provider calls the blocking hook only if all of the following are true:

- The routine is one that is defined as being able to block.
- The specified socket is a blocking socket.
- The request cannot be completed immediately.

(A socket is set to blocking by default, but the **IOCTL FIONBIO** or the **WSAAsyncSelect** function set a socket to nonblocking mode.)

The blocking hook is never called and the application does not need to be concerned with the re-entrancy issues the blocking hook can introduce, if an application follows the guidelines on the following page.

- It uses only nonblocking sockets.
- It uses the **WSAAsyncSelect** and/or the **WSAAsyncGetXByY** routines instead of **select** and the **getXbyY** routines.

If a Windows Sockets 1.1 application invokes an asynchronous or nonblocking operation that takes a pointer to a memory object (a buffer or a global variable, for example) as an argument, it is the responsibility of the application to ensure that the object is available to Windows Sockets throughout the operation. The application must not invoke any Windows function that might affect the mapping or address viability of the memory involved.

## Graceful Shutdown, Linger Options, and Socket Closure

The following material is provided as clarification for the subject of shutting down socket connections closing the sockets. It is important to distinguish the difference between shutting down a socket connection and closing a socket.

Shutting down a socket connection involves an exchange of protocol messages between the two endpoints, hereafter referred to as a shutdown sequence. Two general classes of shutdown sequences are defined: *graceful* and *abortive* (also called *hard*). In a graceful shutdown sequence, any data that has been queued but not yet transmitted can be sent prior to the connection being closed. In an abortive shutdown, any unsent data is lost. The occurrence of a shutdown sequence (graceful or abortive) can also be used to provide an FD\_CLOSE indication to the associated applications signifying that a shutdown is in progress.

Closing a socket, on the other hand, causes the socket handle to become deallocated so that the application can no longer reference or use the socket in any manner.

In Windows Sockets, both the **shutdown** function, and the **WSASendDisconnect** function can be used to initiate a shutdown sequence, while the **closesocket** function is used to deallocate socket handles and free up any associated resources. Some amount of confusion arises, however, from the fact that the **closesocket** function implicitly causes a shutdown sequence to occur if it has not already happened. In fact, it has become a rather common programming practice to rely on this feature and to use **closesocket** to both initiate the shutdown sequence and deallocate the socket handle.

To facilitate this usage, the sockets interface provides for controls by way of the socket option mechanism that allow the programmer to indicate whether the implicit shutdown sequence should be graceful or abortive, and also whether the **closesocket** function should linger (that is not complete immediately) to allow time for a graceful shutdown sequence to complete. These important distinctions and the ramifications of using **closesocket** in this manner are still not widely understood.

By establishing appropriate values for the socket options **SO\_LINGER** and **SO\_DONTLINGER**, the types of behavior on the following page can be obtained with the **closesocket** function.

- Abortive shutdown sequence, immediate return from **closesocket**.
- Graceful shutdown, delaying return until either shutdown sequence completes or a specified time interval elapses. If the time interval expires before the graceful shutdown sequence completes, an abortive shutdown sequence occurs, and **closesocket** returns.
- Graceful shutdown, immediate return—allowing the shutdown sequence to complete in the background. Although this is the default behavior, the application has no way of knowing when (or whether) the graceful shutdown sequence actually completes.

One technique that can be used to minimize the chance of problems occurring during connection teardown is to avoid relying on an implicit shutdown being initiated by **closesocket**. Instead, use one of the two explicit shutdown functions, **shutdown** or **WSASendDisconnect**. This in turn causes an FD\_CLOSE indication to be received by the peer application indicating that all pending data has been received. To illustrate this, the following table shows the functions that would be invoked by the client and server components of an application, where the client is responsible for initiating a graceful shutdown.

Client side	Server side
(1) Invokes <b>shutdown</b> (s, SD_SEND) to signal end of session and that client has no more data to send.	
	(2) Receives FD_CLOSE, indicating graceful shutdown in progress and that all data has been received.
	(3) Sends any remaining response data.
(5a) Gets FD_READ and calls <b>recv</b> to get any response data sent by server.	(4) Invokes <b>shutdown</b> (s, SD_SEND) to indicate server has no more data to send.
(5) Receives FD_CLOSE indication.	(4a) Invokes <b>closesocket</b> .
(6) Invokes <b>closesocket</b> .	

**Note** The timing sequence is maintained from step (1) to step (6) between the client and the server, except for steps (4a) and (5a), which only have local timing significance in the sense that step (5) follows step (5a) on the client side while step (4a) follows step (4) on the server side, with no timing relationship with the remote party.

## Protocol-Independent Out-of-Band Data

The stream socket abstraction includes the notion of out of band (OOB) data. Many protocols allow portions of incoming data to be marked as special in some way, and these special data blocks can be delivered to the user out of the normal sequence. Examples include expedited data in X.25 and other OSI protocols, and urgent data in

BSD Unix's use of TCP. The next section describes OOB data handling in a protocol-independent manner. A discussion of OOB data implemented using TCP urgent data follows it. In the each discussion, the use of **recv** also implies **recvfrom**, **WSARecv**, and **WSARecvFrom**, and references to **WSAAsyncSelect** also apply to **WSAEventSelect**.

### Protocol Independent OOB Data

OOB data is a logically independent transmission channel associated with each pair of connected stream sockets. OOB data may be delivered to the user independently of normal data. The abstraction defines that the OOB data facilities must support the reliable delivery of at least one OOB data block at a time. This data block can contain at least one byte of data, and at least one OOB data block can be pending delivery to the user at any one time. For communications protocols that support in-band signaling (such as TCP, where the urgent data is delivered in sequence with the normal data), the system normally extracts the OOB data from the normal data stream and stores it separately (leaving a gap in the normal data stream). This allows users to choose between receiving the OOB data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to peek' at out-of-band data.

A user can determine if there is any OOB data waiting to be read using the **ioctlsocket SIOCATMARK** function. For protocols where the concept of the position of the OOB data block within the normal data stream is meaningful such as TCP, a Windows Sockets service provider maintains a conceptual marker indicating the position of the last byte of OOB data within the normal data stream. This is not necessary for the implementation of the **ioctlsocket(SIOCATMARK)** functionality—the presence or absence of OOB data is all that is required.

For protocols where the concept of the position of the OOB data block within the normal data stream is meaningful, an application might process out-of-band data inline, as part of the normal data stream. This is achieved by setting the socket option **SO\_OOBINLINE** with **setsockopt**. For other protocols where the OOB data blocks are truly independent of the normal data stream, attempting to set **SO\_OOBINLINE** will result in an error. An application can use the **SIOCATMARK ioctlsocket** command to determine whether there is any unread OOB data preceding the mark. For example, it can use this to resynchronize with its peer by ensuring that all data up to the mark in the data stream is discarded when appropriate.

With **SO\_OOBINLINE** disabled (the default setting):

- Windows Sockets notifies an application of an **FD\_OOB** event, if the application registered for notification with **WSAAsyncSelect**, in exactly the same way **FD\_READ** is used to notify of the presence of normal data. That is, **FD\_OOB** is posted when OOB data arrives with no OOB data previously queued. The **FD\_OOB** is also posted when data is read using the **MSG\_OOB** flag while some OOB data remains queued after the read operation has returned. **FD\_READ** messages are not posted for OOB data.
- Windows Sockets returns from **select** with the appropriate *exceptfds* socket set if OOB data is queued on the socket.

- The application can call **recv** with **MSG\_OOB** to read the urgent data block at any time. The block of OOB data jumps the queue.
- The application can call **recv** without **MSG\_OOB** to read the normal data stream. The OOB data block does not appear in the data stream with normal data. If OOB data remains after any call to **recv**, Windows Sockets notifies the application with **FD\_OOB** or with *exceptfds* when using **select**.
- For protocols where the OOB data has a position within the normal data stream, a single **recv** operation does not span that position. One **recv** returns the normal data before the mark, and a second **recv** is required to begin reading data after the mark.

With **SO\_OOBINLINE** enabled:

- **FD\_OOB** messages are *not* posted for OOB data. OOB data is treated as normal for the purpose of the **select** and **WSAAsyncSelect** functions, and indicated by setting the socket in *readfds* or by sending an **FD\_READ** message respectively.
- The application can not call **recv** with the **MSG\_OOB** flag set to read the OOB data block. The error code **WSAEINVAL** is returned.
- The application can call **recv** without the **MSG\_OOB** flag set. Any OOB data is delivered in its correct order within the normal data stream. OOB data is never mixed with normal data. There must be three read requests to get past the OOB data. The first returns the normal data prior to the OOB data block, the second returns the OOB data, the third returns the normal data following the OOB data. In other words, the OOB data block boundaries are preserved.

The **WSAAsyncSelect** routine is particularly well suited to handling notification of the presence of out-of-band-data when **SO\_OOBINLINE** is off.

## OOB Data in TCP

**Important** The following discussion of out-of-band data, implemented using TCP urgent data, follows the model used in the Berkeley software distribution. Users and implementers should be aware that:

- There are, at present, two conflicting interpretations of RFC 793 (where the concept is introduced).
- The implementation of OOB data in the Berkeley Software Distribution (BSD) does not conform to the Host Requirements laid down in RFC 1122.

Specifically, the TCP urgent pointer in BSD points to the byte after the urgent data byte, and an RFC-compliant TCP urgent pointer points to the urgent data byte. As a result, if an application sends urgent data from a BSD-compatible implementation to an RFC-1122 compatible implementation, the receiver reads the wrong urgent data byte (it reads the byte located after the correct byte in the data stream as the urgent data byte).



To minimize interoperability problems, applications writers are advised not to use OOB data unless this is required to interoperate with an existing service. Windows Sockets suppliers are urged to document the OOB semantics (BSD or RFC 1122) that their product implements.

Arrival of a TCP segment with the URG (for urgent) flag set indicates the existence of a single byte of OOB data within the TCP data stream. The OOB data block is one byte in size. The urgent pointer is a positive offset from the current sequence number in the TCP header that indicates the location of the OOB data block (ambiguously, as noted in the preceding). It might, therefore, point to data that has not yet been received.

If `SO_OOBINLINE` is disabled (the default) when the TCP segment containing the byte pointed to by the urgent pointer arrives, the OOB data block (one byte) is removed from the data stream and buffered. If a subsequent TCP segment arrives with the urgent flag set (and a new urgent pointer), the OOB byte currently queued can be lost as it is replaced by the new OOB data block (as occurs in Berkeley Software Distribution). It is never replaced in the data stream, however.

With `SO_OOBINLINE` enabled, the urgent data remains in the data stream. As a result, the OOB data block is never lost when a new TCP segment arrives containing urgent data. The existing OOB data "mark" is updated to the new position.

## Summary of Windows Sockets 2 Functions

The following tables summarize the functions included in Windows Sockets 2, separated into two groups: Berkeley-style functions, and Microsoft Windows-specific Extension functions that have been ratified as part of the Windows Sockets 2 specification. These tables do not include the Windows Sockets functions known that are used with Registration and Name Resolution.

### Socket Functions

The Windows Sockets specification includes all the following Berkeley-style socket routines that were part of the Windows Sockets 1.1 API.

Routine	Meaning
<b>accept</b> <sup>1</sup>	An incoming connection is acknowledged and associated with an immediately created socket. The original socket is returned to the listening state.
<b>bind</b>	Assigns a local name to an unnamed socket.
<b>closesocket</b>	Removes a socket from the per-process object reference table. Only blocks if <code>SO_LINGER</code> is set with a nonzero time-out on a blocking socket.
<b>connect</b> <sup>1</sup>	Initiates a connection on the specified socket.

Routine	Meaning
<b>getpeername</b>	Retrieves the name of the peer connected to the specified socket.
<b>getsockname</b>	Retrieves the local address to which the specified socket is bound.
<b>getsockopt</b>	Retrieves options associated with the specified socket.
<b>htonl<sup>2</sup></b>	Converts a 32-bit quantity from host-byte order to network-byte order.
<b>htons<sup>2</sup></b>	Converts a 16-bit quantity from host-byte order to network-byte order.
<b>inet_addr<sup>2</sup></b>	Converts a character string representing a number in the Internet standard "." notation to an Internet address value.
<b>inet_ntoa<sup>2</sup></b>	Converts an Internet address value to an ASCII string in "." notation that is, "a.b.c.d".
<b>ioctlsocket</b>	Provides control for sockets.
<b>listen</b>	Listens for incoming connections on a specified socket.
<b>ntohl<sup>2</sup></b>	Converts a 32-bit quantity from network-byte order to host-byte order.
<b>ntohs<sup>2</sup></b>	Converts a 16-bit quantity from network byte order to host byte order.
<b>recv<sup>1</sup></b>	Receives data from a connected or unconnected socket.
<b>recvfrom<sup>1</sup></b>	Receives data from either a connected or unconnected socket.
<b>select<sup>1</sup></b>	Performs synchronous I/O multiplexing.
<b>send<sup>1</sup></b>	Sends data to a connected socket.
<b>sendto<sup>1</sup></b>	Sends data to either a connected or unconnected socket.
<b>setsockopt</b>	Stores options associated with the specified socket.
<b>shutdown</b>	Shuts down part of a full-duplex connection.
<b>socket</b>	Creates an endpoint for communication and returns a socket descriptor.

1 The routine can block if acting on a blocking socket.

2 The routine is retained for backward compatibility with Windows Sockets 1.1, and should only be used for sockets created with AF\_INET address family.

## Microsoft Windows-Specific Extension Functions

The Windows Sockets specification provides a number of extensions to the standard set of Berkeley Sockets routines. Principally, these extended functions allow message or function-based, asynchronous access to network events, as well as enable overlapped I/O. While use of this extended API set is not mandatory for socket-based programming (with the exception of **WSAStartup** and **WSACleanup**), it is recommended for conformance with the Microsoft Windows programming paradigm. For features introduced in Windows Sockets 2, please see *New Concepts, Additions and Changes for Windows Sockets 2*.

Routine	Meaning
<b>WSAAccept</b> <sup>1</sup>	An extended version of <b>accept</b> , which allows for conditional acceptance.
<b>WSAAsyncGetHostByAddr</b> <sup>2, 3</sup>	A set of functions that provide asynchronous versions of the standard Berkeley getXbyY functions. For example, the <b>WSAAsyncGetHostByName</b> function provides an asynchronous, message-based implementation of the standard Berkeley <b>gethostbyname</b> function.
<b>WSAAsyncGetHostByName</b> <sup>2, 3</sup>	
<b>WSAAsyncGetProtoByName</b> <sup>2, 3</sup>	
<b>WSAAsyncGetProtoByNumber</b> <sup>2, 3</sup>	
<b>WSAAsyncGetServByName</b> <sup>2, 3</sup>	
<b>WSAAsyncGetServByPort</b> <sup>2, 3</sup>	
<b>WSAAsyncSelect</b> <sup>3</sup>	
<b>WSACancelAsyncRequest</b> <sup>2, 3</sup>	Cancels an outstanding instance of a <b>WSAAsyncGetXBY</b> function.
<b>WSACleanup</b>	Signs off from the underlying Windows Sockets .dll.
<b>WSACloseEvent</b>	Destroys an event object.
<b>WSAConnect</b> <sup>1</sup>	An extended version of connect which allows for exchange of connect data and QOS specification.
<b>WSACreateEvent</b>	Creates an event object.
<b>WSADuplicateSocket</b>	Allows an underlying socket to be shared by creating a virtual socket.
<b>WSAEnumNetworkEvents</b>	Discovers occurrences of network events.
<b>WSAEnumProtocols</b>	Retrieves information about each available protocol.
<b>WSAEventSelect</b>	Associates network events with an event object.
<b>WSAGetLastError</b> <sup>3</sup>	Obtains details of last Windows Sockets error.
<b>WSAGetOverlappedResult</b>	Gets completion status of overlapped operation.
<b>WSAGetQOSByName</b>	Supplies QOS parameters based on a well-known service name.
<b>WSAHtonl</b>	Extended version of <b>htonl</b> .
<b>WSAhtons</b>	Extended version of <b>htons</b> .
<b>WSAIoctl</b> <sup>1</sup>	Overlapped-capable version of IOCTL.
<b>WSAJoinLeaf</b> <sup>1</sup>	Adds a multipoint leaf to a multipoint session.
<b>WSANTohl</b>	Extended version of <b>ntohl</b> .
<b>WSANTohs</b>	Extended version of <b>ntohs</b> .

Routine	Meaning
<b>WSAProviderConfigChange</b>	Receives notifications of service providers being installed/removed.
<b>WSARecv</b> <sup>1</sup>	An extended version of <b>recv</b> which accommodates scatter/gather I/O, overlapped sockets and provides the <i>flags</i> parameter as in, out.
<b>WSARecvFrom</b> <sup>1</sup>	An extended version of <b>recvfrom</b> which accommodates scatter/gather I/O, overlapped sockets and provides the <i>flags</i> parameter as in, out.
<b>WSAResetEvent</b>	Resets an event object.
<b>WSASend</b> <sup>1</sup>	An extended version of <b>send</b> which accommodates scatter/gather I/O and overlapped sockets.
<b>WSASendTo</b> <sup>1</sup>	An extended version of <b>sendto</b> which accommodates scatter/gather I/O and overlapped sockets.
<b>WSASetEvent</b>	Sets an event object.
<b>WSASetLastError</b> <sup>3</sup>	Sets the error to be returned by a subsequent <b>WSAGetLastError</b> .
<b>WSASocket</b>	An extended version of <b>socket</b> which takes a <b>WSAPROTOCOL_INFO</b> structure as input and allows overlapped sockets to be created.
<b>WSAStartup</b> <sup>3</sup>	Initializes the underlying Windows Sockets .dll.
<b>WSAWaitForMultipleEvents</b> <sup>1</sup>	Blocks on multiple event objects.

1 The routine can block if acting on a blocking socket.

2 The routine is always realized by the name resolution provider associated with the default TCP/IP service provider, if any.

3 The routine was originally a Windows Sockets 1.1 function

## Registration and Name Resolution

Windows Sockets 2 includes a new set of API functions that standardize the way applications access and use the various network naming services. When using these new functions, Windows Sockets 2 applications need not be cognizant of the widely differing protocols associated with name services such as DNS, NIS, X.500, SAP, etc. To maintain full backward compatibility with Windows Sockets 1.1, all of the existing **getXbyY** and asynchronous **WSAAsyncGetXbyY** database lookup functions continue to be supported, but are implemented in the Windows Sockets service provider interface in terms of the new name resolution capabilities. For more information, see the **getservbyname** and **getservbyport** functions. Also, see *Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 SPI*.

## Protocol-Independent Name Resolution

In developing a protocol-independent client/server application, there are two basic requirements that exist with respect to name resolution and registration:

- The ability of the server half of the application (service) to register its existence within (or become accessible to) one or more namespaces.
- The ability of the client application to find the service within a namespace and obtain the required transport protocol and addressing information.

For those accustomed to developing TCP/IP-based applications, this may seem to involve little more than looking up a host address and then using an agreed-upon port number. Other networking schemes however, allow the location of the service, the protocol used for the service, and other attributes to be discovered at run-time. To accommodate the broad diversity of capabilities found in existing name services, the Windows Sockets 2 interface adopts the model described in the following.

### Name Resolution Model

A *namespace* refers to some capability to associate (as a minimum) the protocol and addressing attributes of a network service with one or more human-friendly names. Many namespaces are currently in wide use, including the Internet's Domain Name System (DNS), the bindery and Netware Directory Services (NDS) from Novell®, X.500, etc. These namespaces vary widely in how they are organized and implemented. Some of their properties are particularly important to understand from the perspective of Windows Sockets name resolution.

### Types of Namespaces

There are three different types of namespaces in which a service could be registered:

- Dynamic
- Static
- Persistent

Dynamic namespaces allow services to register with the namespace on the fly, and for clients to discover the available services at run-time. Dynamic namespaces frequently rely on broadcasts to indicate the continued availability of a network service. Examples of dynamic namespaces include the SAP namespace used within a Netware® environment and the NBP namespace used by Appletalk®.

Static namespaces require all of the services to be registered ahead of time, that is, when the namespace is created. The DNS is an example of a static namespace. Although there is a programmatic way to resolve names, there is no programmatic way to register names.

Persistent namespaces allow services to register with the namespace on the fly. Unlike dynamic namespaces however, persistent namespaces retain the registration information in non-volatile storage where it remains until such time as the service

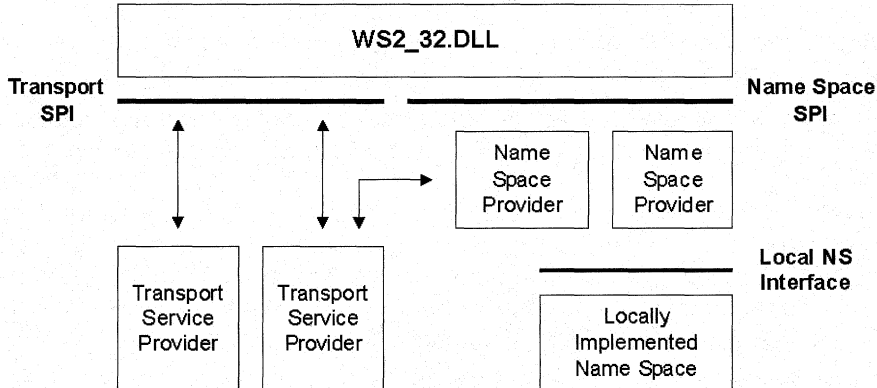
requests that it be removed. Persistent namespaces are typified by directory services such as X.500 and the NDS (Netware Directory Service). These environments allow the adding, deleting, and modification of service properties. In addition, the service object representing the service within the directory service could have a variety of attributes associated with the service. The most important attribute for client applications is the service's addressing information.

## Namespace Organization

Many namespaces are arranged hierarchically. Some, such as X.500 and NDS, allow unlimited nesting. Others allow services to be combined into a single level of hierarchy or group. This is typically referred to as a *workgroup*. When constructing a query, it is often necessary to establish a context point within a namespace hierarchy from which the search will begin.

## Namespace Provider Architecture

Naturally, the programmatic interfaces used to query the various types of namespaces and to register information within a namespace (if supported) differ widely. A *namespace provider* is a locally-resident piece of software that knows how to map between the Windows Sockets namespace SPI and some existing namespace (which could be implemented locally or be accessed through the network). Figure 6-4 shows the namespace provider architecture.



**Figure 6-4: Namespace Provider Architecture.**

Note that it is possible for a given namespace, say DNS, to have more than one namespace provider installed on a given machine.

As mentioned above, the generic term *service* refers to the server-half of a client/server application. In Windows Sockets, a service is associated with a *service class*, and each instance of a particular service has a *service name* which must be unique within the service class. Examples of service classes include FTP Server, SQL Server, XYZ Corp. Employee Info Server, etc. As the example attempts to illustrate, some service classes are well known while others are unique and specific to a particular vertical application.

In either case, every service class is represented by both a class name and a class identifier. The class name does not necessarily need to be unique, but the class identifier must be. Globally Unique Identifiers (GUIDs) are used to represent service class identifiers. For well-known services, class names and class identifiers (GUIDs) have been pre-allocated, and macros are available to convert between, for example, TCP port numbers (in host-byte order) and the corresponding class identifier GUIDs. For other services, the developer chooses the class name and uses the `Uuidgen.exe` utility to generate a GUID for the class identifier.

The notion of a service class exists to allow a set of attributes to be established that are held in common by all instances of a particular service. This set of attributes is provided at the time the service class is defined to Windows Sockets, and is referred to as the service class schema information. When a service is installed and made available on a host machine, that service is considered *instantiated*, and its service name is used to distinguish a particular instance of the service from other instances which may be known to the namespace.

Note that the installation of a service class only needs to occur on machines where the service executes, not on all of the clients which may utilize the service. Where possible, the `Ws2_32.dll` provides service class schema information to a namespace provider at the time an instantiation of a service is to be registered or a service query is initiated. The `Ws2_32.dll` does not, of course, store this information itself, but attempts to retrieve it from a namespace provider that has indicated its ability to supply this data. Since there is no guarantee that the `Ws2_32.dll` can supply the service class schema, namespace providers that need this information must have a fallback mechanism to obtain it through namespace-specific means.

As noted above, the Internet has adopted what can be termed a host-centric service model. Applications needing to locate the transport address of a service generally must first resolve the address of a specific host known to host the service. To this address they add in the well-known port number and thus create a complete transport address. To facilitate the resolution of host names, a special service class identifier has been pre-allocated (`SVCID_HOSTNAME`). A query that specifies `SVCID_HOSTNAME` as the service class and uses the host name the service instance name will, if the query is successful, return host address information.

In Windows Sockets 2, applications that are protocol-independent should avoid the need to comprehend the internal details of a transport address. Thus, the need to first get a host address and then add in the port is problematic. To avoid this, queries may also include the well-known name of a particular service and the protocol over which the service operates, such as FTP or TCP. In this case, a successful query returns a complete transport address for the specified service on the indicated host, and the application is not required to crack open a **SOCKADDR** structure. This is described in more detail in the following.

The Internet's Domain Name System does not have a well-defined means to store service class schema information. As a result, DNS namespace providers can only accommodate well-known TCP/IP services for which a service class GUID has been preallocated.

In practice, this is not a serious limitation since service class GUIDs have been preallocated for the entire set of TCP and UDP ports, and macros are available to retrieve the GUID associated with any TCP or UDP port with the port expressed in host-byte order. Thus, all of the familiar services such as FTP, Telnet, Whois, etc. are well supported.

Continuing with our service class example, instance names of the FTP service may be “alder.intel.com” or “rhino.microsoft.com” while an instance of the XYZ Corp. Employee Info Server might be named “XYZ Corp. Employee Info Server Version 3.5”.

In the first two cases, the combination of the service class GUID for FTP and the machine name (supplied as the service instance name) uniquely identify the desired service. In the third case, the host name where the service resides can be discovered at service query time, so the service instance name does not need to include a host name.

## Summary of Name Resolution Functions

The name resolution functions can be grouped into three categories: Service installation, client queries, and helper functions (and macros). The sections that follow identify the functions in each category and briefly describe their intended use. Key data structures are also described.

### Service Installation

- **WSAInstallServiceClass**
- **WSARemoveServiceClass**
- **WSASetService**

When the required service class does not already exist, an application uses **WSAInstallServiceClass** to install a new service class by supplying a service class name, a GUID for the service class identifier, and a series of **WSANSCLASSINFO** structures. These structures are each specific to a particular namespace, and supply common values such as recommended TCP port numbers or Netware SAP Identifiers. A service class can be removed by calling **WSARemoveServiceClass** and supplying the GUID corresponding to the class identifier.

Once a service class exists, specific instances of a service can be installed or removed through **WSASetService**. This function takes a **WSAQUERYSET** structure as an input parameter along with an operation code and operation flags. The operation code indicates whether the service is being installed or removed. The **WSAQUERYSET** structure provides all of the relevant information about the service including service class identifier, service name (for this instance), applicable namespace identifier and protocol information, and a set of transport addresses at which the service listens. Services should invoke **WSASetService** when they initialize to advertise their presence in dynamic namespaces.



## Client Query

- **WSAEnumNameSpaceProviders**
- **WSALookupServiceBegin**
- **WSALookupServiceNext**
- **WSALookupServiceEnd**

The **WSAEnumNameSpaceProviders** function allows an application to discover which namespaces are accessible through Windows Sockets name resolution facilities. It also allows an application to determine whether a given namespace is supported by more than one namespace provider, and to discover the provider identifier for any particular namespace provider. Using a provider identifier, the application can restrict a query operation to a specified namespace provider.

Windows Sockets' namespace query operations involve a series of calls:

**WSALookupServiceBegin**, followed by one or more calls to **WSALookupServiceNext** and ending with a call to **WSALookupServiceEnd**. **WSALookupServiceBegin** takes a **WSAQUERYSET** structure as input to define the query parameters along with a set of flags to provide additional control over the search operation. It returns a query handle which is used in the subsequent calls to **WSALookupServiceNext** and **WSALookupServiceEnd**.

The application invokes **WSALookupServiceNext** to obtain query results, with results supplied in an application-supplied **WSAQUERYSET** buffer. The application continues to call **WSALookupServiceNext** until the error code **WSA\_E\_NO\_MORE** is returned indicating that all results have been retrieved. The search is then terminated by a call to **WSALookupServiceEnd**. The **WSALookupServiceEnd** function can also be used to cancel a currently pending **WSALookupServiceNext** when called from another thread.

In Windows Socket 2, conflicting error codes are defined for **WSAENOMORE** (10102) and **WSA\_E\_NO\_MORE** (10110). The error code **WSAENOMORE** will be removed in a future version and only **WSA\_E\_NO\_MORE** will remain. For Windows Socket 2, however, applications should check for both **WSAENOMORE** and **WSA\_E\_NO\_MORE** for the widest possible compatibility with Namespace Providers that use either one.

## Helper Functions

- **WSAGetServiceClassNameByClassId**
- **WSAAddressToString**
- **WSAStringToAddress**
- **WSAGetServiceClassInfo**

The name resolution helper functions include a function to retrieve a service class name given a service class identifier, a pair of functions used to translate a transport address between a **SOCKADDR** structure and an ASCII string representation, a function to retrieve the service class schema information for a given service class, and a set of macros for mapping well known services to pre-allocated GUIDs.

The following macros from `Winsock2.h` aid in mapping between well known service classes and these namespaces.

Macro	Description
<code>SVCID_TCP(Port)</code> <code>SVCID_UDP(Port)</code> <code>SVCID_NETWARE(Object Type)</code>	Given a port for TCP/IP or UDP/IP or the object type in the case of Netware, returns the GUID (port number in host order).
<code>IS_SVCID_TCP(GUID)</code> <code>IS_SVCID_UDP(GUID)</code> <code>IS_SVCID_NETWARE(GUID)</code>	Returns TRUE if the GUID is within the allowable range.
<code>SET_TCP_SVCID(GUID, port)</code> <code>SET_UDP_SVCID(GUID, port)</code>	Initializes a GUID structure with the GUID equivalent for a TCP or UDP port number (port number must be in host order).
<code>PORT_FROM_SVCID_TCP(GUID)</code> <code>PORT_FROM_SVCID_UDP(GUID)</code> <code>SAPID_FROM_SVCID_NETWARE(GUID)</code>	Returns the port or object type associated with the GUID (port number in host order).

## Name Resolution Data Structures

There are several important data structures that are used extensively throughout the name resolution functions. These are described in the following.

### Query-Related Data Structures

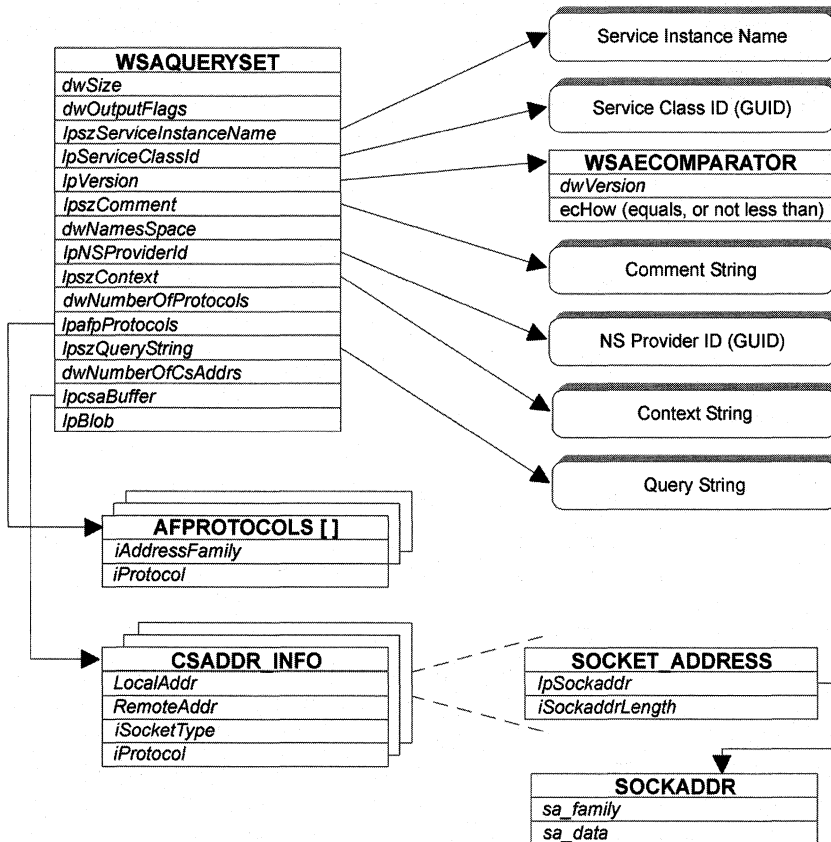
The **WSAQUERYSET** structure is used to form queries for **WSALookupServiceBegin**, and used to deliver query results for **WSALookupServiceNext**. It is a complex structure since it contains pointers to several other structures, some of which reference still other structures. The relationship between **WSAQUERYSET** and the structures it references is illustrated in Figure 6-5.

Within the **WSAQUERYSET** structure, most of the parameters are self explanatory, but some deserve additional explanation. The *dwSize* parameter must always be filled in with `sizeof(WSAQUERYSET)`, as this is used by namespace providers to detect and adapt to different versions of the **WSAQUERYSET** structure that may appear over time.

The *dwOutputFlags* parameter is used by a namespace provider to provide additional information about query results. For details, see **WSALookupServiceNext**.

The **WSAECOMPARATOR** structure referenced by *lpversion* is used for both query constraint and results. For queries, the *dwVersion* parameter indicates the desired version of the service. The *echow* parameter is an enumerated type which specifies how the comparison can be made. The choices are `COMP_EQUALS` which requires that an exact match in version occurs, or `COMP_NOTLESS` which specifies that the service's version number be no less than the value of *dwVersion*.

The interpretation of *dwNameSpace* and *IpNSProviderId* depends upon how the structure is being used and is described further in the individual function descriptions that utilize this structure.



**Figure 6-5: Data Structure Relationships.**

The *lpContext* parameter applies to hierarchical namespaces, and specifies the starting point of a query or the location within the hierarchy where the service resides. The general rules are:

- A value of NULL, blank ("") starts the search at the default context.
- A value of "\" starts the search at the top of the namespace.
- Any other value starts the search at the designated point.

Providers that do not support containment may return an error if anything other than "" or "\" is specified. Providers that support limited containment, such as groups, should accept "", "\", or a designated point. Contexts are namespace specific. If *dwNameSpace* is NS\_ALL, then only "" or "\" should be passed as the context since these are recognized by all namespaces.

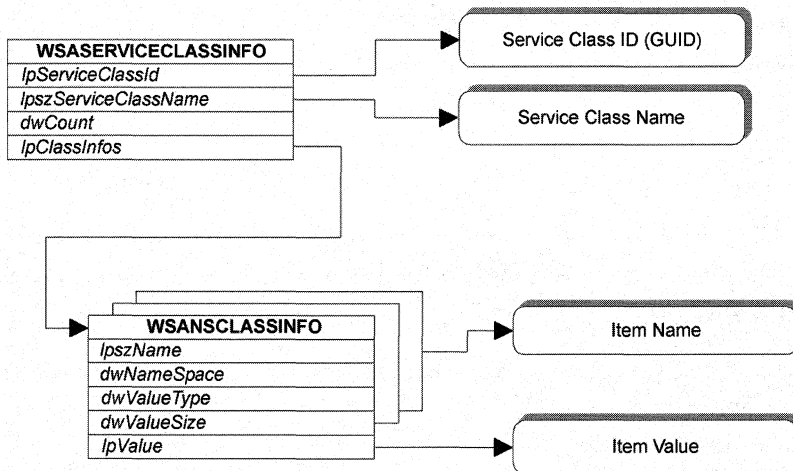
The *lpzQueryString* parameter is used to supply additional, namespace-specific query information such as a string describing a well-known service and transport protocol name, as in “FTP/TCP”.

The **AFPROTOCOLS** structure referenced by *lpafpProtocols* is used for query purposes only, and supplies a list of protocols to constrain the query. These protocols are represented as (address family, protocol) pairs, since protocol values only have meaning within the context of an address family.

The array of **CSADDR\_INFO** structure referenced by *lpcsaBuffer* contain all of the information needed to for either a service to use in establishing a listen, or a client to use in establishing a connection to the service. The *LocalAddr* and *RemoteAddr* parameters both directly contain a **SOCKET\_ADDRESS** structure. A service would create a socket using the tuple (*LocalAddr.lpSockaddr->sa\_family*, *iSocketType*, *iProtocol*). It would bind the socket to a local address using *LocalAddr.lpSockaddr*, and *LocalAddr.lpSockaddrLength*. The client creates its socket with the tuple (*RemoteAddr.lpSockaddr->sa\_family*, *iSocketType*, *iProtocol*), and uses the combination of *RemoteAddr.lpSockaddr*, and *RemoteAddr.lpSockaddrLength* when making a remote connection.

## Service Class Data Structures

When a new service class is installed, a **WSASERVICECLASSINFO** structure must be prepared and supplied. This structure also consists of substructures that contain a series of parameters that apply to specific namespaces. Figure 6-6 shows a class info data structure.



**Figure 6-6: Class Info Data Structure.**

For each service class, there is a single **WSASERVICECLASSINFO** structure. Within the **WSASERVICECLASSINFO** structure, the service class' unique identifier is contained in *IpServiceClassId*, and an associated display string is referenced by *IpServiceClassName*. This is the string that is returned by **WSAGetServiceClassNameByClassId**.

The *IpClassInfos* parameter in the **WSASERVICECLASSINFO** structure references an array of **WSANSCLASSINFO** structures, each of which supplies a named and typed parameter that applies to a specified namespace. Examples of values for the *IpszName* parameter include: "SapId", "TcpPort", "UdpPort", etc. These strings are generally specific to the namespace identified in *dwNameSpace*. Typical values for *dwValueType* might be **REG\_DWORD**, **REG\_SZ**, etc. The *dwValueSize* parameter indicates the length of the data item pointed to by *IpValue*.

The entire collection of data represented in a **WSASERVICECLASSINFO** structure is provided to each namespace provider when **WSAInstallServiceClass** is invoked. Each individual namespace provider then sifts through the list of **WSANSCLASSINFO** structures and retains the information applicable to it.

## Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 API

Windows Sockets 1.1 defined a number of routines that were used for name resolution with TCP/IP (IP version 4) networks. These are customarily called the **getXbyY** functions and include the following.

**gethostname**  
**gethostbyaddr**  
**gethostbyname**  
**getprotobyname**  
**getprotobynumber**  
**getservbyname**  
**getservbyport**

Asynchronous versions of these functions were also defined.

**WSAAsyncGetHostByAddr**  
**WSAAsyncGetHostByName**  
**WSAAsyncGetProtoByName**  
**WSAAsyncGetProtoByNumber**  
**WSAAsyncGetServByName**  
**WSAAsyncGetServByPort**

There are also two functions (now implemented in the *Winsock2.dll*) used to convert dotted Ipv4 internet address notation to and from string and binary representations, respectively.

**inet\_addr**  
**inet\_ntoa**

All of these functions are specific to Ipv4 TCP/IP networks and developers of protocol-independent applications are discouraged from continuing to utilize these transport-specific functions. However, in order to retain strict backward compatibility with Windows Sockets 1.1, all of the above functions continue to be supported as long as at least one namespace provider is present that supports the AF\_INET address family (these functions are not relevant to IP version 6, denoted by AF\_INET6).

The Ws2\_32.dll implements these compatibility functions in terms of the new, protocol-independent name resolution facilities using an appropriate sequence of **WSALookupServiceBegin/Next/End** function calls. The details of how the **getXbyY** functions are mapped to name resolution functions are provided below. The Ws2\_32.dll handles the differences between the asynchronous and synchronous versions of the **getXbyY** functions, so only the implementation of the synchronous **getXbyY** functions are discussed.

## Basic Approach for GetXbyY in the API

Most **getXbyY** functions are translated by the Ws2\_32.dll to a **WSALookupServiceBegin/Next/End** sequence that uses one of a set of special GUIDs as the service class. These GUIDs identify the type of **getXbyY** operation that is being emulated. The query is constrained to those NSPs that support AF\_INET. Whenever a **getXbyY** function returns a hostent or servent structure, the Ws2\_32.dll specifies the LUP\_RETURN\_BLOB flag in **WSALookupServiceBegin** so that the desired information is returned by the NSP. These structures must be modified slightly in that the pointers contained within must be replaced with offsets that are relative to the start of the blob's data. All values referenced by these pointer parameters must, of course, be completely contained within the blob, and all strings are ASCII.

## getprotobyname and getprotobynumber Functions in the API

These functions are implemented within the Ws2\_32.DLL by consulting a local protocols database. They do not result in any name resolution query.

## getservbyname and getservbyport Functions in the API

The **WSALookupServiceBegin** query uses SVCID\_INET\_SERVICEBYNAME as the service class GUID. The *lpzServiceInstanceName* parameter references a string which indicates the service name or service port, and (optionally) the service protocol. The formatting of the string is illustrated as FTP or TCP or 21/TCP or just FTP. The string is not case sensitive. The slash mark, if present, separates the protocol identifier from the preceding part of the string. The Ws2\_32.dll will specify LUP\_RETURN\_BLOB and the NSP will place a **SERVENT** structure in the blob (using offsets instead of pointers as described above). NSPs should honor these other LUP\_RETURN\_\* flags as well.

Flag	Description
LUP_RETURN_NAME	Returns the <i>s_name</i> parameter from <b>SERVENT</b> structure in <i>lpzServiceInstanceName</i> .
LUP_RETURN_TYPE	Returns canonical GUID in <i>lpServiceClassId</i> . It is understood that a service identified as FTP or 21 may be on another port according to locally established conventions. The <i>s_port</i> parameter of the <b>SERVENT</b> structure should indicate where the service can be contacted in the local environment. The canonical GUID returned when LUP_RETURN_TYPE is set should be one of the predefined GUIDs from Svcs.h that corresponds to the port number indicated in the <b>SERVENT</b> structure.

### gethostbyname Function in the API

The **WSALookupServiceBegin** query uses SVCID\_INET\_HOSTADDRBYNAME as the service class GUID. The host name is supplied in *lpzServiceInstanceName*. The Ws2\_32.DLL specifies LUP\_RETURN\_BLOB and the NSP places a **HOSTENT** structure in the blob (using offsets instead of pointers as described above). NSPs should honor these other LUP\_RETURN\_\* flags as well.

Flag	Description
LUP_RETURN_NAME	Returns the <i>h_name</i> parameter from <b>HOSTENT</b> structure in <i>lpzServiceInstanceName</i> .
LUP_RETURN_ADDR	Returns addressing information from <b>HOSTENT</b> in <b>CSADDR_INFO</b> structures, port information is defaulted to zero. Note that this routine does <i>not</i> resolve host names that consist of a dotted internet address.

### gethostbyaddr Function in the API

The **WSALookupServiceBegin** query uses SVCID\_INET\_HOSTNAMEBYADDR as the service class GUID. The host address is supplied in *lpzServiceInstanceName* as a dotted internet string (for example, "192.9.200.120"). The Ws2\_32.DLL specifies LUP\_RETURN\_BLOB and the NSP places a **HOSTENT** structure in the blob (using offsets instead of pointers as described above). NSPs should honor these other LUP\_RETURN\_\* flags as well.

Flag	Description
LUP_RETURN_NAME	Returns the <i>h_name</i> parameter from <b>HOSTENT</b> structure in <i>lpzServiceInstanceName</i> .
LUP_RETURN_ADDR	Returns addressing information from <b>HOSTENT</b> in <b>CSADDR_INFO</b> structures, port information is defaulted to zero.

## gethostname Function in the API

The **WSALookupServiceBegin** query uses **SVCID\_HOSTNAME** as the service class GUID. If *IpszServiceInstanceName* is NULL or references a NULL string (that is ""), the local host is to be resolved. Otherwise, a lookup on a specified host name occurs. For the purposes of emulating **gethostname** the **Ws2\_32.DLL** specifies a null pointer for *IpszServiceInstanceName*, and specifies **LUP\_RETURN\_NAME** so that the host name is returned in the *IpszServiceInstanceName* parameter. If an application uses this query and specifies **LUP\_RETURN\_ADDR** then the host address is provided in a **CSADDR\_INFO** structure. The **LUP\_RETURN\_BLOB** action is undefined for this query. Port information is defaulted to zero unless the *IpszQueryString* references a service such as FTP, in which case the complete transport address of the indicated service is supplied.

## Multipoint and Multicast Semantics

In considering how to support multipoint and multicast semantics in Windows Sockets 2 a number of existing and proposed multipoint/multicast schemes (including IP-multicast, ATM point-to-multipoint connection, ST-II, T.120, H.320 (MCU), and so on) were examined. While alike in some aspects, each is unlike in others. To enable a coherent discussion of the various schemes, it is valuable to first create a taxonomy that characterizes the essential attributes of each. In this document, the term *multipoint* represents both multipoint and multicast.

## Multipoint Taxonomy

The taxonomy described in this section first distinguishes the control plane that concerns itself with the way a multipoint session is established, from the data plane that deals with the transfer of data among session participants.

In the control plane there are two distinct types of session establishment: *rooted* and *nonrooted*. In the case of rooted control, there exists a special participant, called *c\_root*, that is different from the rest of the members of this multipoint session, each of which is called a *c\_leaf*. The *c\_root* must remain present for the whole duration of the multipoint session, as the session is broken up in the absence of the *c\_root*. The *c\_root* usually initiates the multipoint session by setting up the connection to a *c\_leaf*, or a number of *c\_leafs*. The *c\_root* may add more *c\_leafs*, or (in some cases) a *c\_leaf* can join the *c\_root* at a later time. Examples of the rooted control plane can be found in ATM and ST-II.

For a nonrooted control plane, all the members belonging to a multipoint session are leaves, that is, no special participant acting as a *c\_root* exists. Each *c\_leaf* must add itself to a preexisting multipoint session that is always available (as in the case of an IP multicast address), or has been set up through some OOB mechanism which is outside the scope of the Windows Sockets specification.



Another way to look at this is that a *c\_root* still exists, but can be considered to be in the network cloud as opposed to one of the participants. Because a control root still exists, a nonrooted control plane could also be considered to be implicitly rooted. Examples for this kind of implicitly rooted multipoint schemes are:

- A teleconferencing bridge.
- The IP multicast system.
- A Multipoint Control Unit (MCU) in a H.320 video conference.

In the data plane, there are two types of data transfer styles: *rooted* and *nonrooted*. In a rooted data plane, a special participant called *d\_root* exists. Data transfer only occurs between the *d\_root* and the rest of the members of this multipoint session, each of which is referred to as a *d\_leaf*. The traffic could be unidirectional or bi-directional. The data sent out from the *d\_root* is duplicated (if required) and delivered to every *d\_leaf*, while the data from *d\_leafs* only goes to the *d\_root*. In the case of a rooted data plane, no traffic is allowed among *d\_leafs*. An example of a protocol that is rooted in the data plane is ST-II.

In a nonrooted data plane, all the participants are equal, that is, any data they send is delivered to all the other participants in the same multipoint session. Likewise each *d\_leaf* node can receive data from all other *d\_leafs*, and in some cases, from other nodes that are not participating in the multipoint session. No special *d\_root* node exists. IP-multicast is nonrooted in the data plane.

Note that the question of where data unit duplication occurs, or whether a shared single tree or multiple shortest-path trees are used for multipoint distribution are protocol issues, and irrelevant to the interface the application would use to perform multipoint communications. Therefore these issues are not addressed in this appendix or the Windows Sockets interface.

The following table depicts the taxonomy described above and indicates how existing schemes fit into each of the categories. Note that there do not appear to be any existing schemes that employ a nonrooted control plane along with a rooted data plane.

	Rooted control plane	Nonrooted (implicit rooted) control plane
Rooted data plane	ATM, ST-II	No known examples.
Nonrooted data plane	T.120	IP-multicast, H.320 (MCU)

## Windows Sockets 2 Interface Elements for Multipoint and Multicast

The mechanisms that have been incorporated into Windows Sockets 2 for utilizing multipoint capabilities can be summarized as follows:

Three attribute bits in the **WSAPROTOCOL\_INFO** structure.

- Four flags defined for the *dwFlags* parameter of **WSASocket**.
- One function, **WSAJoinLeaf**, for adding leaf nodes into a multipoint session.
- Two **WSAIoctl** command codes for controlling multipoint loopback and the scope of multicast transmissions.

The following paragraphs describe these interface elements in more detail:

- Semantics for Joining Multipoint Leaves
- How Existing Multipoint Protocols Support These Extensions

## Attributes in WSAPROTOCOL\_INFO Structure

In support of the taxonomy described above, three attribute parameters in the **WSAPROTOCOL\_INFO** structure are used to distinguish the schemes used in the control and data planes respectively:

- **XP1\_SUPPORT\_MULTIPPOINT** with a value of 1 indicates this protocol entry supports multipoint communications, and that the following two parameters are meaningful.
- **XP1\_MULTIPPOINT\_CONTROL\_PLANE** indicates whether the control plane is rooted (value = 1) or nonrooted (value = 0).
- **XP1\_MULTIPPOINT\_DATA\_PLANE** indicates whether the data plane is rooted (value = 1) or nonrooted (value = 0).

Note that two **WSAPROTOCOL\_INFO** entries would be present if a multipoint protocol supported both rooted and nonrooted data planes, one entry for each.

The application can use **WSAEnumProtocols** to discover whether multipoint communications is supported for a given protocol and, if so, how it is supported with respect to the control and data planes, respectively.

## Flag Bits for WSASocket

In some instances sockets joined to a multipoint session may have some behavioral differences from point-to-point sockets. For example, a *d\_leaf* socket in a rooted data plane scheme can only send information to the *d\_root* participant. This creates a need for the application to be able to indicate the intended use of a socket coincident with its creation. This is done through four-flag bits that can be set in the *dwFlags* parameter to **WSASocket**:

- **WSA\_FLAG\_MULTIPPOINT\_C\_ROOT**, for the creation of a socket acting as a *c\_root*, and only allowed if a rooted control plane is indicated in the corresponding **WSAPROTOCOL\_INFO** entry.
- **WSA\_FLAG\_MULTIPPOINT\_C\_LEAF**, for the creation of a socket acting as a *c\_leaf*, and only allowed if **XP1\_SUPPORT\_MULTIPPOINT** is indicated in the corresponding **WSAPROTOCOL\_INFO** entry.

- `WSA_FLAG_MULTIPOINT_D_ROOT`, for the creation of a socket acting as a `d_root`, and only allowed if a rooted data plane is indicated in the corresponding `WSAPROTOCOL_INFO` entry.
- `WSA_FLAG_MULTIPOINT_D_LEAF`, for the creation of a socket acting as a `d_leaf`, and only allowed if `XP1_SUPPORT_MULTIPOINT` is indicated in the corresponding `WSAPROTOCOL_INFO` entry.

Note that when creating a multipoint socket, exactly one of the two control-plane flags, and one of the two data-plane flags must be set in `WSASocket`'s `dwFlags` parameter. Thus, the four possibilities for creating multipoint sockets are:

- “`c_root/d_root`”
- “`c_root/d_leaf`”
- “`c_leaf/d_root`”
- “`c_leaf /d_leaf`”

### **SIO\_MULTIPOINT\_LOOPBACK Command Code for `WSAIoctl`**

When `d_leaf` sockets are used in a nonrooted data plane, it is desirable to have traffic that is sent out received back on the same socket. The `SIO_MULTIPOINT_LOOPBACK` command code for `WSAIoctl` is used to enable or disable loopback of multipoint traffic.

### **SIO\_MULTICAST\_SCOPE Command Code for `WSAIoctl`**

When multicasting is employed, it is usually necessary to specify the scope over which the multicast should occur. Scope is defined as the number of routed network segments to be covered. A scope of zero would indicate that the multicast transmission would not be placed on the wire but could be disseminated across sockets within the local host. A scope value of one (the default) indicates that the transmission will be placed on the wire, but will not cross any routers. Higher scope values determine the number of routers that may be crossed. Note that this corresponds to the time-to-live (TTL) parameter in IP multicasting.

The function `WSAJoinLeaf` is used to join a leaf node into the multipoint session. See the following for a discussion on how this function is utilized.

## **Semantics for Joining Multipoint Leaves**

In the following, a multipoint socket is frequently described by defining its role in one of the two planes (control or data). It should be understood that this same socket has a role in the other plane, but this is not mentioned in order to keep the references short. For example when a reference is made to a “`c_root` socket”, this could be either a `c_root/d_root` or a `c_root/d_leaf` socket.

In rooted control plane schemes, new leaf nodes are added to a multipoint session in one or both of two different ways. In the first method, the root uses `WSAJoinLeaf` to initiate a connection with a leaf node and invite it to become a participant. On the leaf

node, the peer application must have created a `c_leaf` socket and used `listen` to set it into listen mode. The leaf node receives an `FD_ACCEPT` indication when invited to join the session, and signals its willingness to join by calling `WSAAccept`. The root application then receives an `FD_CONNECT` indication when the join operation has been completed.

In the second method, the roles are essentially reversed. The root application creates a `c_root` socket and sets it into listen mode. A leaf node wishing to join the session creates a `c_leaf` socket and uses `WSAJoinLeaf` to initiate the connection and request admittance. The root application receives `FD_ACCEPT` when an incoming admittance request arrives, and admits the leaf node by calling `WSAAccept`. The leaf node receives `FD_CONNECT` when it has been admitted.

In a nonrooted control plane, where all nodes are `c_leaf`'s, the `WSAJoinLeaf` is used to initiate the inclusion of a node into an existing multipoint session. An `FD_CONNECT` indication is provided when the join has been completed and the returned socket descriptor is useable in the multipoint session. In the case of IP multicast, this would correspond to the `IP_ADD_MEMBERSHIP` socket option.

(Readers familiar with IP multicast's use of the connectionless UDP protocol may be concerned by the connection-oriented semantics presented here. In particular the notion of using `WSAJoinLeaf` on a UDP socket and waiting for an `FD_CONNECT` indication may be troubling. There is, however, ample precedent for applying connection-oriented semantics to connectionless protocols. It is allowed and sometimes useful, for example, to invoke the standard `connect` function on a UDP socket. The general result of applying connection-oriented semantics to connectionless sockets is a restriction in how such sockets may be used, and this is the case here, as well. A UDP socket used in `WSAJoinLeaf` will have certain restrictions, and waiting for an `FD_CONNECT` indication (which in this case simply indicates that the corresponding IGMP message has been sent) is one such limitation.)

There are therefore, three instances where an application would use `WSAJoinLeaf`:

- Acting as a multipoint root and inviting a new leaf to join the session
- Acting as a leaf making an admittance request to a rooted multipoint session
- Acting as a leaf seeking admittance to a nonrooted multipoint session (for example, IP multicast)

## Using `WSAJoinLeaf`

As mentioned previously, the function `WSAJoinLeaf` is used to join a leaf node into a multipoint session. `WSAJoinLeaf` has the same parameters and semantics as `WSAConnect` except that it returns a socket descriptor (as in `WSAAccept`), and it has an additional `dwFlags` parameter. The `dwFlags` parameter is used to indicate whether the socket will be acting only as a sender, only as a receiver, or both. Only multipoint sockets may be used for input parameter `s` in this function. If the multipoint socket is in nonblocking mode, the returned socket descriptor is not useable until after a

corresponding `FD_CONNECT` indication is received. A root application in a multipoint session may call **WSAJoinLeaf** one or more times in order to add a number of leaf nodes, however at most one multipoint connection request may be outstanding at a time.

The socket descriptor returned by **WSAJoinLeaf** is different depending on whether the input socket descriptor, `s`, is a `c_root` or a `c_leaf`. When used with a `c_root` socket, the `name` parameter designates a particular leaf node to be added and the returned socket descriptor is a `c_leaf` socket corresponding to the newly added leaf node. It is not intended to be used for the exchange of multipoint data, but rather is used to receive `FD_XXX` indications (for example, `FD_CLOSE`) for the connection that exists to the particular `c_leaf`. Some multipoint implementations may also allow this socket to be used for side chats between the root and an individual leaf node. An `FD_CLOSE` indication is received for this socket if the corresponding leaf node calls **closesocket** to drop out of the multipoint session. Symmetrically, invoking **closesocket** on the `c_leaf` socket returned from **WSAJoinLeaf** causes the socket in the corresponding leaf node to get `FD_CLOSE` notification.

When **WSAJoinLeaf** is invoked with a `c_leaf` socket, the `name` parameter contains the address of the root application (for a rooted control scheme) or an existing multipoint session (nonrooted control scheme), and the returned socket descriptor is the same as the input socket descriptor. In a rooted control scheme, the root application puts its `c_root` socket in listening mode by calling **listen**. The standard `FD_ACCEPT` notification is delivered when the leaf node requests to join itself to the multipoint session. The root application uses the usual **accept/WSAAccept** functions to admit the new leaf node. The value returned from either **accept** or **WSAAccept** is also a `c_leaf` socket descriptor just like those returned from **WSAJoinLeaf**. To accommodate multipoint schemes that allow both root-initiated and leaf-initiated joins, it is acceptable for a `c_root` socket that is already in listening mode to be used as in input to **WSAJoinLeaf**.

A multipoint root application is generally responsible for the orderly dismantling of a multipoint session. Such an application may use **shutdown** or **closesocket** on a `c_root` socket to cause all of the associated `c_leaf` sockets, including those returned from **WSAJoinLeaf** and their corresponding `c_leaf` sockets in the remote leaf nodes, to get `FD_CLOSE` notification.

## Semantic Differences Between Multipoint Sockets and Regular Sockets

In the control plane, there are some significant semantic differences between a `c_root` socket and a regular point-to-point socket:

- The `c_root` socket can be used in **WSAJoinLeaf** to join a new a leaf.
- Placing a `c_root` socket into listening mode (by calling **listen**) does not preclude the `c_root` socket from being used in a call to **WSAJoinLeaf** to add a new leaf, or for sending and receiving multipoint data.
- The closing of a `c_root` socket causes all of the associated `c_leaf` sockets to get `FD_CLOSE` notification.

There are no semantic differences between a `c_leaf` socket and a regular socket in the control plane, except that the `c_leaf` socket can be used in **WSAJoinLeaf**, and the use of `c_leaf` socket in **listen** indicates that only multipoint connection requests should be accepted.

In the data plane, the semantic differences between the `d_root` socket and a regular point-to-point socket are

- The data sent on the `d_root` socket is delivered to all the leaves in the same multipoint session.
- The data received on the `d_root` socket may be from any of the leaves.

The `d_leaf` socket in the rooted data plane has no semantic difference from the regular socket, however, in the nonrooted data plane, the data sent on the `d_leaf` socket goes to all the other leaf nodes, and the data received could be from any other leaf nodes. As mentioned earlier, the information about whether the `d_leaf` socket is in a rooted or nonrooted data plane is contained in the corresponding **WSAPROTOCOL\_INFO** structure for the socket.

## How Existing Multipoint Protocols Support These Extensions

In this section we indicate how IP multicast and ATM point-to-multipoint capabilities can be accessed through the Windows Sockets 2 multipoint functions. We chose these two as examples because they are popular and well understood.

### IP Multicast

IP multicast falls into the category of nonrooted data plane and nonrooted control plane. All applications play a leaf role. Currently, most IP multicast implementations use a set of socket options proposed by Steve Deering to the IETF. Five operations are thus made available:

- `IP_MULTICAST_TTL`—Sets time to live, controls scope of multicast session
- `IP_MULTICAST_IF`—Determines interface to be used for multicasting.
- `IP_ADD_MEMBERSHIP`—Joins a specified multicast session.
- `IP_DROP_MEMBERSHIP`—Drops out of a multicast session.
- `IP_MULTICAST_LOOP`—Controls loopback of multicast traffic.

Setting the time-to-live for an IP multicast socket maps directly to using the `SIO_MULTICAST_SCOPE` command code for **WSAIoctl**.

The method for determining the IP interface to be used for multicasting is through a TCP/IP-specific socket option as described in the *Windows Sockets 2 Protocol-Specific Annex*. The remaining three operations are covered well with the Windows Sockets 2 semantics described here.

The application would open sockets with `c_leaf/d_leaf` flags in **WSASocket**. It would use **WSAJoinLeaf** to add itself to a multicast group on the default interface designated for multicast operations. If the flag in **WSAJoinLeaf** indicates that this socket is only a sender, then the join operation is essentially a no-op and no IGMP messages need to be sent. Otherwise, an IGMP packet is sent out to the router to indicate interests in receiving packets sent to the specified multicast address. Since the application created special `c_leaf/d_leaf` sockets used only for performing multicast, the standard **closesocket** function would be used to drop out of the multicast session. The `SIO_MULTIPOINT_LOOPBACK` command code for **WSAIoctl** provides a generic control mechanism for determining whether data sent on a `d_leaf` socket in a nonrooted multipoint scheme can also be received on the same socket.

---

**Note** The Windows Sockets version of the `IP_MULTICAST_LOOP` option is semantically different than the UNIX version of the `IP_MULTICAST_LOOP` option. In Windows Sockets, the `IP_MULTICAST_LOOP` option applies only to the *receive* path. In contrast, the UNIX version of the `IP_MULTICAST_LOOP` option applies to the *send* path. For example, applications ON and OFF (which are easier to track than X and Y) join the same group on the same interface; application ON sets the `IP_MULTICAST_LOOP` option on, application OFF sets the `IP_MULTICAST_LOOP` option off. If ON and OFF are Windows Sockets applications, OFF can send to ON, but ON cannot send to OFF. In contrast, if ON and OFF are UNIX applications, ON can send to OFF, but OFF cannot send to ON.

---

## ATM Point to Multipoint

ATM falls into the category of rooted data and rooted control planes. An application acting as the root would create `c_root` sockets and counterparts running on leaf nodes would utilize `c_leaf` sockets. The root application uses **WSAJoinLeaf** to add new leaf nodes. The corresponding applications on the leaf nodes will have set their `c_leaf` sockets into listen mode. **WSAJoinLeaf** with a `c_root` socket specified is mapped to the Q.2931 ADDPARTY message. The leaf-initiated join is not supported in ATM UNI 3.1, but is supported in ATM UNI 4.0. Thus **WSAJoinLeaf** with a `c_leaf` socket specified is mapped to the appropriate ATM UNI 4.0 message.

There are additional considerations to bear in mind for ATM point-to-multipoint:

- The addition of new leaves to the ATM point-to-multipoint session is invitation-based only. The root invites leaves—which have already their **accept** function call—by calling the **WSAJoinLeaf** function.
- The flow of data in an ATM point-to-multipoint session is from root-to-leaves only; leaves cannot use the same session to send information to the root.
- Only one leaf per ATM adapter is allowed.

## Additional Windows Socket Information

This section contains information on the Windows Sockets 2 header file, additional Windows Sockets reference material, and the error codes encountered in programming for Windows Sockets 2.

### Windows Sockets 2 API Header File—Winsock2.h

New versions of Winsock2.h will appear periodically as new identifiers are allocated by the Windows Sockets Identifier Clearinghouse. The clearinghouse can be reached through the world wide web at:

<http://www.stardust.com/winsock/>

### Socket Options Specific to Microsoft Service Providers

Microsoft's service providers support additional socket options not included in the Windows Sockets 2 API:

Socket Option for Windows NT 4.0 Only

Socket Option for Windows NT 4.0 and Windows 95

#### Socket Option for Windows NT 4.0 Only

The following socket options are Microsoft-specific extensions for connect and disconnect data and options, and are used only by non-TCP/IP transports such as DECNet, OSI TP4, etc. These are only used in the Microsoft implementation of Windows Sockets on Windows NT 4.0.

- SO\_CONNDATA
- SO\_CONNOPT
- SO\_DISCDATA
- SO\_DISCOPT
- SO\_CONNDATALEN
- SO\_CONNOPTLEN
- SO\_DISCDATALEN
- SO\_DISCOPTLEN

The following socket options are Microsoft-specific extensions for controlling the size of datagrams:

- SO\_MAXDG
- SO\_MAXPATHDG



## Socket Option for Windows NT 4.0 and Windows 95

SO\_SNDTIMEO  
SO\_RCVTIMEO

### Details on SO\_SNDTIMEO and SO\_RCVTIMEO

These two options set up time-outs for the **send**, **sendto**, **recv**, and **recvfrom** functions. You can obtain the same functionality by calling **select** with a time-out just before the I/O call, but these options offer a significant improvement in performance by avoiding a kernel transition and the other overhead of the **select** call. For any code whose performance is very critical, applications should use these time-out options rather than **select**.

You can set these options on any type of socket in any state. The default value for these options is zero, which refers to an infinite time-out. Any other setting is the time-out, in milliseconds. It is valid to set the time-out to any value, but values less than 500 milliseconds (half a second) are interpreted to be 500 milliseconds.

To set a send time-out, use:

```
int timeout = TIMEOUT_VALUE;
int err;
SOCKET s;

s = socket( ... );
err = setsockopt(
    s,
    SOL_SOCKET,
    SO_SNDTIMEO,
    (char *)&timeout,
    sizeof(timeout));
if (err != NO_ERROR) {
    /* failed for some reason... */
}
```

The `TIMEOUT_VALUE` is the needed time-out in milliseconds. To set a receive time-out, substitute `SO_RCVTIMEO` for `SO_SNDTIMEO` in the preceding example.

After setting one of these options to a nonzero value, I/O through the Windows Sockets calls fails with the error `WSAETIMEDOUT` if the request cannot be satisfied within the specified time-out. If a request times out, an application has no guarantees as to how much data was actually sent or received in the I/O call.

The following socket option is used in conjunction with the MS Extension function **AcceptEx**.

SO\_UPDATE\_ACCEPT\_CONTEXT

## Additional Documentation

This specification is intended to cover the Windows Sockets interface in detail. Many details of specific protocols and Windows, however, are intentionally omitted in the interest of brevity, and this specification often assumes background knowledge of these topics. For more information, the following references may be helpful:

### Networking Books

Jones, A. and Ohlund, J., *Network Programming for Microsoft Windows*, (Microsoft Press, 1999).

Braden, R.[1989], *RFC 1122, Requirements for Internet Hosts--Communication Layers*, Internet Engineering Task Force.

Comer, D., *Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture*, (Prentice Hall, 1991).

Comer, D. and Stevens, D., *Internetworking with TCP/IP Volume II: Design, Implementation, and Internals* (Prentice Hall, 1991).

Comer, D. and Stevens, D., *Internetworking with TCP/IP Volume III: Client-Server Programming and Applications* (Prentice Hall, 1991).

Leffler, S. et al., *An Advanced 4.3BSD Interprocess Communication Tutorial*.

Stevens, W.R., *Unix Network Programming* (Prentice Hall, 1990).

Stevens, W.R.. *TCP/IP Illustrated, Volume 1: The Protocols* (Addison-Wesley, 1994).

Wright, G.R. and Stevens, W.R., *TCP/IP Illustrated Volume 2: The Implementation* (Addison-Wesley, 1995).

### Windows Sockets Books

Jones, A. and Ohlund, J., *Network Programming for Microsoft Windows* (Microsoft Press, 1999).

Bonner, P., *Network Programming with Windows Sockets* (Prentice Hall, 1995).

Dumas, A., *Programming Windows Sockets* (Sams Publishing, 1995).

Quinn, B. and Shute, D., *Windows Sockets Network Programming* (Addison-Wesley, 1995).

Roberts, D., *Developing for the Internet with Winsock* (Coriolis Group Books, 1995).



---

## CHAPTER 7

# Error Codes in the Winsock API

The following is a list of possible error codes returned by the **WSAGetLastError** call, along with their extended explanations. Errors are listed in alphabetical order by error macro. Some error codes defined in Winsock2.h are not returned from any function—these are not included in this topic.

## Error Codes

### **WSAEACCES** **(10013)**

*Permission denied.*

An attempt was made to access a socket in a way forbidden by its access permissions. An example is using a broadcast address for **sendto** without broadcast permission being set using **setsockopt(SO\_BROADCAST)**.

Another possible reason for the **WSAEACCES** error is that when the **bind** function is called (on Windows NT 4 SP4 or later), another application, service, or kernel mode driver is bound to the same address with exclusive access. Such exclusive access is a new feature of Windows NT 4 SP4 and later, and is implemented by using the **SO\_EXCLUSIVEADDRUSE** option.

### **WSAEADDRINUSE** **(10048)**

*Address already in use.*

Typically, only one usage of each socket address (protocol/IP address/port) is permitted. This error occurs if an application attempts to **bind** a socket to an IP address/port that has already been used for an existing socket, or a socket that wasn't closed properly, or one that is still in the process of closing. For server applications that need to **bind** multiple sockets to the same port number, consider using **setsockopt(SO\_REUSEADDR)**. Client applications usually need not call **bind** at all—connect chooses an unused port automatically. When **bind** is called with a wildcard address (involving **ADDR\_ANY**), a **WSAEADDRINUSE** error could be delayed until the specific address is committed. This could happen with a call to another function later, including **connect**, **listen**, **WSAConnect**, or **WSAJoinLeaf**.

**WSAEADDRNOTAVAIL  
(10049)**

*Cannot assign requested address.*

The requested address is not valid in its context. This normally results from an attempt to **bind** to an address that is not valid for the local machine. This can also result from **connect**, **sendto**, **WSAConnect**, **WSAJoinLeaf**, or **WSASendTo** when the remote address or port is not valid for a remote machine (for example, address or port 0).

**WSAEAFNOSUPPORT  
(10047)**

*Address family not supported by protocol family.*

An address incompatible with the requested protocol was used. All sockets are created with an associated address family (that is, AF\_INET for Internet Protocols) and a generic protocol type (that is, SOCK\_STREAM). This error is returned if an incorrect protocol is explicitly requested in the **socket** call, or if an address of the wrong family is used for a socket, for example, in **sendto**.

**WSAEALREADY  
(10037)**

*Operation already in progress.*

An operation was attempted on a nonblocking socket with an operation already in progress—that is, calling **connect** a second time on a nonblocking socket that is already connecting, or canceling an asynchronous request (**WSAAsyncGetXbyY**) that has already been canceled or completed.

**WSAECONNABORTED  
(10053)**

*Software caused connection abort.*

An established connection was aborted by the software in your host machine, possibly due to a data transmission time-out or protocol error.

**WSAECONNREFUSED  
(10061)**

*Connection refused.*

No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host—that is, one with no server application running.

**WSAECONNRESET**  
**(10054)**

*Connection reset by peer.*

An existing connection was forcibly closed by the remote host. This normally results if the peer application on the remote host is suddenly stopped, the host is rebooted, or the remote host uses a hard close (see **setsockopt** for more information on the **SO\_LINGER** option on the remote socket.) This error may also result if a connection was broken due to keep-alive activity detecting a failure while one or more operations are in progress. Operations that were in progress fail with **WSAENETRESET**. Subsequent operations fail with **WSAECONNRESET**.

**WSAEDESTADDRREQ**  
**(10039)**

*Destination address required.*

A required address was omitted from an operation on a socket. For example, this error is returned if **sendto** is called with the remote address of **ADDR\_ANY**.

**WSAEFAULT**  
**(10014)**

*Bad address.*

The system detected an invalid pointer address in attempting to use a pointer argument of a call. This error occurs if an application passes an invalid pointer value, or if the length of the buffer is too small. For instance, if the length of an argument, which is a **SOCKADDR** structure, is smaller than the size of (**SOCKADDR**).

**WSAEHOSTDOWN**  
**(10064)**

*Host is down.*

A socket operation failed because the destination host is down. A socket operation encountered a dead host. Networking activity on the local host has not been initiated. These conditions are more likely to be indicated by the error **WSAETIMEDOUT**.

**WSAEHOSTUNREACH**  
**(10065)**

*No route to host.*

A socket operation was attempted to an unreachable host. See **WSAENETUNREACH**.

**WSAEINPROGRESS**  
**(10036)**

*Operation now in progress.*

A blocking operation is currently executing. Windows Sockets only allows a single blocking operation—per task or thread—to be outstanding, and if any other function call is made (whether or not it references that or any other socket) the function fails with the **WSAEINPROGRESS** error.

**WSAEINTR**  
**(10004)**

*Interrupted function call.*

A blocking operation was interrupted by a call to **WSACancelBlockingCall**.

**WSAEINVAL**  
**(10022)**

*Invalid argument.*

Some invalid argument was supplied (for example, specifying an invalid level to the **setsockopt** function). In some instances, it also refers to the current state of the socket—for instance, calling **accept** on a socket that is not listening.

**WSAEISCONN**  
**(10056)**

*Socket is already connected.*

A connect request was made on an already-connected socket. Some implementations also return this error if **sendto** is called on a connected SOCK\_DGRAM socket (for SOCK\_STREAM sockets, the *to* parameter in **sendto** is ignored) although other implementations treat this as a legal occurrence.

**WSAEMFILE**  
**(10024)**

*Too many open files.*

Too many open sockets. Each implementation may have a maximum number of socket handles available, either globally, per process, or per thread.

**WSAEMSGSIZE**  
**(10040)**

*Message too long.*

A message sent on a datagram socket was larger than the internal message buffer or some other network limit, or the buffer used to receive a datagram was smaller than the datagram itself.

**WSAENETDOWN**  
**(10050)**

*Network is down.*

A socket operation encountered a dead network. This could indicate a serious failure of the network system (that is, the protocol stack that the Windows Sockets DLL runs over), the network interface, or the local network itself.

**WSAENETRESET**  
**(10052)**

*Network dropped connection on reset.*

The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress. It can also be returned by **setsockopt** if an attempt is made to set SO\_KEEPALIVE on a connection that has already failed.

**WSAENETUNREACH**  
(10051)

*Network is unreachable.*

A socket operation was attempted to an unreachable network. This usually means the local software knows no route to reach the remote host.

**WSAENOBUFS**  
(10055)

*No buffer space available.*

An operation on a socket could not be performed because the system lacked sufficient buffer space or because a queue was full.

**WSAENOPROTOPT**  
(10042)

*Bad protocol option.*

An unknown, invalid or unsupported option or level was specified in a **getsockopt** or **setsockopt** call.

**WSAENOTCONN**  
(10057)

*Socket is not connected.*

A request to send or receive data was disallowed because the socket is not connected and (when sending on a datagram socket using **sendto**) no address was supplied. Any other type of operation might also return this error—for example, **setsockopt** setting SO\_KEEPALIVE if the connection has been reset.

**WSAENOTSOCK**  
(10038)

*Socket operation on nonsocket.*

An operation was attempted on something that is not a socket. Either the socket handle parameter did not reference a valid socket, or for **select**, a member of an **fd\_set** was not valid.

**WSAEOPNOTSUPP**  
(10045)

*Operation not supported.*

The attempted operation is not supported for the type of object referenced. Usually this occurs when a socket descriptor to a socket that cannot support this operation is trying to accept a connection on a datagram socket.

**WSAEPFNOSUPPORT**  
(10046)

*Protocol family not supported.*

The protocol family has not been configured into the system or no implementation for it exists. This message has a slightly different meaning from **WSAEAFNOSUPPORT**. However, it is interchangeable in most cases, and all Windows Sockets functions that return one of these messages also specify **WSAEAFNOSUPPORT**.



**WSAEPROCLIM  
(10067)**

*Too many processes.*

A Windows Sockets implementation may have a limit on the number of applications that can use it simultaneously. **WSAStartup** may fail with this error if the limit has been reached.

**WSAEPROTONOSUPPORT  
(10043)**

*Protocol not supported.*

The requested protocol has not been configured into the system, or no implementation for it exists. For example, a **socket** call requests a SOCK\_DGRAM socket, but specifies a stream protocol.

**WSAEPROTOTYPE  
(10041)**

*Protocol wrong type for socket.*

A protocol was specified in the **socket** function call that does not support the semantics of the socket type requested. For example, the ARPA Internet UDP protocol cannot be specified with a socket type of SOCK\_STREAM.

**WSAESHUTDOWN  
(10058)**

*Cannot send after socket shutdown.*

A request to send or receive data was disallowed because the socket had already been shut down in that direction with a previous **shutdown** call. By calling **shutdown** a partial close of a socket is requested, which is a signal that sending or receiving, or both have been discontinued.

**WSAESOCKTNOSUPPORT  
(10044)**

*Socket type not supported.*

The support for the specified socket type does not exist in this address family. For example, the optional type SOCK\_RAW might be selected in a **socket** call, and the implementation does not support SOCK\_RAW sockets at all.

**WSAETIMEDOUT  
(10060)**

*Connection timed out.*

A connection attempt failed because the connected party did not properly respond after a period of time, or the established connection failed because the connected host has failed to respond.

**WSATYPE\_NOT\_FOUND  
(10109)**

*Class type not found.*

The specified class was not found.

**WSAEWOULDBLOCK**  
(10035)

*Resource temporarily unavailable.*

This error is returned from operations on nonblocking sockets that cannot be completed immediately, for example **recv** when no data is queued to be read from the socket. It is a nonfatal error, and the operation should be retried later. It is normal for WSAEWOULDBLOCK to be reported as the result from calling **connect** on a nonblocking SOCK\_STREAM socket, since some time must elapse for the connection to be established.

**WSAHOST\_NOT\_FOUND**  
(11001)

*Host not found.*

No such host is known. The name is not an official host name or alias, or it cannot be found in the database(s) being queried. This error may also be returned for protocol and service queries, and means that the specified name could not be found in the relevant database.

**WSA\_INVALID\_HANDLE**  
(OS dependent)

*Specified event object handle is invalid.*

An application attempts to use an event object, but the specified handle is not valid.

**WSA\_INVALID\_PARAMETER**  
(OS dependent)

*One or more parameters are invalid.*

An application used a Windows Sockets function which directly maps to a Win32 function. The Win32 function is indicating a problem with one or more parameters.

**WSA\_INVALIDPROC**  
(OS dependent)

*Invalid procedure table from service provider.*

A service provider returned a bogus procedure table to Ws2\_32.dll. (Usually caused by one or more of the function pointers being null.)

**WSA\_INVALIDPROVIDER**  
(OS dependent)

*Invalid service provider version number.*

A service provider returned a version number other than 2.0.

**WSA\_IO\_INCOMPLETE**  
(OS dependent)

*Overlapped I/O event object not in signaled state.*

The application has tried to determine the status of an overlapped operation which is not yet completed. Applications that use **WSAGetOverlappedResult** (with the *fWait* flag set to FALSE) in a polling mode to determine when an overlapped operation has completed, get this error code until the operation is complete.

**WSA\_IO\_PENDING**  
(OS dependent)

*Overlapped operations will complete later.*

The application has initiated an overlapped operation that cannot be completed immediately. A completion indication will be given later when the operation has been completed.

**WSA\_NOT\_ENOUGH\_MEMORY**  
(OS dependent)

*Insufficient memory available.*

An application used a Windows Sockets function that directly maps to a Win32 function. The Win32 function is indicating a lack of required memory resources.

**WSANOTINITIALISED**  
(10093)

*Successful WSAStartup not yet performed.*

Either the application hasn't called **WSAStartup** or **WSAStartup** failed. The application may be accessing a socket that the current active task does not own (that is, trying to share a socket between tasks), or **WSACleanup** has been called too many times.

**WSANO\_DATA**  
(11004)

*Valid name, no data record of requested type.*

The requested name is valid and was found in the database, but it does not have the correct associated data being resolved for. The usual example for this is a host name-to-address translation attempt (using **gethostbyname** or **WSAAsyncGetHostByName**) which uses the DNS (Domain Name Server). An MX record is returned but no A record—indicating the host itself exists, but is not directly reachable.

**WSANO\_RECOVERY**  
(11003)

*This is a nonrecoverable error.*

This indicates some sort of nonrecoverable error occurred during a database lookup. This may be because the database files (for example, BSD-compatible HOSTS, SERVICES, or PROTOCOLS files) could not be found, or a DNS request was returned by the server with a severe error.

**WSAPROVIDERFAILEDINIT**  
(OS dependent)

*Unable to initialize a service provider.*

Either a service provider's DLL could not be loaded (**LoadLibrary** failed) or the provider's **WSPStartup/NSPStartup** function failed.

**WSASYSCALLFAILURE****(OS dependent)***System call failure.*

Returned when a system call that should never fail does. For example, if a call to **WaitForMultipleObjects** fails or one of the registry functions fails trying to manipulate the protocol/name space catalogs.

**WSASYSNOTREADY****(10091)***Network subsystem is unavailable.*

This error is returned by **WSAStartup** if the Windows Sockets implementation cannot function at this time because the underlying system it uses to provide network services is currently unavailable. Users should check:

- That the appropriate Windows Sockets DLL file is in the current path.
- That they are not trying to use more than one Windows Sockets implementation simultaneously. If there is more than one Winsock DLL on your system, be sure the first one in the path is appropriate for the network subsystem currently loaded.
- The Windows Sockets implementation documentation to be sure all necessary components are currently installed and configured correctly.

**WSATRY\_AGAIN****(11002)***Nonauthoritative host not found.*

This is usually a temporary error during host name resolution and means that the local server did not receive a response from an authoritative server. A retry at some time later may be successful.

**WSAVERNOTSUPPORTED****(10092)***Winsock.dll version out of range.*

The current Windows Sockets implementation does not support the Windows Sockets specification version requested by the application. Check that no old Windows Sockets DLL files are being accessed.

**WSAEDISCON****(10101)***Graceful shutdown in progress.*

Returned by **WSARecv** and **WSARecvFrom** to indicate that the remote party has initiated a graceful shutdown sequence.

**WSA\_OPERATION\_ABORTED****(OS dependent)***Overlapped operation aborted.*

An overlapped operation was canceled due to the closure of the socket, or the execution of the **SIO\_FLUSH** command in **WSAIoctl**.



## CHAPTER 8

# Winsock 2 Functions

## Windows Sockets 2 Functions

### accept

The Windows Sockets **accept** function permits an incoming connection attempt on a socket.

```
SOCKET accept (  
    SOCKET s,  
    struct sockaddr FAR *addr,  
    int FAR *addrlen  
);
```

#### Parameters

*s*

[in] Descriptor identifying a socket that has been placed in a listening state with the **listen** function. The connection is actually made with the socket that is returned by **accept**.

*addr*

[out] Optional pointer to a buffer that receives the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the address family established when the socket was created.

*addrlen*

[out] Optional pointer to an integer that contains the length of *addr*.

#### Return Values

If no error occurs, **accept** returns a value of type **SOCKET** that is a descriptor for the new socket. This returned value is a handle for the socket on which the actual connection is made.

Otherwise, a value of **INVALID\_SOCKET** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

The integer referred to by *addrlen* initially contains the amount of space pointed to by *addr*. On return it will contain the actual length in bytes of the address returned.

## Remarks

The **accept** function extracts the first connection on the queue of pending connections on socket *s*. It then creates a new socket and returns a handle to the new socket. The newly created socket is the socket that will handle the actual connection and has the same properties as socket *s*, including the asynchronous events registered with the **WSAAsyncSelect** or **WSAEventSelect** functions.

The **accept** function can block the caller until a connection is present if no pending connections are present on the queue, and the socket is marked as blocking. If the socket is marked as nonblocking and no pending connections are present on the queue, **accept** returns an error as described in the following. After the successful completion of **accept** returns a new socket handle, the accepted socket cannot be used to accept more connections. The original socket remains open and listens for new connection requests.

The parameter *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the address family in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned.

The **accept** function is used with connection-oriented socket types such as **SOCK\_STREAM**.

If *addr* and/or *addrlen* are equal to NULL, then no information about the remote address of the accepted socket is returned.

## Error Codes

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>addrlen</i> parameter is too small or <i>addr</i> is not a valid part of the user address space.
WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The <b>listen</b> function was not invoked prior to <b>accept</b> .
WSAEMFILE	The queue is nonempty upon entry to <b>accept</b> and there are no descriptors available.
WSAENOBUFS	No buffer space is available.

---

Error code	Meaning
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The referenced socket is not a type that supports connection-oriented service.
WSAEWOULDBLOCK	The socket is marked as nonblocking and no connections are present to be accepted.

### Notes for ATM

The following are important issues associated with connection setup, and must be considered when using Asynchronous Transfer Mode (ATM) with Windows Sockets 2:

- The **accept** and **WSAAccept** functions do not necessarily set the remote address and address length parameters. Therefore, when using ATM the caller should use the **WSAAccept** function and place **ATM\_CALLING\_PARTY\_NUMBER\_IE** in the **ProviderSpecific** member of the **QOS** structure, which itself is included in the *lpSQOS* parameter of the callback function used in accordance with **WSAAccept**.
- When using the **accept** function, realize that the function may return before connection establishment has traversed the entire distance between sender and receiver. This is because the accept function returns as soon as it receives a **CONNECT ACK** message; in ATM a **CONNECT ACK** message is returned by the next switch in the path as soon as a **CONNECT** message is processed (rather than the **CONNECT ACK** being sent by the end node to which the connection is ultimately established). As such, applications should realize that if data is sent immediately following receipt of a **CONNECT ACK** message, data loss is possible, since the connection may not have been established “all the way” between sender and receiver.

#### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in *Winsock2.h*.

**Library:** Use *Ws2\_32.lib*.

#### + See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **bind**, **connect**, **listen**, **select**, **socket**, **WSAAsyncSelect**, **WSAAccept**

---

## AcceptEx

The Windows Sockets **AcceptEx** function accepts a new connection, returns the local and remote address, and receives the first block of data sent by the client application. The **AcceptEx** function is not supported on Windows 95/98.



---

**Note** This function is a Microsoft-specific extension to the Windows Sockets specification. For more information, see *Microsoft Extensions and Windows Sockets 2.2*.

---

```

BOOL AcceptEx (
    SOCKET          sListenSocket,
    SOCKET          sAcceptSocket,
    PVOID           lpOutputBuffer,
    DWORD           dwReceiveDataLength,
    DWORD           dwLocalAddressLength,
    DWORD           dwRemoteAddressLength,
    LPDWORD         lpdwBytesReceived,
    LPOVERLAPPED   lpOverlapped
);

```

## Parameters

### *sListenSocket*

[in] Descriptor identifying a socket that has already been called with the **listen** function. A server application waits for attempts to connect on this socket.

### *sAcceptSocket*

[in] Descriptor identifying a socket on which to accept an incoming connection. This socket must not be bound or connected.

### *lpOutputBuffer*

[in] Pointer to a buffer that receives the first block of data sent on a new connection, the local address of the server, and the remote address of the client. The receive data is written to the first part of the buffer starting at offset zero, while the addresses are written to the latter part of the buffer. If this parameter is set to NULL, no receive will be performed, nor will local or remote addresses be available through the use of **GetAcceptExSockaddrs** function calls.

### *dwReceiveDataLength*

[in] Number of bytes in *lpOutputBuffer* that will be used for actual receive data at the beginning of the buffer. This size should not include the size of the local address of the server, nor the remote address of the client; they are appended to the output buffer. If *dwReceiveDataLength* is zero, accepting the connection will not result in a receive operation. Instead, **AcceptEx** completes as soon as a connection arrives, without waiting for any data.

### *dwLocalAddressLength*

[in] Number of bytes reserved for the local address information. This value must be at least 16 bytes more than the maximum address length for the transport protocol in use.

### *dwRemoteAddressLength*

[in] Number of bytes reserved for the remote address information. This value must be at least 16 bytes more than the maximum address length for the transport protocol in use.

*lpdwBytesReceived*

[out] Pointer to a **DWORD** that receives the count of bytes received. This parameter is set only if the operation completes synchronously. If it returns **ERROR\_IO\_PENDING** and is completed later, then this **DWORD** is never set and you must obtain the number of bytes read from the completion notification mechanism.

*lpOverlapped*

[in] An **OVERLAPPED** structure that is used to process the request. This parameter *must* be specified; it cannot be null.

**Return Values**

If no error occurs, the **AcceptEx** function completed successfully and a value of **TRUE** is returned.

If the function fails, **AcceptEx** returns **FALSE**. The **WSAGetLastError** function can then be called to return extended error information. If **WSAGetLastError** returns **ERROR\_IO\_PENDING**, then the operation was successfully initiated and is still in progress.

**Remarks**

The **AcceptEx** function combines several socket functions into a single API/kernel transition. The **AcceptEx** function, when successful, performs three tasks:

- A new connection is accepted.
- Both the local and remote addresses for the connection are returned.
- The first block of data sent by the remote is received.

A program can make a connection to a socket more quickly using **AcceptEx** instead of the **accept** function.

A single output buffer receives the data: the local socket address (the server), and the remote socket address (the client).

Using a single buffer improves performance, but the **GetAcceptExSockaddrs** function must be called to parse the buffer into its three distinct parts.

The buffer size for the local and remote address must be 16 bytes more than the size of the **SOCKADDR** structure for the transport protocol in use because the addresses are written in an internal format. For example, the size of a **SOCKADDR\_IN** (the address structure for TCP/IP) is 16 bytes. Therefore, a buffer size of at least 32 bytes must be specified for the local and remote addresses.

The **AcceptEx** function uses overlapped I/O, unlike the Windows Sockets 1.1 **accept** function. If your application uses **AcceptEx**, it can service a large number of clients with a relatively small number of threads. As with all overlapped Win32 functions, either Win32 events or completion ports can be used as a completion notification mechanism.

Another key difference between the **AcceptEx** function and the Windows Sockets 1.1 **accept** function is that the **AcceptEx** function requires the caller to already have two sockets:

- One that specifies the socket on which to listen.
- One that specifies the socket on which to accept the connection.

The *sAcceptSocket* parameter must be an open socket that is neither bound nor connected.

The *IpNumberOfBytesTransferred* parameter of the **GetQueuedCompletionStatus** function or the **GetOverlappedResult** function indicates the number of bytes received in the request.

When this operation is successfully completed, *sAcceptHandle* can be passed, but to the following functions only:

**ReadFile**  
**WriteFile**  
**send**  
**recv**  
**TransmitFile**  
**closesocket**

---

**Note** If you have called the **TransmitFile** function with both the **TF\_DISCONNECT** and **TF\_REUSE\_SOCKET** flags, the specified socket has been returned to a state in which it is neither bound nor connected. You can then pass the handle of the socket to the **AcceptEx** function in the *sAcceptSocket* parameter.

---

When the **AcceptEx** function returns, the socket *sAcceptSocket* is in the default state for a connected socket. The socket *sAcceptSocket* does not inherit the properties of the socket associated with *sListenSocket* parameter until **SO\_UPDATE\_ACCEPT\_CONTEXT** is set on the socket. Use the **setsockopt** function to set the **SO\_UPDATE\_ACCEPT\_CONTEXT** option, specifying *sAcceptSocket* as the socket handle and *sListenSocket* as the option value.

For example:

```
err = setsockopt( sAcceptSocket,  
                SOL_SOCKET,  
                SO_UPDATE_ACCEPT_CONTEXT,  
                (char *)&sListenSocket,  
                sizeof(sListenSocket) );
```

Use the **getsockopt** function with the **SO\_CONNECT\_TIME** option to check whether a connection has been accepted. If it has been accepted, you can determine how long the connection has been established. The return value is the number of seconds that the socket has been connected. If the socket is not connected, the **getsockopt** returns

0xFFFFFFFF. Checking a connection like this is necessary to see if connections that have been established for a while, without receiving any data. It is recommended that you terminate those connections.

For example:

```
INT seconds;
INT bytes = sizeof(seconds);
err = getsockopt( sAcceptSocket, SOL_SOCKET, SO_CONNECT_TIME,
                (char *)&seconds, (PINT)&bytes );
if ( err != NO_ERROR ) {
    printf( "getsockopt(SO_CONNECT_TIME) failed: %ld\n", WSAGetLastError( ) );
    exit(1);
}
```

### Notes for ATM

There are important issues associated with connection setup when using Asynchronous Transfer Mode (ATM) with Windows Sockets 2. Please see the Remarks section in the Windows Sockets 2 **accept** function documentation for important ATM connection setup information.

#### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later. A Microsoft-specific extension.

**Header:** Declared in Mswsock.h.

**Library:** Use Mswsock.lib.

---

## bind

The Windows Sockets **bind** function associates a local address with a socket.

```
int bind (
    SOCKET                s,
    const struct sockaddr FAR *name,
    int                   namelen
);
```

### Parameters

*s*

[in] Descriptor identifying an unbound socket.

*name*

[in] Address to assign to the socket from the **SOCKADDR** structure.

*namelen*

[in] Length of the value in the *name* parameter.

## Return Values

If no error occurs, **bind** returns zero. Otherwise, it returns `SOCKET_ERROR`, and a specific error code can be retrieved by calling **WSAGetLastError**.

## Remarks

The **bind** function is used on an unconnected socket before subsequent calls to the **connect** or **listen** functions. It is used to bind to either connection-oriented (stream) or connectionless (datagram) sockets. When a socket is created with a call to the **socket** function, it exists in a name space (address family), but it has no name assigned to it. Use the **bind** function to establish the local association of the socket by assigning a local name to an unnamed socket.

A name consists of three parts when using the Internet address family:

- The address family.
- A host address.
- A port number that identifies the application.

In Windows Sockets 2, the *name* parameter is not strictly interpreted as a pointer to a **SOCKADDR** structure. It is cast this way for Windows Sockets 1.1 compatibility. Service providers are free to regard it as a pointer to a block of memory of size *namelen*. The first 2 bytes in this block (corresponding to the **sa\_family** member of the **SOCKADDR** structure) *must* contain the address family that was used to create the socket. Otherwise, an error `WSAEFAULT` occurs.

If an application does not care what local address is assigned, specify the manifest constant value `ADDR_ANY` for the **sa\_data** member of the *name* parameter. This allows the underlying service provider to use any appropriate network address, potentially simplifying application programming in the presence of multihomed hosts (that is, hosts that have more than one network interface and address).

For TCP/IP, if the port is specified as zero, the service provider assigns a unique port to the application with a value between 1024 and 5000. The application can use **getsockname** after calling **bind** to learn the address and the port that has been assigned to it. If the Internet address is equal to `INADDR_ANY`, **getsockname** cannot necessarily supply the address until the socket is connected, since several addresses can be valid if the host is multihomed. Binding to a specific port number other than port 0 is discouraged for client applications, since there is a danger of conflicting with another socket already using that port number.

## Notes for IrDA Sockets

- The `Af_irda.h` header file must be explicitly included.
- Local names are not exposed in IrDA. IrDA client sockets therefore, must never call the **bind** function before the **connect** function. If the IrDA socket was previously bound to a service name using **bind**, the **connect** function will fail with `SOCKET_ERROR`.

- If the service name is of the form “LSAP-SELxxx”, where xxx is a decimal integer in the range 1–127, the address indicates a specific LSAP-SEL xxx rather than a service name. Service names such as these allow server applications to accept incoming connections directed to a specific LSAP-SEL, without first performing an ISA service name query to get the associated LSAP-SEL. One example of this service name type is a non-Windows device that does not support IAS.

## Error Codes

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	
WSAEADDRINUSE	A process on the machine is already bound to the same fully-qualified address and the socket has not been marked to allow address reuse with <b>SO_REUSEADDR</b> . For example, the IP address and port are bound in the <code>af_inet</code> case). (See the <b>SO_REUSEADDR</b> socket option under <b>setsockopt</b> .)
WSAEADDRNOTAVAIL	The specified address is not a valid address for this machine.
WSAEFAULT	The <i>name</i> or <i>namelen</i> parameter is not a valid part of the user address space, the <i>namelen</i> parameter is too small, the <i>name</i> parameter contains an incorrect address format for the associated address family, or the first two bytes of the memory block specified by <i>name</i> does not match the address family associated with the socket descriptor <i>s</i> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The socket is already bound to an address.
WSAENOBUFS	Not enough buffers available, too many connections.
WSAENOTSOCK	The descriptor is not a socket.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

### + See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **connect**, **getsockname**, **listen**, **setsockopt**, **socket**, **WSACancelBlockingCall**

# closesocket

The Windows Sockets **closesocket** function closes an existing socket.

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

## Parameters

*s*

[in] Descriptor identifying the socket to close.

## Return Values

If no error occurs, **closesocket** returns zero. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

## Remarks

The **closesocket** function closes a socket. Use it to release the socket descriptor *s* so that further references to *s* fail with the error **WSAENOTSOCK**. If this is the last reference to an underlying socket, the associated naming information and queued data are discarded. Any pending blocking, asynchronous calls issued by any thread in this process are canceled without posting any notification messages.

Any pending overlapped send and receive operations (**WSASend/WSASendTo/WSARecv/WSARecvFrom** with an overlapped socket) issued by any thread in this process are also canceled. Any event, completion routine, or completion port action specified for these overlapped operations is performed. The pending overlapped operations fail with the error status **WSA\_OPERATION\_ABORTED**.

An application should always have a matching call to **closesocket** for each successful call to **socket** to return any socket resources to the system.

The semantics of **closesocket** are affected by the socket options **SO\_LINGER** and **SO\_DONTLINGER** as follows (**SO\_DONTLINGER** is enabled by default; **SO\_LINGER** is disabled).

Option	Interval	Type of close	Wait for close?
<b>SO_DONTLINGER</b>	Do not care	Graceful	No
<b>SO_LINGER</b>	Zero	Hard	No
<b>SO_LINGER</b>	Nonzero	Graceful	Yes

If **SO\_LINGER** is set with a zero time-out interval (that is, the **LINGER** structure members **l\_onoff** is not zero and **l\_linger** is zero), **closesocket** is not blocked even if queued data has not yet been sent or acknowledged. This is called a hard or abortive close, because the socket's virtual circuit is reset immediately, and any unsent data is lost. Any **recv** call on the remote side of the circuit will fail with **WSAECONNRESET**.

If **SO\_LINGER** is set with a nonzero time-out interval on a blocking socket, the **closesocket** call blocks on a blocking socket until the remaining data has been sent or until the time-out expires. This is called a graceful disconnect. If the time-out expires before all data has been sent, the Windows Sockets implementation terminates the connection before **closesocket** returns.

Enabling **SO\_LINGER** with a nonzero time-out interval on a nonblocking socket is not recommended. In this case, the call to **closesocket** will fail with an error of **WSAEWOULDBLOCK** if the close operation cannot be completed immediately. If **closesocket** fails with **WSAEWOULDBLOCK** the socket handle is still valid, and a disconnect is not initiated. The application must call **closesocket** again to close the socket. If **SO\_DONTLINGER** is set on a stream socket by setting the **I\_onoff** member of the **LINGER** structure to zero, the **closesocket** call will return immediately and does not receive **WSAEWOULDBLOCK** whether the socket is blocking or nonblocking. However, any data queued for transmission will be sent, if possible, before the underlying socket is closed. This is also called a graceful disconnect. In this case, the Windows Sockets provider cannot release the socket and other resources for an arbitrary period, thus affecting applications that expect to use all available sockets. This is the default behavior (**SO\_DONTLINGER** is set by default).

---

**Note** To ensure that all data is sent and received on a connection, an application should call **shutdown** before calling **closesocket** (see *Graceful shutdown, linger options, and socket closure* for more information). Also note, an **FD\_CLOSE** network event is *not* posted after **closesocket** is called.

---

Here is a summary of **closesocket** behavior:

- If **SO\_DONTLINGER** is enabled (the default setting) it always returns immediately—connection is gracefully closed in the background.
- If **SO\_LINGER** is enabled with a zero time-out: it always returns immediately—connection is reset/terminated.
- If **SO\_LINGER** is enabled with a nonzero time-out:
  - with a blocking socket, it blocks until all data sent or time-out expires.
  - with a nonblocking socket, it returns immediately indicating failure.

For additional information please see *Graceful shutdown, linger options, and socket closure* for more information.

### Notes for IrDA Sockets

- The `Af_irda.h` header file must be explicitly included.
- For Windows CE, **WSAEINTR** is not supported.
- The standard linger options are supported.



- Although IrDA does not provide a graceful close, IrDA will defer closing until receive queues are purged. Thus, an application can send data and immediately call the **socket** function, and be confident that the receiver will copy the data before receiving an FD\_CLOSE message.

### Notes for ATM

The following are important issues associated with connection teardown when using Asynchronous Transfer Mode (ATM) and Windows Sockets 2:

- Using the **closesocket** or **shutdown** functions with SD\_SEND or SD\_BOTH results in a RELEASE signal being sent out on the control channel. Due to ATM's use of separate signal and data channels, it is possible that a RELEASE signal could reach the remote end before the last of the data reaches its destination, resulting in a loss of that data. One possible solution is programming a sufficient delay between the last data sent and the closesocket or shutdown function calls for an ATM socket.
- Half Close is not supported by ATM.
- Both abortive and graceful disconnects result in a RELEASE signal being sent out with the same cause field. In either case, received data at the remote end of the socket is still delivered to the application. See *Graceful Shutdown, Linger Options, and Socket Closure* for more information.

### Error Codes

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	The (blocking) Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
WSAEWOULDBLOCK	The socket is marked as nonblocking and <b>SO_LINGER</b> is set to a nonzero time-out value.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

Windows Sockets Programming Considerations Overview, **Socket Functions**, **accept**, **ioctlsocket**, **setsockopt**, **socket**, **WSAAsyncSelect**, **WSADuplicateSocket**

## connect

The Windows Sockets **connect** function establishes a connection to a specified socket.

```
int connect (  
    SOCKET                s,  
    const struct sockaddr FAR *name,  
    int                   namelen  
);
```

### Parameters

*s*

[in] Descriptor identifying an unconnected socket.

*name*

[in] Name of the socket to which the connection should be established.

*namelen*

[in] Length of *name*.

### Return Values

If no error occurs, **connect** returns zero. Otherwise, it returns **SOCKET\_ERROR**, and a specific error code can be retrieved by calling **WSAGetLastError**.

On a blocking socket, the return value indicates success or failure of the connection attempt.

With a nonblocking socket, the connection attempt cannot be completed immediately. In this case, **connect** will return **SOCKET\_ERROR**, and **WSAGetLastError** will return **WSAEWOULDBLOCK**. In this case, there are three possible scenarios:

- Use the **select** function to determine the completion of the connection request by checking to see if the socket is writeable.
- If the application is using **WSAAsyncSelect** to indicate interest in connection events, then the application will receive an **FD\_CONNECT** notification indicating that the **connect** operation is complete (successfully or not).
- If the application is using **WSAEventSelect** to indicate interest in connection events, then the associated event object will be signaled indicating that the **connect** operation is complete (successfully or not).

Until the connection attempt completes on a nonblocking socket, all subsequent calls to **connect** on the same socket will fail with the error code `WSAEALREADY`, and `WSAEISCONN` when the connection completes successfully. Due to ambiguities in version 1.1 of the Windows Sockets specification, error codes returned from **connect** while a connection is already pending may vary among implementations. As a result, it is not recommended that applications use multiple calls to connect to detect connection completion. If they do, they must be prepared to handle `WSAEINVAL` and `WSAEWOULDBLOCK` error values the same way that they handle `WSAEALREADY`, to assure robust execution.

If the error code returned indicates the connection attempt failed (that is, `WSAECONNREFUSED`, `WSAENETUNREACH`, `WSAETIMEDOUT`) the application can call **connect** again for the same socket.

### Remarks

The **connect** function is used to create a connection to the specified destination. If socket *s*, is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound.

For connection-oriented sockets (for example, type `SOCK_STREAM`), an active connection is initiated to the foreign host using *name* (an address in the name space of the socket; for a detailed description, see *bind* and `SOCKADDR`).

When the socket call completes successfully, the socket is ready to send and receive data. If the address member of the structure specified by the *name* parameter is all zeroes, **connect** will return the error `WSAEADDRNOTAVAIL`. Any attempt to reconnect an active connection will fail with the error code `WSAEISCONN`.

For connection-oriented, nonblocking sockets, it is often not possible to complete the connection immediately. In such a case, this function returns the error `WSAEWOULDBLOCK`. However, the operation proceeds.

When the success or failure outcome becomes known, it may be reported in one of two ways, depending on how the client registers for notification.

- If the client uses the **select** function, success is reported in the `writelfds` set and failure is reported in the `exceptfds` set.
- If the client uses the functions **WSAAsyncSelect** or **WSAEventSelect**, the notification is announced with `FD_CONNECT` and the error code associated with the `FD_CONNECT` indicates either success or a specific reason for failure.

For a connectionless socket (for example, type **SOCK\_DGRAM**), the operation performed by **connect** is merely to establish a default destination address that can be used on subsequent **send/WSASend** and **recv/WSARecv** calls. Any datagrams received from an address other than the destination address specified will be discarded. If the address member of the structure specified by *name* is all zeroes, the socket will be disconnected. Then, the default remote address will be indeterminate, so **send/WSASend** and **recv/WSARecv** calls will return the error code **WSAENOTCONN**. However, **sendto/WSASendTo** and **recvfrom/WSARecvFrom** can still be used. The default destination can be changed by simply calling **connect** again, even if the socket is already connected. Any datagrams queued for receipt are discarded if *name* is different from the previous **connect**.

For connectionless sockets, *name* can indicate any valid address, including a broadcast address. However, to connect to a broadcast address, a socket must use **setsockopt** to enable the **SO\_BROADCAST** option. Otherwise, **connect** will fail with the error code **WSAEACCES**.

When a connection between sockets is broken, the sockets should be discarded and recreated. When a problem develops on a connected socket, the application must discard and recreate the needed sockets in order to return to a stable point.

### Notes for IrDA Sockets

- The `Af_irda.h` header file must be explicitly included.
- If an existing IrDA connection is detected at the media-access level, **WSAENETDOWN** is returned.
- If active connections to a device with a different address exist, **WSAEADDRINUSE** is returned.
- **For Windows CE only:** If IAS name resolution fails because another IAS query is in progress, **WSAEINPROGRESS** is returned. In this situation, retrying the operation at one-second intervals is recommended.
- If the socket is already connected or an exclusive/multiplexed mode change failed, **WSAEISCONN** is returned.
- If the socket was previously bound to a local service name to accept incoming connections using **bind**, **WSAEINVAL** is returned. Note that once a socket is bound, it cannot be used for establishing an outbound connection.

IrDA implements the connect function with addresses of the form **sockaddr\_irda**. Typically, a client application will create a socket with the socket function, scan the immediate vicinity for IrDA devices with the **IRLMP\_ENUMDEVICES** socket option, choose a device from the returned list, form an address, and call **connect**. There is no difference between blocking and nonblocking semantics.

## Error Codes

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The socket's local address is already in use and the socket was not marked to allow address reuse with <code>SO_REUSEADDR</code> . This error usually occurs when executing <b>bind</b> , but could be delayed until this function if the <b>bind</b> was to a partially wildcard address (involving <code>ADDR_ANY</code> ) and if a specific address needs to be committed at the time of this function.
WSAEINTR	The blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	A nonblocking <b>connect</b> call is in progress on the specified socket.
<hr/> <p><b>Note</b> In order to preserve backward compatibility, this error is reported as <code>WSAEINVAL</code> to Windows Sockets 1.1 applications that link to either <code>Winsock.dll</code> or <code>Wsock32.dll</code>.</p> <hr/>	
WSAEADDRNOTAVAIL	The remote address is not a valid address (such as <code>ADDR_ANY</code> ).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAEFAULT	The <i>name</i> or the <i>namelen</i> parameter is not a valid part of the user address space, the <i>namelen</i> parameter is too small, or the <i>name</i> parameter contains incorrect address format for the associated address family.
WSAEINVAL	The parameter <i>s</i> is a listening socket.
WSAEISCONN	The socket is already connected (connection-oriented sockets only).
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the connection cannot be completed immediately.
WSAEACCES	Attempt to connect datagram socket to broadcast address failed because <b>setsockopt</b> option <code>SO_BROADCAST</code> is not enabled.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **accept**, **bind**, **getsockname**, **select**, **socket**, **WSAAsyncSelect**, **WSAConnect**

---

## EnumProtocols

**Important** The **EnumProtocols** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is *obsolete*. For the convenience of Windows Sockets 1.1 developers, the reference material is included.

The **WSAEnumProtocols** function provides equivalent functionality in Windows Sockets 2.

The **EnumProtocols** function obtains information about a specified set of network protocols that are active on a local host.

```
INT EnumProtocols (
    LPINT lpiProtocols,      // pointer to array of protocol
                           // identifiers
    LPVOID lpProtocolBuffer, // pointer to buffer to receive protocol
                           // information
    LPDWORD lpdwBufferLength // pointer to variable that specifies
                           // the size of the receiving buffer
);
```

### Parameters

#### *lpiProtocols*

Pointer to a null-terminated array of protocol identifiers. The **EnumProtocols** function obtains information about the protocols specified by this array.

If *lpiProtocols* is NULL, the function obtains information about all available protocols.

The following protocol identifier values are defined.

Value	Protocol
IPPROTO_TCP	TCP/IP, a connection/stream-oriented protocol.
IPPROTO_UDP	User Datagram Protocol (UDP/IP), a connectionless datagram protocol.

Value	Protocol
ISOPROTO_TP4	ISO connection-oriented transport protocol.
NSPROTO_IPX	IPX.
NSPROTO_SPX	SPX.
NSPROTO_SPXII	SPX II.

***IpProtocolBuffer***

Pointer to a buffer that the function fills with an array of **PROTOCOL\_INFO** data structures.

***lpdwBufferLength***

Pointer to a variable that, on input, specifies the size, in bytes, of the buffer pointed to by *IpProtocolBuffer*.

On output, the function sets this variable to the minimum buffer size needed to retrieve all of the requested information. For the function to succeed, the buffer must be at least this size.

**Return Values**

If the function succeeds, the return value is the number of **PROTOCOL\_INFO** data structures written to the buffer pointed to by *IpProtocolBuffer*.

If the function fails, the return value is **SOCKET\_ERROR** (-1). To get extended error information, call **GetLastError**. **GetLastError** can return the following extended error code.

Error code	Meaning
ERROR_INSUFFICIENT_BUFFER	The buffer pointed to by <i>IpProtocolBuffer</i> was too small to receive all of the relevant <b>PROTOCOL_INFO</b> structures. Call the function with a buffer at least as large as the value returned in <i>*lpdwBufferLength</i> .

**Remarks**

In the following sample code, the **EnumProtocols** function obtains information about all protocols that are available on a system. The code then examines each of the protocols in greater detail.

```
SOCKET
OpenConnection (
    PTSTR ServiceName,
    PGUID ServiceType,
    BOOL Reliable,
    BOOL MessageOriented,
    BOOL StreamOriented,
    BOOL Connectionless,
```



```
PINT ProtocolUsed
)
{
    // local variables
    INT protocols[MAX_PROTOCOLS+1];
    BYTE buffer[2048];
    DWORD bytesRequired;
    INT err;
    PPROTOCOL_INFO protocolInfo;
    PCSADDR_INFO csaddrInfo;
    INT protocolCount;
    INT addressCount;
    INT i;
    DWORD protocolIndex;
    SOCKET s;

    // First look up the protocols installed on this machine.
    //
    bytesRequired = sizeof(buffer);
    err = EnumProtocols( NULL, buffer, &bytesRequired );
    if ( err <= 0 )
        return INVALID_SOCKET;

    // Walk through the available protocols and pick out the ones which
    // support the desired characteristics.
    //
    protocolCount = err;
    protocolInfo = (PPROTOCOL_INFO)buffer;

    for ( i = 0, protocolIndex = 0;
          i < protocolCount && protocolIndex < MAX_PROTOCOLS;
          i++, protocolInfo++ ) {

        // If connection-oriented support is requested, then check if
        // supported by this protocol. We assume here that connection-
        // oriented support implies fully reliable service.
        //

        if ( Reliable ) {
            // Check to see if the protocol is reliable. It must
            // guarantee both delivery of all data and the order in
            // which the data arrives.
            //
            if ( (protocolInfo->dwServiceFlags &
                  XP_GUARANTEED_DELIVERY) == 0
```

*(continued)*



*(continued)*

```

        ||
        (protocolInfo->dwServiceFlags &
         XP_GUARANTEED_ORDER) == 0 ) {

        continue;
    }

    // Check to see that the protocol matches the stream/message
    // characteristics requested.
    //
    if ( StreamOriented &&
        (protocolInfo->dwServiceFlags & XP_MESSAGE_ORIENTED)
         != 0 &&
        (protocolInfo->dwServiceFlags & XP_PSEUDO_STREAM)
         == 0 ) {
        continue;
    }

    if ( MessageOriented &&
        (protocolInfo->dwServiceFlags & XP_MESSAGE_ORIENTED)
         == 0 ) {
        continue;
    }

    }
else if ( Connectionless ) {
    // Make sure that this is a connectionless protocol.
    //
    if ( (protocolInfo->dwServiceFlags & XP_CONNECTIONLESS)
         != 0 )
        continue;
    }

    // This protocol fits all the criteria. Add it to the list of
    // protocols in which we're interested.
    //
    protocols[protocolIndex++] = protocolInfo->iProtocol;
}

```


### ! Requirements

**Version:** Requires Windows Sockets 1.1. A Microsoft-specific extension. Obsolete for Windows Sockets 2.0.

**Header:** Declared in Nspapi.h.

**Library:** Use Wsock32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT/2000.

 See Also

**GetAddressByName**, **PROTOCOL\_INFO**

## GetAcceptExSockaddrs

The Windows Sockets **GetAcceptExSockaddrs** function parses the data obtained from a call to the **AcceptEx** function and passes the local and remote addresses to a **SOCKADDR** structure.

**Note** This function is a Microsoft-specific extension to the Windows Sockets specification. For more information, see *Microsoft Extensions and Windows Sockets 2*.

```
VOID GetAcceptExSockaddrs (  
    PVOID                IpOutputBuffer,  
    DWORD                dwReceiveDataLength,  
    DWORD                dwLocalAddressLength,  
    DWORD                dwRemoteAddressLength,  
    LPSOCKADDR           *LocalSockaddr,  
    LPINT                LocalSockaddrLength,  
    LPSOCKADDR           *RemoteSockaddr,  
    LPINT                RemoteSockaddrLength  
);
```

### Parameters

#### *IpOutputBuffer*

[in] Pointer to a buffer that receives the first block of data sent on a connection resulting from an **AcceptEx** call. Must be the same *IpOutputBuffer* parameter that was passed to the **AcceptEx** function.

#### *dwReceiveDataLength*

[in] Number of bytes in the buffer used for receiving the first data. This value must be equal to the *dwReceiveDataLength* parameter that was passed to the **AcceptEx** function.

#### *dwLocalAddressLength*

[in] Number of bytes reserved for the local address information. Must be equal to the *dwLocalAddressLength* parameter that was passed to the **AcceptEx** function.

#### *dwRemoteAddressLength*

[in] Number of bytes reserved for the remote address information. This value must be equal to the *dwRemoteAddressLength* parameter that was passed to the **AcceptEx** function.

*LocalSockaddr*

[out] Pointer to the **SOCKADDR** structure that receives the local address of the connection (the same information that would be returned by the Windows Sockets **getsockname** function). This parameter must be specified.

*LocalSockaddrLength*

[out] Size of the local address. This parameter must be specified.

*RemoteSockaddr*

[out] Pointer to the **SOCKADDR** structure that receives the remote address of the connection (the same information that would be returned by the Windows Sockets **getpeername** function). This parameter must be specified.

*RemoteSockaddrLength*

[out] Size of the local address. This parameter must be specified.

**Return Values**

This function does not return a value.

**Remarks**

The **GetAcceptExSockaddrs** function is used exclusively with the **AcceptEx** function to parse the first data that the socket receives into local and remote addresses. You are required to use this function because the **AcceptEx** function writes address information in an internal (TDI) format. The **GetAcceptExSockaddrs** routine is required to locate the **SOCKADDR** structures in the buffer.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later. A Microsoft-specific extension.

**Header:** Declared in Mswsock.h.

**Library:** Use Mswsock.lib.

---

## GetAddressByName

The **GetAddressByName** function queries a name space, or a set of default name spaces, in order to obtain network address information for a specified network service. This process is known as service name resolution. A network service can also use the function to obtain local address information that it can use with the **bind** function.

---

**Important** The **GetAddressByName** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is *obsolete*. For the convenience of Windows Sockets 1.1 developers, the reference material is as follows.

The functions detailed in *Protocol-Independent Name Resolution* provide equivalent functionality in Windows Sockets 2.

---

```

INT GetAddressByName (
    DWORD dwNameSpace,      // name space to query for service
                          // address information
    LPGUID lpServiceType,   // the type of the service
    LPTSTR lpServiceName,   // the name of the service
    LPINT lpProtocols,      // points to array of
                          // protocol identifiers
    DWORD dwResolution,    // set of bit flags that specify
                          // aspects of name resolution
    LPSERVICE_ASYNC_INFO lpServiceAsyncInfo,
                          // reserved for future use,
                          // must be NULL
    LPVOID lpCsaddrBuffer,  // points to buffer to receive
                          // address information
    LPDWORD lpdwBufferLength, // points to variable
                          //with address buffer size
                          //information
    LPTSTR lpAliasBuffer,   // points to buffer to
                          //receive alias information
    LPDWORD lpdwAliasBufferLength
                          // points to variable with alias
                          // buffer size information
);

```

## Parameters

### *dwNameSpace*

Specifies the name space, or a set of default name spaces, that the operating system will query for network address information.

Use one of the following constants to specify a name space.

Value	Name space
NS_DEFAULT	A set of default name spaces. The function queries each name space within this set. The set of default name spaces typically includes all the name spaces installed on the system. System administrators, however, can exclude particular name spaces from the set. This is the value that most applications should use for <i>dwNameSpace</i> .
NS_DNS	The Domain Name System used in the Internet for host name resolution.
NS_NETBT	The NetBIOS over TCP/IP layer. All Windows NT/Windows 2000 systems register their computer names with NetBIOS. This name space is used to convert a computer name to an IP address that uses this registration. Note that NS_NETBT can access a WINS server to perform the resolution.

(continued)

*(continued)*

Value	Name space
NS_SAP	The Netware Service Advertising Protocol. This can access the Netware bindery if appropriate. NS_SAP is a dynamic name space that allows registration of services.
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\hosts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and looks up host names and IP addresses in cache of host to IP address mappings.

Most calls to **GetAddressByName** should use the special value NS\_DEFAULT. This lets a client get by with no knowledge of which name spaces are available on an internetwork. The system administrator determines name space access. Name spaces can come and go without the client having to be aware of the changes.

#### *IpServiceType*

Pointer to a globally unique identifier (GUID) that specifies the type of the network service. The header file Svcguid.h includes definitions of several GUID service types, and macros for working with them.

#### *IpServiceName*

Pointer to a zero-terminated string that uniquely represents the service name. For example, "MY SNA SERVER".

Setting *IpServiceName* to NULL is the equivalent of setting *dwResolution* to RES\_SERVICE. The function operates in its second mode, obtaining the local address to which a service of the specified type should bind. The function stores the local address within the **LocalAddr** member of the **CSADDR\_INFO** structures stored into *\*IpCsaddrBuffer*.

If *dwResolution* is set to RES\_SERVICE, the function ignores the *IpServiceName* parameter.

If *dwNameSpace* is set to NS\_DNS, *\*IpServiceName* is the name of the host.

#### *IpiProtocols*

Pointer to a zero-terminated array of protocol identifiers. The function restricts a name resolution attempt to name space providers that offer these protocols. This lets the caller limit the scope of the search.

If *IpiProtocols* is NULL, the function obtains information on all available protocols.

#### *dwResolution*

Set of bit flags that specify aspects of the service name resolution process. The following bit flags are defined:

Value	Meaning
RES_SERVICE	If set, the function obtains the address to which a service of the specified type should bind. This is the equivalent of setting <i>lpServiceName</i> to NULL.  If this flag is clear, normal name resolution occurs.
RES_FIND_MULTIPLE	If this flag is set, the operating system performs an extensive search of all name spaces for the service. It asks every appropriate name space to resolve the service name. If this flag is clear, the operating system stops looking for service addresses as soon as one is found.
RES_SOFT_SEARCH	This flag is valid if the name space supports multiple levels of searching.  If this flag is valid and set, the operating system performs a simple and quick search of the name space. This is useful if an application only needs to obtain easy-to-find addresses for the service.  If this flag is valid and clear, the operating system performs a more extensive search of the name space.

#### *lpServiceAsyncInfo*

Reserved for future use; must be set to NULL.

#### *lpCsaddrBuffer*

Pointer to a buffer to receive one or more **CSADDR\_INFO** data structures. The number of structures written to the buffer depends on the amount of information found in the resolution attempt. You should assume that multiple structures will be written, although in many cases there will only be one.

#### *lpdwBufferLength*

Pointer to a variable that, upon input, specifies the size, in bytes, of the buffer pointed to by *lpCsaddrBuffer*.

Upon output, this variable contains the total number of bytes required to store the array of **CSADDR\_INFO** structures. If this value is less than or equal to the input value of *\*lpdwBufferLength*, and the function is successful, this is the number of bytes actually stored in the buffer. If this value is greater than the input value of *\*lpdwBufferLength*, the buffer was too small, and the output value of *\*lpdwBufferLength* is the minimal required buffer size.

#### *lpAliasBuffer*

Pointer to a buffer to receive alias information for the network service.

If a name space supports aliases, the function stores an array of zero-terminated name strings into the buffer pointed to by *lpAliasBuffer*. There is a double zero-terminator at the end of the list. The first name in the array is the service's primary name. Names that follow are aliases. An example of a name space that supports aliases is DNS.

If a name space does not support aliases, it stores a double zero-terminator into the buffer.

This parameter is optional, and can be set to NULL.

#### *lpdwAliasBufferLength*

Pointer to a variable that, upon input, specifies the size, in bytes, of the buffer pointed to by *lpAliasBuffer*.

Upon output, this variable contains the total number of bytes required to store the array of name strings. If this value is less than or equal to the input value of *\*lpdwAliasBufferLength*, and the function is successful, this is the number of bytes actually stored in the buffer. If this value is greater than the input value of *\*lpdwAliasBufferLength*, the buffer was too small, and the output value of *\*lpdwAliasBufferLength* is the minimal required buffer size.

If *lpAliasBuffer* is NULL, *lpdwAliasBufferLength* is meaningless and can also be NULL.

### Return Values

If the function succeeds, the return value is the number of **CSADDR\_INFO** data structures written to the buffer pointed to by *lpCsaddrBuffer*.

If the function fails, the return value is **SOCKET\_ERROR**(-1). To get extended error information, call **GetLastError**. **GetLastError** can return the following extended error value.

Error code	Meaning
<b>ERROR_INSUFFICIENT_BUFFER</b>	The buffer pointed to by <i>lpCsaddrBuffer</i> was too small to receive all of the relevant <b>CSADDR_INFO</b> structures. Call the function with a buffer at least as large as the value returned in <i>*lpdwBufferLength</i> .

### Remarks

This function is a more powerful version of the Windows Sockets function **gethostbyname**. The **GetAddressByName** function works with multiple name services.

The **GetAddressByName** function lets a client obtain a Windows Sockets address for a network service. The client specifies the service of interest by its service type and service name.

Many name services support a default prefix or suffix that the name service provider considers when resolving service names. For example, in the DNS name space, if a domain is named "nt.microsoft.com", and "ftp millikan" is provided as input, the DNS software fails to resolve "millikan", but successfully resolves "millikan.nt.microsoft.com".

Note that the **GetAddressByName** function can search for a service address in two ways: within a particular name space, or within a set of default name spaces. Using a default name space, an administrator can specify that certain name spaces will be searched for service addresses only if specified by name. An administrator or name space setup program can also control the ordering of name space searches.



**!** Requirements

**Version:** Requires Windows Sockets 1.1. A Microsoft-specific extension. Obsolete for Windows Sockets 2.0.

**Header:** Declared in Nspapi.h.

**Library:** Use Wsock32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT/2000.

**+** See Also

gethostbyname, CSADDR\_INFO

---

## gethostbyaddr

The Windows Sockets **gethostbyaddr** function retrieves the host information corresponding to a network address.

```
struct HOSTENT FAR * gethostbyaddr (
    const char FAR    *addr,
    int               len,
    int               type
);
```

### Parameters

*addr*

[in] Pointer to an address in network byte order.

*len*

[in] Length of the address.

*type*

[in] Type of the address, such as the AF\_INET address family type (defined as TCP, UDP, and other associated Internet protocols). Address family types and their corresponding values are defined in the winsock2.h header file.

### Return Values

If no error occurs, **gethostbyaddr** returns a pointer to the **HOSTENT** structure. Otherwise, it returns a NULL pointer, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.

(continued)



*(continued)*

Error code	Meaning
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative host not found, or server failed.
WSANO_RECOVERY	A nonrecoverable error occurred.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEAFNOSUPPORT	The <i>type</i> specified is not supported by the Windows Sockets implementation.
WSAEFAULT	The <i>addr</i> parameter is not a valid part of the user address space, or the <i>len</i> parameter is too small.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .

**Remarks**

The **gethostbyaddr** function returns a pointer to the **HOSTENT** structure that contains the name and address corresponding to the given network address.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

**gethostbyname**, **HOSTENT**, **WSAAsyncGetHostByAddr**

---

## gethostbyname

The Windows Sockets **gethostbyname** function retrieves host information corresponding to a host name from a host database.

```
struct hostent FAR *gethostbyname (
    const char FAR *name
);
```

**Parameters**

*name*

[out] Pointer to the null-terminated name of the host to resolve.

## Return Values

If no error occurs, **gethostbyname** returns a pointer to the **HOSTENT** structure described above. Otherwise, it returns a NULL pointer and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative host not found, or server failure.
WSANO_RECOVERY	A nonrecoverable error occurred.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>name</i> parameter is not a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .

## Remarks

The **gethostbyname** function returns a pointer to a **HOSTENT** structure—a structure allocated by Windows Sockets. The **HOSTENT** structure contains the results of a successful search for the host specified in the *name* parameter.

The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other Windows Sockets function calls.

The **gethostbyname** function cannot resolve IP address strings passed to it. Such a request is treated exactly as if an unknown host name were passed. Use **inet\_addr** to convert an IP address string the string to an actual IP address, then use another function, **gethostbyaddr**, to obtain the contents of the **HOSTENT** structure.


The **gethostbyname** function resolves the string returned by a successful call to **gethostname**.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

 See Also**gethostbyaddr, WSAAsyncGetHostByName**

---

## gethostname

The Windows Sockets **gethostname** function returns the standard host name for the local machine.

```
int gethostname (  
    char FAR    *name,  
    int        namelen  
);
```

### Parameters

*name*

[out] Pointer to a buffer that receives the local host name.

*namelen*

[in] Length of the buffer.

### Return Values

If no error occurs, **gethostname** returns zero. Otherwise, it returns **SOCKET\_ERROR** and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSAEFAULT	The <i>name</i> parameter is not a valid part of the user address space, or the buffer size specified by <i>namelen</i> parameter is too small to hold the complete host name.
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.

### Remarks

The **gethostname** function returns the name of the local host into the buffer specified by the *name* parameter. The host name is returned as a null-terminated string. The form of the host name is dependent on the Windows Sockets provider—it can be a simple host name, or it can be a fully qualified domain name. However, it is guaranteed that the name returned will be successfully parsed by **gethostbyname** and **WSAAsyncGetHostByName**.

---

**Note** If no local host name has been configured, **gethostname** must succeed and return a token host name that **gethostbyname** or **WSAAsyncGetHostByName** can resolve.

---

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

**+** See Also

**gethostbyname**, **WSAAsyncGetHostByName**

---

## GetNameByType

The **GetNameByType** function obtains the name of a network service. The network service is specified by its service type.

---

**Important** The **GetNameByType** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is *obsolete*. For the convenience of Windows Sockets 1.1 developers, the reference material is as follows.

The functions detailed in *Protocol-Independent Name Resolution* provide equivalent functionality in Windows Sockets 2.

---

```
INT GetNameByType (
    LPGUID lpServiceType, // points to network
                        //service type GUID
    LPTSTR lpServiceName, // points to buffer to receive name
                        // of network service
    DWORD dwNameLength // points to variable that
                        // specifies buffer size
);
```

### Parameters

*lpServiceType*

Pointer to a globally unique identifier (GUID) that specifies the type of the network service. The header file `Svcguid.h` includes definitions of several GUID service types, and macros for working with them.

*lpServiceName*

Pointer to a buffer to receive a zero-terminated string that uniquely represents the name of the network service.

***dwNameLength***

Pointer to a variable that, on input, specifies the size of the buffer pointed to by *lpServiceName*. On output, the variable contains the actual size of the service name string.

**Return Values**

If the function succeeds, the return value is not `SOCKET_ERROR` (-1).

If the function fails, the return value is `SOCKET_ERROR` (-1). To get extended error information, call **GetLastError**.

**! Requirements**

**Version:** Requires Windows Sockets 1.1. A Microsoft-specific extension. Obsolete for Windows Sockets 2.0.

**Header:** Declared in `Nspapi.h`.

**Library:** Use `Wsock32.lib`.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT/2000.

**+ See Also****GetTypeByName**

---

## getpeername

The Windows Sockets **getpeername** function retrieves the name of the peer to which a socket is connected.

```
int getpeername (  
    SOCKET          s,  
    struct sockaddr FAR *name,  
    int FAR        *namelen  
);
```

**Parameters**

*s*

[in] Descriptor identifying a connected socket.

*name*

[out] The structure that receives the name of the peer.

*namelen*

[in/out] Pointer to the size of the *name* structure.

## Return Values

If no error occurs, **getpeername** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>name</i> or the <i>namelen</i> parameter is not a valid part of the user address space, or the <i>namelen</i> parameter is too small.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.

## Remarks

The **getpeername** function retrieves the name of the peer connected to the socket *s* and stores it in the **SOCKADDR** structure identified by *name*. The **getpeername** function can be used only on a connected socket. For datagram sockets, only the name of a peer specified in a previous **connect** call will be returned—any name specified by a previous **sendto** call will *not* be returned by **getpeername**.

On call, the *namelen* argument contains the size of the *name* buffer, in bytes. On return, the *namelen* parameter contains the actual size in bytes of the name returned.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Wsock32.lib`.

### + See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **bind**, **getsockname**, **socket**

---

## getprotobyname

The Windows Sockets **getprotobyname** function retrieves the protocol information corresponding to a protocol name.

```

struct PROTOENT FAR * getprotobyname (
    const char FAR *name
);

```

## Parameters

*name*

[in] Pointer to a null-terminated protocol name.

## Return Values

If no error occurs, **getprotobyname** returns a pointer to the **PROTOENT**. Otherwise, it returns a NULL pointer and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	A nonauthoritative Protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>name</i> parameter is not a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .

## Remarks

The **getprotobyname** function returns a pointer to the **PROTOENT** structure containing the name(s) and protocol number that correspond to the protocol specified in the *name* parameter. All strings are null-terminated. The **PROTOENT** structure is allocated by the Windows Sockets library. An application must never attempt to modify this structure or to free any of its components. Furthermore, like **HOSTENT**, only one copy of this structure is allocated per thread, so the application should copy any information that it needs before issuing any other Windows Sockets function calls.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

**+** See Also

`getprotobyname`, `WSAAsyncGetProtoByName`

## getprotobyname

The Windows Sockets **getprotobyname** function retrieves protocol information corresponding to a protocol number.

```
struct PROTOENT FAR * getprotobyname (
    int    number
);
```

### Parameters

*number*

[in] Protocol number, in host byte order.

### Return Values

If no error occurs, **getprotobyname** returns a pointer to the **PROTOENT** structure. Otherwise, it returns a NULL pointer and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	A nonauthoritative Protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .



## Remarks

This **getprotobynumber** function returns a pointer to the **PROTOENT** structure as previously described in **getprotobyname**. The contents of the structure correspond to the given protocol number.

The pointer that is returned points to the structure allocated by Windows Sockets. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information that it needs before issuing any other Windows Sockets function calls.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

### + See Also

**getprotobyname**, **WSAAsyncGetProtoByNumber**

---

# getservbyname

The Windows Sockets **getservbyname** function retrieves service information corresponding to a service name and protocol.

```
struct servent FAR * getservbyname (  
    const char FAR *name,  
    const char FAR *proto  
);
```

## Parameters

*name*

[in] Pointer to a null-terminated service name.

*proto*

[in] Optional pointer to a null-terminated protocol name. If this pointer is NULL, **getservbyname** returns the first service entry where *name* matches the **s\_name** member of the **SERVENT** structure or the **s\_aliases** member of the **SERVENT** structure. Otherwise, **getservbyname** matches both the *name* and the *proto*.

## Return Values

If no error occurs, **getservbyname** returns a pointer to the **SERVENT** structure. Otherwise, it returns a NULL pointer and a specific error number can be retrieved by calling **WSAGetLastError**.

---

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Service not found.
WSATRY_AGAIN	A nonauthoritative Service not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .

### Remarks

The **getservbyname** function returns a pointer to the **SERVENT** structure containing the name(s) and service number that match the string in the *name* parameter. All strings are null-terminated.

The pointer that is returned points to the **SERVENT** structure allocated by the Windows Sockets library. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other Windows Sockets function calls.

#### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

#### + See Also

**getservbyport**, **WSAAsyncGetServByName**

---

## getservbyport

The Windows Sockets **getservbyport** function retrieves service information corresponding to a port and protocol.

```

struct servent FAR * getservbyport (
    int          port,
    const char FAR *proto
);

```

### Parameters

*port*

[in] Port for a service, in network byte order.

*proto*

[in] Optional pointer to a protocol name. If this is NULL, **getservbyport** returns the first service entry for which the *port* matches the **s\_port** of the **SERVENT** structure. Otherwise, **getservbyport** matches both the *port* and the *proto* parameters.

### Return Values

If no error occurs, **getservbyport** returns a pointer to the **SERVENT** structure. Otherwise, it returns a NULL pointer and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Service not found.
WSATRY_AGAIN	A nonauthoritative Service not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>proto</i> parameter is not a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .

### Remarks

The **getservbyport** function returns a pointer to a **SERVENT** structure as it does in the **getservbyname** function.

The **SERVENT** structure is allocated by Windows Sockets. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other Windows Sockets function calls.

#### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

#### + See Also

**getservbyname, WSAAsyncGetServByPort**

## GetService

**Important** The **GetService** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is *obsolete*. For the convenience of Windows Sockets 1.1 developers, the reference material is as follows.

The functions detailed in *Protocol-Independent Name Resolution* provide equivalent functionality in Windows Sockets 2.

The **GetService** function obtains information about a network service in the context of a set of default name spaces or a specified name space. The network service is specified by its type and name. The information about the service is obtained as a set of **NS\_SERVICE\_INFO** data structures.

```

INT GetService (
    DWORD dwNameSpace, // specifies name space or
                       // spaces to search
    PGUID lpGuid,      // points to a GUID service type
    LPTSTR lpServiceName, // points to a service name
    DWORD dwProperties, // specifies service information be
                       // to obtained
    LPVOID lpBuffer,   // points to buffer to receive
                       // service information
    LPDWORD lpdwBufferSize, // points to size of buffer, size
                             // of service information
    LPSERVICE_ASYNC_INFO lpServiceAsyncInfo
                             // reserved for future use,
                             // must be NULL
);

```

## Parameters

### *dwNameSpace*

Specifies the name space, or a set of default name spaces, that the operating system queries for information about the specified network service.

Use one of the following constants to specify a name space.

Value	Name space
NS_DEFAULT	A set of default name spaces. The operating system queries each name space within this set. The set of default name spaces typically includes all the name spaces installed on the system. System administrators, however, can exclude particular name spaces from the set. NS_DEFAULT is the value that most applications should use for <i>dwNameSpace</i> .
NS_DNS	The Domain Name System used in the Internet for host name resolution.
NS_NETBT	The NetBIOS over TCP/IP layer. All Windows NT/Windows 2000 systems register their computer names with NetBIOS. This name space is used to resolve a computer name into an IP address using this registration. Note that NS_NETBT can access a WINS server to perform the resolution.
NS_SAP	The Netware Service Advertising Protocol. This can access the Netware bindery if appropriate. NS_SAP is a dynamic name space that allows registration of services.
NS_TCPIP_HOSTS	Looks up host names and IP addresses in the <systemroot>\system32\drivers\etc\hosts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and looks up host names and IP addresses in cache of host to IP address mappings.

Most calls to **GetService** should use the special value NS\_DEFAULT. This lets a client get by without knowing available name spaces on an internetwork. The system administrator determines name space access. Name spaces can come and go without the client having to be aware of the changes.

### *IpGuid*

Pointer to a globally unique identifier (GUID) that specifies the type of the network service. The header file Svcguid.h includes GUID service types from many well-known services within the DNS and SAP name spaces.

### *IpServiceName*

Pointer to a zero-terminated string that uniquely represents the service name. For example, "MY SNA SERVER".

*dwProperties*

Set of bit flags that specify the service information that the function obtains. Each of these bit flag constants, other than **PROP\_ALL**, corresponds to a particular member of the **SERVICE\_INFO** data structure. If the flag is set, the function puts information into the corresponding member of the data structures stored in *\*lpBuffer*. The following bit flags are defined.

Value	Name space
PROP_COMMENT	If this flag is set, the function stores data in the <b>lpComment</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_LOCALE	If this flag is set, the function stores data in the <b>lpLocale</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_DISPLAY_HINT	If this flag is set, the function stores data in the <b>dwDisplayHint</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_VERSION	If this flag is set, the function stores data in the <b>dwVersion</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_START_TIME	If this flag is set, the function stores data in the <b>dwTime</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_MACHINE	If this flag is set, the function stores data in the <b>lpMachineName</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_ADDRESSES	If this flag is set, the function stores data in the <b>lpServiceAddress</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_SD	If this flag is set, the function stores data in the <b>ServiceSpecificInfo</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_ALL	If this flag is set, the function stores data in all of the members of the data structures stored in <i>*lpBuffer</i> .

*lpBuffer*

Pointer to a buffer to receive an array of **NS\_SERVICE\_INFO** structures and associated service information. Each **NS\_SERVICE\_INFO** structure contains service information in the context of a particular name space. Note that if *dwNameSpace* is **NS\_DEFAULT**, the function stores more than one structure into the buffer; otherwise, just one structure is stored.

Each **NS\_SERVICE\_INFO** structure contains a **SERVICE\_INFO** structure. The members of these **SERVICE\_INFO** structures will contain valid data based on the bit flags that are set in the *dwProperties* parameter. If a member's corresponding bit flag is not set in *dwProperties*, the member's value is undefined.

The function stores the **NS\_SERVICE\_INFO** structures in a consecutive array, starting at the beginning of the buffer. The pointers in the contained **SERVICE\_INFO** structures point to information that is stored in the buffer between the end of the **NS\_SERVICE\_INFO** structures and the end of the buffer.

#### *lpdwBufferSize*

Pointer to a variable that, on input, contains the size, in bytes, of the buffer pointed to by *lpBuffer*. On output, this variable contains the number of bytes required to store the requested information. If this output value is greater than the input value, the function has failed due to insufficient buffer size.

#### *lpServiceAsyncInfo*

Reserved for future use. Must be set to NULL.

### Return Values

If the function succeeds, the return value is the number of **NS\_SERVICE\_INFO** structures stored in *\*lpBuffer*. Zero indicates that no structures were stored.

If the function fails, the return value is **SOCKET\_ERROR** (-1). To get extended error information, call **GetLastError**. **GetLastError** can return one of the following extended error values.

Error code	Meaning
ERROR_INSUFFICIENT_BUFFER	The buffer pointed to by <i>lpBuffer</i> is too small to receive all of the requested information. Call the function with a buffer at least as large as the value returned in <i>*lpdwBufferSize</i> .
ERROR_SERVICE_NOT_FOUND	The specified service was not found, or the specified name space is not in use. The function return value is zero in this case.

#### ! Requirements

**Version:** Requires Windows Sockets 1.1. A Microsoft-specific extension. Obsolete for Windows Sockets 2.0.

**Header:** Declared in *Nspapi.h*.

**Library:** Use *Wsock32.lib*.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT/2000.

#### + See Also

**SetService, NS\_SERVICE\_INFO, SERVICE\_INFO**

# getsockname

The Windows Sockets **getsockname** function retrieves the local name for a socket.

```
int getsockname (
    SOCKET          s,
    struct sockaddr FAR *name,
    int FAR         *namelen
);
```

## Parameters

*s*

[in] Descriptor identifying a socket.

*name*

[out] Receives the address (name) of the socket.

*namelen*

[in/out] Size of the *name* buffer.

## Return Values

If no error occurs, **getsockname** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>name</i> or the <i>namelen</i> parameter is not a valid part of the user address space, or the <i>namelen</i> parameter is too small.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound to an address with <b>bind</b> , or <code>ADDR_ANY</code> is specified in <b>bind</b> but connection has not yet occurs.

## Remarks

The **getsockname** function retrieves the current name for the specified socket descriptor in *name*. It is used on the bound or connected socket specified by the *s* parameter. The local association is returned. This call is especially useful when a **connect** call has been made without doing a **bind** first; the **getsockname** function provides the only way to determine the local association that has been set by the system.



On call, the *namelen* argument contains the size of the *name* buffer, in bytes. On return, the *namelen* parameter contains the actual size in bytes of the *name* parameter.

The **getsockname** function does not always return information about the host address when the socket has been bound to an unspecified address, unless the socket has been connected with **connect** or **accept** (for example, using `ADDR_ANY`). A Windows Sockets application must not assume that the address will be specified unless the socket is connected. The address that will be used for the socket is unknown unless the socket is connected when used in a multihomed host. If the socket is using a connectionless protocol, the address may not be available until I/O occurs on the socket.

#### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later. Not supported on Windows 95.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Wsock32.lib`.

#### + See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **bind**, **getpeername**, **socket**

---

## getsockopt

The Windows Sockets **getsockopt** function retrieves a socket option.

```
int getsockopt (  
    SOCKET          s,  
    int             level,  
    int             optname,  
    char FAR       *optval,  
    int FAR        *optlen  
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*level*

[in] Level at which the option is defined; the supported levels include `SOL_SOCKET` and `IPPROTO_TCP`. See the *Windows Sockets 2 Protocol-Specific Annex* (a separate document included with the Platform SDK) for more information on protocol-specific levels.

*optname*

[in] Socket option for which the value is to be retrieved.

*optval*

[out] Pointer to the buffer in which the value for the requested option is to be returned.

*optlen*

[in/out] Pointer to the size of the *optval* buffer.

## Return Values

If no error occurs, **getsockopt** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	One of the <i>optval</i> or the <i>optlen</i> parameters is not a valid part of the user address space, or the <i>optlen</i> parameter is too small.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The <i>level</i> parameter is unknown or invalid.
WSAENOPROTOPT	The option is unknown or unsupported by the indicated protocol family.
WSAENOTSOCK	The descriptor is not a socket.

## Remarks

The **getsockopt** function retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in *optval*. Options can exist at multiple protocol levels, but they are always present at the uppermost socket level. Options affect socket operations, such as the packet routing and OOB data transfer.

The value associated with the selected option is returned in the buffer *optval*. The integer pointed to by *optlen* should originally contain the size of this buffer; on return, it will be set to the size of the value returned. For `SO_LINGER`, this will be the size of a **LINGER** structure. For most other options, it will be the size of an integer.

The application is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specified.

If the option was never set with **setsockopt**, then **getsockopt** returns the default value for the option.

The following options are supported for **getsockopt**. The Type column identifies the type of data addressed by *optval*.

*level* = SOL\_SOCKET

Value	Type	Meaning
<b>SO_ACCEPTCONN</b>	BOOL	Socket is listening.
<b>SO_BROADCAST</b>	BOOL	Socket is configured for the transmission of broadcast messages.
<b>SO_CONDITIONAL_ACCEPT</b>	BOOL	Returns current socket state, either from a previous call to <b>setsockopt</b> or the system default.
<b>SO_DEBUG</b>	BOOL	Debugging is enabled.
<b>SO_DONTLINGER</b>	BOOL	If TRUE, the <b>SO_LINGER</b> option is disabled.
<b>SO_DONTROUTE</b>	BOOL	Routing is disabled. Not supported on ATM sockets.
<b>SO_ERROR</b>	int	Retrieves error status and clear.
<b>SO_GROUP_ID</b>	GROUP	Reserved.
<b>SO_GROUP_PRIORITY</b>	int	Reserved.
<b>SO_KEEPAIVE</b>	BOOL	Keep-alives are being sent. Not supported on ATM sockets.
<b>SO_LINGER</b>	<b>LINGER</b> structure	Returns the current linger options.
<b>SO_MAX_MSG_SIZE</b>	unsigned int	Maximum size of a message for message-oriented socket types (for example, <b>SOCK_DGRAM</b> ). Has no meaning for stream oriented sockets.
<b>SO_OOINLINE</b>	BOOL	OOB data is being received in the normal data stream. (See section Windows Sockets 1.1 Blocking Routines and EINPROGRESS for a discussion of this topic.)
<b>SO_PROTOCOL_INFO</b>	<b>WSAPROTOCOL_INFO</b>	Description of protocol information for protocol that is bound to this socket.
<b>SO_RCVBUF</b>	int	Buffer size for receives.
<b>SO_REUSEADDR</b>	BOOL	The socket can be bound to an address which is already in use. Not applicable for ATM sockets.
<b>SO_SNDBUF</b>	int	Buffer size for sends.
<b>SO_TYPE</b>	int	The type of the socket (for example, <b>SOCK_STREAM</b> ).
<b>PVD_CONFIG</b>	Service Provider Dependent	An opaque data structure object from the service provider associated with socket <i>s</i> . This object stores the current configuration information of the service provider. The exact format of this data structure is service provider specific.

*level = IPPROTO\_TCP*

Value	Type	Meaning
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.

*level = NSPROTO\_IPX*

**Note** Windows 2000 and Windows NT support all IPX options. Windows 95 and Windows 98 support only the following:

IPX\_PTYPE  
 IPX\_FILTERPTYPE  
 IPX\_DSTYPE  
 IPX\_RECVHDR  
 IPX\_MAXSIZE  
 IPX\_ADDRESS

Value	Type	Meaning
IPX_PTYPE	int	Obtains the IPX packet type.
IPX_FILTERPTYPE	int	Obtains the receive filter packet type
IPX_DSTYPE	int	Obtain the value of the data stream field in the SPX header on every packet sent.
IPX_EXTENDED_ADDRESS	BOOL	Find out whether extended addressing is enabled.
IPX_RECVHDR	BOOL	Find out whether the protocol header is sent up on all receive headers.
IPX_MAXSIZE	int	Obtain the maximum data size that can be sent.
IPX_ADDRESS	IPX_ADDRESS_DATA structure	Obtain information about a specific adapter to which IPX is bound. Adapter numbering is base zero. The <b>adapternum</b> member is filled in upon return.
IPX_GETNETINFO	IPX_NETNUM_DATA structure	Obtain information about a specific IPX network number. If not available in the cache, uses RIP to obtain information.

(continued)

*(continued)*

Value	Type	Meaning
<b>IPX_GETNETINFO_NORIP</b>	<b>IPX_NETNUM_DATA</b> structure	Obtain information about a specific IPX network number. If not available in the cache, will <i>not</i> use RIP to obtain information, and returns error.
<b>IPX_SPXGETCONNECTIONSTATUS</b>	<b>IPX_SPXCONNSTATUS_DATA</b> structure	Obtains information about a connected SPX socket.
<b>IPX_ADDRESS_NOTIFY</b>	<b>IPX_ADDRESS_DATA</b> structure	Obtains status notification when changes occur on an adapter to which IPX is bound.
<b>IPX_MAX_ADAPTER_NUM</b>	int	Obtains maximum number of adapters present, numbered as base zero.
<b>IPX_RERIPNETNUMBER</b>	<b>IPX_NETNUM_DATA</b> structure	Similar to <b>IPX_GETNETINFO</b> , but forces IPX to use RIP for resolution, even if the network information is in the local cache.
<b>IPX_IMMEDIATESPXACK</b>	BOOL	Directs SPX connections not to delay before sending an ACK. Applications without back-and-forth traffic should set this to TRUE to increase performance.

BSD options not supported for **getsockopt** are as follows.

Value	Type	Meaning
<b>SO_RCVLOWAT</b>	int	Receives low watermark.
<b>SO_RCVTIMEO</b>	int	Receives time-out.
<b>SO_SNDLOWAT</b>	int	Sends low watermark.
<b>SO_SNDTIMEO</b>	int	Sends time-out.
<b>TCP_MAXSEG</b>	int	Receives TCP maximum-segment size.

Calling **getsockopt** with an unsupported option will result in an error code of WSAENOPROTOOPT being returned from **WSAGetLastError**.

### **SO\_DEBUG**

Windows Sockets service providers are encouraged (but not required) to supply output debug information if the **SO\_DEBUG** option is set by an application. The mechanism for generating the debug information and the form it takes are beyond the scope of this document.

### SO\_ERROR

The **SO\_ERROR** option returns and resets the per socket-based error code, which is different from the per thread based-error code that is handled using the **WSAGetLastError** and **WSASetLastError** function calls. A successful call using the socket does not reset the socket based error code returned by the **SO\_ERROR** option.

### SO\_GROUP\_ID

This option is reserved. This option is also exclusive to **getsockopt**; the value should be NULL.

### SO\_GROUP\_PRIORITY

This option is reserved.

### SO\_KEEPAIVE

An application can request that a TCP/IP service provider enable the use of keep-alive packets on TCP connections by turning on the **SO\_KEEPAIVE** socket option. A Windows Sockets provider need not support the use of keep-alive: if it does, the precise semantics are implementation-specific but should conform to section 4.2.3.6 of RFC 1122: *Requirements for Internet Hosts—Communication Layers*. If a connection is dropped as the result of keep-alives the error code WSAENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with WSAENOTCONN. **SO\_KEEPAIVE** is not supported on ATM sockets, and requests to enable the use of keep-alive packets on an ATM socket results in an error being returned by the socket.

### SO\_LINGER

**SO\_LINGER** controls the action taken when unsent data is queued on a socket and a **closesocket** is performed. See **closesocket** for a description of the way in which the **SO\_LINGER** settings affect the semantics of **closesocket**. The application gets the current behavior by retrieving a **LINGER** structure (pointed to by the *optval* parameter).

### SO\_MAX\_MSG\_SIZE

This is a get-only socket option that indicates the maximum outbound (send) size of a message for message-oriented socket types (for example, **SOCK\_DGRAM**) as implemented by a particular service provider. It has no meaning for byte stream oriented sockets. There is no provision to find out the maximum inbound-message size

### SO\_PROTOCOL\_INFO

This is a get-only option that supplies the **WSAPROTOCOL\_INFO** structure associated with this socket. See **WSAEnumProtocols** for more information about this structure.

### SO\_SNDBUF

When a Windows Sockets implementation supports the **SO\_RCVBUF** and **SO\_SNDBUF** options, an application can request different buffer sizes (larger or smaller). The call to **setsockopt** can succeed even if the implementation did not provide the whole amount requested. An application must call this function with the same option to check the buffer size actually provided.

### SO\_REUSEADDR

By default, a socket cannot be bound (see *bind*) to a local address that is already in use. On occasion, however, it can be necessary to reuse an address in this way. Because every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the Windows Sockets provider that a **bind** on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the **SO\_REUSEADDR** socket option for the socket before issuing the **bind**. Note that the option is interpreted only at the time of the **bind**: it is therefore unnecessary (but harmless) to set the option on a socket that is not to be bound to an existing address, and setting or resetting the option after the **bind** has no effect on this or any other socket. **SO\_REUSEADDR** is not applicable for ATM sockets, and although requests to reuse an address do not result in an error, they have no effect on when an ATM socket is in use.

### PVD\_CONFIG

This option retrieves an opaque data structure object from the service provider associated with socket *s*. This object stores the current configuration information of the service provider. The exact format of this data structure is service provider specific.

### TCP\_NODELAY

The **TCP\_NODELAY** option is specific to TCP/IP service providers. The Nagle algorithm is disabled if the **TCP\_NODELAY** option is enabled (and vice versa). The Nagle algorithm (described in RFC 896) is very effective in reducing the number of small packets sent by a host. The process involves buffering send data when there is unacknowledged data already in flight or buffering send data until a full-size packet can be sent. It is highly recommended that Windows Sockets implementations enable the Nagle Algorithm by default because, for the vast majority of application protocols, the Nagle Algorithm can deliver significant performance enhancements. However, for some applications this algorithm can impede performance, and **setsockopt** with the same option can be used to turn it off. These are applications where many small messages are sent, and the time delays between the messages are maintained.

### Notes for IrDA Sockets

- The `Af_irda.h` header file must be explicitly included.
- Windows CE does not support the `WSAENETDOWN` return value. Windows NT/Windows 2000 will return `WSAENETDOWN` to indicate the underlying transceiver driver failed to initialize with the IrDA protocol stack.
- IrDA supports several special socket options:

Value	Type	Meaning
<code>IRLMP_ENUMDEVICES</code>	<code>*DEVICELIST</code>	Describes devices in range.
<code>IRLMP_IAS_QUERY</code>	<code>*IAS_QUERY</code>	Retrieve IAS attributes.

Before an IrDA socket connection can be initiated, a device address must be obtained by performing a **getsockopt**(,,IRLMP\_ENUMDEVICES,,) function call, which returns a list of all available IrDA devices. A device address returned from the function call is copied into a **SOCKADDR\_IRDA** structure, which in turn is used by a subsequent call to the **connect** function call.

Discovery can be performed in two ways:

First, performing a **getsockopt** function call with the **IRLMP\_ENUMDEVICES** option causes a single discovery to be run on each idle adapter. The list of discovered devices and cached devices (on active adapters) is returned immediately. The following code demonstrates this approach.

```
SOCKADDR_IRDA DestSockAddr = { AF_IRDA, 0, 0, 0, 0, "SampleIrDAService" };

#define DEVICE_LIST_LEN    10

unsigned char    DevListBuff[sizeof(DEVICELIST) -
                          sizeof(IRDA_DEVICE_INFO) +
                          (sizeof(IRDA_DEVICE_INFO) * DEVICE_LIST_LEN)];
int             DevListLen    = sizeof(DevListBuff);
PDEVICELIST     pDevList      = (PDEVICELIST) &DevListBuff;

pDevList->numDevice = 0;

// Sock is not in connected state
if (getsockopt(Sock, SOL_IRLMP, IRLMP_ENUMDEVICES,
              (char *) pDevList, &DevListLen) == SOCKET_ERROR)
{
    // WSAGetLastError
}

if (pDevList->numDevice == 0)
{
    // no devices discovered or cached
    // not a bad idea to run a couple of times
}
else
{
    // one per discovered device
    for (i = 0; i < (int) pDevList->numDevice; i++)
    {
        // typedef struct _IRDA_DEVICE_INFO
        // {
        //     u_char    irdaDeviceID[4];
        //     char      irdaDeviceName[22];
    }
```

(continued)



*(continued)*

```

        //      u_char      irdaDeviceHints1;
        //      u_char      irdaDeviceHints2;
        //      u_char      irdaCharSet;
        // } _IRDA_DEVICE_INFO;

        // pDevList->Device[i]. see _IRDA_DEVICE_INFO for fields
        // display the device names and let the user select one
    }
}

// assume the user selected the first device [0]
memcpy(&DestSockAddr.irdaDeviceID[0], &pDevList->Device[0].irdaDeviceID[0], 4);

if (connect(Sock, (const struct sockaddr *) &DestSockAddr,
           sizeof(SOCKADDR_IRDA)) == SOCKET_ERROR)
{
    // WSAGetLastError
}

```

The second approach to performing discovery of IrDA device addresses is to perform a lazy discovery; in this approach, the application is not notified until the discovered devices list changes from the last discovery run by the stack.

The **DEVICELIST** structure shown in the Type column in the previous table is an extendible array of device descriptions. IrDA fills in as many device descriptions as can fit in the supplied buffer. The device description consists of a device identifier necessary to form a `sockaddr_irda` structure, and a displayable string describing the device.

The **IAS\_QUERY** structure shown in the Type column in the previous table is used to retrieve a single attribute of a single class from a peer device's IAS database. The application specifies the device and class to query and the attribute and attribute type. Note that the device would have been obtained previously by a call to **getsockopt**(`IRLMP_ENUMDEVICES`). It is expected that the application allocates a buffer, of the necessary size, for the returned parameters.

Many **SO** level socket options are not meaningful to IrDA; only **SO\_LINGER** and **SO\_DONTLINGER** are specifically supported.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

**+** See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **setsockopt**, **socket**, **WSAAsyncSelect**, **WSAConnect**, **WSAGetLastError**, **WSASetLastError**

## GetTypeByName

**Important** The **GetTypeByName** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is *obsolete*. For the convenience of Windows Sockets 1.1 developers, the reference material is as follows.

The functions detailed in *Protocol-Independent Name Resolution* provide equivalent functionality in Windows Sockets 2.

The **GetTypeByName** function obtains a service type **GUID** for a network service specified by name.

```
INT GetTypeByName (
    LPTSTR    lpServiceName, // points to the name of the
                          // network service
    PGUID     lpServiceType // points to a variable to receive
                          // network service type
);
```

### Parameters

#### *lpServiceName*

Pointer to a zero-terminated string that uniquely represents the name of the service. For example, "MY SNA SERVER".

#### *lpServiceType*

Pointer to a variable to receive a Globally Unique Identifier (**GUID**) that specifies the type of the network service. The header file `Svcguid.h` includes definitions of several **GUID** service types and macros for working with them.

### Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is `SOCKET_ERROR(-1)`. To get extended error information, call **GetLastError**. **GetLastError** can return the following extended error value.

Value	Meaning
ERROR_SERVICE_DOES_NOT_EXIST	The specified service type is unknown.

**!** Requirements

**Version:** Requires Windows Sockets 1.1. A Microsoft-specific extension. Obsolete for Windows Sockets 2.0.

**Header:** Declared in Nspapi.h.

**Library:** Use Wsock32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT/2000.

**+** See Also

**GetNameByType**

---

## htonl

The Windows Sockets **htonl** function converts a **u\_long** from host to TCP/IP network byte order (which is big-endian).

```
u_long htonl (  
    u_long  hostlong  
);
```

### Parameters

*hostlong*

[in] 32-bit number in host byte order.

### Return Values

The **htonl** function returns the value in TCP/IP's network byte order.

### Remarks

The **htonl** function takes a 32-bit number in host byte order and returns a 32-bit number in the network byte order used in TCP/IP networks.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+ See Also**

Windows Sockets Programming Considerations Overview, Socket Functions, **htons**, **ntohl**, **ntohs**, **WSAhtoni**, **WSAhtons**, **WSANTohl**, **WSANTohs**

## htons

The Windows Sockets **htons** function converts a **u\_short** from host to TCP/IP network byte order (which is big-endian).

```
u_short htons (  
    u_short  hostshort  
);
```

### Parameters

*hostshort*

[in] 16-bit number in host byte order.

### Remarks

The **htons** function takes a 16-bit number in host byte order and returns a 16-bit number in network byte order used in TCP/IP networks.

### Return Values

The **htons** function returns the value in TCP/IP network byte order.

**! Requirements**

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+ See Also**

Windows Sockets Programming Considerations Overview, Socket Functions, **htoni**, **ntohl**, **ntohs**, **WSAhtoni**, **WSAhtons**, **WSANTohl**, **WSANTohs**

## inet\_addr

The Windows Sockets **inet\_addr** function converts a string containing an (IPv4) Internet Protocol dotted address into a proper address for the **IN\_ADDR** structure.

```
unsigned long inet_addr (  
    const char FAR *cp  
);
```

## Parameters

*cp*

[in] Null-terminated character string representing a number expressed in the Internet standard "." (dotted) notation.

## Return Values

If no error occurs, **inet\_addr** returns an unsigned long value containing a suitable binary representation of the Internet address given. If the string in the *cp* parameter does not contain a legitimate Internet address, for example if a portion of an "a.b.c.d" address exceeds 255, then **inet\_addr** returns the value `INADDR_NONE`.

## Remarks

The **inet\_addr** function interprets the character string specified by the *cp* parameter. This string represents a numeric Internet address expressed in the Internet standard "." notation. The value returned is a number suitable for use as an Internet address. All Internet addresses are returned in IP's network order (bytes ordered from left to right). If you pass in " " (a space) to the **inet\_addr** function, **inet\_addr** returns zero.

## Internet Addresses

Values specified using the "." notation take one of the following forms:

a.b.c.d a.b.c a.b a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the 4 bytes of an Internet address. When an Internet address is viewed as a 32-bit integer quantity on the Intel architecture, the bytes referred to above appear as "d.c.b.a". That is, the bytes on an Intel processor are ordered from right to left.

The parts that make up an address in "." notation can be decimal, octal or hexadecimal as specified in the C language. Numbers that start with "0x" or "0X" imply hexadecimal. Numbers that start with "0" imply octal. All other numbers are interpreted as decimal.

Internet address value	Meaning
"4.3.2.16"	Decimal
"004.003.002.020"	Octal
"0x4.0x3.0x2.0x10"	Hexadecimal
"4.003.002.0x10"	Mix

---

**Note** The following notations are only used by Berkeley, and nowhere else on the Internet. For compatibility with their software, they are supported as specified.

---

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most 2 bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as "128.net.host".

When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the right-most 3 bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as “net.host”.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

### + See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **inet\_ntoa**

---

## inet\_ntoa

The Windows Sockets **inet\_ntoa** function converts an (IPv4) Internet network address into a string in Internet standard dotted format.

```
char FAR * inet_ntoa (  
    struct in_addr in  
);
```

### Parameters

*in*

[in] Structure that represents an Internet host address.

### Return Values

If no error occurs, **inet\_ntoa** returns a character pointer to a static buffer containing the text address in standard “.” notation. Otherwise, it returns NULL.

### Remarks

The **inet\_ntoa** function takes an Internet address structure specified by the *in* parameter and returns an ASCII string representing the address in “.” (dot) notation as in “a.b.c.d.” The string returned by **inet\_ntoa** resides in memory that is allocated by Windows Sockets. The application should not make any assumptions about the way in which the memory is allocated. The data is guaranteed to be valid until the next Windows Sockets function call within the same thread—but no longer. Therefore, the data should be copied before another Windows Sockets call is made.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **inet\_addr**

---

## ioctlsocket

The Windows Sockets **ioctlsocket** function controls the I/O mode of a socket.

```
int ioctlsocket (
    SOCKET      s,
    long        cmd,
    u_long FAR  *argp
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*cmd*

[in] Command to perform on the socket *s*.

*argp*

[in/out] Pointer to a parameter for *cmd*.

### Return Values

Upon successful completion, the **ioctlsocket** returns zero. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor <i>s</i> is not a socket.
WSAEFAULT	The <i>argp</i> parameter is not a valid part of the user address space.

## Remarks

The **ioctlsocket** function can be used on any socket in any state. It is used to set or retrieve operating parameters associated with the socket, independent of the protocol and communications subsystem. Here are the supported commands to use in the *cmd* parameter and their semantics:

### FIONBIO

Use with a nonzero *argp* parameter to enable the nonblocking mode of socket *s*. The *argp* parameter is zero if nonblocking is to be disabled. The *argp* parameter points to an unsigned long value. When a socket is created, it operates in blocking mode by default (nonblocking mode is disabled). This is consistent with BSD sockets.

The **WSAAsyncSelect** and **WSAEventSelect** functions automatically set a socket to nonblocking mode. If **WSAAsyncSelect** or **WSAEventSelect** has been issued on a socket, then any attempt to use **ioctlsocket** to set the socket back to blocking mode will fail with **WSAEINVAL**.

To set the socket back to blocking mode, an application must first disable **WSAAsyncSelect** by calling **WSAAsyncSelect** with the *lEvent* parameter equal to zero, or disable **WSAEventSelect** by calling **WSAEventSelect** with the *lNetworkEvents* parameter equal to zero.

### FIONREAD

Use to determine the amount of data pending in the network's input buffer that can be read from socket *s*. The *argp* parameter points to an unsigned long value in which **ioctlsocket** stores the result. If *s* is stream oriented (for example, type **SOCK\_STREAM**), **FIONREAD** returns the amount of data that can be read in a single call to the **recv** function; this might not be the same as the total amount of data queued on the socket. If *s* is message oriented (for example, type **SOCK\_DGRAM**), **FIONREAD** returns the size of the first datagram (message) queued on the socket.

### SIOCATMARK

Use to determine whether or not all OOB data has been read. (See section Windows Sockets 1.1 Blocking Routines and EINPROGRESS for a discussion on out of band (OOB) data.) This applies only to a stream oriented socket (for example, type **SOCK\_STREAM**) that has been configured for in-line reception of any OOB data (**SO\_OOBINLINE**). If no OOB data is waiting to be read, the operation returns **TRUE**. Otherwise, it returns **FALSE**, and the next **recv** or **recvfrom** performed on the socket will retrieve some or all of the data preceding the mark. The application should use the **SIOCATMARK** operation to determine whether any data remains. If there is any normal data preceding the urgent (out of band) data, it will be received in order. (A **recv** or **recvfrom** will never mix OOB and normal data in the same call.) The *argp* parameter points to an unsigned long value in which **ioctlsocket** stores the Boolean result.



## Compatibility

This **ioctlsocket** function performs only a subset of functions on a socket when compared to the **ioctl** function found in Berkeley sockets. The **ioctlsocket** function has no command parameter equivalent to the **FIOASYNC** of **ioctl**, and **SIOCATMARK** is the only socket-level command that is supported by **ioctlsocket**.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

### + See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **getsockopt**, **setsockopt**, **socket**, **WSAAsyncSelect**, **WSAEventSelect**, **WSAIoctl**

---

# listen

The Windows Sockets **listen** function places a socket a state where it is listening for an incoming connection.

```
int listen (
    SOCKET    s,
    int      backlog
);
```

## Parameters

**s**

[in] Descriptor identifying a bound, unconnected socket.

**backlog**

[in] Maximum length of the queue of pending connections. If set to `SOMAXCONN`, the underlying service provider responsible for socket *s* will set the backlog to a maximum reasonable value. There is no standard provision to obtain the actual backlog value.

## Return Values

If no error occurs, **listen** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

### Error code

### Meaning

`WSANOTINITIALISED`

A successful **WSAStartup** call must occur before using this function.

`WSAENETDOWN`

The network subsystem has failed.

Error code	Meaning
WSAEADDRINUSE	The socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs during execution of the <b>bind</b> function, but could be delayed until this function if the <b>bind</b> was to a partially wildcard address (involving ADDR_ANY) and if a specific address needs to be committed at the time of this function.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The socket has not been bound with <b>bind</b> .
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The referenced socket is not of a type that supports the <b>listen</b> operation.

### Remarks

To accept connections, a socket is first created with the **socket** function and bound to a local address with the **bind** function, a backlog for incoming connections is specified with **listen**, and then the connections are accepted with the **accept** function. Sockets that are connection oriented, those of type **SOCK\_STREAM** for example, are used with **listen**. The socket *s* is put into passive mode where incoming connection requests are acknowledged and queued pending acceptance by the process.

The **listen** function is typically used by servers that can have more than one connection request at a time. If a connection request arrives and the queue is full, the client will receive an error with an indication of WSAECONNREFUSED.

If there are no available socket descriptors, **listen** attempts to continue to function. If descriptors become available, a later call to **listen** or **accept** will refill the queue to the current or most recent backlog, if possible, and resume listening for incoming connections.

An application can call **listen** more than once on the same socket. This has the effect of updating the current backlog for the listening socket. Should there be more pending connections than the new backlog value, the excess pending connections will be reset and dropped.

### Notes for IrDA Sockets

- The `Af_irda.h` header file must be explicitly included.
- **Windows CE only:** The *backlog* parameter is currently limited (silently) to 2. As in 4.3BSD, illegal values (less than 1 or greater than 5) are replaced by the nearest valid value.

### Compatibility

The *backlog* parameter is limited (silently) to a reasonable value as determined by the underlying service provider. Illegal values are replaced by the nearest legal value. There is no standard provision to find out the actual backlog value.

#### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

#### + See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **accept**, **connect**, **socket**

---

## ntohl

The Windows Sockets `ntohl` function converts a `u_long` from TCP/IP network order to host byte order (which is little-endian on Intel processors).

```
u_long ntohl (  
    u_long  netlong  
);
```

### Parameters

*netlong*

[in] 32-bit number in TCP/IP network byte order.

### Return Values

The `ntohl` function always returns a value in host byte order. If the *netlong* parameter was already in host byte order, then no operation is performed.

### Remarks

The `ntohl` function takes a 32-bit number in TCP/IP network byte order and returns a 32-bit number in host byte order.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **htonl**, **htons**, **ntohs**, **WSAhtonl**, **WSAhtons**, **WSANTohl**, **WSANTohs**

## ntohs

The Windows Sockets **ntohs** function converts a **u\_short** from TCP/IP network byte order to host byte order (which is little-endian on Intel processors).

```
u_short ntohs (  
    u_short  netshort  
);
```

**Parameters**

*netshort*

[in] 16-bit number in TCP/IP network byte order.

**Return Values**

The **ntohs** function returns the value in host byte order. If the *netshort* parameter was already in host byte order, then no operation is performed.

**Remarks**

The **ntohs** function takes a 16-bit number in TCP/IP network byte order and returns a 16-bit number in host byte order.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **htonl**, **htons**, **ntohl**, **WSAhtonl**, **WSAhtons**, **WSANTohl**, **WSANTohs**

## recv

The Windows Sockets **recv** function receives data from a connected socket.

```
int recv (
    SOCKET      s,
    char FAR    *buf,
    int         len,
    int         flags
);
```

### Parameters

*s*

[in] Descriptor identifying a connected socket.

*buf*

[out] Buffer for the incoming data.

*len*

[in] Length of *buf*.

*flags*

[in] Flag specifying the way in which the call is made.

### Return Values

If no error occurs, **recv** returns the number of bytes received. If the connection has been gracefully closed, the return value is zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>buf</i> parameter is not completely contained in a valid part of the user address space.
WSAENOTCONN	The socket is not connected.
WSAEINTR	The (blocking) call was canceled through <b>WSACancelBlockingCall</b> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENETRESET	The connection has been broken due to the <i>keep-alive</i> activity detecting a failure while the operation was in progress.
WSAENOTSOCK	The descriptor is not a socket.

Error code	Meaning
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to receive on a socket after <b>shutdown</b> has been invoked with <b>how</b> set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the receive operation would block.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAEINVAL	The socket has not been bound with <b>bind</b> , or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled or (for byte stream sockets only) <i>len</i> was zero or negative.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
WSAETIMEDOUT	The connection has been dropped because of a network failure or because the peer system failed to respond.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer usable. On a UDP-datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.

### Remarks

The **recv** function is used to read incoming data on connection-oriented sockets, or connectionless sockets. When using a connection-oriented protocol, the sockets must be connected before calling **recv**. When using a connectionless protocol, the sockets must be bound before calling **recv**.

The local address of the socket must be known. For server applications, use an explicit **bind** function or an implicit **accept** or **WSAAccept** function. Explicit binding is discouraged for client applications. For client applications, the socket can become bound implicitly to a local address using **connect**, **WSAConnect**, **sendto**, **WSASendTo**, or **WSAJoinLeaf**.

For connected or connectionless sockets, the **recv** function restricts the addresses from which received messages are accepted. The function only returns messages from the remote address specified in the connection. Messages from other addresses are (silently) discarded.

For connection-oriented sockets (type **SOCK\_STREAM** for example), calling **recv** will return as much information as is currently available—up to the size of the buffer supplied. If the socket has been configured for in-line reception of OOB data (socket option **SO\_OOBLINE**) and OOB data is yet unread, only OOB data will be returned. The application can use the **ioctlsocket** or **WSAIoctl SIOCATMARK** command to determine whether any more OOB data remains to be read.

For connectionless sockets (type **SOCK\_DGRAM** or other message-oriented sockets), data is extracted from the first enqueued datagram (message) from the destination address specified by the **connect** function.

If the datagram or message is larger than the buffer supplied, the buffer is filled with the first part of the datagram, and **recv** generates the error **WSAEMSGSIZE**. For unreliable protocols (for example, UDP) the excess data is lost; for reliable protocols, the data is retained by the service provider until it is successfully read by calling **recv** with a large enough buffer.

If no incoming data is available at the socket, the **recv** call blocks and waits for data to arrive according to the blocking rules defined for **WSARecv** with the **MSG\_PARTIAL** flag not set unless the socket is nonblocking. In this case, a value of **SOCKET\_ERROR** is returned with the error code set to **WSAEWOULDBLOCK**. The **select**, **WSAAsyncSelect**, or **WSAEventSelect** functions can be used to determine when more data arrives.

If the socket is connection oriented and the remote side has shut down the connection gracefully, and all data has been received, a **recv** will complete immediately with zero bytes received. If the connection has been reset, a **recv** will fail with the error **WSAECONNRESET**.

The *flags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

Value	Meaning
<b>MSG_PEEK</b>	Peeks at the incoming data. The data is copied into the buffer but is not removed from the input queue. The function then returns the number of bytes currently pending to receive.
<b>MSG_OOB</b>	Processes OOB data. (See section <i>DECnet Out-of-band data</i> for a discussion of this topic.)

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **recvfrom**, **select**, **send**, **socket**, **WSAAsyncSelect**, **WSARecvEx**

## recvfrom

The Windows Sockets **recvfrom** function receives a datagram and stores the source address.

```
int recvfrom (
    SOCKET          s,
    char FAR*      buf,
    int            len,
    int            flags,
    struct sockaddr FAR *from,
    int FAR        *fromlen
);
```

### Parameters

*s*

[in] Descriptor identifying a bound socket.

*buf*

[out] Buffer for the incoming data.

*len*

[in] Length of *buf*.

*flags*

[in] Indicator specifying the way in which the call is made.

*from*

[out] Optional pointer to a buffer that will hold the source address upon return.

*fromlen*

[in/out] Optional pointer to the size of the *from* buffer.

### Return Values

If no error occurs, **recvfrom** returns the number of bytes received. If the connection has been gracefully closed, the return value is zero. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.



Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>buf</i> or <i>from</i> parameters are not part of the user address space, or the <i>fromlen</i> parameter is too small to accommodate the peer address.
WSAEINTR	The (blocking) call was canceled through <b>WSACancelBlockingCall</b> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The socket has not been bound with <b>bind</b> , or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled, or (for byte stream-style sockets only) <i>len</i> was zero or negative.
WSAEISCONN	The socket is connected. This function is not permitted with a connected socket, whether the socket is connection-oriented or connectionless.
WSAENETRESET	The connection has been broken due to the keep-alive activity detecting a failure while the operation was in progress.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to <b>recvfrom</b> on a socket after <b>shutdown</b> has been invoked with <i>how</i> set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the <b>recvfrom</b> operation would block.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer usable. On a UPD-datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.

## Remarks

The **recvfrom** function reads incoming data on both connected and unconnected sockets and captures the address from which the data was sent. The socket must not be connected. The local address of the socket must be known. For server applications, this is usually done explicitly through **bind**. Explicit binding is discouraged for client applications. For client applications using this function, the socket can become bound implicitly to a local address through **sendto**, **WSASendTo**, or **WSAJoinLeaf**.

For stream-oriented sockets such as those of type **SOCK\_STREAM**, a call to **recvfrom** returns as much information as is currently available—up to the size of the buffer supplied. If the socket has been configured for inline reception of OOB data (socket option **SO\_OOBINLINE**) and OOB data is yet unread, only OOB data will be returned. The application can use the **ioctlsocket** or **WSAIoctl SIOCATMARK** command to determine whether any more OOB data remains to be read. The *from* and *fromlen* parameters are ignored for connection-oriented sockets.

For message-oriented sockets, data is extracted from the first enqueued message, up to the size of the buffer supplied. If the datagram or message is larger than the buffer supplied, the buffer is filled with the first part of the datagram, and **recvfrom** generates the error **WSAEMSGSIZE**. For unreliable protocols (for example, UDP) the excess data is lost.

If the *from* parameter is nonzero and the socket is not connection oriented, (type **SOCK\_DGRAM** for example), the network address of the peer that sent the data is copied to the corresponding **SOCKADDR** structure. The value pointed to by *fromlen* is initialized to the size of this structure and is modified, on return, to indicate the actual size of the address stored in the **SOCKADDR** structure.

If no incoming data is available at the socket, the **recvfrom** function blocks and waits for data to arrive according to the blocking rules defined for **WSARecv** with the **MSG\_PARTIAL** flag not set unless the socket is nonblocking. In this case, a value of **SOCKET\_ERROR** is returned with the error code set to **WSAEWOULDBLOCK**. The **select**, **WSAAsyncSelect**, or **WSAEventSelect** can be used to determine when more data arrives.

If the socket is connection oriented and the remote side has shut down the connection gracefully, the call to **recvfrom** will complete immediately with zero bytes received. If the connection has been reset **recvfrom** will fail with the error **WSAECONNRESET**.

The *flags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

Value	Meaning
MSG_PEEK	Peeks at the incoming data. The data is copied into the buffer but is not removed from the input queue, and the function returns the number of bytes currently pending to receive.
MSG_OOB	Processes OOB data. (See section <i>DECnet Out-Of-band data</i> for a discussion of this topic.)

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

### + See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **recv**, **send**, **socket**, **WSAAsyncSelect**, **WSAEventSelect**

## select

The Windows Sockets **select** function determines the status of one or more sockets, waiting if necessary, to perform synchronous I/O.

```
int select (
    int                nfds,
    fd_set FAR        *readfds,
    fd_set FAR        *writefds,
    fd_set FAR        *exceptfds,
    const struct timeval FAR *timeout
);
```

### Parameters

*nfds*

[in] Ignored. The *nfds* parameter is included only for compatibility with Berkeley sockets.

*readfds*

[in/out] Optional pointer to a set of sockets to be checked for readability.

*writefds*

[in/out] Optional pointer to a set of sockets to be checked for writability.

*exceptfds*

[in/out] Optional pointer to a set of sockets to be checked for errors.

*timeout*

[in] Maximum time for **select** to wait, provided in the form of a **TIMEVAL** structure. Set the *timeout* parameter to NULL for blocking operation.

## Return Values

The **select** function returns the total number of socket handles that are ready and contained in the **fd\_set** structures, zero if the time limit expired, or **SOCKET\_ERROR** if an error occurred. If the return value is **SOCKET\_ERROR**, **WSAGetLastError** can be used to retrieve a specific error code.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAEFAULT	The Windows Sockets implementation was unable to allocate needed resources for its internal operations, or the <i>readfds</i> , <i>writefds</i> , <i>exceptfds</i> , or <i>timeval</i> parameters are not part of the user address space.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	The <i>time-out</i> value is not valid, or all three descriptor parameters were NULL.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	One of the descriptor sets contains an entry that is not a socket.

## Remarks

The **select** function is used to determine the status of one or more sockets. For each socket, the caller can request information on read, write, or error status. The set of sockets for which a given status is requested is indicated by an **fd\_set** structure. The sockets contained within the **fd\_set** structures must be associated with a single service provider. For the purpose of this restriction, sockets are considered to be from the same service provider if the **WSAPROTOCOL\_INFO** structures describing their protocols have the same *providerId* value. Upon return, the structures are updated to reflect the subset of these sockets that meet the specified condition. The **select** function returns the number of sockets meeting the conditions. A set of macros is provided for manipulating an **fd\_set** structure. These macros are compatible with those used in the Berkeley software, but the underlying representation is completely different.

The parameter *readfds* identifies the sockets that are to be checked for readability. If the socket is currently in the **listen** state, it will be marked as readable if an incoming connection request has been received such that an **accept** is guaranteed to complete

without blocking. For other sockets, readability means that queued data is available for reading such that a call to **recv**, **WSARecv**, **WSARecvFrom**, or **recvfrom** is guaranteed not to block.

For connection-oriented sockets, readability can also indicate that a request to close the socket has been received from the peer. If the virtual circuit was closed gracefully, and all data was received, then a **recv** will return immediately with zero bytes read. If the virtual circuit was reset, then a **recv** will complete immediately with an error code such as **WSAECONNRESET**. The presence of OOB data will be checked if the socket option **SO\_OOBINLINE** has been enabled (see *setsockopt*).

The parameter *writfds* identifies the sockets that are to be checked for writability. If a socket is processing a **connect** call (nonblocking), a socket is writeable if the connection establishment successfully completes. If the socket is not processing a **connect** call, writability means a **send**, **sendto**, or **WSASendto** are guaranteed to succeed. However, they can block on a blocking socket if the *len* parameter exceeds the amount of outgoing system buffer space available. It is not specified how long these guarantees can be assumed to be valid, particularly in a multithreaded environment.

The parameter *exceptfds* identifies the sockets that are to be checked for the presence of OOB data (see section DECnet Out-of-band data for a discussion of this topic) or any exceptional error conditions.

---

**Important** OOB data will only be reported in this way if the option **SO\_OOBINLINE** is **FALSE**. If a socket is processing a **connect** call (nonblocking), failure of the connect attempt is indicated in *exceptfds* (application must then call **getsockopt** **SO\_ERROR** to determine the error value to describe why the failure occurred). This document does not define which other errors will be included.

---

Any two of the parameters, *readfds*, *writfds*, or *exceptfds*, can be given as **NULL**. At least one must be non-**NULL**, and any non-**NULL** descriptor set must contain at least one handle to a socket.

Summary: A socket will be identified in a particular set when **select** returns if:

*readfds*:

- If **listen** has been called and a connection is pending, **accept** will succeed.
- Data is available for reading (includes OOB data if **SO\_OOBINLINE** is enabled).
- Connection has been closed/reset/terminated.

*writfds*:

- If processing a **connect** call (nonblocking), connection has succeeded.
- Data can be sent.

*exceptfds*:

- If processing a **connect** call (nonblocking), connection attempt failed.
- OOB data is available for reading (only if `SO_OOINLINE` is disabled).

Four macros are defined in the header file `Winsock2.h` for manipulating and checking the descriptor sets. The variable `FD_SETSIZE` determines the maximum number of descriptors in a set. (The default value of `FD_SETSIZE` is 64, which can be modified by defining `FD_SETSIZE` to another value before including `Winsock2.h`.) Internally, socket handles in an `fd_set` structure are not represented as bit flags as in Berkeley Unix. Their data representation is opaque. Use of these macros will maintain software portability between different socket environments. The macros to manipulate and check `fd_set` contents are:

**FD\_CLR**(*s*, \**set*)

Removes the descriptor *s* from *set*.

**FD\_ISSET**(*s*, \**set*)

Nonzero if *s* is a member of the *set*. Otherwise, zero.

**FD\_SET**(*s*, \**set*)

Adds descriptor *s* to *set*.

**FD\_ZERO**(\**set*)

Initializes the *set* to the NULL set.

The parameter *time-out* controls how long the **select** can take to complete. If *time-out* is a NULL pointer, **select** will block indefinitely until at least one descriptor meets the specified criteria. Otherwise, *time-out* points to a **TIMEVAL** structure that specifies the maximum time that **select** should wait before returning. When **select** returns, the contents of the **TIMEVAL** structure are not altered. If **TIMEVAL** is initialized to {0, 0}, **select** will return immediately; this is used to poll the state of the selected sockets. If **select** returns immediately, then the **select** call is considered nonblocking and the standard assumptions for nonblocking calls apply. For example, the blocking hook will *not* be called, and Windows Sockets will *not* yield.

---

**Note** The **select** function has no effect on the persistence of socket events registered with **WSAAsyncSelect** or **WSAEventSelect**.

---

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

## + See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **accept**, **connect**, **recv**, **recvfrom**, **send**, **WSAAsyncSelect**, **WSAEventSelect**, **TIMEVAL**

## send

The Windows Sockets **send** function sends data on a connected socket.

```
int send (
    SOCKET          s,
    const char FAR *buf,
    int             len,
    int             flags
);
```

### Parameters

*s*

[in] Descriptor identifying a connected socket.

*buf*

[in] Buffer containing the data to be transmitted.

*len*

[in] Length of the data in *buf*.

*flags*

[in] Indicator specifying the way in which the call is made.

### Return Values

If no error occurs, **send** returns the total number of bytes sent, which can be less than the number indicated by *len* for nonblocking sockets. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set. Call <b>setsockopt</b> with the <b>SO_BROADCAST</b> parameter to allow the use of the broadcast address.
WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through <b>WSACancelBlockingCall</b> .

---

<b>Error code</b>	<b>Meaning</b>
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>buf</i> parameter is not completely contained in a valid part of the user address space.
WSAENETRESET	The connection has been broken due to the keep-alive activity detecting a failure while the operation was in progress.
WSAENOBUFS	No buffer space is available.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to send on a socket after <b>shutdown</b> has been invoked with <b>how</b> set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the requested operation would block.
WSAEMSGSIZE	The socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEHOSTUNREACH	The remote host cannot be reached from this host at this time.
WSAEINVAL	The socket has not been bound with <b>bind</b> , or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. For UDP sockets, the remote host was unable to deliver a previously sent UDP datagram and responded with a "Port Unreachable" ICMP packet. The application should close the socket as it is no longer usable.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.



## Remarks

The **send** function is used to write outgoing data on a connected socket. For message-oriented sockets, care must be taken not to exceed the maximum packet size of the underlying provider, which can be obtained by using **getsockopt** to retrieve the value of socket option **SO\_MAX\_MSG\_SIZE**. If the data is too long to pass atomically through the underlying protocol, the error **WSAEMSGSIZE** is returned, and no data is transmitted.

The successful completion of a **send** does not indicate that the data was successfully delivered.

If no buffer space is available within the transport system to hold the data to be transmitted, **send** will block unless the socket has been placed in nonblocking mode. On nonblocking stream oriented sockets, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both client and server machines. The **select**, **WSAAsyncSelect** or **WSAEventSelect** functions can be used to determine when it is possible to send more data.

Calling **send** with a zero *len* parameter is permissible and will be treated by implementations as successful. In such cases, **send** will return zero as a valid value. For message-oriented sockets, a zero-length transport datagram is sent.

The *flags* parameter can be used to influence the behavior of the function beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Sends OOB data (stream-style socket such as <b>SOCK_STREAM</b> only. Also see <i>DECnet Out-Of-band data</i> for a discussion of this topic).

## Notes for IrDA Sockets

The `Af_irda.h` header file must be explicitly included.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

### **+** See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **recv**, **recvfrom**, **select**, **sendto**, **socket**, **WSAAsyncSelect**, **WSAEventSelect**

## sendto

The Windows Sockets **sendto** function sends data to a specific destination.

```
int sendto (
    SOCKET                s,
    const char FAR       *buf,
    int                  len,
    int                  flags,
    const struct sockaddr FAR *to,
    int                  tolen
);
```

### Parameters

*s*

[in] Descriptor identifying a (possibly connected) socket.

*buf*

[in] Buffer containing the data to be transmitted.

*len*

[in] Length of the data in *buf*.

*flags*

[in] Indicator specifying the way in which the call is made.

*to*

[in] Optional pointer to the address of the target socket.

*tolen*

[in] Size of the address in *to*.

### Return Values

If no error occurs, **sendto** returns the total number of bytes sent, which can be less than the number indicated by *len*. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.

(continued)

*(continued)*

Error code	Meaning
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set. Call <b>setsockopt</b> with the <b>SO_BROADCAST</b> parameter to allow the use of the broadcast address.
WSAEINVAL	An unknown flag was specified, or <b>MSG_OOB</b> was specified for a socket with <b>SO_OOBINLINE</b> enabled.
WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>buf</i> or <i>to</i> parameters are not part of the user address space, or the <i>toLen</i> parameter is too small.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOBUFS	No buffer space is available.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to send to on a socket after <b>shutdown</b> has been invoked with <b>how</b> set to <b>SD_SEND</b> or <b>SD_BOTH</b> .
WSAEWOULDBLOCK	The socket is marked as nonblocking and the requested operation would block.
WSAEMSGSIZE	The socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEHOSTUNREACH	The remote host cannot be reached from this host at this time.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. For UDP sockets, the remote host was unable to deliver a previously sent UDP datagram and responded with a "Port Unreachable" ICMP packet. The application should close the socket as it is no longer usable.

Error code	Meaning
WSAEADDRNOTAVAIL	The remote address is not a valid address, for example, ADDR_ANY.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDRREQ	A destination address is required.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.

### Remarks

The **sendto** function is used to write outgoing data on a socket. For message-oriented sockets, care must be taken not to exceed the maximum packet size of the underlying subnets, which can be obtained by using **getsockopt** to retrieve the value of socket option **SO\_MAX\_MSG\_SIZE**. If the data is too long to pass atomically through the underlying protocol, the error **WSAEMSGSIZE** is returned and no data is transmitted.

The *to* parameter can be any valid address in the socket's address family, including a broadcast or any multicast address. To send to a broadcast address, an application must have used **setsockopt** with **SO\_BROADCAST** enabled. Otherwise, **sendto** will fail with the error code **WSAEACCES**. For TCP/IP, an application can send to any multicast address.

If the socket is unbound, unique values are assigned to the local association by the system, and the socket is then marked as bound. An application can use **getsockname** to determine the local socket name in this case.

The successful completion of a **sendto** does not indicate that the data was successfully delivered.

The **sendto** function is normally used on a connectionless socket to send a datagram to a specific peer socket identified by the *to* parameter. Even if the connectionless socket has been previously connected to a specific address, the *to* parameter overrides the destination address for that particular datagram only. On a connection-oriented socket, the *to* and *toLen* parameters are ignored, making **sendto** equivalent to **send**.

### For Sockets Using IP (Version 4)

To send a broadcast (on a **SOCK\_DGRAM** only), the address in the *to* parameter should be constructed using the special IP address **INADDR\_BROADCAST** (defined in **Winsock2.h**), together with the intended port number. It is generally inadvisable for a broadcast datagram to exceed the size at which fragmentation can occur, which implies that the data portion of the datagram (excluding headers) should not exceed 512 bytes.

If no buffer space is available within the transport system to hold the data to be transmitted, **sendto** will block unless the socket has been placed in a nonblocking mode. On nonblocking, stream oriented sockets, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both the client and server systems. The **select**, **WSAAsyncSelect** or **WSAEventSelect** function can be used to determine when it is possible to send more data.

Calling **sendto** with a *len* of zero is permissible and will return zero as a valid value. For message-oriented sockets, a zero-length transport datagram is sent.

The *flags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Sends OOB data (stream-style socket such as <b>SOCK_STREAM</b> only. Also see DECnet Out-Of-band data for a discussion of this topic.)

#### Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

#### See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **recv**, **recvfrom**, **select**, **send**, **socket**, **WSAAsyncSelect**, **WSAEventSelect**

---

## SetService

The **SetService** function registers or removes from the registry a network service within one or more name spaces. The function can also add or remove a network service type within one or more name spaces.

**Important** The **SetService** function is *obsolete*. For the convenience of Windows Sockets 1.1 developers, the reference material is as follows.

The functions detailed in *Protocol-Independent Name Resolution* provide equivalent functionality in Windows Sockets 2.

---

```

INT SetService (
    DWORD dwNameSpace, // specifies name space(s) to
                       // operate within
    DWORD dwOperation, // specifies operation to perform
    DWORD dwFlags,     // set of bit flags that modify
                       // function operation
    LPSERVICE_INFO lpServiceInfo,
                       // points to structure containing
                       // service information
    LPSERVICE_ASYNC_INFO lpServiceAsyncInfo,
                       // reserved for future use,
                       // must be NULL
    LPDWORD lpdwStatusFlags
                       // points to set of status bit flags
);

```

## Parameters

### *dwNameSpace*

Name space, or a set of default name spaces, within which the function will operate.

Use one of the following constants to specify a name space.

Value	Name space
NS_DEFAULT	A set of default name spaces. The function queries each name space within this set. The set of default name spaces typically includes all the name spaces installed on the system. System administrators, however, can exclude particular name spaces from the set. NS_DEFAULT is the value that most applications should use for <i>dwNameSpace</i> .
NS_DNS	The Domain Name System used in the Internet to resolve the name of the host.
NS_NDS	The NetWare 4 provider.
NS_NETBT	The NetBIOS over TCP/IP layer. All Windows NT/Windows 2000 and Windows 95 systems register their computer names with NetBIOS. This name space is used to convert a computer name to an IP address that uses this registration.
NS_SAP	The NetWare Service Advertising Protocol. This can access the Netware bindery, if appropriate. NS_SAP is a dynamic name space that enables the registration of services.
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\hosts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and lookup value in the cache of host to IP address mappings.

*dwOperation*

Specifies the operation that the function will perform. Use one of the following values to specify an operation:

Value	Meaning
SERVICE_REGISTER	Register the network service with the name space. This operation can be used with the SERVICE_FLAG_DEFER and SERVICE_FLAG_HARD bit flags.
SERVICE_DEREGISTER	Remove from the registry the network service from the name space. This operation can be used with the SERVICE_FLAG_DEFER and SERVICE_FLAG_HARD bit flags.
SERVICE_FLUSH	Perform any operation that was called with the SERVICE_FLAG_DEFER bit flag set to one.
SERVICE_ADD_TYPE	Add a service type to the name space.  For this operation, use the <b>ServiceSpecificInfo</b> member of the <b>SERVICE_INFO</b> structure pointed to by <i>IpServiceInfo</i> to pass a <b>SERVICE_TYPE_INFO_ABS</b> structure. You must also set the <b>ServiceType</b> member of the <b>SERVICE_INFO</b> structure. Other <b>SERVICE_INFO</b> members are ignored.
SERVICE_DELETE_TYPE	Remove a service type, added by a previous call specifying the SERVICE_ADD_TYPE operation, from the name space.

*dwFlags*

Set of bit flags that modify the function's operation. You can set one or more of the following bit flags:

Value	Name space
SERVICE_FLAG_DEFER	This bit flag is valid only if the operation is SERVICE_REGISTER or SERVICE_DEREGISTER.  If this bit flag is one, and it is valid, the name-space provider should defer the registration or deregistration operation until a SERVICE_FLUSH operation is requested.
SERVICE_FLAG_HARD	This bit flag is valid only if the operation is SERVICE_REGISTER or SERVICE_DEREGISTER.  If this bit flag is one, and it is valid, the name-space provider updates any relevant persistent store information when the operation is performed.  For example: If the operation involves deregistration in a name space that uses a persistent store, the name-space provider would remove the relevant persistent store information.

*IpServiceInfo*

Pointer to a **SERVICE\_INFO** structure that contains information about the network service or service type.

*lpServiceAsyncInfo*

Reserved for future use. Must be set to NULL.

*lpdwStatusFlags*

Set of bit flags that receive function status information. The following bit flag is defined:

Value	Meaning
SET_SERVICE_PARTIAL_SUCCESS	One or more name-space providers were unable to successfully perform the requested operation.

**Return Values**

If the function fails, the return value is `SOCKET_ERROR`. To get extended error information, call **GetLastError**. **GetLastError** can return the following extended error value.

Error code	Meaning
<code>ERROR_ALREADY_REGISTERED</code>	The function tried to register a service that was already registered.

**! Requirements**

**Version:** Requires Windows Sockets 1.1. Not supported on Windows 95. Obsolete for Windows Sockets 2.0.

**Header:** Declared in `Nspapi.h`.

**Library:** Use `Wsock32.lib`.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT/2000.

**+ See Also**

`GetService`, `SERVICE_INFO`, `SERVICE_TYPE_INFO_ABS`

## setsockopt

The Windows Sockets **setsockopt** function sets a socket option.

```
int setsockopt (
    SOCKET          s,
    int             level,
    int             optname,
    const char FAR *optval,
    int             optlen
);
```



## Parameters

*s*

[in] Descriptor identifying a socket.

*level*

[in] Level at which the option is defined; the supported *levels* include SOL\_SOCKET and IPPROTO\_TCP. See the *Windows Sockets 2 Protocol-Specific Annex* (a separate document included with the Platform SDK) for more information on protocol-specific levels.

*optname*

[in] Socket option for which the value is to be set.

*optval*

[in] Pointer to the buffer in which the value for the requested option is supplied.

*optlen*

[in] Size of the *optval* buffer.

## Return Values

If no error occurs, **setsockopt** returns zero. Otherwise, a value of SOCKET\_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	<i>optval</i> is not in a valid part of the process address space or <i>optlen</i> parameter is too small.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	<i>level</i> is not valid, or the information in <i>optval</i> is not valid.
WSAENETRESET	Connection has timed out when SO_KEEPALIVE is set.
WSAENOPROTOPT	The option is unknown or unsupported for the specified provider or socket
WSAENOTCONN	Connection has been reset when SO_KEEPALIVE is set.
WSAENOTSOCK	The descriptor is not a socket.

## Remarks

The **setsockopt** function sets the current value for a socket option associated with a socket of any type, in any state. Although options can exist at multiple protocol levels, they are always present at the uppermost socket level. Options affect socket operations, such as whether expedited data (OOB data for example) is received in the normal data stream, and whether broadcast messages can be sent on the socket.

---

**Note** If the **setsockopt** function is called before the **bind** function, TCP/IP options will not be checked with TCP/IP until the bind occurs. In this case, the **setsockopt** function call will always succeed, but the **bind** function call *may* fail because of an early **setsockopt** failing.

---

There are two types of socket options: Boolean options that enable or disable a feature or behavior, and options that require an integer value or structure. To enable a Boolean option, *optval* points to a nonzero integer. To disable the option *optval* points to an integer equal to zero. The *optlen* parameter should be equal to **sizeof(int)** for Boolean options. For other options, *optval* points to the an integer or structure that contains the desired value for the option, and *optlen* is the length of the integer or structure.

The following options are supported for **setsockopt**. For default values of these options, see the description. The Type identifies the type of data addressed by *optval*.

*level* = SOL\_SOCKET

Value	Type	Meaning
SO_BROADCAST	BOOL	Allows transmission of broadcast messages on the socket.
SO_CONDITIONAL_ACCEPT	BOOL	Enables sockets to delay the acknowledgment of a connection until after the <b>WSAAccept</b> condition function is called.
SO_DEBUG	BOOL	Records debugging information.
SO_DONTLINGER	BOOL	Does not block close waiting for unsent data to be sent. Setting this option is equivalent to setting SO_LINGER with <i>l_onoff</i> set to zero.
SO_DONTROUTE	BOOL	Does not route: sends directly to interface. Not supported on ATM sockets (results in an error).
SO_GROUP_PRIORITY	int	Reserved.
SO_KEEPAIVE	BOOL	Sends keep-alives. Not supported on ATM sockets (results in an error).
SO_LINGER	structure <b>LINGER</b>	Lingers on close if unsent data is present.
SO_OOBLINE	BOOL	Receives OOB data in the normal data stream. (See section DECnet Out-Of-band data for a discussion of this topic.)
SO_RCVBUF	int	Specifies the total per-socket buffer space reserved for receives. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.

(continued)

*(continued)*

Value	Type	Meaning
SO_REUSEADDR	BOOL	Allows the socket to be bound to an address that is already in use. (See <b>bind</b> .) Not applicable on ATM sockets.
SO_EXCLUSIVEADDRUSE	BOOL	Enables a socket to be bound for exclusive access. Requires Windows NT 4.0 SP4 or Windows 2000.
SO_SNDBUF	int	Specifies the total per-socket buffer space reserved for sends. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.
PVD_CONFIG	Service Provider Dependent	This object stores the configuration information for the service provider associated with socket <i>s</i> . The exact format of this data structure is service provider specific.

*level = IPPROTO\_TCP*<sup>1</sup>

Value	Type	Meaning
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.

<sup>1</sup> Included for backward compatibility with Windows Sockets 1.1.

*level = NSPROTO\_IPX*

**Note** Windows 2000 and Windows NT support all IPX options. Windows 95 and Windows 98 support only the following:

IPX\_PTYPE  
 IPX\_FILTERPTYPE  
 IPX\_DSTTYPE  
 IPX\_RECVDHDR  
 IPX\_MAXSIZE (used with the **getsockopt** function)  
 IPX\_ADDRESS (used with the **getsockopt** function)

Value	Type	Meaning
IPX_PTYPE	int	Sets the IPX packet type.
IPX_FILTERPTYPE	int	Sets the receive filter packet type
IPX_STOPFILTERPTYPE	int	Stop filtering the filter type set with IPX_FILTERPTYPE
IPX_DSTTYPE	int	Set the value of the data stream field in the SPX header on every packet sent.

Value	Type	Meaning
<b>IPX_EXTENDED_ADDRESS</b>	BOOL	Set whether extended addressing is enabled.
<b>IPX_RECVHDR</b>	BOOL	Set whether the protocol header is sent up on all receive headers.
<b>IPX_RECEIVE_BROADCAST</b>	BOOL	Indicates broadcast packets are likely on the socket. Set to TRUE by default. Applications that do not use broadcasts should set this to FALSE for better system performance.
<b>IPX_IMMEDIATESPXACK</b>	BOOL	Directs SPX connections not to delay before sending an ACK. Applications without back-and-forth traffic should set this to TRUE to increase performance.

BSD options not supported for **setsockopt** are:

Value	Type	Meaning
<b>SO_ACCEPTCONN</b>	BOOL	Socket is listening.
<b>SO_RCVLOWAT</b>	int	Receives low watermark.
<b>SO_RCVTIMEO</b>	int	Receives time-out (available in Microsoft implementation of Windows Sockets 2).
<b>SO_SNDLOWAT</b>	int	Sends low watermark.
<b>SO_SNDTIMEO</b>	int	Sends time-out (available in Microsoft implementation of Windows Sockets 2).
<b>SO_TYPE</b>	int	Type of the socket.

### **SO\_CONDITIONAL\_ACCEPT**

Setting this socket option to TRUE delays the acknowledgment of a connection until after the **WSAAccept** condition function is called. If FALSE, the connection may be accepted before the condition function is called, but the connection will be disconnected if the condition function rejects the call. This option must be set before calling the **listen** function, otherwise **WSAEINVAL** is returned.

**SO\_CONDITIONAL\_ACCEPT** is only supported for TCP and ATM.

TCP sets **SO\_CONDITIONAL\_ACCEPT** to FALSE by default, and therefore by default the connection will be accepted before the **WSAAccept** condition function is called. When set to TRUE, the conditional decision must be made within the TCP connection timeout. **CF\_DEFER** connections are still subject to the timeout.

ATM sets **SO\_CONDITIONAL\_ACCEPT** to TRUE by default.

**SO\_DEBUG**

Windows Sockets service providers are encouraged (but not required) to supply output debug information if the **SO\_DEBUG** option is set by an application. The mechanism for generating the debug information and the form it takes are beyond the scope of this document.

**SO\_GROUP\_PRIORITY**

Reserved.

**SO\_KEEPAIVE**

An application can request that a TCP/IP provider enable the use of keep-alive packets on TCP connections by turning on the **SO\_KEEPAIVE** socket option. A Windows Sockets provider need not support the use of keep-alives. If it does, the precise semantics are implementation-specific but should conform to section 4.2.3.6 of RFC 1122: *Requirements for Internet Hosts—Communication Layers*. If a connection is dropped as the result of keep-alives the error code WSAENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with WSAENOTCONN.

**SO\_LINGER**

The **SO\_LINGER** option controls the action taken when unsend data is queued on a socket and a **closesocket** is performed. See **closesocket** for a description of the way in which the **SO\_LINGER** settings affect the semantics of **closesocket**. The application sets the desired behavior by creating a **LINGER** structure (pointed to by the *optval* parameter) with these members **l\_onoff** and **l\_linger** set appropriately.

**SO\_REUSEADDR**

By default, a socket cannot be bound (see **bind**) to a local address that is already in use. On occasion, however, it can be necessary to reuse an address in this way. Since every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the Windows Sockets provider that a **bind** on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the **SO\_REUSEADDR** socket option for the socket before issuing the **bind**. The option is interpreted only at the time of the **bind**. It is therefore unnecessary and harmless to set the option on a socket that is not to be bound to an existing address. Setting or resetting the option after the **bind** has no effect on this or any other socket.

**SO\_RCVBUF** and **SO\_SNDBUF**

When a Windows Sockets implementation supports the **SO\_RCVBUF** and **SO\_SNDBUF** options, an application can request different buffer sizes (larger or smaller). The call to **setsockopt** can succeed even when the implementation did not provide the whole amount requested. An application must call **getsockopt** with the same option to check the buffer size actually provided.

**PVD\_CONFIG**

This object stores the configuration information for the service provider associated with the socket specified in the *s* parameter. The exact format of this data structure is specific to each service provider.

## TCP\_NODELAY

The **TCP\_NODELAY** option is specific to TCP/IP service providers. The Nagle algorithm is disabled if the **TCP\_NODELAY** option is enabled (and vice versa). The process involves buffering send data when there is unacknowledged data already in flight or buffering send data until a full-size packet can be sent. It is highly recommended that TCP/IP service providers enable the Nagle Algorithm by default, and for the vast majority of application protocols the Nagle Algorithm can deliver significant performance enhancements. However, for some applications this algorithm can impede performance, and **TCP\_NODELAY** can be used to turn it off. These are applications where many small messages are sent, and the time delays between the messages are maintained. Application writers should not set **TCP\_NODELAY** unless the impact of doing so is well-understood and desired because setting **TCP\_NODELAY** can have a significant negative impact on network and application performance.

## Notes for IrDA Sockets

- The `Af_irda.h` header file must be explicitly included.
- IrDA provides the following settable socket option:

Value	Type	Meaning
IRLMP_IAS_SET	*IAS_SET	Sets IAS attributes

The `IRLMP_IAS_SET` socket option enables the application to set a single attribute of a single class in the local IAS. The application specifies the class to set, the attribute, and attribute type. The application is expected to allocate a buffer of the necessary size for the passed parameters.

IrDA provides an IAS database that stores IrDA-based information. Limited access to the IAS database is available through the Windows Sockets 2 interface, but such access is not normally used by applications, and exists primarily to support connections to non-Windows devices that are not compliant with the Windows Sockets 2 IrDA conventions.

The following structure, **IAS\_SET**, is used with the `IRLMP_IAS_SET` setsockopt option to manage the local IAS database:

```
typedef struct _IAS_SET
{
    char    irdaClassName[IAS_MAX_CLASSNAME];
    char    irdaAttribName[IAS_MAX_ATTRIBNAME];
    u_long  irdaAttribType;
    union
    {
        LONG irdaAttribInt;
        struct
        {
            u_short  Len;
        }
    }
}
```

(continued)

(continued)

```

        u_char   OctetSeq[IAS_MAX_OCTET_STRING];
    } irdaAttribOctetSeq;
    struct
    {
        u_char   Len;
        u_char   CharSet;
        u_char   UsrStr[IAS_MAX_USER_STRING];
    } irdaAttribUsrStr;
    } irdaAttribute;
} IAS_SET, *PIAS_SET, FAR *LPIAS_SET;

```

The following structure, **IAS\_QUERY**, is used with the `IRLMP_IAS_QUERY` setsockopt option to query a peer's IAS database:

```

typedef struct _WINDOWS_IAS_QUERY
{
    u_char   irdaDeviceID[4];
    char     irdaClassName[IAS_MAX_CLASSNAME];
    char     irdaAttribName[IAS_MAX_ATTRIBNAME];
    u_long   irdaAttribType;
    union
    {
        LONG   irdaAttribInt;
        struct
        {
            u_long   Len;
            u_char   OctetSeq[IAS_MAX_OCTET_STRING];
        } irdaAttribOctetSeq;
        struct
        {
            u_long   Len;
            u_long   CharSet;
            u_char   UsrStr[IAS_MAX_USER_STRING];
        } irdaAttribUsrStr;
    } irdaAttribute;
} IAS_QUERY, *PIAS_QUERY, FAR *LPIAS_QUERY;

```

Many `SO_` level socket options are not meaningful to IrDA. Only `SO_LINGER` is specifically supported.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

**+** See Also

Windows Sockets Programming Considerations Overview, `Socket Functions`, `bind`, `getsockopt`, `ioctlsocket`, `socket`, `WSAAsyncSelect`, `WSAEventSelect`

## shutdown

The Windows Sockets **shutdown** function disables sends or receives on a socket.

```
int shutdown (  
    SOCKET s,  
    int how  
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*how*

[in] Flag that describes what types of operation will no longer be allowed.

### Return Values

If no error occurs, **shutdown** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	The <i>how</i> parameter is not valid, or is not consistent with the socket type. For example, <code>SD_SEND</code> is used with a <code>UNI_RECV</code> socket type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.

### Remarks

The **shutdown** function is used on all types of sockets to disable reception, transmission, or both.



If the *how* parameter is `SD_RECEIVE`, subsequent calls to the **recv** function on the socket will be disallowed. This has no effect on the lower protocol layers. For TCP sockets, if there is still data queued on the socket waiting to be received, or data arrives subsequently, the connection is reset, since the data cannot be delivered to the user. For UDP sockets, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated.

If the *how* parameter is `SD_SEND`, subsequent calls to the **send** function are disallowed. For TCP sockets, a FIN will be sent after all data is sent and acknowledged by the receiver.

Setting *how* to `SD_BOTH` disables both sends and receives as described above.

The **shutdown** function does not close the socket. Any resources attached to the socket will *not* be freed until **closesocket** is invoked.

To assure that all data is sent and received on a connected socket before it is closed, an application should use **shutdown** to close connection before calling **closesocket**. For example, to initiate a graceful disconnect:

1. Call **WSAAsyncSelect** to register for `FD_CLOSE` notification.
2. Call **shutdown** with *how*=`SD_SEND`.
3. When `FD_CLOSE` received, call **recv** until zero returned, or `SOCKET_ERROR`.
4. Call **closesocket**.

---

**Note** The **shutdown** function does not block regardless of the **SO\_LINGER** setting on the socket.

---

An application should not rely on being able to reuse a socket after it has been shut down. In particular, a Windows Sockets provider is not required to support the use of **connect** on a socket that has been shut down.

### Notes for ATM

There are important issues associated with connection teardown when using Asynchronous Transfer Mode (ATM) and Windows Sockets 2. For more information about these important considerations, see the section titled **Notes for ATM** in the **Remarks** section of the **closesocket** function reference.

#### Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

### See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **connect**, **socket**

## socket

The Windows Sockets **socket** function creates a socket that is bound to a specific service provider.

```
SOCKET socket (
    int    af,
    int    type,
    int    protocol
);
```

### Parameters

*af*

[in] Address family specification.

*type*

[in] Type specification for the new socket.

The following are the only two *type* specifications supported for Windows Sockets 1.1:

Type	Explanation
<b>SOCK_STREAM</b>	Provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. Uses TCP for the Internet address family.
<b>SOCK_DGRAM</b>	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses UDP for the Internet address family.

In Windows Sockets 2, many new socket types will be introduced and no longer need to be specified, since an application can dynamically discover the attributes of each available transport protocol through the **WSAEnumProtocols** function. Socket type definitions appear in Winsock2.h, which will be periodically updated as new socket types, address families, and protocols are defined.

*protocol*

[in] Protocol to be used with the socket that is specific to the indicated address family.

### Return Values

If no error occurs, **socket** returns a descriptor referencing the new socket. Otherwise, a value of **INVALID\_SOCKET** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem or the associated service provider has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	The specified protocol is not supported.
WSAEPROTOTYPE	The specified protocol is the wrong type for this socket.
WSAESOCKTNOSUPPORT	The specified socket type is not supported in this address family.
WSAEBADF	For Windows CE AF_IRDA sockets only: the shared serial port is busy.

### Remarks

The **socket** function causes a socket descriptor and any related resources to be allocated and bound to a specific transport-service provider. Windows Sockets will utilize the first available service provider that supports the requested combination of address family, socket type and protocol parameters. The socket that is created will have the overlapped attribute as a default. For Microsoft operating systems, the Microsoft-specific socket option, **SO\_OPENTYPE**, defined in *Mswsock.h* can affect this default. See Microsoft-specific documentation for a detailed description of **SO\_OPENTYPE**.

Sockets without the overlapped attribute can be created by using **WSASocket**. All functions that allow overlapped operation (**WSASend**, **WSARecv**, **WSASendTo**, **WSARecvFrom**, and **WSAIoctl**) also support nonoverlapped usage on an overlapped socket if the values for parameters related to overlapped operation are NULL.

When selecting a protocol and its supporting service provider this procedure will only choose a base protocol or a protocol chain, not a protocol layer by itself. Unchained protocol layers are not considered to have partial matches on *type* or *af* either. That is, they do not lead to an error code of **WSAEAFNOSUPPORT** or **WSAEPROTONOSUPPORT** if no suitable protocol is found.

---

**Important** The manifest constant `AF_UNSPEC` continues to be defined in the header file but its use is *strongly discouraged*, as this can cause ambiguity in interpreting the value of the *protocol* parameter.

---

Connection-oriented sockets such as **SOCK\_STREAM** provide full-duplex connections, and must be in a connected state before any data can be sent or received on it. A connection to another socket is created with a **connect** call. Once connected, data can be transferred using **send** and **recv** calls. When a session has been completed, a **closesocket** must be performed.

The communications protocols used to implement a reliable, connection-oriented socket ensure that data is not lost or duplicated. If data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and subsequent calls will fail with the error code set to `WSAETIMEDOUT`.

Connectionless, message-oriented sockets allow sending and receiving of datagrams to and from arbitrary peers using **sendto** and **recvfrom**. If such a socket is connected to a specific peer, datagrams can be sent to that peer using **send** and can be received only from this peer using **recv**.

Support for sockets with type **RAW** is not required, but service providers are encouraged to support raw sockets as practicable.

### Notes for IrDA Sockets

- The `Af_irda.h` header file must be explicitly included.
- Only **SOCK\_STREAM** is supported; the **SOCK\_DGRAM** type is not supported by IrDA.
- The *protocol* parameter is always set to 0 for IrDA.

---

**Note** On Windows NT/Windows 2000, raw socket support requires administrative privileges.

---

#### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

#### + See Also

Windows Sockets Programming Considerations Overview, Socket Functions, **accept**, **bind**, **connect**, **getsockname**, **getsockopt**, **ioctlsocket**, **listen**, **recv**, **recvfrom**, **select**, **send**, **sendto**, **setsockopt**, **shutdown**, **WSASocket**

# TransmitFile

The Windows Sockets **TransmitFile** function transmits file data over a connected socket handle. This function uses the operating system's cache manager to retrieve the file data, and provides high-performance file data transfer over sockets.

---

**Note** This function is a Microsoft-specific extension to the Windows Sockets specification. For more information, see *Microsoft Extensions and Windows Sockets 2*.

---

```

BOOL TransmitFile (
    SOCKET                hSocket,
    HANDLE                hFile,
    DWORD                nNumberOfBytesToWrite,
    DWORD                nNumberOfBytesPerSend,
    LPOVERLAPPED         lpOverlapped,
    LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
    DWORD                dwFlags
);

```

## Parameters

### *hSocket*

Handle to a connected socket. The **TransmitFile** function will transmit the file data over this socket. The socket specified by *hSocket* must be a connection-oriented socket; the **TransmitFile** function does not support datagram sockets. Sockets of type **SOCK\_STREAM**, **SOCK\_SEQPACKET**, or **SOCK\_RDM** are connection-oriented sockets.

### *hFile*

Handle to the open file that the **TransmitFile** function transmits. Since operating system reads the file data sequentially, you can improve caching performance by opening the handle with **FILE\_FLAG\_SEQUENTIAL\_SCAN**. The *hFile* parameter is optional; if the *hFile* parameter is NULL, only data in the header and/or the tail buffer is transmitted; any additional action, such as socket disconnect or reuse, is performed as specified by the *dwFlags* parameter.

### *nNumberOfBytesToWrite*

Number of file bytes to transmit. The **TransmitFile** function completes when it has sent the specified number of bytes, or when an error occurs, whichever occurs first.

Set *nNumberOfBytesToWrite* to zero in order to transmit the entire file.

### *nNumberOfBytesPerSend*

Size of each block of data sent in each send operation, in bytes. This specification is used by Windows' sockets layer. To select the default send size, set *nNumberOfBytesPerSend* to zero.

The *nNumberOfBytesPerSend* parameter is useful for message protocols that have limitations on the size of individual send requests.

### *lpOverlapped*

Pointer to an **OVERLAPPED** structure. If the socket handle has been opened as overlapped, specify this parameter in order to achieve an overlapped (asynchronous) I/O operation. By default, socket handles are opened as overlapped.

You can use *lpOverlapped* to specify an offset within the file at which to start the file data transfer by setting the **Offset** and **OffsetHigh** member of the **OVERLAPPED** structure. If *lpOverlapped* is NULL, the transmission of data always starts at the current byte offset in the file.

When *lpOverlapped* is not NULL, the overlapped I/O might not finish before **TransmitFile** returns. In that case, the **TransmitFile** function returns FALSE, and **GetLastError** returns ERROR\_IO\_PENDING. This enables the caller to continue processing while the file transmission operation completes. Windows will set the event specified by the **hEvent** member of the **OVERLAPPED** structure, or the socket specified by *hSocket*, to the signaled state upon completion of the data transmission request.

### *lpTransmitBuffers*

Pointer to a **TRANSMIT\_FILE\_BUFFERS** data structure that contains pointers to data to send before and after the file data is sent. Set the *lpTransmitBuffers* parameter to NULL if you want to transmit only the file data.

### *dwFlags*

The *dwFlags* parameter has six settings:

#### TF\_DISCONNECT

Start a transport-level disconnect after all the file data has been queued for transmission.

#### TF\_REUSE\_SOCKET

Prepare the socket handle to be reused. When the **TransmitFile** request completes, the socket handle can be passed to the **AcceptEx** function. It is only valid if TF\_DISCONNECT is also specified.

#### TF\_USE\_DEFAULT\_WORKER

Directs the Windows Sockets service provider to use the system's default thread to process long **TransmitFile** requests. The system default thread can be adjusted using the following registry parameter as a **REG\_DWORD**:

CurrentControlSet\Services\afd\Parameters\TransmitWorker

#### TF\_USE\_SYSTEM\_THREAD

Directs the Windows Sockets service provider to use system threads to process long **TransmitFile** requests.

#### TF\_USE\_KERNEL\_APC

Directs the driver to use kernel Asynchronous Procedure Calls (APCs) instead of worker threads to process long **TransmitFile** requests. Long **TransmitFile** requests are defined as requests that require more than a single read from the file or a cache; the request therefore depends on the size of the file and the specified length of the send packet.

Use of `TF_USE_KERNEL_APC` can deliver significant performance benefits. It is possible (though unlikely), however, that the thread in which context **TransmitFile** is initiated is being used for heavy computations; this situation may prevent APCs from launching. Note that the Windows Sockets kernel mode driver uses normal kernel APCs, which launch whenever a thread is in a wait state, which differs from user-mode APCs, which launch whenever a thread is in an *alertable* wait state initiated in user mode).

#### `TF_WRITE_BEHIND`

Complete the **TransmitFile** request immediately, without pending. If this flag is specified and **TransmitFile** succeeds, then the data has been accepted by the system but not necessarily acknowledged by the remote end. Do not use this setting with the `TF_DISCONNECT` and `TF_REUSE_SOCKET` flags.

### Return Values

If the **TransmitFile** function succeeds, the return value is `TRUE`. Otherwise, the return value is `FALSE`. To get extended error information, call **GetLastError**. The function returns `FALSE` if an overlapped I/O operation is not complete before **TransmitFile** returns. In that case, **GetLastError** returns `ERROR_IO_PENDING`.

### Remarks

The Windows NT Server/Windows 2000 Server optimizes the **TransmitFile** function for high performance. The Windows NT Workstation/Windows 2000 Professional optimizes the function for minimum memory and resource utilization. Expect better performance results when using **TransmitFile** on Windows NT Server/Windows 2000 Server.

---

**Note** **TransmitFile** is not functional on transports that perform their own buffering. Transports with the `TDI_SERVICE_INTERNAL_BUFFERING` flag set, such as ADSP, perform their own buffering. Because **TransmitFile** achieves its performance gains by sending data directly from the file cache. Transports that run out of buffer space on a particular connection are not handled by **TransmitFile**, and as a result of running out of buffer space on the connection, **TransmitFile** returns `STATUS_DEVICE_NOT_READY`.

---

#### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later. A Microsoft-specific extension. Not supported on Windows 95.

**Header:** Declared in `Mswsock.h`.

**Library:** Use `Mswsock.lib`.

#### + See Also

**OVERLAPPED, TRANSMIT\_FILE\_BUFFERS**

# WSAAccept

The Windows Sockets **WSAAccept** function conditionally accepts a connection based on the return value of a condition function, provides QOS flow specifications, and allows the transfer of connection data.

```
SOCKET WSAAccept (  
    SOCKET                s,  
    struct sockaddr FAR   *addr,  
    LPINT                addrLen,  
    LPCONDITIONPROC      lpfnCondition,  
    DWORD                dwCallbackData  
);
```

## Parameters

*s*

[in] Descriptor identifying a socket that is listening for connections after a call to the **listen** function.

*addr*

[out] Optional pointer to a buffer that receives the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the address family established when the socket was created.

*addrLen*

[in/out] Optional pointer to an integer that contains the length of the address *addr*.

*lpfnCondition*

[in] Procedure instance address of the optional, application-supplied condition function that will make an accept/reject decision based on the caller information passed in as parameters.

*dwCallbackData*

[in] Callback data passed back to the application as the value of the *dwCallbackData* parameter of the condition function. This parameter is not interpreted by Windows Sockets.

## Return Values

If no error occurs, **WSAAccept** returns a value of type **SOCKET** that is a descriptor for the accepted socket. Otherwise, a value of **INVALID\_SOCKET** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

The integer referred to by *addrLen* initially contains the amount of space pointed to by *addr*. On return it will contain the actual length in bytes of the address returned.



Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAECONNREFUSED	The connection request was forcefully rejected as indicated in the return value of the condition function (CF_REJECT).
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>addrLen</i> parameter is too small or the <i>addr</i> or <i>lfnCondition</i> are not part of the user address space.
WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress.
WSAEINVAL	<b>listen</b> was not invoked prior to <b>WSAAccept</b> , the return value of the condition function is not a valid one, or any case where the specified socket is in an invalid state.
WSAEMFILE	The queue is nonempty upon entry to <b>WSAAccept</b> and there are no socket descriptors available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The referenced socket is not a type that supports connection-oriented service.
WSATRY_AGAIN	The acceptance of the connection request was deferred as indicated in the return value of the condition function (CF_DEFER).
WSAEWOULDBLOCK	The socket is marked as nonblocking and no connections are present to be accepted.
WSAEACCES	The connection request that was offered has timed out or been withdrawn.

### Remarks

The **WSAAccept** function extracts the first connection on the queue of pending connections on socket *s*, and checks it against the condition function, provided the condition function is specified (that is, not NULL). If the condition function returns CF\_ACCEPT, **WSAAccept** creates a new socket. The newly created socket has the same properties as socket *s* including asynchronous events registered with **WSAAsyncSelect** or with **WSAEventSelect**. If the condition function returns CF\_REJECT, **WSAAccept** rejects the connection request. The condition function runs in the same thread as this function does, and should return as soon as possible. If the decision cannot be made immediately, the condition function should return CF\_DEFER to indicate that no decision has been made, and no action about this connection request

should be taken by the service provider. When the application is ready to take action on the connection request, it will invoke **WSAAccept** again and return either **CF\_ACCEPT** or **CF\_REJECT** as a return value from the condition function.

A socket in default mode (blocking) will block until a connection is present when an application calls **WSAAccept** and no connections are pending on the queue.

A socket in nonblocking mode (blocking) fails with the error **WSAEWOULDBLOCK** when an application calls **WSAAccept** and no connections are pending on the queue. After **WSAAccept** succeeds and returns a new socket handle, the accepted socket cannot be used to accept any more connections. The original socket remains open and listens for new connection requests.

The *addr* parameter is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the address family in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*. On return, it will contain the actual length (in bytes) of the address returned. This call is used with connection-oriented socket types such as **SOCK\_STREAM**. If *addr* and/or *addrlen* are equal to **NULL**, then no information about the remote address of the accepted socket is returned. Otherwise, these two parameters will be filled in regardless of whether the condition function is specified or what it returns.

A prototype of the condition function is as follows:

```
int CALLBACK ConditionFunc (
    IN LPWSABUF      IpCallerId,
    IN LPWSABUF      IpCallerData,
    IN OUT LPQOS     IpSQOS,
    IN OUT LPQOS     IpGQOS,
    IN LPWSABUF      IpCalleeId,
    OUT LPWSABUF     IpCalleeData,
    OUT GROUP FAR    *g,
    IN DWORD         dwCallbackData
);
```

The **ConditionFunc** is a placeholder for the application-supplied callback function. The actual condition function must reside in a DLL or application module. It is exported in the module definition file. Use **MakeProcInstance** to get a procedure-instance address for the callback function.

The *IpCallerId* parameter points to a **WSABUF** structure that contains the address of the connecting entity, where its *len* parameter is the length of the buffer in bytes, and its *buf* parameter is a pointer to the buffer. The *IpCallerData* is a value parameter that contains any user data. The information in these parameters is sent along with the connection request. If no caller identification or caller data is available, the corresponding parameters will be **NULL**. Many network protocols do not support connect-time caller data. Most conventional network protocols can be expected to support caller identifier information at connection-request time. The *buf* portion of the **WSABUF** pointed to by

*IpCallerId* points to a **SOCKADDR**. The **SOCKADDR** structure is interpreted according to its address family (typically by casting the **SOCKADDR** to some type specific to the address family).

The *IpSQOS* parameter references the **FLOWSPEC** structures for socket *s* specified by the caller, one for each direction, followed by any additional provider-specific parameters. The sending or receiving flow specification values will be ignored as appropriate for any unidirectional sockets. A NULL value for indicates that there is no caller supplied QOS and that no negotiation is possible. A non-NULL *IpSQOS* pointer indicates that a QOS negotiation is to occur or that the provider is prepared to accept the QOS request without negotiation.

The *IpGQOS* parameter is reserved, and should be NULL.

The *IpCalleeId* is a value parameter that contains the local address of the connected entity. The *buf* portion of the WSABUF pointed to by *IpCalleeId* points to a **SOCKADDR**. The **SOCKADDR** structure is interpreted according to its address family (typically by casting the **SOCKADDR** to some type specific to the address family).

The *IpCalleeData* is a result parameter used by the condition function to supply user data back to the connecting entity. The *IpCalleeData->len* initially contains the length of the buffer allocated by the service provider and pointed to by *IpCalleeData->buf*. A value of zero means passing user data back to the caller is not supported. The condition function should copy up to *IpCalleeData->len* bytes of data into *IpCalleeData->buf*, and then update *IpCalleeData->len* to indicate the actual number of bytes transferred. If no user data is to be passed back to the caller, the condition function should set *IpCalleeData->len* to zero. The format of all address and user data is specific to the address family to which the socket belongs.

The *dwCallbackData* parameter value passed to the condition function is the value passed as the *dwCallbackData* parameter in the original **WSAAccept** call. This value is interpreted only by the Windows Socket version 2 client. This allows a client to pass some context information from the **WSAAccept** call site through to the condition function. This also provides the condition function with any additional information required to determine whether to accept the connection or not. A typical usage is to pass a (suitably cast) pointer to a data structure containing references to application-defined objects with which this socket is associated.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **accept**, **bind**, **connect**, **getsockopt**, **listen**, **select**, **socket**, **WSAAsyncSelect**, **WSAConnect**

# WSAAddressToString

The Windows Sockets **WSAAddressToString** function converts all components of a **SOCKADDR** structure into a human-readable string representation of the address.

This is intended to be used mainly for display purposes. If the caller wants the translation to be done by a particular provider, it should supply the corresponding **WSAPROTOCOL\_INFO** structure in the *lpProtocolInfo* parameter.

```
INT WSAAddressToString (  
    LPSOCKADDR          lpsaAddress,  
    DWORD               dwAddressLength,  
    LPWSAPROTOCOL_INFO lpProtocolInfo,  
    OUT LPTSTR          lpzAddressString,  
    IN OUT LPDWORD     lpdwAddressStringLength  
);
```

## Parameters

### *lpsaAddress*

[in] Pointer to the **SOCKADDR** structure to translate into a string.

### *dwAddressLength*

[in] Length of the address in **SOCKADDR**, which may vary in size with different protocols.

### *lpProtocolInfo*

[in] (Optional) The **WSAPROTOCOL\_INFO** structure for a particular provider. If this is NULL, the call is routed to the provider of the first protocol supporting the address family indicated in *lpsaAddress*.

### *lpzAddressString*

[in] Buffer that receives the human-readable address string.

### *lpdwAddressStringLength*

[in/out] On input, the length of the *AddressString* buffer. On output, returns the length of the string actually copied into the buffer. If the supplied buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size in bytes.

## Return Values

If no error occurs, **WSAAddressToString** returns a value of zero. Otherwise, the value **SOCKET\_ERROR** is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSAEFAULT	The specified <i>IpcsAddress</i> , <i>IpProtocolInfo</i> , <i>IpszAddressString</i> are not all in the address space of the process, or the <i>IpszAddressString</i> buffer is too small. Pass in a larger buffer.
WSAEINVAL	The specified address is not a valid socket address, or there was no transport provider supporting its indicated address family.
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application <i>must</i> first call <b>WSAStartup</b> before calling any Windows Sockets functions.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT/2000.

## WSAAsyncGetHostByAddr

The Windows Sockets **WSAAsyncGetHostByAddr** function asynchronously retrieves host information that corresponds to an address.

```
HANDLE WSAAsyncGetHostByAddr (
    HWND          hWnd,
    unsigned int  wMsg,
    const char FAR *addr,
    int           len,
    int           type,
    char FAR      *buf,
    int           buflen
);
```

### Parameters

*hWnd*

[in] Handle of the window that will receive a message when the asynchronous request completes.

*wMsg*

[in] Message to be received when the asynchronous request completes.

*addr*

[in] Pointer to the network address for the host. Host addresses are stored in network byte order.

*len*

[in] Length of the address.

*type*

[in] Type of the address.

*buf*

[out] Pointer to the data area to receive the **HOSTENT** data. The data area *must* be larger than the size of a **HOSTENT** structure because the supplied data area is used by Windows Sockets to contain a **HOSTENT** structure and all of the data referenced by members of the **HOSTENT** structure. A buffer of MAXGETHOSTSTRUCT bytes is recommended.

*buflen*

[in] Size of data area for the *buf* parameter.

## Return Values

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetHostByAddr** returns a nonzero value of type **HANDLE** that is the asynchronous task handle (not to be confused with a Windows **HTASK**) for the request. This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest**, or it can be used to match up asynchronous operations and completion messages by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncGetHostByAddr** returns a zero value, and a specific error number can be retrieved by calling **WSAGetLastError**.

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

Error code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	<i>addr</i> or <i>buf</i> is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative host not found, or SERVERFAIL.

(continued)

(continued)

Error code	Meaning
WSANO_RECOVERY	Nonrecoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

Error Code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

## Remarks

The **WSAAsyncGetHostByAddr** function is an asynchronous version of **gethostbyaddr**. It is used to retrieve the host name and address information that corresponds to a network address. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer supplied to the original function call contains a **HOSTENT** structure. To access the members of this structure, the original buffer address is cast to a **HOSTENT** structure pointer and accessed as appropriate.

If the error code is **WSAENOBUFS**, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByAddr** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer supplied to this function is used by Windows Sockets to construct a structure together with the contents of data areas referenced by members of the same **HOSTENT** structure. To avoid the **WSAENOBUFFS** error, the application should provide a buffer of at least **MAXGETHOSTSTRUCT** bytes (as defined in **Winsock2.h**).

The error code and buffer length should be extracted from the *IParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLLEN**, defined in **Winsock2.h** as:

```
#define WSAGETASYNCERROR(IParam)      HIWORD(IParam)
#define WSAGETASYNCBUFLLEN(IParam)   LOWORD(IParam)
```

The use of these macros will maximize the portability of the source code for the application.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in **Winsock2.h**.

**Library:** Use **Ws2\_32.lib**.

### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **gethostbyaddr**, **HOSTENT**, **WSACancelAsyncRequest**

## WSAAsyncGetHostByName

The Windows Sockets **WSAAsyncGetHostByName** function asynchronously retrieves host information corresponding to a host name.

```
HANDLE WSAAsyncGetHostByName (
    HWND          hWnd,
    unsigned int  wMsg,
    const char FAR *name,
    char FAR      *buf,
    int           buflen
);
```

### Parameters

*hWnd*

[in] Handle of the window that will receive a message when the asynchronous request completes.

*wMsg*

[in] Message to be received when the asynchronous request completes.

*name*

[in] Pointer to the null-terminated name of the host.



*buf*

[out] Pointer to the data area to receive the **HOSTENT** data. The data area must be larger than the size of a **HOSTENT** structure because the supplied data area is used by Windows Sockets to contain a **HOSTENT** structure and all of the data referenced by members of the **HOSTENT** structure. A buffer of MAXGETHOSTSTRUCT bytes is recommended.

*buflen*

[in] Size of data area for the *buf* parameter.

## Return Values

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetHostByName** returns a nonzero value of type HANDLE that is the asynchronous task handle (not to be confused with a Windows HTASK) for the request. This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest**, or it can be used to match up asynchronous operations and completion messages by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncGetHostByName** returns a zero value, and a specific error number can be retrieved by calling **WSAGetLastError**.

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

Error code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The <i>name</i> or <i>buf</i> parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	A nonauthoritative host not found, or SERVERFAIL.
WSANO_RECOVERY	Nonrecoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

---

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

### Remarks

The **WSAAsyncGetHostByName** function is an asynchronous version of **gethostbyname**, and is used to retrieve host name and address information corresponding to a host name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque asynchronous task handle that which the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in *Winsock2.h*. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer supplied to the original function call contains a **HOSTENT** structure. To access the elements of this structure, the original buffer address should be cast to a **HOSTENT** structure pointer and accessed as appropriate.

If the error code is **WSAENOBUFS**, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByAddr** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer supplied to this function is used by Windows Sockets to construct a **HOSTENT** structure together with the contents of data areas referenced by members of the same **HOSTENT** structure. To avoid the **WSAENOBUFS** error, the application should provide a buffer of at least **MAXGETHOSTSTRUCT** bytes (as defined in *Winsock2.h*).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in *Winsock2.h* as:

```
#define WSAGETASYNCERROR(1Param)      HIWORD(1Param)
#define WSAGETASYNCBUFLen(1Param)    LOWORD(1Param)
```

The use of these macros will maximize the portability of the source code for the application.

**WSAAsyncGetHostByName** is guaranteed to resolve the string returned by a successful call to **gethostname**.

#### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **gethostbyname**, **HOSTENT**, **WSACancelAsyncRequest**

---

## WSAAsyncGetProtoByName

The Windows Sockets **WSAAsyncGetProtoByName** function gets protocol information corresponding to a protocol name asynchronously.

```
HANDLE WSAAsyncGetProtoByName (
    HWND          hWnd,
    unsigned int  wParam,
    const char FAR *name,
    char FAR      *buf,
    int           buflen
):
```

### Parameters

*hWnd*

[in] Handle of the window that will receive a message when the asynchronous request completes.

*wMsg*

[in] Message to be received when the asynchronous request completes.

*name*

[in] Pointer to the null-terminated protocol name to be resolved.

*buf*

[out] Pointer to the data area to receive the **PROTOENT** data. The data area must be larger than the size of a **PROTOENT** structure because the data area is used by Windows Sockets to contain a **PROTOENT** structure and all of the data that is referenced by members of the **PROTOENT** structure. A buffer of **MAXGETHOSTSTRUCT** bytes is recommended.

*buflen*

[out] Size of data area for the *buf* parameter.

## Return Values

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetProtoByName** returns a nonzero value of type **HANDLE** that is the asynchronous task handle for the request (not to be confused with a Windows **HTASK**). This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest**, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncGetProtoByName** returns a zero value, and a specific error number can be retrieved by calling **WSAGetLastError**.

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

Error code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The <i>name</i> or <i>buf</i> parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	A nonauthoritative protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

### Remarks

The **WSAAsyncGetProtoByName** function is an asynchronous version of **getprotobyname**. It is used to retrieve the protocol name and number from the Windows Sockets database corresponding to a given protocol name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer supplied to the original function call contains a **PROTOENT** structure. To access the members of this structure, the original buffer address should be cast to a **PROTOENT** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByAddr** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer supplied to this function is used by Windows Sockets to construct a **PROTOENT** structure together with the contents of data areas referenced by members of the same **PROTOENT** structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLen**, defined in Winsock2.h as:

```
#define WSAGETASYNCERROR(1Param)      HIWORD(1Param)
#define WSAGETASYNCBUFLen(1Param)    LOWORD(1Param)
```

The use of these macros will maximize the portability of the source code for the application.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **getprotobyname**, **WSACancelAsyncRequest**

## WSAAsyncGetProtoByNumber

The Windows Sockets **WSAAsyncGetProtoByNumber** function asynchronously retrieves protocol information corresponding to a protocol number.

```
HANDLE WSAAsyncGetProtoByNumber (
    HWND          hWnd,
    unsigned int  wParam,
    int           number,
    char FAR     *buf,
    int           buflen
);
```

### Parameters

*hWnd*

[in] Handle of the window that will receive a message when the asynchronous request completes.

*wMsg*

[in] Message to be received when the asynchronous request completes.

*number*

[in] Protocol number to be resolved, in host byte order.

*buf*

[out] Pointer to the data area to receive the **PROTOENT** data. The data area must be larger than the size of a **PROTOENT** structure because the data area is used by Windows Sockets to contain a **PROTOENT** structure and all of the data that is referenced by members of the **PROTOENT** structure. A buffer of **MAXGETHOSTSTRUCT** bytes is recommended.

*buflen*

[in] Size of data area for the *buf* parameter.

## Return Values

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetProtoByNumber** returns a nonzero value of type **HANDLE** that is the asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest**, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncGetProtoByNumber** returns a zero value, and a specific error number can be retrieved by calling **WSAGetLastError**.

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

Error code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The <i>buf</i> parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	Nonauthoritative protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

## Remarks

The **WSAAsyncGetProtoByNumber** function is an asynchronous version of **getprotobynumber**, and is used to retrieve the protocol name and number corresponding to a protocol number. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in *Winsock2.h*. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer supplied to the original function call contains a **PROTOENT** structure. To access the members of this structure, the original buffer address is cast to a **PROTOENT** structure pointer and accessed as appropriate.

If the error code is **WSAENOBUFS**, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByAddr** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer supplied to this function is used by Windows Sockets to construct a **PROTOENT** structure together with the contents of data areas referenced by members of the same **PROTOENT** structure. To avoid the **WSAENOBUFS** error noted above, the application should provide a buffer of at least **MAXGETHOSTSTRUCT** bytes (as defined in *Winsock2.h*).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in *Winsock2.h* as:

```
#define WSAGETASYNCERROR(lParam) HIWORD(lParam)
#define WSAGETASYNCBUFLEN(lParam) LOWORD(lParam)
```

The use of these macros will maximize the portability of the source code for the application.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in *Winsock2.h*.

**Library:** Use *Ws2\_32.lib*.



**+** See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **getprotobynumber**, **WSACancelAsyncRequest**

---

## WSAAsyncGetServByName

The Windows Sockets **WSAAsyncGetServByName** function asynchronously retrieves service information corresponding to a service name and port.

```
HANDLE WSAAsyncGetServByName (  
    HWND                hWnd,  
    unsigned int        wParam,  
    const char FAR      *name,  
    const char FAR      *proto,  
    char FAR            *buf,  
    int                 buflen  
);
```

### Parameters

*hWnd*

[in] Handle of the window that should receive a message when the asynchronous request completes.

*wParam*

[in] Message to be received when the asynchronous request completes.

*name*

[in] Pointer to a null-terminated service name.

*proto*

[in] Pointer to a protocol name. This can be NULL, in which case **WSAAsyncGetServByName** will search for the first service entry for which *s\_name* or one of the *s\_aliases* matches the given *name*. Otherwise, **WSAAsyncGetServByName** matches both *name* and *proto*.

*buf*

[out] Pointer to the data area to receive the **SERVENT** data. The data area must be larger than the size of a **SERVENT** structure because the data area supplied is used by Windows Sockets to contain a **SERVENT** structure and all of the data that is referenced by members of the **SERVENT** structure. A buffer of MAXGETHOSTSTRUCT bytes is recommended.

*buflen*

[in] Size of data area for the *buf* parameter.

## Return Values

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetServByName** returns a nonzero value of type **HANDLE** that is the asynchronous task handle for the request (not to be confused with a Windows **HTASK**). This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest**, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncServByName** returns a zero value, and a specific error number can be retrieved by calling **WSAGetLastError**.

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

Error code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	<i>buf</i> is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative service not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

## Remarks

The **WSAAsyncGetServByName** function is an asynchronous version of **getservbyname** and is used to retrieve service information corresponding to a service name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in *Winsock2.h*. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer supplied to the original function call contains a **SERVENT** structure. To access the members of this structure, the original buffer address should be cast to a **SERVENT** structure pointer and accessed as appropriate.

If the error code is *WSAENOBUFFS*, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByAddr** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer supplied to this function is used by Windows Sockets to construct a **SERVENT** structure together with the contents of data areas referenced by members of the same **SERVENT** structure. To avoid the *WSAENOBUFFS* error, the application should provide a buffer of at least *MAXGETHOSTSTRUCT* bytes (as defined in *Winsock2.h*).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLen**, defined in *Winsock2.h* as:

```
#define WSAGETASYNCERROR(lParam)      HIWORD(lParam)
#define WSAGETASYNCBUFLen(lParam)    LOWORD(lParam)
```

The use of these macros will maximize the portability of the source code for the application.

## ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in *Winsock2.h*.

**Library:** Use *Ws2\_32.lib*.

**+ See Also**

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **getservbyname**, **WSACancelAsyncRequest**

## WSAAsyncGetServByPort

The Windows Sockets **WSAAsyncGetServByPort** function gets service information corresponding to a port and protocol asynchronously.

```
HANDLE WSAAsyncGetServByPort (
    HWND          hWnd,
    unsigned int  wParam,
    int           port,
    const char FAR *proto,
    char FAR      *buf,
    int           buflen
);
```

### Parameters

***hWnd***

[in] Handle of the window that should receive a message when the asynchronous request completes.

***wParam***

[in] Message to be received when the asynchronous request completes.

***port***

[in] Port for the service, in network byte order.

***proto***

[in] Pointer to a protocol name. This can be NULL, in which case **WSAAsyncGetServByPort** will search for the first service entry for which *s\_port* match the given *port*. Otherwise, **WSAAsyncGetServByPort** matches both *port* and *proto*.

***buf***

[out] Pointer to the data area to receive the **SERVENT** data. The data area must be larger than the size of a **SERVENT** structure because the data area supplied is used by Windows Sockets to contain a **SERVENT** structure and all of the data that is referenced by members of the **SERVENT** structure. A buffer of MAXGETHOSTSTRUCT bytes is recommended.

***buflen***

[in] Size of data area for the *buf* parameter.

## Return Values

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetServByPort** returns a nonzero value of type HANDLE that is the asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest**, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncGetServByPort** returns a zero value, and a specific error number can be retrieved by calling **WSAGetLastError**.

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

Error code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	<i>proto</i> or <i>buf</i> is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer port not found.
WSATRY_AGAIN	Nonauthoritative port not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

## Remarks

The **WSAAsyncGetServByPort** function is an asynchronous version of **getservbyport**, and is used to retrieve service information corresponding to a port number. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in *Winsock2.h*. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer supplied to the original function call contains a **SERVENT** structure. To access the members of this structure, the original buffer address should be cast to a **SERVENT** structure pointer and accessed as appropriate.

If the error code is *WSAENOBUFFS*, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByAddr** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer supplied to this function is used by Windows Sockets to construct a **SERVENT** structure together with the contents of data areas referenced by members of the same **SERVENT** structure. To avoid the *WSAENOBUFFS* error, the application should provide a buffer of at least *MAXGETHOSTSTRUCT* bytes (as defined in *Winsock2.h*).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in *Winsock2.h* as:

```
#define WSAGETASYNCERROR(lParam)    HIWORD(lParam)
#define WSAGETASYNCBUFLEN(lParam)  LOWORD(lParam)
```

The use of these macros will maximize the portability of the source code for the application.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in *Winsock2.h*.

**Library:** Use *Ws2\_32.lib*.

### See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **getservbyport**, **WSACancelAsyncRequest**

## WSAAsyncSelect

The Windows Sockets **WSAAsyncSelect** function requests Windows message-based notification of network events for a socket.

```
int WSAAsyncSelect (
    SOCKET          s,
    HWND           hWnd,
    unsigned int    wMsg,
    long           lEvent
);
```

### Parameters

*s*

[in] Descriptor identifying the socket for which event notification is required.

*hWnd*

[in] Handle identifying the window that will receive a message when a network event occurs.

*wMsg*

[in] Message to be received when a network event occurs.

*lEvent*

[in] Bitmask that specifies a combination of network events in which the application is interested.

### Return Values

If the **WSAAsyncSelect** function succeeds, the return value is zero provided the application's declaration of interest in the network event set was successful. Otherwise, the value **SOCKET\_ERROR** is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid such as the window handle not referring to an existing window, or the specified socket is in an invalid state.

Error code	Meaning
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor is not a socket.

Additional error codes can be set when an application window receives a message. This error code is extracted from the *IPParam* in the reply message using the WSAGETSELECTERROR macro. Possible error codes for each network event are shown in the following table.

#### Event: FD\_CONNECT

Error code	Meaning
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAEFAULT	The <i>namelen</i> parameter is incorrect.
WSAEINVAL	The socket is already bound to an address.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more file descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTCONN	The socket is not connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.

#### Event: FD\_CLOSE

Error code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was terminated due to a time-out or other failure.



Event: FD\_READ

Event: FD\_WRITE

Event: FD\_OOB

Event: FD\_ACCEPT

Event: FD\_QOS

Event: FD\_GROUP\_QOS

Event: FD\_ADDRESS\_LIST\_CHANGE

Error code	Meaning
WSAENETDOWN	The network subsystem has failed.

Event: FD\_ROUTING\_INTERFACE\_CHANGE

Error code	Meaning
WSAENETUNREACH	The specified destination is no longer reachable.
WSAENETDOWN	The network subsystem has failed.

### Remarks

The **WSAAsyncSelect** function is used to request that *Ws2\_32.dll* should send a message to the window *hWnd* whenever it detects any of the network events specified by the *lEvent* parameter. The message that should be sent is specified by the *wMsg* parameter. The socket for which notification is required is identified by the *s* parameter.

The **WSAAsyncSelect** function automatically sets socket *s* to nonblocking mode, regardless of the value of *lEvent*. See the **ioctlsocket** functions for information on how to set the nonblocking socket back to blocking mode.

The *lEvent* parameter is constructed by using the bitwise OR operator with any of the values specified in the following table.

Value	Meaning
FD_READ	Wants to receive notification of readiness for reading.
FD_WRITE	Wants to receive notification of readiness for writing.
FD_OOB	Wants to receive notification of the arrival of OOB data.
FD_ACCEPT	Wants to receive notification of incoming connections.
FD_CONNECT	Wants to receive notification of completed connection or multipoint join operation.
FD_CLOSE	Wants to receive notification of socket closure.
FD_QOS	Wants to receive notification of socket Quality of Service (QOS) changes.

Value	Meaning
FD_GROUP_QOS	Reserved.
FD_ROUTING_INTERFACE_CHANGE	Wants to receive notification of routing interface changes for the specified destination(s).
FD_ADDRESS_LIST_CHANGE	Wants to receive notification of local address list changes for the socket's protocol family.

Issuing a **WSAAsyncSelect** for a socket cancels any previous **WSAAsyncSelect** or **WSAEventSelect** for the same socket. For example, to receive notification for both reading and writing, the application must call **WSAAsyncSelect** with both **FD\_READ** and **FD\_WRITE**, as follows:

```
rc = WSAAsyncSelect(s, hWnd, wParam, FD_READ|FD_WRITE);
```

It is not possible to specify different messages for different events. The following code will *not* work; the second call will cancel the effects of the first, and only **FD\_WRITE** events will be reported with message **wMsg2**:

```
rc = WSAAsyncSelect(s, hWnd, wParam1, FD_READ);
rc = WSAAsyncSelect(s, hWnd, wParam2, FD_WRITE);
```

To cancel all notification indicating that Windows Sockets should send no further messages related to network events on the socket, *lEvent* is set to zero.

```
rc = WSAAsyncSelect(s, hWnd, 0, 0);
```

Although **WSAAsyncSelect** immediately disables event message posting for the socket in this instance, it is possible that messages could be waiting in the application's message queue. Therefore, the application must be prepared to receive network event messages even after cancellation. Closing a socket with **closesocket** also cancels **WSAAsyncSelect** message sending, but the same caveat about messages in the queue still applies.

The socket created by the **accept** function has the same properties as the listening socket used to accept it. Consequently, **WSAAsyncSelect** events set for the listening socket also apply to the accepted socket. For example, if a listening socket has **WSAAsyncSelect** events **FD\_ACCEPT**, **FD\_READ**, and **FD\_WRITE**, then any socket accepted on that listening socket will also have **FD\_ACCEPT**, **FD\_READ**, and **FD\_WRITE** events with the same *wMsg* value used for messages. If a different *wMsg* or events are desired, the application should call **WSAAsyncSelect**, passing the accepted socket and the desired new information.

When one of the nominated network events occurs on the specified socket *s*, the application's window *hWnd* receives message *wMsg*. The *wParam* parameter identifies the socket on which a network event has occurred. The low word of *lParam* specifies the network event that has occurred. The high word of *lParam* contains any error code. The error code be any error as defined in `Winsock2.h`.

---

**Note** Upon receipt of an event notification message, the **WSAGetLastError** function cannot be used to check the error value because the error value returned can differ from the value in the high word of *IPParam*.

---

The error and event codes can be extracted from the *IPParam* using the macros **WSAGETSELECTERROR** and **WSAGETSELECTEVENT**, defined in *Winsock2.h* as:

```
#define WSAGETSELECTERROR(IPParam)      HIWORD(IPParam)
#define WSAGETSELECTEVENT(IPParam)     LOWORD(IPParam)
```

The use of these macros will maximize the portability of the source code for the application.

The possible network event codes that can be returned are shown in the following table.

Value	Meaning
FD_READ	Socket <i>s</i> ready for reading.
FD_WRITE	Socket <i>s</i> ready for writing.
FD_OOB	OOB data ready for reading on socket <i>s</i> .
FD_ACCEPT	Socket <i>s</i> ready for accepting a new incoming connection.
FD_CONNECT	Connection or multipoint <i>join</i> operation initiated on socket <i>s</i> completed.
FD_CLOSE	Connection identified by socket <i>s</i> has been closed.
FD_QOS	Quality of Service associated with socket <i>s</i> has changed.
FD_GROUP_QOS	Reserved.
FD_ROUTING_INTERFACE_CHANGE	Local interface that should be used to send to the specified destination has changed.
FD_ADDRESS_LIST_CHANGE	The list of addresses of the socket's protocol family to which the application client can bind has changed.

Although **WSAAsyncSelect** can be called with interest in multiple events, the application window will receive a single message for each network event.

As in the case of the **select** function, **WSAAsyncSelect** will frequently be used to determine when a data transfer operation (**send** or **recv**) can be issued with the expectation of immediate success. Nevertheless, a robust application must be prepared for the possibility that it can receive a message and issue a Windows Sockets 2 call that returns **WSAEWOULDBLOCK** immediately. For example, the following sequence of events is possible:

1. Data arrives on socket *s*; Windows Sockets 2 posts **WSAAsyncSelect** message
2. Application processes some other message
3. While processing, application issues an **ioctlsocket(s, FIONREAD...)** and notices that there is data ready to be read

4. Application issues a **recv(s,...)** to read the data
5. Application loops to process next message, eventually reaching the **WSAAsyncSelect** message indicating that data is ready to read
6. Application issues **recv(s,...)**, which fails with the error **WSAEWOULDBLOCK**.

Other sequences are possible.

The `Ws2_32.dll` will *not* continually flood an application with messages for a particular network event. Having successfully posted notification of a particular event to an application window, no further message(s) for that network event will be posted to the application window until the application makes the function call that implicitly reenables notification of that network event.

Event	Reenabling function
FD_READ	<b>recv</b> , <b>recvfrom</b> , <b>WSARecv</b> , or <b>WSARecvFrom</b> .
FD_WRITE	<b>send</b> , <b>sendto</b> , <b>WSASend</b> , or <b>WSASendTo</b> .
FD_OOB	<b>recv</b> , <b>recvfrom</b> , <b>WSARecv</b> , or <b>WSARecvFrom</b> .
FD_ACCEPT	<b>accept</b> or <b>WSAAccept</b> unless the error code is <b>WSATRY_AGAIN</b> indicating that the condition function returned <b>CF_DEFER</b> .
FD_CONNECT	NONE.
FD_CLOSE	NONE.
FD_QOS	<b>WSAIoctl</b> with command <b>SIO_GET_QOS</b> .
FD_GROUP_QOS	Reserved.
FD_ROUTING_INTERFACE_CHANGE	<b>WSAIoctl</b> with command <b>SIO_ROUTING_INTERFACE_CHANGE</b> .
FD_ADDRESS_LIST_CHANGE	<b>WSAIoctl</b> with command <b>SIO_ADDRESS_LIST_CHANGE</b> .

Any call to the reenabling routine, even one that fails, results in reenabling of message posting for the relevant event.

For **FD\_READ**, **FD\_OOB**, and **FD\_ACCEPT** events, message posting is level-triggered. This means that if the reenabling routine is called and the relevant condition is still met after the call, a **WSAAsyncSelect** message is posted to the application. This allows an application to be event-driven and not be concerned with the amount of data that arrives at any one time. Consider the following sequence:

1. Network transport stack receives 100 bytes of data on socket **s** and causes Windows Sockets 2 to post an **FD\_READ** message.
2. The application issues **recv( s, buffptr, 50, 0)** to read 50 bytes.
3. Another **FD\_READ** message is posted since there is still data to be read.

With these semantics, an application need not read all available data in response to an FD\_READ message—a single **recv** in response to each FD\_READ message is appropriate. If an application issues multiple **recv** calls in response to a single FD\_READ, it can receive multiple FD\_READ messages. Such an application can need to disable FD\_READ messages before starting the **recv** calls by calling **WSAAsyncSelect** with the FD\_READ event not set.

The FD\_QOS and FD\_GROUP\_QOS events are considered edge triggered. A message will be posted exactly once when a quality of service change occurs. Further messages will *not* be forthcoming until either the provider detects a further change in quality of service or the application renegotiates the quality of service for the socket.

The FD\_ROUTING\_INTERFACE\_CHANGE message is posted when the local interface that should be used to reach the destination specified in **WSAIoctl** with SIO\_ROUTING\_INTERFACE\_CHANGE changes *after* such IOCTL has been issued.

The FD\_ADDRESS\_LIST\_CHANGE message is posted when the list of addresses to which the application can bind changes *after* **WSAIoctl** with SIO\_ADDRESS\_LIST\_CHANGE has been issued.

If any event has already happened when the application calls **WSAAsyncSelect** or when the reenabling function is called, then a message is posted as appropriate. For example, consider the following sequence:

1. An application calls **listen**.
2. A connect request is received but not yet accepted.
3. The application calls **WSAAsyncSelect** specifying that it wants to receive FD\_ACCEPT messages for the socket. Due to the persistence of events, Windows Sockets 2 posts an FD\_ACCEPT message immediately.

The FD\_WRITE event is handled slightly differently. An FD\_WRITE message is posted when a socket is first connected with **connect** or **WSAConnect** (after FD\_CONNECT, if also registered) or accepted with **accept** or **WSAAccept**, and then after a send operation fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD\_WRITE message and lasting until a send returns WSAEWOULDBLOCK. After such a failure the application will be notified that sends are again possible with an FD\_WRITE message.

The FD\_OOB event is used only when a socket is configured to receive OOB data separately. (See section DECnet Out-Of-band data for a discussion of this topic.) If the socket is configured to receive OOB data inline, the OOB (expedited) data is treated as normal data and the application should register an interest in, and will receive, FD\_READ events, not FD\_OOB events. An application can set or inspect the way in which OOB data is to be handled by using **setsockopt** or **getsockopt** for the SO\_OOBINLINE option.

The error code in an `FD_CLOSE` message indicates whether the socket close was graceful or abortive. If the error code is zero, then the close was graceful; if the error code is `WSAECONNRESET`, then the socket's virtual circuit was reset. This only applies to connection-oriented sockets such as **SOCK\_STREAM**.

The `FD_CLOSE` message is posted when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the `FD_CLOSE` is posted when the connection goes into the `TIME_WAIT` or `CLOSE_WAIT` states. This results from the remote end performing a **shutdown** on the send side or a **closesocket**. `FD_CLOSE` should only be posted after all data is read from a socket, but an application should check for remaining data upon receipt of `FD_CLOSE` to avoid any possibility of losing data.

Please note your application will receive **ONLY** an `FD_CLOSE` message to indicate closure of a virtual circuit, and only when all the received data has been read if this is a graceful close. It will *not* receive an `FD_READ` message to indicate this condition.

The `FD_QOS` message is posted when any parameter in the flow specification associated with socket `s` has changed. Applications should use **WSAIoctl** with command `SIO_GET_QOS` to get the current QOS for socket `s`.

The `FD_ROUTING_INTERFACE_CHANGE` and `FD_ADDRESS_LIST_CHANGE` events are considered edge triggered as well. A message will be posted exactly once when a change occurs after the application has requested the notification by issuing **WSAIoctl** with `SIO_ROUTING_INTERFACE_CHANGE` or `SIO_ADDRESS_LIST_CHANGE` correspondingly. Further messages will not be forthcoming until the application reissues the `IOCTL` and another change is detected since the `IOCTL` has been issued.

Here is a summary of events and conditions for each asynchronous notification message.

- **FD\_READ:**

1. When **WSAAsyncSelect** is called, if there is data currently available to receive.
2. When data arrives, if `FD_READ` is not already posted.
3. After **recv** or **recvfrom** is called (with or without `MSG_PEEK`), if data is still available to receive.

**Note** When **setsockopt** `SO_OOBINLINE` is enabled, data includes both normal data and OOB data in the instances noted above.

- **FD\_WRITE:**

1. When **WSAAsyncSelect** called, if a **send** or **sendto** is possible.
2. After **connect** or **accept** called, when connection established.
3. After **send** or **sendto** fail with `WSAEWOULDBLOCK`, when **send** or **sendto** are likely to succeed.
4. After **bind** on a connectionless socket. `FD_WRITE` may or may not occur at this time (implementation-dependent). In any case, a connectionless socket is always writable immediately after a **bind** operation.

- **FD\_OOB:** Only valid when **setsockopt** SO\_OOBINLINE is disabled (default).
  1. When **WSAAsyncSelect** called, if there is OOB data currently available to receive with the MSG\_OOB flag.
  2. When OOB data arrives, if FD\_OOB not already posted.
  3. After **recv** or **recvfrom** called with *or without* MSG\_OOB flag, if OOB data is still available to receive.
- **FD\_ACCEPT:**
  1. When **WSAAsyncSelect** called, if there is currently a connection request available to accept.
  2. When a connection request arrives, if FD\_ACCEPT not already posted.
  3. After **accept** called, if there is another connection request available to accept.
- **FD\_CONNECT:**
  1. When **WSAAsyncSelect** called, if there is currently a connection established.
  2. After **connect** called, when connection is established (even when **connect** succeeds immediately, as is typical with a datagram socket).
  3. After calling **WSAJoinLeaf**, when join operation completes.
  4. After **connect**, **WSAConnect**, or **WSAJoinLeaf** was called with a nonblocking, connection-oriented socket. The initial operation returned with a specific error of WSAEWOULDBLOCK, but the network operation went ahead. Whether the operation eventually succeeds or not, when the outcome has been determined, FD\_CONNECT happens. The client should check the error code to determine whether the outcome was successful or failed.
- **FD\_CLOSE:** Only valid on connection-oriented sockets (for example, **SOCK\_STREAM**)
  1. When **WSAAsyncSelect** called, if socket connection has been closed.
  2. After remote system initiated graceful close, when no data currently available to receive (note: if data has been received and is waiting to be read when the remote system initiates a graceful close, the FD\_CLOSE is not delivered until all pending data has been read).
  3. After local system initiates graceful close with **shutdown** and remote system has responded with “End of Data” notification (for example, TCP FIN), when no data currently available to receive.
  4. When remote system terminates connection (for example, sent TCP RST), and *IParam* will contain WSAECONNRESET error value.

---

**Note** FD\_CLOSE is *not* posted after **closesocket** is called.

---

- **FD\_QOS:**
  1. When **WSAAsyncSelect** called, if the quality of service associated with the socket has been changed.
  2. After **WSAIoctl** with SIO\_GET\_QOS called, when the quality of service is changed.

- **FD\_GROUP\_QOS** Reserved.
- **FD\_ROUTING\_INTERFACE\_CHANGE:**  
After **WSAIoctl** with **SIO\_ROUTING\_INTERFACE\_CHANGE** called, when the local interface that should be used to reach the destination specified in the IOCTL changes.
- **FD\_ADDRESS\_LIST\_CHANGE:**  
After **WSAIoctl** with **SIO\_ADDRESS\_LIST\_CHANGE** called, when the list of local addresses to which the application can bind changes.

#### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **select**, **WSAEventSelect**

## WSACancelAsyncRequest

The Windows Sockets **WSACancelAsyncRequest** function cancels an incomplete asynchronous operation.

```
int WSACancelAsyncRequest (
    HANDLE  hAsyncTaskHandle
);
```

### Parameters

*hAsyncTaskHandle*

[in] Handle that specifies the asynchronous operation to be canceled.

### Return Values

The value returned by **WSACancelAsyncRequest** is zero if the operation was successfully canceled. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that the specified asynchronous task handle was invalid.

(continued)



*(continued)*

Error code	Meaning
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	The asynchronous routine being canceled has already completed.

---

**Note** It is unclear whether the application can usefully distinguish between `WSAEINVAL` and `WSAEALREADY`, since in both cases the error indicates that there is no asynchronous operation in progress with the indicated handle. [Trivial exception: zero is always an invalid asynchronous task handle.] The Windows Sockets specification does not prescribe how a conformant Windows Sockets provider should distinguish between the two cases. For maximum portability, a Windows Sockets application should treat the two errors as equivalent.

---

### Remarks

The **`WSACancelAsyncRequest`** function is used to cancel an asynchronous operation that was initiated by one of the **`WSAAsyncGetXByY`** functions such as **`WSAAsyncGetHostByName`**. The operation to be canceled is identified by the *hAsyncTaskHandle* parameter, which should be set to the asynchronous task handle as returned by the initiating **`WSAAsyncGetXByY`** function.

An attempt to cancel an existing asynchronous **`WSAAsyncGetXByY`** operation can fail with an error code of `WSAEALREADY` for two reasons. First, the original operation has already completed and the application has dealt with the resultant message. Second, the original operation has already completed but the resultant message is still waiting in the application window queue.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **`WSAAsyncGetHostByAddr`**, **`WSAAsyncGetHostByName`**, **`WSAAsyncGetProtoByName`**, **`WSAAsyncGetProtoByNumber`**, **`WSAAsyncGetServByName`**, **`WSAAsyncGetServByPort`**

---

## WSACancelBlockingCall

The **WSACancelBlockingCall** function has been removed in compliance with the Windows Sockets 2 specification, revision 2.2.0.

The function is not exported directly by the `Ws2_32.dll` and Windows Sockets 2 applications should not use this function. Windows Sockets 1.1 applications that call this function are still supported through the `Winsock.dll` and `Wsock32.dll`.

Blocking hooks are generally used to keep a single-threaded GUI application responsive during calls to blocking functions. Instead of using blocking hooks, an applications should use a separate thread (separate from the main GUI thread) for network activity.

### ! Requirements

**Version:** Requires Windows Sockets 1.1. Obsolete for Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

---

## WSACleanup

The Windows Sockets **WSACleanup** function terminates use of the `Ws2_32.dll`.

```
int WSACleanup (void);
```

### Parameters

This function has no parameters.

### Return Values

The return value is zero if the operation was successful. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Attempting to call **WSACleanup** from within a blocking hook and then failing to check the return code is a common programming error in Windows Socket 1.1 applications. If an application needs to quit while a blocking call is outstanding, the application must first cancel the blocking call with **WSACancelBlockingCall** then issue the **WSACleanup** call once control has been returned to the application.

In a multithreaded environment, **WSACleanup** terminates Windows Sockets operations for all threads.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.

### Remarks

An application or DLL is required to perform a successful **WSAStartup** call before it can use Windows Sockets services. When it has completed the use of Windows Sockets, the application or DLL must call **WSACleanup** to deregister itself from a Windows Sockets implementation and allow the implementation to free any resources allocated on behalf of the application or DLL. Any pending blocking or asynchronous calls issued by any thread in this process are canceled without posting any notification messages or without signaling any event objects. Any pending overlapped send and receive operations (**WSASend/WSASendTo/WSARecv/WSARecvFrom** with an overlapped socket) issued by any thread in this process are also canceled without setting the event object or invoking the completion routine, if specified. In this case, the pending overlapped operations fail with the error status `WSA_OPERATION_ABORTED`.

Sockets that were open when **WSACleanup** was called are reset and automatically deallocated as if **closesocket** were called; sockets that have been closed with **closesocket** but that still have pending data to be sent can be affected—the pending data can be lost if the `Ws2_32.dll` is unloaded from memory as the application exits. To insure that all pending data is sent, an application should use **shutdown** to close the connection, then wait until the close completes before calling **closesocket** and **WSACleanup**. All resources and internal state, such as queued unposted-posted messages, must be deallocated so as to be available to the next user.

There must be a call to **WSACleanup** for every successful call to **WSAStartup** made by a task. Only the final **WSACleanup** for that task does the actual cleanup; the preceding calls simply decrement an internal reference count in the `Ws2_32.dll`.

#### Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

#### See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **closesocket**, **shutdown**, **WSAStartup**

# WSACloseEvent

The Windows Sockets **WSACloseEvent** function closes an open event object handle.

```
BOOL WSACloseEvent (
    WSAEVENT  hEvent
);
```

## Parameters

*hEvent*

[in] Object handle identifying the open event.

## Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_INVALID_HANDLE	The <i>hEvent</i> is not a valid event object handle.

## Remarks

The handle to the event object is closed so that further references to this handle will fail with the error **WSA\_INVALID\_HANDLE**.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSACreateEvent**, **WSAEnumNetworkEvents**, **WSAEventSelect**, **WSAGetOverlappedResult**, **WSARecv**, **WSARecvFrom**, **WSAResetEvent**, **WSASend**, **WSASendTo**, **WSASetEvent**, **WSAWaitForMultipleEvents**

# WSAConnect

The Windows Sockets **WSAConnect** function establishes a connection to another socket application, exchanges connect data, and specifies needed quality of service based on the supplied **FLOWSPEC** structure.

```
int WSAConnect (
    SOCKET          s,
    const struct sockaddr FAR *name,
    int            namelen,
    LPWSABUF       lpCallerData,
    LPWSABUF       lpCalleeData,
    LPQOS          lpSQOS,
    LPQOS          lpGQOS
);
```

## Parameters

*s*

[in] Descriptor identifying an unconnected socket.

*name*

[in] Name of the socket in the other application to which to connect.

*namelen*

[in] Length of the *name*.

*lpCallerData*

[in] Pointer to the user data that is to be transferred to the other socket during connection establishment.

*lpCalleeData*

[out] Pointer to the user data that is to be transferred back from the other socket during connection establishment.

*lpSQOS*

[in] Pointer to the **FLOWSPEC** structures for socket *s*, one for each direction.

*lpGQOS*

[in] Reserved. Should be NULL.

## Return Values

If no error occurs, **WSAConnect** returns zero. Otherwise, it returns **SOCKET\_ERROR**, and a specific error code can be retrieved by calling **WSAGetLastError**. On a blocking socket, the return value indicates success or failure of the connection attempt.

With a nonblocking socket, the connection attempt cannot be completed immediately. In this case, **WSAConnect** will return **SOCKET\_ERROR**, and **WSAGetLastError** will return **WSAEWOULDBLOCK**; the application could therefore:

- Use **select** to determine the completion of the connection request by checking if the socket is writeable.

- If your application is using **WSAAsyncSelect** to indicate interest in connection events, then your application will receive an `FD_CONNECT` notification when the connect operation is complete (successful or not).
- If your application is using **WSAEventSelect** to indicate interest in connection events, then the associated event object will be signaled when the connect operation is complete (successful or not).

For a nonblocking socket, until the connection attempt completes all subsequent calls to **WSAConnect** on the same socket will fail with the error code `WSAEALREADY`.

If the return error code indicates the connection attempt failed (that is, `WSAECONNREFUSED`, `WSAENETUNREACH`, `WSAETIMEDOUT`) the application can call **WSAConnect** again for the same socket.

Error code	Meaning
<code>WSANOTINITIALISED</code>	A successful <b>WSAStartup</b> call must occur before using this function.
<code>WSAENETDOWN</code>	The network subsystem has failed.
<code>WSAEADDRINUSE</code>	The local address of the socket is already in use and the socket was not marked to allow address reuse with <code>SO_REUSEADDR</code> . This error usually occurs during the execution of <b>bind</b> , but could be delayed until this function if the <b>bind</b> function operates on a partially wildcard address (involving <code>ADDR_ANY</code> ) and if a specific address needs to be “committed” at the time of this function.
<code>WSAEINTR</code>	The (blocking) Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
<code>WSAEINPROGRESS</code>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<code>WSAEALREADY</code>	A nonblocking <b>connect/WSAConnect</b> call is in progress on the specified socket.
<code>WSAEADDRNOTAVAIL</code>	The remote address is not a valid address (such as <code>ADDR_ANY</code> ).
<code>WSAEAFNOSUPPORT</code>	Addresses in the specified family cannot be used with this socket.
<code>WSAECONNREFUSED</code>	The attempt to connect was rejected.
<code>WSAEFAULT</code>	The <i>name</i> or the <i>namelen</i> parameter is not a valid part of the user address space, the <i>namelen</i> parameter is too small, the buffer length for <i>lpCalleeData</i> , <i>lpSQOS</i> , and <i>lpGQOS</i> are too small, or the buffer length for <i>lpCallerData</i> is too large.
<code>WSAEINVAL</code>	The parameter <i>s</i> is a listening socket.
<code>WSAEISCONN</code>	The socket is already connected (connection-oriented sockets only).
<code>WSAENETUNREACH</code>	The network cannot be reached from this host at this time.
<code>WSAENOBUFS</code>	No buffer space is available. The socket cannot be connected.

(continued)

*(continued)*

Error code	Meaning
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The <b>FLOWSPEC</b> structures specified in <i>lpSQOS</i> and <i>lpGQOS</i> cannot be satisfied.
WSAEPROTONOSUPPORT	The <i>lpCallerData</i> argument is not supported by the service provider.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the connection cannot be completed immediately.
WSAEACCES	Attempt to connect datagram socket to broadcast address failed because <b>setsockopt</b> <code>SO_BROADCAST</code> is not enabled.

### Remarks

The **WSAConnect** function is used to create a connection to the specified destination, and to perform a number of other ancillary operations that occur at connect time. If the socket, *s*, is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound.

For connection-oriented sockets (for example, type **SOCK\_STREAM**), an active connection is initiated to the foreign host using *name* (an address in the name space of the socket; for a detailed description, please see **bind**). When this call completes successfully, the socket is ready to send/receive data. If the address parameter of the *name* structure is all zeroes, **WSAConnect** will return the error `WSAEADDRNOTAVAIL`. Any attempt to reconnect an active connection will fail with the error code `WSAEISCONN`.

For connection-oriented, nonblocking sockets, it is often not possible to complete the connection immediately. In such cases, this function returns the error `WSAEWOULDBLOCK`. However, the operation proceeds. When the success or failure outcome becomes known, it may be reported in one of several ways depending on how the client registers for notification. If the client uses **select**, success is reported in the *writelfds* set and failure is reported in the *exceptfds* set. If the client uses **WSAAsyncSelect** or **WSAEventSelect**, the notification is announced with `FD_CONNECT` and the error code associated with the `FD_CONNECT` indicates either success or a specific reason for failure.

For a connectionless socket (for example, type **SOCK\_DGRAM**), the operation performed by **WSAConnect** is merely to establish a default destination address so that the socket can be used on subsequent connection-oriented send and receive operations (**send**, **WSASend**, **recv**, and **WSARecv**). Any datagrams received from an address other than the destination address specified will be discarded. If the entire name structure is all zeros (not just the address parameter of the name structure), then the socket will be disconnected. Then, the default remote address will be indeterminate, so **send/WSASend** and **recv/WSARecv** calls will return the error code `WSAENOTCONN`. However, **sendto/WSASendTo** and **recvfrom/WSARecvFrom** can still be used.

The default destination can be changed by simply calling **WSAConnect** again, even if the socket is already connected. Any datagrams queued for receipt are discarded if *name* is different from the previous **WSAConnect**.

For connectionless sockets, *name* can indicate any valid address, including a broadcast address. However, to connect to a broadcast address, a socket must have **setsockopt** **SO\_BROADCAST** enabled. Otherwise, **WSAConnect** will fail with the error code **WSAEACCES**.

On connectionless sockets, exchange of user-to-user data is not possible and the corresponding parameters will be silently ignored.

The application is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specifies.

The *lpCallerData* is a value parameter that contains any user data that is to be sent along with the connection request. If *lpCallerData* is NULL, no user data will be passed to the peer. The *lpCalleeData* is a result parameter that will contain any user data passed back from the other socket as part of the connection establishment in a **WSABUF** structure. The member *lpCalleeData->len* initially contains the length of the buffer allocated by the application and pointed to by *lpCalleeData->buf*. *lpCalleeData->len* will be set to zero if no user data has been passed back. The *lpCalleeData* information will be valid when the connection operation is complete. For blocking sockets, the connection operation completes when the **WSAConnect** function returns. For nonblocking sockets, completion will be after the **FD\_CONNECT** notification has occurred. If *lpCalleeData* is NULL, no user data will be passed back. The exact format of the user data is specific to the address family to which the socket belongs.

At connect time, an application can use the *lpSQOS* parameter to override any previous quality of service specification made for the socket through **WSAIoctl** with the **SIO\_SET\_QOS** opcode.

*lpSQOS* specifies the **FLOWSPEC** structures for socket *s*, one for each direction, followed by any additional provider-specific parameters. If either the associated transport provider in general or the specific type of socket in particular cannot honor the quality of service request, an error will be returned as indicated in the following. The sending or receiving flow specification values will be ignored, respectively, for any unidirectional sockets. If no provider-specific parameters are supplied, the *buf* and *len* parameters of *lpSQOS->ProviderSpecific* should be set to NULL and zero, respectively. A NULL value for *lpSQOS* indicates no application supplied quality of service.

When connected sockets become closed for whatever reason, they should be discarded and recreated. It is safest to assume that when things go awry for any reason on a connected socket, the application must discard and recreate the needed sockets in order to return to a stable point.



**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **accept**, **bind**, **connect**, **getsockname**, **getsockopt**, **select**, **socket**, **WSAAsyncSelect**, **WSAEventSelect**

---

## WSACreateEvent

The Windows Sockets **WSACreateEvent** function creates a new event object.

```
WSAEVENT WSACreateEvent (void);
```

### Parameters

This function has no parameters.

### Return Values

If no error occurs, **WSACreateEvent** returns the handle of the event object. Otherwise, the return value is `WSA_INVALID_EVENT`. To get extended error information, call **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to create the event object.

### Remarks

The **WSACreateEvent** function is used to create an event object created that is manual reset with an initial state of nonsignaled. Windows Sockets 2 event objects are system objects in Win32 environments. Therefore, if a Win32 application desires auto reset events, it can call the native **WSACreateEvent** Win32 function directly. The scope of an event object is limited to the process in which it is created.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSACloseEvent**, **WSAEnumNetworkEvents**, **WSAEventSelect**, **WSAGetOverlappedResult**, **WSARecv**, **WSARecvFrom**, **WSAResetEvent**, **WSASend**, **WSASendTo**, **WSASetEvent**, **WSAWaitForMultipleEvents**

## WSADuplicateSocket

The Windows Sockets **WSADuplicateSocket** function returns a **WSAPROTOCOL\_INFO** structure that can be used to create a new socket descriptor for a shared socket. The **WSADuplicateSocket** function cannot be used on a QOS-enabled socket.

```
int WSADuplicateSocket (
    SOCKET                s,
    DWORD                dwProcessId,
    LPWSAPROTOCOL_INFO  lpProtocolInfo
);
```

### Parameters

*s*

[in] Descriptor identifying the local socket.

*dwProcessId*

[in] Process identifier of the target process in which the duplicated socket will be used.

*lpProtocolInfo*

[out] Pointer to a buffer, allocated by the client, that is large enough to contain a **WSAPROTOCOL\_INFO** structure. The service provider copies the protocol information structure contents to this buffer.

### Return Values

If no error occurs, **WSADuplicateSocket** returns zero. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lpProtocolInfo</i> argument is not a valid part of the user address space.

### Remarks

The **WSADuplicateSocket** function is used to enable socket sharing between processes. A source process calls **WSADuplicateSocket** to obtain a special **WSAPROTOCOL\_INFO** structure. It uses some interprocess communications (IPC) mechanism to pass the contents of this structure to a target process, which in turn uses it in a call to **WSASocket** to obtain a descriptor for the duplicated socket. The special **WSAPROTOCOL\_INFO** structure can only be used once by the target process.

Sockets can be shared among threads in a given process without using the **WSADuplicateSocket** function because a socket descriptor is valid in all threads of a process.

One possible scenario for establishing and handing off a shared socket is illustrated in the following table.

Source process	IPC	Destination process
1) <b>WSASocket</b> , <b>WSAConnect</b>		
2) Request target process identifier	⇒	
		3) Receive process identifier request and respond
4) Receive process identifier	⇐	
5) Call <b>WSADuplicateSocket</b> to get a special <b>WSAPROTOCOL_INFO</b> structure		

Source process	IPC	Destination process
6) Send <b>WSAPROTOCOL_INFO</b> structure to target	⇒	7) Receive <b>WSAPROTOCOL_INFO</b> structure 8) Call <b>WSASocket</b> to create shared socket descriptor.
10) <b>closesocket</b>		9) Use shared socket for data exchange

The descriptors that reference a shared socket can be used independently for I/O. However, the Windows Sockets interface does not implement any type of access control, so it is up to the processes involved to coordinate their operations on a shared socket. Shared sockets are typically used to having one process that is responsible for creating sockets and establishing connections, and other processes that are responsible for information exchange.

All of the state information associated with a socket is held in common across all the descriptors because the socket descriptors are duplicated and not the actual socket. For example, a **setsockopt** operation performed using one descriptor is subsequently visible using a **getsockopt** from any or all descriptors. A process can call **closesocket** on a duplicated socket and the descriptor will become deallocated. The underlying socket, however, will remain open until **closesocket** is called by the last remaining descriptor.

Notification on shared sockets is subject to the usual constraints of **WSAAsyncSelect** and **WSAEventSelect**. Issuing either of these calls using any of the shared descriptors cancels any previous event registration for the socket, regardless of which descriptor was used to make that registration. Thus, a shared socket cannot deliver **FD\_READ** events to process A and **FD\_WRITE** events to process B. For situations when such tight coordination is required, developers would be advised to use threads instead of separate processes.

---

**Note** The **WSADuplicateSocket** function cannot be used on a QOS-enabled socket. Attempting to do so will result in the return of a **WSAEINVAL** error.

---

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

**Unicode:** Implemented as Unicode and ANSI versions on all platforms.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSASocket**

## WSAEnumNameSpaceProviders

The Windows Sockets **WSAEnumNameSpaceProviders** function retrieves information about available name spaces.

```
INT WINAPI WSAEnumNameSpaceProviders (
    IN OUT LPDWORD          lpdwBufferLength,
    OUT LPWSANAMESPACE_INFO lpnspBuffer
);
```

### Parameters

#### *lpdwBufferLength*

[in/out] On input, the number of bytes contained in the buffer pointed to by *lpnspBuffer*. On output (if the function fails, and the error is WSAEFAULT), the minimum number of bytes to pass for the *lpnspBuffer* to retrieve all the requested information. The passed-in buffer must be sufficient to hold all of the name space information.

#### *lpnspBuffer*

[out] Buffer that is filled with **WSANAMESPACE\_INFO** structures. The returned structures are located consecutively at the head of the buffer. Variable sized information referenced by pointers in the structures point to locations within the buffer located between the end of the fixed sized structures and the end of the buffer. The number of structures filled in is the return value of **WSAEnumNameSpaceProviders**.

### Return Values

The **WSAEnumNameSpaceProviders** function returns the number of **WSANAMESPACE\_INFO** structures copied into *lpnspBuffer*. Otherwise, the value **SOCKET\_ERROR** is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSAEFAULT	The buffer length was too small to receive all the relevant <b>WSANAMESPACE_INFO</b> structures and associated information. Passes in a buffer at least as large as the value returned in <i>lpdwBufferLength</i> .
WSANOTINITIALIZED	The Ws2_32.dll has not been initialized. The application must first call <b>WSAStartup</b> before calling any Windows Sockets functions.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on all platforms.

## WSAEnumNetworkEvents

The Windows Sockets **WSAEnumNetworkEvents** function discovers occurrences of network events for the indicated socket, clear internal network event records, and reset event objects (optional).

```
int WSAEnumNetworkEvents (
    SOCKET          s,
    WSAEVENT       hEventObject,
    LPWSANETWORKEVENTS lpNetworkEvents
);
```

### Parameters

*s*

[in] Descriptor identifying the socket.

*hEventObject*

[in] Optional handle identifying an associated event object to be reset.

*lpNetworkEvents*

[out] Pointer to a **WSANETWORKEVENTS** structure that is filled with a record of network events that occurred and any associated error codes.

### Return Values

The return value is zero if the operation was successful. Otherwise, the value **SOCKET\_ERROR** is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lpNetworkEvents</i> argument is not a valid part of the user address space.

## Remarks

The **WSAEnumNetworkEvents** function is used to discover which network events have occurred for the indicated socket since the last invocation of this function. It is intended for use in conjunction with **WSAEventSelect**, which associates an event object with one or more network events. The recording of network events commences when **WSAEventSelect** is called with a nonzero *INetworkEvents* parameter and remains in effect until another call is made to **WSAEventSelect** with the *INetworkEvents* parameter set to zero, or until a call is made to **WSAAsyncSelect**.

**WSAEnumNetworkEvents** only reports network activity and errors nominated through **WSAEventSelect**. See the descriptions of **select** and **WSAAsyncSelect** to find out how those functions report network activity and errors.

The socket's internal record of network events is copied to the structure referenced by *lpNetworkEvents*, after which the internal network events record is cleared. If the *hEventObject* parameter is not NULL, the indicated event object is also reset. The Windows Sockets provider guarantees that the operations of copying the network event record, clearing it and resetting any associated event object are automatic, such that the next occurrence of a nominated network event will cause the event object to become set. In the case of this function returning SOCKET\_ERROR, the associated event object is not reset and the record of network events is not cleared.

The *INetworkEvents* member of the **WSANETWORKEVENTS** structure indicates which of the FD\_XXX network events have occurred. The *iErrorCode* array is used to contain any associated error codes with the array index corresponding to the position of event bits in *INetworkEvents*. Identifiers such as FD\_READ\_BIT and FD\_WRITE\_BIT can be used to index the *iErrorCode* array. Note that only those elements of the *iErrorCode* array are set that correspond to the bits set in *INetworkEvents* parameter. Other parameters are not modified (this is important for backward compatibility with the applications that are not aware of new FD\_ROUTING\_INTERFACE\_CHANGE and FD\_ADDRESS\_LIST\_CHANGE events).

The following error codes can be returned along with the corresponding network event.

### Event: FD\_CONNECT

Error code	Meaning
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection

**Event: FD\_CLOSE**

Error code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was terminated due to a time-out or other failure.

**Event: FD\_READ****Event: FD\_WRITE****Event: FD\_OOB****Event: FD\_ACCEPT****Event: FD\_QOS****Event: FD\_GROUP\_QOS****Event: FD\_ADDRESS\_LIST\_CHANGE**

Error code	Meaning
WSAENETDOWN	The network subsystem has failed.

**Event: FD\_ROUTING\_INTERFACE\_CHANGE**

Error code	Meaning
WSAENETUNREACH	The specified destination is no longer reachable.
WSAENETDOWN	The network subsystem has failed.

**! Requirements**

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+ See Also**

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSAEventSelect**

---

## WSAEnumProtocols

The Windows Sockets **WSAEnumProtocols** function retrieves information about available transport protocols.



```

int WSAEnumProtocols (
    LPINT          lpiProtocols,
    LPWSAPROTOCOL_INFO lpProtocolBuffer,
    ILPDWORD       lpdwBufferLength
);

```

## Parameters

### *lpiProtocols*

[in] Null-terminated array of iProtocol values. This parameter is optional; if *lpiProtocols* is NULL, information on all available protocols is returned. Otherwise, information is retrieved only for those protocols listed in the array.

### *lpProtocolBuffer*

[out] Buffer that is filled with **WSAPROTOCOL\_INFO** structures.

### *lpdwBufferLength*

[in/out] On input, the count of bytes in the *lpProtocolBuffer* buffer passed to **WSAEnumProtocols**. On output, the minimum buffer size that can be passed to **WSAEnumProtocols** to retrieve all the requested information. This routine has no ability to enumerate over multiple calls; the passed-in buffer must be large enough to hold all entries in order for the routine to succeed. This reduces the complexity of the API and should not pose a problem because the number of protocols loaded on a machine is typically small.

## Return Values

If no error occurs, **WSAEnumProtocols** returns the number of protocols to be reported. Otherwise, a value of **SOCKET\_ERROR** is returned and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAENOBUFS	The buffer length was too small to receive all the relevant <b>WSAPROTOCOL_INFO</b> structures and associated information. Pass in a buffer at least as large as the value returned in <i>lpdwBufferLength</i> .
WSAEFAULT	One or more of the <i>lpiProtocols</i> , <i>lpProtocolBuffer</i> , or <i>lpdwBufferLength</i> arguments are not a valid part of the user address space.

## Remarks

The **WSAEnumProtocols** function is used to discover information about the collection of transport protocols and protocol chains installed on the local machine. Since layered protocols are only usable by applications when installed in protocol chains, information on layered protocols is not included in *lpProtocolBuffer*. The *lpiProtocols* parameter can be used as a filter to constrain the amount of information provided. Often, *lpiProtocols* will be supplied as a NULL pointer that will cause the function to return information on all available transport protocols and protocol chains.

A **WSAPROTOCOL\_INFO** structure is provided in the buffer pointed to by *lpProtocolBuffer* for each requested protocol. If the supplied buffer is not large enough (as indicated by the input value of *lpdwBufferLength*), the value pointed to by *lpdwBufferLength* will be updated to indicate the required buffer size. The application should then obtain a large enough buffer and call this **WSAEnumProtocols** again.

The order in which the **WSAPROTOCOL\_INFO** structures appear in the buffer coincides with the order in which the protocol entries were registered by the service provider using the *Ws2\_32.dll*, or with any subsequent reordering that occurred through the Windows Sockets applet or DLL supplied for establishing default TCP/IP providers.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Winsock2.h*.

**Library:** Use *Ws2\_32.lib*.

**Unicode:** Implemented as Unicode and ANSI versions on all platforms.

### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions

---

## WSAEventSelect

The Windows Sockets **WSAEventSelect** function specifies an event object to be associated with the supplied set of FD\_XXX network events.

```
int WSAEventSelect (
    SOCKET      s,
    WSAEVENT   hEventObject,
    long        lNetworkEvents
);
```

### Parameters

*s*

[in] Descriptor identifying the socket.

*hEventObject*

[in] Handle identifying the event object to be associated with the supplied set of FD\_XXX network events.

*INetworkEvents*

[in] Bitmask that specifies the combination of FD\_XXX network events in which the application has interest.

**Return Values**

The return value is zero if the application's specification of the network events and the associated event object was successful. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

As in the case of the **select** and **WSAAsyncSelect** functions, **WSAEventSelect** will frequently be used to determine when a data transfer operation (**send** or **recv**) can be issued with the expectation of immediate success. Nevertheless, a robust application must be prepared for the possibility that the event object is set and it issues a Windows Sockets call that returns `WSAEWOULDBLOCK` immediately. For example, the following sequence of operations is possible:

- Data arrives on socket *s*; Windows Sockets sets the **WSAEventSelect** event object.
- The application does some other processing.
- While processing, the application issues an **ioctlsocket(s, FIONREAD...)** and notices that there is data ready to be read.
- The application issues a **recv(s,...)** to read the data.
- The application eventually waits on the event object specified in **WSAEventSelect**, which returns immediately indicating that data is ready to read.
- The application issues **recv(s,...)**, which fails with the error `WSAEWOULDBLOCK`.

Having successfully recorded the occurrence of the network event (by setting the corresponding bit in the internal network event record) and signaled the associated event object, no further actions are taken for that network event until the application makes the function call that implicitly reenables the setting of that network event and signaling of the associated event object.

<b>Network event</b>	<b>Re-enabling function</b>
FD_READ	<b>recv</b> , <b>recvfrom</b> , <b>WSARecv</b> , or <b>WSARecvFrom</b> .
FD_WRITE	<b>send</b> , <b>sendto</b> , <b>WSASend</b> , or <b>WSASendTo</b> .
FD_OOB	<b>recv</b> , <b>recvfrom</b> , <b>WSARecv</b> , or <b>WSARecvFrom</b> .
FD_ACCEPT	<b>accept</b> or <b>WSAAccept</b> unless the error code returned is <code>WSATRY_AGAIN</code> indicating that the condition function returned <code>CF_DEFER</code> .
FD_CONNECT	None.

Network event	Re-enabling function
FD_CLOSE	None.
FD_QOS	<b>WSAIoctl</b> with command SIO_GET_QOS.
FD_GROUP_QOS	Reserved.
FD_ROUTING_INTERFACE_CHANGE	<b>WSAIoctl</b> with command SIO_ROUTING_INTERFACE_CHANGE.
FD_ADDRESS_LIST_CHANGE	<b>WSAIoctl</b> with command SIO_ADDRESS_LIST_CHANGE.

Any call to the reenabling routine, even one that fails, results in reenabling of recording and signaling for the relevant network event and event object.

For FD\_READ, FD\_OOB, and FD\_ACCEPT network events, network event recording and event object signaling are level-triggered. This means that if the reenabling routine is called and the relevant network condition is still valid after the call, the network event is recorded and the associated event object is set. This allows an application to be event-driven and not be concerned with the amount of data that arrives at any one time. Consider the following sequence:

1. Transport provider receives 100 bytes of data on socket *s* and causes *Ws2\_32.dll* to record the FD\_READ network event and set the associated event object.
2. The application issues **recv( s, buffptr, 50, 0)** to read 50 bytes.
3. The transport provider causes *Ws2\_32.dll* to record the FD\_READ network event and sets the associated event object again since there is still data to be read.

With these semantics, an application need not read all available data in response to an FD\_READ network event—a single **recv** in response to each FD\_READ network event is appropriate.

The FD\_QOS event is considered edge triggered. A message will be posted exactly once when a quality of service change occurs. Further messages will *not* be forthcoming until either the provider detects a further change in quality of service or the application renegotiates the quality of service for the socket.

The FD\_ROUTING\_INTERFACE\_CHANGE and FD\_ADDRESS\_LIST\_CHANGE events are considered edge triggered as well. A message will be posted exactly once when a change occurs *after* the application has requested the notification by issuing **WSAIoctl** with SIO\_ROUTING\_INTERFACE\_CHANGE or SIO\_ADDRESS\_LIST\_CHANGE correspondingly. Other messages will not be forthcoming until the application reissues the **IOCTL** and another change is detected since the **IOCTL** has been issued.

If a network event has already happened when the application calls **WSAEventSelect** or when the reenabling function is called, then a network event is recorded and the associated event object is set as appropriate. For example, consider the following sequence:

1. An application calls **listen**.
2. A connect request is received but not yet accepted.
3. The application calls **WSAEventSelect** specifying that it is interested in the FD\_ACCEPT network event for the socket. Due to the persistence of network events, Windows Sockets records the FD\_ACCEPT network event and sets the associated event object immediately.

The FD\_WRITE network event is handled slightly differently. An FD\_WRITE network event is recorded when a socket is first connected with **connect/WSAConnect** or accepted with **accept/WSAAccept**, and then after a send fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD\_WRITE network event setting and lasting until a send returns WSAEWOULDBLOCK. After such a failure the application will find out that sends are again possible when an FD\_WRITE network event is recorded and the associated event object is set.

The FD\_OOB network event is used only when a socket is configured to receive OOB data separately. If the socket is configured to receive OOB data inline, the OOB (expedited) data is treated as normal data and the application should register an interest in, and will get FD\_READ network event, not FD\_OOB network event. An application can set or inspect the way in which OOB data is to be handled by using **setsockopt** or **getsockopt** for the SO\_OOBINLINE option.

The error code in an FD\_CLOSE network event indicates whether the socket close was graceful or abortive. If the error code is zero, then the close was graceful; if the error code is WSAECONNRESET, then the socket's virtual circuit was reset. This only applies to connection-oriented sockets such as **SOCK\_STREAM**.

The FD\_CLOSE network event is recorded when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the FD\_CLOSE is recorded when the connection goes into the TIME\_WAIT or CLOSE\_WAIT states. This results from the remote end performing a **shutdown** on the send side or a **closesocket**. FD\_CLOSE being posted after all data is read from a socket. An application should check for remaining data upon receipt of FD\_CLOSE to avoid any possibility of losing data.

Please note Windows Sockets will record only an FD\_CLOSE network event to indicate closure of a virtual circuit. It will *not* record an FD\_READ network event to indicate this condition.

The FD\_QOS network event is recorded when any parameter in the flow specification associated with socket *s*. Applications should use **WSAIoctl** with command SIO\_GET\_QOS to get the current QOS for socket *s*.

The FD\_ROUTING\_INTERFACE\_CHANGE network event is recorded when the local interface that should be used to reach the destination specified in **WSAIoctl** with SIO\_ROUTING\_INTERFACE\_CHANGE changes *after* such IOCTL has been issued.

The `FD_ADDRESS_LIST_CHANGE` network event is recorded when the list of addresses of protocol family for the socket to which the application can bind changes after **WSAIoctl** with `SIO_ADDRESS_LIST_CHANGE` has been issued.

Error code	Meaning
<code>WSANOTINITIALISED</code>	A successful <b>WSAStartup</b> call must occur before using this function.
<code>WSAENETDOWN</code>	The network subsystem has failed.
<code>WSAEINVAL</code>	Indicates that one of the specified parameters was invalid, or the specified socket is in an invalid state.
<code>WSAEINPROGRESS</code>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<code>WSAENOTSOCK</code>	The descriptor is not a socket.

### Remarks

The **WSAEventSelect** function is used to specify an event object, *hEventObject*, to be associated with the selected `FD_XXX` network events, *INetworkEvents*. The socket for which an event object is specified is identified by the *s* parameter. The event object is set when any of the nominated network events occur.

The **WSAEventSelect** function operates very similarly to **WSAAsyncSelect**, the difference being the actions taken when a nominated network event occurs. The **WSAAsyncSelect** function causes an application-specified Windows message to be posted. The **WSAEventSelect** sets the associated event object and records the occurrence of this event in an internal network event record. An application can use **WSAWaitForMultipleEvents** to wait or poll on the event object, and use **WSAEnumNetworkEvents** to retrieve the contents of the internal network event record and thus determine which of the nominated network events have occurred.

**WSAEventSelect** is the only function that causes network activity and errors to be recorded and retrievable through **WSAEnumNetworkEvents**. See the descriptions of **select** and **WSAAsyncSelect** to find out how those functions report network activity and errors.

The **WSAEventSelect** function automatically sets socket *s* to nonblocking mode, regardless of the value of *INetworkEvents*. See **ioctlsocket/WSAIoctl** about how to set the socket back to blocking mode.

The *INetworkEvents* parameter is constructed by using the bitwise OR operator with any of the values specified in the following table.

Value	Meaning
<code>FD_READ</code>	Wants to receive notification of readiness for reading.
<code>FD_WRITE</code>	Wants to receive notification of readiness for writing.

(continued)

*(continued)*

Value	Meaning
FD_OOB	Wants to receive notification of the arrival of OOB data.
FD_ACCEPT	Wants to receive notification of incoming connections.
FD_CONNECT	Wants to receive notification of completed connection or multipoint join operation.
FD_CLOSE	Wants to receive notification of socket closure.
FD_QOS	Wants to receive notification of socket (QOS changes.
FD_GROUP_QOS	Reserved.
FD_ROUTING_INTERFACE_CHANGE	Wants to receive notification of routing interface changes for the specified destination.
FD_ADDRESS_LIST_CHANGE	Wants to receive notification of local address list changes for the address family of the socket.

Issuing a **WSAEventSelect** for a socket cancels any previous **WSAAsyncSelect** or **WSAEventSelect** for the same socket and clears the internal network event record. For example, to associate an event object with both reading and writing network events, the application must call **WSAEventSelect** with both **FD\_READ** and **FD\_WRITE**, as follows:

```
rc = WSAEventSelect(s, hEventObject, FD_READ|FD_WRITE);
```

It is not possible to specify different event objects for different network events. The following code will *not* work; the second call will cancel the effects of the first, and only the **FD\_WRITE** network event will be associated with *hEventObject2*:

```
rc = WSAEventSelect(s, hEventObject1, FD_READ);
rc = WSAEventSelect(s, hEventObject2, FD_WRITE); //bad
```

To cancel the association and selection of network events on a socket, *INetworkEvents* should be set to zero, in which case the *hEventObject* parameter will be ignored.

```
rc = WSAEventSelect(s, hEventObject, 0);
```

Closing a socket with **closesocket** also cancels the association and selection of network events specified in **WSAEventSelect** for the socket. The application, however, still must call **WSACloseEvent** to explicitly close the event object and free any resources.

The socket created when the **accept** function is called has the same properties as the listening socket used to accept it. Any **WSAEventSelect** association and network events selection set for the listening socket apply to the accepted socket. For example, if a listening socket has **WSAEventSelect** association of *hEventObject* with **FD\_ACCEPT**, **FD\_READ**, and **FD\_WRITE**, then any socket accepted on that listening socket will also have **FD\_ACCEPT**, **FD\_READ**, and **FD\_WRITE** network events associated with the same *hEventObject*. If a different *hEventObject* or network events are desired, the application should call **WSAEventSelect**, passing the accepted socket and the desired new information.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSAAsyncSelect**, **WSACloseEvent**, **WSACreateEvent**, **WSAEnumNetworkEvents**, **WSAWaitForMultipleEvents**

---

## WSAGetLastError

The Windows Sockets **WSAGetLastError** function gets the error status for the last operation that failed.

```
int WSAGetLastError (void);
```

### Parameters

This function has no parameters.

### Return Values

The return value indicates the error code for this thread's last Windows Sockets operation that failed.

### Remarks

The **WSAGetLastError** function returns the last network error that occurred. When a particular Windows Sockets function indicates that an error has occurred, this function should be called to retrieve the appropriate error code. This error code can be different from the error code obtained from **getsockopt** SO\_ERROR, which is socket-specific since **WSAGetLastError** is for all thread-specific sockets.

A successful function call, or a call to **WSAGetLastError**, does not reset the error code. To reset the error code, use the **WSASetLastError** function call with *iError* set to zero. A **getsockopt** SO\_ERROR also resets the error code to zero.

The **WSAGetLastError** function should not be used to check for an error value on receipt of an asynchronous message. In this case, the error value is passed in the *IPParam* parameter of the message, and this can differ from the value returned by **WSAGetLastError**.



### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **getsockopt**, **WSASetLastError**

## WSAGetOverlappedResult

The Windows Sockets **WSAGetOverlappedResult** function returns the results of an overlapped operation on the specified socket.

```

BOOL WSAGetOverlappedResult (
    SOCKET                s,
    LPWSAOVERLAPPED     lpOverlapped,
    LPDWORD              lpcbTransfer,
    BOOL                 fWait,
    LPDWORD              lpdwFlags
);

```

### Parameters

*s*

[in] Descriptor identifying the socket. This is the same socket that was specified when the overlapped operation was started by a call to **WSARecv**, **WSARecvFrom**, **WSASend**, **WSASendTo**, or **WSAIocctl**.

*lpOverlapped*

[in] Pointer to a **WSAOVERLAPPED** structure that was specified when the overlapped operation was started.

*pcbTransfer*

[out] Pointer to a 32-bit variable that receives the number of bytes that were actually transferred by a send or receive operation, or by **WSAIocctl**.

*fWait*

[in] Flag that specifies whether the function should wait for the pending overlapped operation to complete. If **TRUE**, the function does not return until the operation has been completed. If **FALSE** and the operation is still pending, the function returns **FALSE** and the **WSAGetLastError** function returns **WSA\_IO\_INCOMPLETE**. The *fWait* parameter may be set to **TRUE** only if the overlapped operation selected the event-based completion notification.

*lpdwFlags*

[out] Pointer to a 32-bit variable that will receive one or more flags that supplement the completion status. If the overlapped operation was initiated through **WSARecv** or **WSARecvFrom**, this parameter will contain the results value for *lpFlags* parameter.

**Return Values**

If **WSAGetOverlappedResult** succeeds, the return value is TRUE. This means that the overlapped operation has completed successfully and that the value pointed to by *lpcbTransfer* has been updated. If **WSAGetOverlappedResult** returns FALSE, this means that either the overlapped operation has not completed, the overlapped operation completed but with errors, or the overlapped operation's completion status could not be determined due to errors in one or more parameters to **WSAGetOverlappedResult**. On failure, the value pointed to by *lpcbTransfer* will *not* be updated. Use **WSAGetLastError** to determine the cause of the failure (either of **WSAGetOverlappedResult** or of the associated overlapped operation).

<b>Error code</b>	<b>Meaning</b>
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSA_INVALID_HANDLE	The <i>hEvent</i> parameter of the <b>WSAOVERLAPPED</b> structure does not contain a valid event object handle.
WSA_INVALID_PARAMETER	One of the parameters is unacceptable.
WSA_IO_INCOMPLETE	The <i>fWait</i> parameter is FALSE and the I/O operation has not yet completed.
WSAEFAULT	One or more of the <i>lpOverlapped</i> , <i>lpcbTransfer</i> , or <i>lpdwFlags</i> arguments are not a valid part of the user address space.

**Remarks**

The **WSAGetOverlappedResult** function reports the results of the last overlapped operation for the specified socket. The **WSAOverlappedResult** function is passed the socket descriptor and the **WSAOVERLAPPED** structure that was specified when the overlapped function was called. A pending operation is indicated when the function that started the operation returns FALSE and the **WSAGetLastError** function returns WSA\_IO\_PENDING. When an I/O operation such as **WSARecv** is pending, the function that started the operation resets the *hEvent* member of the **WSAOVERLAPPED** structure to the nonsignaled state. Then, when the pending operation has completed, the system sets the event object to the signaled state.

If the *fWait* parameter is TRUE, **WSAGetOverlappedResult** determines whether the pending operation has been completed by waiting for the event object to be in the signaled state. A client may set the *fWait* parameter to TRUE, but only if it selected event-based completion notification when the I/O operation was requested. If another form of notification was selected, the usage of the *hEvent* parameter of the **WSAOVERLAPPED** structure is different, and setting *fWait* to TRUE causes unpredictable results.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSAAccept**, **WSAConnect**, **WSACreateEvent**, **WSAIoctl**, **WSARecv**, **WSARecvFrom**, **WSASend**, **WSASendTo**, **WSAWaitForMultipleEvents**

---

## WSAGetQOSByName

The Windows Sockets **WSAGetQOSByName** function initializes a **QOS** structure based on a named template, or it supplies a buffer to retrieve an enumeration of the available template names.

```
BOOL WSAGetQOSByName (  
    SOCKET        s,  
    LPWSABUF     lpQOSName,  
    LPQOS        lpQOS  
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*lpQOSName*

[in out] Pointer to a specific quality of service template.

*lpQOS*

[out] Pointer to the **QOS** structure to be filled.

### Return Values

If **WSAGetQOSByName** succeeds, the return value is TRUE. If the function fails, the return value is FALSE. To get extended error information, call **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lpQOSName</i> or <i>lpQOS</i> parameter are not a valid part of the user address space, or the buffer length for <i>lpQOS</i> is too small.
WSAENVAL	The specified QOS template name is invalid.

### Remarks

The **WSAGetQOSByName** function is used by applications to initialize a **QOS** structure to a set of known values appropriate for a particular service class or media type. These values are stored in a template that is referenced by a well-known name. The client may retrieve these values by setting the *buf* parameter of the **WSABUF** structure indicated by *lpQOSName*, which points to a string of nonzero length specifying a template name. In this case, the usage of *lpQOSName* is IN only, and results are returned through *lpQOS*.

Alternatively, the client may use this function to retrieve an enumeration of available template names. The client may do this by setting the *buf* parameter of the **WSABUF** indicated by *lpQOSName* to a zero-length null-terminated string. In this case the buffer indicated by *buf* is overwritten with a sequence of as many available, null-terminated template names up to the number of bytes available in *buf* as indicated by the *len* parameter of the **WSABUF** indicated by *lpQOSName*. The list of names itself is terminated by a zero-length name. When the **WSAGetQOSByName** function is used to retrieve template names, the *lpQOS* parameter is ignored.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **getsockopt**, **WSAAccept**, **WSAConnect**

## WSAGetServiceClassInfo

The Windows Sockets **WSAGetServiceClassInfo** function retrieves all of the class information (schema) pertaining to a specified service class from a specified name space provider.

```

INT WSAGetServiceClassInfo (
    LPGUID          lpProviderId,
    LPGUID          lpServiceClassId,
    LPDWORD         lpdwBufferLength,
    LPWSASERVICECLASSINFO lpServiceClassInfo
);

```

### Parameters

#### *lpProviderId*

[in] Pointer to a GUID that identifies a specific name space provider.

#### *lpServiceClassId*

[in] Pointer to a GUID identifying the service class.

#### *lpdwBufferLength*

[in/out] On input, the number of bytes contained in the buffer pointed to by *lpServiceClassInfos*. On output, if the function fails and the error is WSAEFAULT, then it contains the minimum number of bytes to pass for the *lpServiceClassInfo* to retrieve the record.

#### *lpServiceClassInfo*

[out] Pointer to the service class information from the indicated name space provider for the specified service class.

### Return Values

The return value is zero if the **WSAGetServiceClassInfo** was successful. Otherwise, the value **SOCKET\_ERROR** is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSAEACCESS	The calling routine does not have sufficient privileges to access the information.
WSAEFAULT	The buffer referenced by <i>lpServiceClassInfo</i> is too small. Pass in a larger buffer.
WSAEINVAL	The specified service class identifier or name space provider identifier is invalid.
WSANOTINITIALIZED	The Ws2_32.dll has not been initialized. The application must first call <b>WSAStartup</b> before calling any Windows Sockets functions.

Error code	Meaning
WSATYPE NOT FOUND	The specified class was not found.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

### Remarks

The **WSAGetServiceClassInfo** function retrieves service class information but the service class information retrieved from a particular name space provider might not be the complete set of class information that was supplied when the service class was installed. Individual name space providers are only required to retain service class information that is applicable to the name spaces that they support. See the section *Service Class Data Structures* for more information.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

**Unicode:** Implemented as Unicode and ANSI versions on all platforms.

## WSAGetServiceClassNameByClassId

The Windows Sockets **WSAGetServiceClassNameByClassId** function returns the name of the service associated with the given type. This name is the generic service name, like FTP or SNA, and not the name of a specific instance of that service.

```
INT WSAGetServiceClassNameByClassId (
    LPGUID          lpServiceClassId,
    LPTSTR          lpszServiceClassName,
    LPDWORD         lpdwBufferLength
);
```

### Parameters

*lpServiceClassId*

[in] Pointer to the GUID for the service class.

*lpszServiceClassName*

[out] Pointer to the service name.

*lpdwBufferLength*

[in/out] On input, the length of the buffer returned by *lpszServiceClassName*. On output, the length of the service name copied into *lpszServiceClassName*.

## Return Values

The **WSAGetServiceClassNameByClassId** function returns a value of zero if successful. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSAEFAULT	The specified buffer referenced by <i>lpServiceClassName</i> is too small. Pass in a larger buffer.
WSA_INVALID_PARAMETER	The <i>lpServiceClassId</i> parameter specified is invalid.
WSANOTINITIALIZED	The <code>Ws2_32.dll</code> has not been initialized. The application must first call <b>WSAStartup</b> before calling any Windows Sockets functions.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

## WSAHtonl

The Windows Sockets **WSAHtonl** function converts a **u\_long** from host byte order to network byte order.

```
int WSAHtonl (
    SOCKET          s,
    u_long          hostlong,
    u_long FAR     *lpnetlong
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*hostlong*

[in] 32-bit number in host byte order.

*lpnetlong*

[out] Pointer to a 32-bit number in network byte order.

## Return Values

If no error occurs, **WSAHtonl** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lpnetlong</i> parameter is not completely contained in a valid part of the user address space.

## Remarks

The **WSAHtonl** function takes a 32-bit number in host byte order and returns a 32-bit number pointed to by the *lpnetlong* parameter in the network byte order associated with socket *s*.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **htonl**, **htons**, **ntohl**, **ntohs**, **WSANTohl**, **WSAHtons**, **WSANTohs**

# WSAHtons

The Windows Sockets **WSAHtons** function converts a **u\_short** from host byte order to network byte order.

```
int WSAHtons (
    SOCKET          s,
    u_short         hostshort,
    u_short FAR    *lpnetshort
);
```

## Parameters

*s*

[in] Descriptor identifying a socket.



*hostshort*

[in] 16-bit number in host byte order.

*lpNetshort*

[out] Pointer to a 16-bit number in network byte order.

### Return Values

If no error occurs, **WSAHtons** returns zero. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lpNetshort</i> parameter is not completely contained in a valid part of the user address space.

### Remarks

The **WSAHtons** function takes a 16-bit number in host byte order and returns a 16-bit number pointed to by the *lpNetshort* parameter in the network byte order associated with socket *s*.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **htonl**, **htons**, **ntohl**, **ntohs**, **WSAhtonl**, **WSANtohl**, **WSANtohs**

## WSAInstallServiceClass

The Windows Sockets **WSAInstallServiceClass** function registers a service class schema within a name space. This schema includes the class name, class identifier, and any name space-specific information that is common to all instances of the service, such as the SAP identifier or object identifier.

```
INT WSAInstallServiceClass (
    LPWSASERVICECLASSINFO lpServiceClassInfo
);
```

## Parameters

### *IpServiceClassInfo*

[in] Service class to name space specific-type mapping information. Multiple mappings can be handled at one time.

See the section Service Class Data Structures for a description of pertinent data structures.

## Return Values

The return value is zero if the operation was successful. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSAEACCES	The calling function does not have sufficient privileges to install the service.
WSAEALREADY	Service class information has already been registered for this service class identifier. To modify service class information, first use <b>WSARemoveServiceClass</b> , and then reinstall with updated class information data.
WSAEINVAL	The service class information was invalid or improperly structured.
WSANOTINITIALIZED	The <code>Ws2_32.dll</code> has not been initialized. The application must first call <b>WSAStartup</b> before calling any Windows Sockets functions.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

### Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

**Unicode:** Implemented as Unicode and ANSI versions on all platforms.

## WSAIoctl

The Windows Sockets **WSAIoctl** function controls the mode of a socket.

```
int WSAIoctl (
    SOCKET          s,
    DWORD          dwIoControlCode,
    LPVOID         lpvInBuffer,
```

(continued)

(continued)

```

DWORD          cbInBuffer,
LPVOID         lpvOutBuffer,
DWORD          cbOutBuffer,
LPDWORD        lpcbBytesReturned,
LPWSAOVERLAPPED lpOverlapped,
LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);

```

## Parameters

*s*

[in] Descriptor identifying a socket.

*dwIoControlCode*

[in] Control code of operation to perform.

*lpvInBuffer*

[in] Pointer to the input buffer.

*cbInBuffer*

[in] Size of the input buffer.

*lpvOutBuffer*

[out] Pointer to the output buffer.

*cbOutBuffer*

[in] Size of the output buffer.

*lpcbBytesReturned*

[out] Pointer to actual number of bytes of output.

*lpOverlapped*

[in] Pointer to a **WSAOVERLAPPED** structure (ignored for nonoverlapped sockets).

*lpCompletionRoutine*

[in] Pointer to the completion routine called when the operation has been completed (ignored for nonoverlapped sockets).

## Return Values

Upon successful completion, the **WSAIoctl** returns zero. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>lpvInBuffer</i> , <i>lpvOutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> , or <i>lpCompletionRoutine</i> argument is not totally contained in a valid part of the user address space, or the <i>cbInBuffer</i> or <i>cbOutBuffer</i> argument is too small.

Error code	Meaning
WSAEINVAL	<i>dwIoControlCode</i> is not a valid command, or a supplied input parameter is not acceptable, or the command is not applicable to the type of socket supplied.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAENOTSOCK	The descriptor <i>s</i> is not a socket.
WSAEOPNOTSUPP	The specified ioctl command cannot be realized. (For example, the <b>FLOWSPEC</b> structures specified in SIO_SET_QOS cannot be satisfied.)
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the requested operation would block.

### Remarks

The **WSAIoctl** function is used to set or retrieve operating parameters associated with the socket, the transport protocol, or the communications subsystem.

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a nonoverlapped socket. For a nonoverlapped socket, *lpOverlapped* and *lpCompletionRoutine* parameters are ignored, which cause the function to behave like the standard **ioctlsocket** function except that **WSAIoctl** can block if socket *s* is in blocking mode. If socket *s* is in nonblocking mode, this function can return WSAEWOLDBLOCK when the specified operation cannot be finished immediately. In this case, the application may change the socket to blocking mode and reissue the request or wait for the corresponding network event (such as FD\_ROUTING\_INTERFACE\_CHANGE or FD\_ADDRESS\_LIST\_CHANGE in the case of SIO\_ROUTING\_INTERFACE\_CHANGE or SIO\_ADDRESS\_LIST\_CHANGE) using a Windows message-based (using **WSAAsyncSelect**) or event (using **WSAEventSelect**)-based notification mechanism.

For overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The final completion status is retrieved through **WSAGetOverlappedResult**. The *lpcbBytesReturned* parameter is ignored.

Any IOCTL may block indefinitely, depending on the service provider's implementation. If the application cannot tolerate blocking in a **WSAIoctl** call, overlapped I/O would be advised for IOCTLs that are especially likely to block including:

SIO_FINDROUTE	SIO_SET_QOS
SIO_FLUSH	SIO_SET_GROUP_QOS
SIO_GET_QOS	SIO_ROUTING_INTERFACE_CHANGE
SIO_GET_GROUP_QOS	SIO_ADDRESS_LIST_CHANGE

Some protocol-specific IOCTLs may also be especially likely to block. Check the relevant protocol-specific annex for any available information.

It is possible to adopt an encoding scheme that preserves the currently defined **ioctlsocket** opcodes while providing a convenient way to partition the opcode identifier space in as much as the *dwIoControlCode* parameter is now a 32-bit entity. The *dwIoControlCode* parameter is built to allow for protocol and vendor independence when adding new control codes while retaining backward compatibility with the Windows Sockets 1.1 and Unix control codes. The *dwIoControlCode* parameter has the following form.

<b>I</b>	<b>O</b>	<b>V</b>	<b>T</b>	<b>Vendor/address family</b>	<b>code</b>
3	3	2	22	2 2 2 2 2 2 1 1 1 1	1 1 1 1 1 1
1	0	9	87	6 5 4 3 2 1 0 9 8 7 6	5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

**I** is set if the input buffer is valid for the code, as with **IOC\_IN**.

**O** is set if the output buffer is valid for the code, as with **IOC\_OUT**. Codes with both input and output parameters set both **I** and **O**.

**V** is set if there are no parameters for the code, as with **IOC\_VOID**.

**T** is a 2-bit quantity that defines the type of IOCTL. The following values are defined:

- 0** The IOCTL is a standard Unix IOCTL code, as with **FIONREAD** and **FIONBIO**.
- 1** The IOCTL is a generic Windows Sockets 2 IOCTL code. New IOCTL codes defined for Windows Sockets 2 will have **T == 1**.
- 2** The IOCTL applies only to a specific address family.
- 3** The IOCTL applies only to a specific vendor's provider. This type allows companies to be assigned a vendor number that appears in the **Vendor/Address family** parameter. Then, the vendor can define new IOCTLs specific to that vendor without having to register the IOCTL with a clearinghouse, thereby providing vendor flexibility and privacy.

**Vendor/Address family** An 11-bit quantity that defines the vendor who owns the code (if **T == 3**) or that contains the address family to which the code applies (if **T == 2**). If this is a Unix IOCTL code (**T == 0**) then this parameter has the same value as the code on Unix. If this is a generic Windows Sockets 2 IOCTL (**T == 1**) then this parameter can be used as an extension of the code parameter to provide additional code values.

**Code** The 16-bit quantity that contains the specific IOCTL code for the operation.

The following Unix IOCTL codes (commands) are supported.

### **FIONBIO**

Enable or disable nonblocking mode on socket *s*. *lpvInBuffer* points at an **unsigned long**, which is nonzero if nonblocking mode is to be enabled and zero if it is to be disabled. When a socket is created, it operates in blocking mode (that is, nonblocking mode is disabled). This is consistent with BSD sockets.

The **WSAAsyncSelect** or **WSAEventSelect** routine automatically sets a socket to nonblocking mode. If **WSAAsyncSelect** or **WSAEventSelect** has been issued on a socket, then any attempt to use **WSAIoctl** to set the socket back to blocking mode will fail with **WSAEINVAL**. To set the socket back to blocking mode, an application must first disable **WSAAsyncSelect** by calling **WSAAsyncSelect** with the *lEvent* parameter equal to zero, or disable **WSAEventSelect** by calling **WSAEventSelect** with the *lNetworkEvents* parameter equal to zero.

### **FIONREAD**

Determine the amount of data that can be read atomically from socket *s*. *lpvOutBuffer* points at an **unsigned long** in which **WSAIoctl** stores the result. If *s* is stream oriented (for example, type **SOCK\_STREAM**), **FIONREAD** returns the total amount of data that can be read in a single receive operation; this is normally the same as the total amount of data queued on the socket (since data stream is byte-oriented, this is not guaranteed). If *s* is message oriented (for example, type **SOCK\_DGRAM**), **FIONREAD** returns the size of the first datagram (message) queued on the socket.

### **SIOCATMARK**

Determine whether or not all OOB data has been read. This applies only to a socket of stream-style (for example, type **SOCK\_STREAM**) that has been configured for inline reception of any OOB data (**SO\_OOBINLINE**). If no OOB data is waiting to be read, the operation returns **TRUE**. Otherwise, it returns **FALSE**, and the next receive operation performed on the socket will retrieve some or all of the data preceding the mark; the application should use the **SIOCATMARK** operation to determine whether any remains. If there is any normal data preceding the urgent (out of band) data, it will be received in order. (Note that **recv** operations will never mix OOB and normal data in the same call.) *lpvOutBuffer* points at a **BOOL** in which **WSAIoctl** stores the result.

The following Windows Sockets 2 commands are supported.

### **SIO\_ASSOCIATE\_HANDLE** (opcode setting: I, T==1)

Associate this socket with the specified handle of a companion interface. The input buffer contains the integer value corresponding to the manifest constant for the companion interface (for example, **TH\_NETDEV** and **TH\_TAPI**), followed by a value that is a handle of the specified companion interface, along with any other required information. Refer to the appropriate section in the Windows Sockets 2 Protocol-Specific Annex (a separate document) for details specific to a particular companion interface. The total size is reflected in the input buffer length. No output buffer is required. The **WSAENOPROTOOPT** error code is indicated for service providers that do not support this ioctl. The handle associated by this ioctl can be retrieved using **SIO\_TRANSLATE\_HANDLE**.

A companion interface might be used, for example, if a particular provider provides (1) a great deal of additional controls over the behavior of a socket and (2) the controls are provider-specific enough that they do not map to existing Windows Socket functions or ones likely to be defined in the future. It is recommend that the Component Object Model (COM) be used instead of this ioctl to discover and track

other interfaces that might be supported by a socket. This ioctl is present for (reverse) compatibility with systems where COM is not available or cannot be used for some other reason.

**SIO\_ENABLE\_CIRCULAR\_QUEUEING** (opcode setting: V, T==1)

Indicates to the underlying message-oriented service provider that a newly arrived message should never be dropped because of a buffer queue overflow. Instead, the oldest message in the queue should be eliminated in order to accommodate the newly arrived message. No input and output buffers are required. Note that this ioctl is only valid for sockets associated with unreliable, message-oriented protocols. The WSAENOPROTOOPT error code is indicated for service providers that do not support this ioctl.

**SIO\_FIND\_ROUTE** (opcode setting: O, T==1)

When issued, this ioctl requests that the route to the remote address specified as a **SOCKADDR** in the input buffer be discovered. If the address already exists in the local cache, its entry is invalidated. In the case of Novell's IPX, this call initiates an IPX GetLocalTarget (GLT), which queries the network for the given remote address.

**SIO\_FLUSH** (opcode setting: V, T==1)

Discards current contents of the sending queue associated with this socket. No input and output buffers are required. The WSAENOPROTOOPT error code is indicated for service providers that do not support this ioctl.

**SIO\_GET\_BROADCAST\_ADDRESS** (opcode setting: O, T==1)

This ioctl fills the output buffer with a **SOCKADDR** structure containing a suitable broadcast address for use with **sendto/WSASendTo**.

**SIO\_GET\_EXTENSION\_FUNCTION\_POINTER** (opcode setting: O, I, T==1)

Retrieve a pointer to the specified extension function supported by the associated service provider. The input buffer contains a globally unique identifier (GUID) whose value identifies the extension function in question. The pointer to the desired function is returned in the output buffer. Extension function identifiers are established by service provider vendors and should be included in vendor documentation that describes extension function capabilities and semantics.

**SIO\_GET\_QOS** (opcode setting: O, T==1)

Reserved for future use with sockets. Retrieve the **QOS** structure associated with the socket. The input buffer is optional. Some protocols (for example, RSVP) allow the input buffer to be used to qualify a quality of service request. The **QOS** structure will be copied into the output buffer. The output buffer must be sized large enough to be able to contain the full **QOS** structure. The WSAENOPROTOOPT error code is indicated for service providers that do not support quality of service.

A sender may not call SIO\_GET\_QOS until the socket is connected.

A receiver may call SIO\_GET\_QOS soon as it is bound.

**SIO\_GET\_GROUP\_QOS** (opcode setting: O, I, T==1)

Reserved.

**SIO\_MULTIPOINT\_LOOPBACK** (opcode setting: I, T==1)

Controls whether data sent in a multipoint session will also be received by the same socket on the local host. A value of TRUE causes loopback reception to occur while a value of FALSE prohibits this. By default, loopback is enabled.

**SIO\_MULTICAST\_SCOPE** (opcode setting: I, T==1)

Specifies the scope over which multicast transmissions will occur. Scope is defined as the number of routed network segments to be covered. A scope of zero would indicate that the multicast transmission would not be placed on the wire but could be disseminated across sockets within the local host. A scope value of one (the default) indicates that the transmission will be placed on the wire, but will *not* cross any routers. Higher scope values determine the number of routers that can be crossed. Note that this corresponds to the time-to-live (TTL) parameter in IP multicasting. By default, scope is 1.

**SIO\_RCVALL**

Enables a socket to receive all IP packets on the network. The socket handle passed to the **WSAIoctl** function must be of AF\_INET address family, **SOCK\_RAW** socket type, and IPPROTO\_IP protocol. The socket also must be bound to an explicit local interface, which means that you cannot bind to INADDR\_ANY.

Once the socket is bound and the ioctl set, calls to the **WSARecv** or **recv** functions return IP datagrams passing through the given interface. Note that you must supply a sufficiently large buffer. Setting this ioctl requires Administrator privilege on the local machine. SIO\_RCVALL is available in Windows 2000 and later versions of Windows.

**SIO\_RCVALL\_MCAST**

Enables a socket to receive all multicast IP traffic on the network (that is, all IP packets destined for IP addresses in the range of 224.0.0.0 to 239.255.255.255). The socket handle passed to the **WSAIoctl** function must be of AF\_INET address family, **SOCK\_RAW** socket type, and IPPROTO\_UDP protocol. The socket also must be bound to an explicit local interface, which means that you cannot bind to INADDR\_ANY.

Once the socket is bound and the ioctl set, calls to the **WSARecv** or **recv** functions return multicast IP datagrams passing through the given interface. Note that you must supply a sufficiently large buffer. Setting this ioctl requires Administrator privilege on the local machine. **SIO\_RCVALL\_MCAST** is available only in Windows 2000 and later versions of Windows.

**SIO\_RCVALL\_IGMPMCAST**

Enables a socket to receive all IGMP multicast IP traffic on the network, without receiving other multicast IP traffic. The socket handle passed to the **WSAIoctl** function must be of AF\_INET address family, **SOCK\_RAW** socket type, and IPPROTO\_IGMP protocol. The socket also must be bound to an explicit local interface, which means that you cannot bind to INADDR\_ANY.

Once the socket is bound and the ioctl set, calls to the **WSARecv** or **recv** functions return multicast IP datagrams passing through the given interface. Note that you must



supply a sufficiently large buffer. Setting this ioctl requires Administrator privilege on the local machine. **SIO\_RCVALL\_IGMPMC** is available only in Windows 2000 and later versions of Windows.

### **SIO\_KEEPALIVE\_VALS**

Enables the per-connection setting of **keep-alive** option, keepalive time, and keepalive interval. The argument structure for **SIO\_KEEPALIVE\_VALS** is as follows:

```
/* Argument structure for SIO_KEEPALIVE_VALS */
struct tcp_keepalive {
    u_long  onoff;
    u_long  keepalivetime;
    u_long  keepaliveinterval;
};
```

### **SIO\_SET\_QOS** (opcode setting: I, T==1)

Associate the supplied **QOS** structure with the socket. No output buffer is required, the **QOS** structure will be obtained from the input buffer. The **WSAENOPROTOOPT** error code is indicated for service providers that do not support quality of service.

### **SIO\_SET\_GROUP\_QOS** (opcode setting: I, T==1)

Reserved.

### **SIO\_TRANSLATE\_HANDLE** (opcode setting: I, O, T==1)

To obtain a corresponding handle for socket *s* that is valid in the context of a companion interface (for example, **TH\_NETDEV** and **TH\_TAPI**). A manifest constant identifying the companion interface along with any other needed parameters are specified in the input buffer. The corresponding handle will be available in the output buffer upon completion of this function. Refer to the appropriate section in Windows Sockets 2 Protocol-Specific Annex for details specific to a particular companion interface. The **WSAENOPROTOOPT** error code is indicated for service providers that do not support this ioctl for the specified companion interface. This ioctl retrieves the handle associated using **SIO\_TRANSLATE\_HANDLE**.

It is recommend that the Component Object Model (COM) be used instead of this ioctl to discover and track other interfaces that might be supported by a socket. This ioctl is present for backward compatibility with systems where COM is not available or cannot be used for some other reason.

### **SIO\_ROUTING\_INTERFACE\_QUERY** (opcode setting: I, O, T==1)

To obtain the address of the local interface (represented as **SOCKADDR** structure) which should be used to send to the remote address specified in the input buffer (as **SOCKADDR**). Remote multicast addresses may be submitted in the input buffer to get the address of the preferred interface for multicast transmission. In any case, the interface address returned may be used by the application in a subsequent **bind()** request.

Note that routes are subject to change. Therefore, applications cannot rely on the information returned by **SIO\_ROUTING\_INTERFACE\_QUERY** to be persistent.

Applications may register for routing change notifications through the **SIO\_ROUTING\_INTERFACE\_CHANGE** IOCTL which provides for notification

through either overlapped I/O or `FD_ROUTING_INTERFACE_CHANGE` event. The following sequence of actions can be used to guarantee that the application always has current routing interface information for a given destination:

- Issue `SIO_ROUTING_INTERFACE_CHANGE` IOCTL
- Issue `SIO_ROUTING_INTERFACE_QUERY` IOCTL
- Whenever **`SIO_ROUTING_INTERFACE_CHANGE` IOCTL** notifies the application of routing change (either through overlapped I/O or by signaling `FD_ROUTING_INTERFACE_CHANGE` event), the whole sequence of actions should be repeated.

If output buffer is not large enough to contain the interface address, `SOCKET_ERROR` is returned as the result of this IOCTL and **`WSAGetLastError`** returns `WSAEFAULT`. The required size of the output buffer will be returned in *lpcbBytesReturned* in this case. Note the `WSAEFAULT` error code is also returned if the *lpvInBuffer*, *lpvOutBuffer* or *lpcbBytesReturned* parameter is not totally contained in a valid part of the user address space.

If the destination address specified in the input buffer cannot be reached through any of the available interfaces, `SOCKET_ERROR` is returned as the result of this IOCTL and **`WSAGetLastError`** returns `WSAENETUNREACH` or even `WSAENETDOWN` if all of the network connectivity is lost.

#### **`SIO_ROUTING_INTERFACE_CHANGE`** (opcode setting: I, T==1)

To receive notification of the interface change that should be used to reach the remote address in the input buffer (specified as a **`SOCKADDR`** structure). No output information will be provided upon completion of this IOCTL; the completion merely indicates that routing interface for a given destination has changed and should be queried again through **`SIO_ROUTING_INTERFACE_QUERY`**.

It is assumed (although not required) that the application uses overlapped I/O to be notified of routing interface change through completion of **`SIO_ROUTING_INTERFACE_CHANGE`** request. Alternatively, if the **`SIO_ROUTING_INTERFACE_CHANGE` IOCTL** is issued on nonblocking socket *and* without overlapped parameters (*lpOverlapped* / *CompletionRoutine* are set `NULL`), it will complete immediately with error `WSAEWOULDBLOCK`, and the application can then wait for routing change events through call to **`WSAEventSelect`** or **`WSAAsyncSelect`** with `FD_ROUTING_INTERFACE_CHANGE` bit set in the network event bitmask.

It is recognized that routing information remains stable in most cases so that requiring the application to keep multiple outstanding IOCTLs to get notifications about all destinations that it is interested in as well as having service provider to keep track of all them will unnecessarily tie significant system resources. This situation can be avoided by extending the meaning of the input parameters and relaxing the service provider requirements as follows:

- The application can specify a protocol family specific wildcard address (same as one used in **bind** call when requesting to bind to any available address) to request notifications of any routing changes. This allows the application to keep only one outstanding **SIO\_ROUTING\_INTERFACE\_CHANGE** for all the sockets/destinations it has and then use **SIO\_ROUTING\_INTERFACE\_QUERY** to get the actual routing information.
- Service provider has the option to ignore the information supplied by the application in the input buffer of the **SIO\_ROUTING\_INTERFACE\_CHANGE** (as though the application specified a wildcard address) and complete the **SIO\_ROUTING\_INTERFACE\_CHANGE IOCTL** or signal **FD\_ROUTING\_INTERFACE\_CHANGE** event in the event of any routing information change (not just the route to the destination specified in the input buffer).

### **SIO\_ADDRESS\_LIST\_QUERY** (opcode setting: I, O, T==1)

To obtain a list of local transport addresses of the socket's protocol family to which the application can bind. The list returned in the output buffer using the following format:

```
typedef struct _SOCKET_ADDRESS_LIST {
    INT iAddressCount;
    SOCKET_ADDRESS Address[1];
} SOCKET_ADDRESS_LIST, FAR * LPSOCKET_ADDRESS_LIST;
Members:
    iAddressCount - number of address structures in the list;
    Address - array of protocol family specific address structures.
```

**Note** In Win32 Plug-n-Play environments addresses can be added and removed dynamically. Therefore, applications cannot rely on the information returned by **SIO\_ADDRESS\_LIST\_QUERY** to be persistent. Applications may register for address change notifications through the **SIO\_ADDRESS\_LIST\_CHANGE IOCTL** which provides for notification through either overlapped I/O or **FD\_ADDRESS\_LIST\_CHANGE** event. The following sequence of actions can be used to guarantee that the application always has current address list information:

- Issue **SIO\_ADDRESS\_LIST\_CHANGE IOCTL**
- Issue **SIO\_ADDRESS\_LIST\_QUERY IOCTL**
- Whenever **SIO\_ADDRESS\_LIST\_CHANGE IOCTL** notifies the application of address list change (either through overlapped I/O or by signaling **FD\_ADDRESS\_LIST\_CHANGE** event), the whole sequence of actions should be repeated.

If output buffer is not large enough to contain the address list, **SOCKET\_ERROR** is returned as the result of this IOCTL and **WSAGetLastError** returns **WSAEFAULT**. The required size of the output buffer will be returned in *lpcbBytesReturned* in this case. Note the **WSAEFAULT** error code is also returned if the *lpvInBuffer*, *lpvOutBuffer*, or *lpcbBytesReturned* parameter is not totally contained in a valid part of the user address space.

**SIO\_ADDRESS\_LIST\_CHANGE** (opcode setting: T==1)

To receive notification of changes in the list of local transport addresses of the socket's protocol family to which the application can bind. No output information will be provided upon completion of this IOCTL; the completion merely indicates that list of available local address has changed and should be queried again through **SIO\_ADDRESS\_LIST\_QUERY**.

It is assumed (although not required) that the application uses overlapped I/O to be notified of change by completion of **SIO\_ADDRESS\_LIST\_CHANGE** request. Alternatively, if the **SIO\_ADDRESS\_LIST\_CHANGE** IOCTL is issued on a nonblocking socket and without overlapped parameters (*lpOverlapped* / *lpCompletionRoutine* are set to NULL), it will complete immediately with error **WSAEWOULDBLOCK**. The application can then wait for address list change events through a call to **WSAEventSelect** or **WSAAsyncSelect** with **FD\_ADDRESS\_LIST\_CHANGE** bit set in the network event bitmask.

If an overlapped operation completes immediately, **WSAIoctl** returns a value of zero and the *lpcbBytesReturned* parameter is updated with the number of bytes in the output buffer. If the overlapped operation is successfully initiated and will complete later, this function returns **SOCKET\_ERROR** and indicates error code **WSA\_IO\_PENDING**. In this case, *lpcbBytesReturned* is not updated. When the overlapped operation completes the amount of data in the output buffer is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **WSAGetOverlappedResult**.

When called with an overlapped socket, the *lpOverlapped* parameter must be valid for the duration of the overlapped operation. The *lpOverlapped* parameter contains the address of a **WSAOVERLAPPED** structure.

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents** or **WSAGetOverlappedResult** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to TRUE. In this case, the usage of the *hEvent* parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The prototype of the completion routine is as follows:

```
void CALLBACK CompletionRoutine (
    IN DWORD          dwError,
    IN DWORD          cbTransferred,
    IN LPWSAOVERLAPPED lpOverlapped,
```

(continued)

(continued)

```
IN DWORD          dwFlags
):
```

This **CompletionRoutine** is a placeholder for an application-defined or library-defined function. The *dwError* parameter specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* parameter specifies the number of bytes returned. Currently, there are no flag values defined and *dwFlags* will be zero. The **CompletionRoutine** function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. The completion routines can be called in any order, not necessarily in the same order the overlapped operations are completed.

### Compatibility

The IOCTL codes with **T == 0** are a subset of the IOCTL codes used in Berkeley sockets. In particular, there is no command that is equivalent to **FIOASYNC**.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **getsockopt**, **ioctlsocket**, **setsockopt**, **socket**, **WSASocket**

---

## WSAIsBlocking

This function has been removed in compliance with the Windows Sockets 2 specification, revision 2.2.0.

The Windows Socket **WSAIsBlocking** function is not exported directly by the Ws2\_32.dll, and Windows Sockets 2 applications should not use this function. Windows Sockets 1.1 applications that call this function are still supported through the Winsock.dll and Wsock32.dll.

Blocking hooks are generally used to keep a single-threaded GUI application responsive during calls to blocking functions. Instead of using blocking hooks, an applications should use a separate thread (separate from the main GUI thread) for network activity.

# WSAJoinLeaf

The Windows Sockets **WSAJoinLeaf** function joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the supplied **FLOWSPEC** structures.

```
SOCKET WSAJoinLeaf (  
    SOCKET                s,  
    const struct sockaddr FAR *name,  
    int                   namelen,  
    LPWSABUF              lpCallerData,  
    LPWSABUF              lpCalleeData,  
    LPQOS                 lpSQOS,  
    LPQOS                 lpGQOS,  
    DWORD                 dwFlags  
);
```

## Parameters

*s*

[in] Descriptor identifying a multipoint socket.

*name*

[in] Name of the peer to which the socket is to be joined.

*namelen*

[in] Length of *name*.

*lpCallerData*

[in] Pointer to the user data that is to be transferred to the peer during multipoint session establishment.

*lpCalleeData*

[out] Pointer to the user data that is to be transferred back from the peer during multipoint session establishment.

*lpSQOS*

[in] Pointer to the **FLOWSPEC** structures for socket *s*, one for each direction.

*lpGQOS*

[in] Reserved.

*dwFlags*

[in] Flags to indicate that the socket is acting as a sender, receiver, or both.

## Return Values

If no error occurs, **WSAJoinLeaf** returns a value of type **SOCKET** that is a descriptor for the newly created multipoint socket. Otherwise, a value of **INVALID\_SOCKET** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

On a blocking socket, the return value indicates success or failure of the join operation.

With a nonblocking socket, successful initiation of a join operation is indicated by a return of a valid socket descriptor. Subsequently, an `FD_CONNECT` indication will be given on the original socket `s` when the join operation completes, either successfully or otherwise. The application must use either `WSAAsyncSelect` or `WSAEventSelect` with interest registered for the `FD_CONNECT` event in order to determine when the join operation has completed and checks the associated error code to determine the success or failure of the operation. The `select` function cannot be used to determine when the join operation completes.

Also, until the multipoint session join attempt completes all subsequent calls to `WSAJoinLeaf` on the same socket will fail with the error code `WSAEALREADY`. After the `WSAJoinLeaf` operation completes successfully, a subsequent attempt will usually fail with the error code `WSAEISCONN`. An exception to the `WSAEISCONN` rule occurs for a `c_root` socket that allows root-initiated joins. In such a case, another join may be initiated after a prior `WSAJoinLeaf` operation completes.

If the return error code indicates the multipoint session join attempt failed (that is, `WSAECONNREFUSED`, `WSAENETUNREACH`, `WSAETIMEDOUT`) the application can call `WSAJoinLeaf` again for the same socket.

Error code	Meaning
<code>WSANOTINITIALISED</code>	A successful <b>WSAStartup</b> call must occur before using this function.
<code>WSAENETDOWN</code>	The network subsystem has failed.
<code>WSAEADDRINUSE</code>	The socket's local address is already in use and the socket was not marked to allow address reuse with <code>SO_REUSEADDR</code> . This error usually occurs at the time of <b>bind</b> , but could be delayed until this function if the <b>bind</b> was to a partially wildcard address (involving <code>ADDR_ANY</code> ) and if a specific address needs to be committed at the time of this function.
<code>WSAEINTR</code>	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
<code>WSAEINPROGRESS</code>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<code>WSAEALREADY</code>	A nonblocking <b>WSAJoinLeaf</b> call is in progress on the specified socket.
<code>WSAEADDRNOTAVAIL</code>	The remote address is not a valid address (such as <code>ADDR_ANY</code> ).
<code>WSAEAFNOSUPPORT</code>	Addresses in the specified family cannot be used with this socket.
<code>WSAECONNREFUSED</code>	The attempt to join was forcefully rejected.

Error code	Meaning
WSAEFAULT	The <i>name</i> or the <i>namelen</i> parameter is not a valid part of the user address space, the <i>namelen</i> parameter is too small, the buffer length for <i>lpCalleeData</i> , <i>lpSQOS</i> , and <i>lpGQOS</i> are too small, or the buffer length for <i>lpCallerData</i> is too large.
WSAEISCONN	The socket is already a member of the multipoint session.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be joined.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The <b>FLOWSPEC</b> structures specified in <i>lpSQOS</i> and <i>lpGQOS</i> cannot be satisfied.
WSAEPROTONOSUPPORT	The <i>lpCallerData</i> augment is not supported by the service provider.
WSAETIMEDOUT	The attempt to join timed out without establishing a multipoint session.

### Remarks

The **WSAJoinLeaf** function is used to join a leaf node to a multipoint session, and to perform a number of other ancillary operations that occur at session join time as well. If the socket, *s*, is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound.

The **WSAJoinLeaf** function has the same parameters and semantics as **WSAConnect** except that it returns a socket descriptor (as in **WSAAccept**), and it has an additional *dwFlags* parameter. Only multipoint sockets created using **WSASocket** with appropriate multipoint flags set can be used for input parameter *s* in this function. The returned socket descriptor will *not* be useable until after the join operation completes. For example, if the socket is in nonblocking mode after a corresponding **FD\_CONNECT** indication has been received from **WSAAsyncSelect** or **WSAEventSelect** on the original socket *s*, except that **closesocket** may be invoked on this new socket descriptor to cancel a pending join operation. A root application in a multipoint session may call **WSAJoinLeaf** one or more times in order to add a number of leaf nodes, however at most one multipoint connection request may be outstanding at a time. Refer to Multipoint and Multicast Semantics for additional information.

For nonblocking sockets it is often not possible to complete the connection immediately. In such a case, this function returns an as-yet unusable socket descriptor and the operation proceeds. There is no error code such as **WSAEWOULDBLOCK** in this case, since the function has effectively returned a successful start indication. When the final outcome success or failure becomes known, it may be reported through



**WSAAsyncSelect** or **WSAEventSelect** depending on how the client registers for notification on the original socket *s*. In either case, the notification is announced with **FD\_CONNECT** and the error code associated with the **FD\_CONNECT** indicates either success or a specific reason for failure. The **select** function cannot be used to detect completion notification for **WSAJoinLeaf**.

The socket descriptor returned by **WSAJoinLeaf** is different depending on whether the input socket descriptor, *s*, is a *c\_root* or a *c\_leaf*. When used with a *c\_root* socket, the *name* parameter designates a particular leaf node to be added and the returned socket descriptor is a *c\_leaf* socket corresponding to the newly added leaf node. The newly created socket has the same properties as *s*, including asynchronous events registered with **WSAAsyncSelect** or with **WSAEventSelect**. It is not intended to be used for exchange of multipoint data, but rather is used to receive network event indications (for example, **FD\_CLOSE**) for the connection that exists to the particular *c\_leaf*. Some multipoint implementations can also allow this socket to be used for side chats between the root and an individual leaf node. An **FD\_CLOSE** indication will be received for this socket if the corresponding leaf node calls **closesocket** to drop out of the multipoint session. Symmetrically, invoking **closesocket** on the *c\_leaf* socket returned from **WSAJoinLeaf** will cause the socket in the corresponding leaf node to get an **FD\_CLOSE** notification.

When **WSAJoinLeaf** is invoked with a *c\_leaf* socket, the *name* parameter contains the address of the root application (for a rooted control scheme) or an existing multipoint session (nonrooted control scheme), and the returned socket descriptor is the same as the input socket descriptor. In other words, a new socket descriptor is *not* allocated. In a rooted control scheme, the root application would put its *c\_root* socket in listening mode by calling **listen**. The standard **FD\_ACCEPT** notification will be delivered when the leaf node requests to join itself to the multipoint session. The root application uses the usual **accept/WSAAccept** functions to admit the new leaf node. The value returned from either **accept** or **WSAAccept** is also a *c\_leaf* socket descriptor just like those returned from **WSAJoinLeaf**. To accommodate multipoint schemes that allow both root-initiated and leaf-initiated joins, it is acceptable for a *c\_root* socket that is already in listening mode to be used as an input to **WSAJoinLeaf**.

The application is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specifies.

The *lpCallerData* is a value parameter that contains any user data that is to be sent along with the multipoint session join request. If *lpCallerData* is **NULL**, no user data will be passed to the peer. The *lpCalleeData* is a result parameter that will contain any user data passed back from the peer as part of the multipoint session establishment. The *lpCalleeData->len* initially contains the length of the buffer allocated by the application and pointed to by *lpCalleeData->buf*. *lpCalleeData->len* will be set to zero if no user data has been passed back. The *lpCalleeData* information will be valid when the multipoint join operation is complete.

For blocking sockets, this will be when the **WSAJoinLeaf** function returns. For nonblocking sockets, this will be after the join operation has completed. For example, this could occur after **FD\_CONNECT** notification on the original socket *s*). If *lpCalleeData* is **NULL**, no user data will be passed back. The exact format of the user data is specific to the address family to which the socket belongs.

At multipoint session establishment time, an application can use the *lpSQOS* parameter to override any previous quality of service specification made for the socket through **WSAIoctl** with the **SIO\_SET\_QOS** opcode.

The *lpSQOS* parameter specifies the **FLOWSPEC** structures for socket *s*, one for each direction, followed by any additional provider-specific parameters. If either the associated transport provider in general or the specific type of socket in particular cannot honor the quality of service request, an error will be returned as indicated in the following. The respective sending or receiving flow specification values will be ignored for any unidirectional sockets. If no provider-specific parameters are supplied, the *buf* and *len* parameters of *lpSQOS->ProviderSpecific* should be set to **NULL** and zero, respectively. A **NULL** value for *lpSQOS* indicates no application-supplied quality of service.

The *dwFlags* parameter is used to indicate whether the socket will be acting only as a sender (**JL\_SENDER\_ONLY**), only as a receiver (**JL\_RECEIVER\_ONLY**), or both (**JL\_BOTH**).

When connected sockets break (that is, become closed for whatever reason), they should be discarded and recreated. It is safest to assume that when things go awry for any reason on a connected socket, the application must discard and recreate the needed sockets in order to return to a stable point.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **accept**, **bind**, **select**, **WSAAccept**, **WSAAsyncSelect**, **WSAEventSelect**, **WSASocket**

---

## WSALookupServiceBegin

The Windows Sockets **WSALookupServiceBegin** function initiates a client query that is constrained by the information contained within a **WSAQUERYSET** structure.

**WSALookupServiceBegin** only returns a handle, which should be used by subsequent calls to **WSALookupServiceNext** to get the actual results.

```

INT WSALookupServiceBegin (
    LPWSAQUERYSET    IpqsRestrictions,
    DWORD            dwControlFlags,
    LPHANDLE         IpLookup
);

```

## Parameters

### *IpqsRestrictions*

[in] Pointer to the search criteria. See the following for details.

### *dwControlFlags*

[in] Flag that controls the depth of the search:

Flag	Description
LUP_DEEP	Queries deep as opposed to just the first level.
LUP_CONTAINERS	Returns containers only.
LUP_NOCONTAINERS	Does not return any containers.
LUP_FLUSHCACHE	If the provider has been caching information, ignores the cache, and queries the name space itself.
LUP_FLUSHPREVIOUS	Used as a value for the <i>dwControlFlags</i> argument in <b>WSALookupServiceNext</b> . Setting this flag instructs the provider to discard the last result set, which was too large for the supplied buffer, and move on to the next result set.
LUP_NEAREST	If possible, returns results in the order of distance. The measure of distance is provider specific.
LUP_RES_SERVICE	This indicates whether prime response is in the remote or local part of <b>CSADDR_INFO</b> structure. The other part needs to be usable in either case.
LUP_RETURN_ALIASES	Any available alias information is to be returned in successive calls to <b>WSALookupServiceNext</b> , and each alias returned will have the <b>RESULT_IS_ALIAS</b> flag set.
LUP_RETURN_NAME	Retrieves the name as <i>IpzServiceInstanceName</i> .
LUP_RETURN_TYPE	Retrieves the type as <i>IpServiceClassId</i> .
LUP_RETURN_VERSION	Retrieves the version as <i>IpVersion</i> .
LUP_RETURN_COMMENT	Retrieves the comment as <i>IpzComment</i> .
LUP_RETURN_ADDR	Retrieves the addresses as <i>IpzsaBuffer</i> .
LUP_RETURN_BLOB	Retrieves the private data as <i>IpBlob</i> .
LUP_RETURN_ALL	Retrieves all of the information.

*lphLookup*

[out] Handle to be used when calling **WSALookupServiceNext** in order to start retrieving the results set.

**Return Values**

The return value is zero if the operation was successful. Otherwise, the value **SOCKET\_ERROR** is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

<b>Error code</b>	<b>Meaning</b>
<b>WSAEINVAL</b>	One or more parameters were missing or invalid for this provider.
<b>WSANO_DATA</b>	The name was found in the database but no data matching the given restrictions was located.
<b>WSANOTINITIALIZED</b>	The <b>Ws2_32.dll</b> has not been initialized. The application must first call <b>WSAStartup</b> before calling any Windows Sockets functions.
<b>WSASERVICE_NOT_FOUND</b>	No such service is known. The service cannot be found in the specified name space.
<b>WSA NOT ENOUGH MEMORY</b>	There was insufficient memory to perform the operation.

**Remarks**

If **LUP\_CONTAINERS** is specified in a call, all other restriction values should be avoided. If any are supplied, it is up to the name service provider to decide if it can support this restriction over the containers. If it cannot, it should return an error.

Some name service providers can have other means of finding containers. For example, containers might all be of some well-known type, or of a set of well-known types, and therefore a query restriction can be created for finding them. No matter what other means the name service provider has for locating containers, **LUP\_CONTAINERS** and **LUP\_NOCONTAINERS** take precedence. Hence, if a query restriction is given that includes containers, specifying **LUP\_NOCONTAINERS** will prevent the container items from being returned. Similarly, no matter the query restriction, if **LUP\_CONTAINERS** is given, only containers should be returned. If a name space does not support containers, and **LUP\_CONTAINERS** is specified, it should simply return **WSANO\_DATA**.

The preferred method of obtaining the containers within another container, is the call:

```
dwStatus = WSALookupServiceBegin(
    lpgsRestrictions,
    LUP_CONTAINERS,
    lphLookup);
```

This call is followed by the requisite number of **WSALookupServiceNext** calls. This will return all containers contained immediately within the starting context; that is, it is not a deep query. With this, one can map the address space structure by walking the hierarchy, perhaps enumerating the content of selected containers. Subsequent uses of **WSALookupServiceBegin** use the containers returned from a previous call.

As mentioned above, a **WSAQUERYSET** structure is used as an input parameter to **WSALookupBegin** in order to qualify the query. The following table indicates how the **WSAQUERYSET** is used to construct a query. When a parameter is marked as *(Optional)* a NULL pointer can be supplied, indicating that the parameter will *not* be used as a search criteria. See section *Query-Related Data Structures* for additional information.

<b>WSAQUERYSET member name</b>	<b>Query interpretation</b>
<b>dwSize</b>	Must be set to sizeof( <b>WSAQUERYSET</b> ). This is a versioning mechanism.
<b>dwOutputflags</b>	Ignored for queries.
<b>LpszServiceInstanceName</b>	<i>(Optional)</i> Referenced string contains service name. The semantics for wildcarding within the string are not defined, but can be supported by certain name space providers.
<b>LpServiceClassId</b>	<i>(Required)</i> The GUID corresponding to the service class.
<b>LpVersion</b>	<i>(Optional)</i> References desired version number and provides version comparison semantics (that is, version must match exactly, or version must be not less than the value supplied).
<b>LpszComment</b>	Ignored for queries.
<b>DwNameSpace<sup>1</sup></b>	Identifier of a single name space in which to constrain the search, or NS_ALL to include all name spaces.
<b>LpNSProviderId</b>	<i>(Optional)</i> References the GUID of a specific name space provider, and limits the query to this provider only.
<b>LpszContext</b>	<i>(Optional)</i> Specifies the starting point of the query in a hierarchical name space.
<b>DwNumberOfProtocols</b>	Size of the protocol constraint array, can be zero.
<b>LpafpProtocols</b>	<i>(Optional)</i> References an array of <b>AFPROTOCOLS</b> structure. Only services that utilize these protocols will be returned.

WSAQUERYSET member name	Query interpretation
<b>LpszQueryString</b>	<i>(Optional)</i> Some name spaces (such as whois++) support enriched SQL-like queries that are contained in a simple text string. This parameter is used to specify that string.
<b>DwNumberOfCsAddrs</b>	Ignored for queries.
<b>LpcsaBuffer</b>	Ignored for queries.
<b>LpBlob</b>	<i>(Optional)</i> This is a pointer to a provider-specific entity.

<sup>1</sup> See the Important note that follows.

**Important** In most instances, applications interested in only a particular transport protocol should constrain their query by address family and protocol rather than by name space. This would allow an application that needs to locate a TCP/IP service, for example, to have its query processed by all available name spaces such as the local hosts file, DNS, and NIS.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT/2000.

#### + See Also

**WSALookupServiceEnd, WSALookupServiceNext**

## WSALookupServiceEnd

The Windows Sockets **WSALookupServiceEnd** function is called to free the handle after previous calls to **WSALookupServiceBegin** and **WSALookupServiceNext**.

If you call **WSALookupServiceEnd** from another thread while an existing **WSALookupServiceNext** is blocked, the end call will have the same effect as a cancel and will cause the **WSALookupServiceNext** call to return immediately.

```
INT WSALookupServiceEnd (
    HANDLE    hLookup
);
```

## Parameters

*hLookup*

[in] Handle previously obtained by calling **WSALookupServiceBegin**.

## Return Values

The return value is zero if the operation was successful. Otherwise, the value **SOCKET\_ERROR** is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSA_INVALID_HANDLE	The handle is not valid.
WSANOTINITIALIZED	The Ws2_32.dll has not been initialized. The application must first call <b>WSAStartup</b> before calling any Windows Sockets functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

### + See Also

**WSALookupServiceBegin**, **WSALookupServiceNext**

# WSALookupServiceNext

The Windows Sockets **WSALookupServiceNext** function is called after obtaining a handle from a previous call to **WSALookupServiceBegin** in order to retrieve the requested service information.

The provider will pass back a **WSAQUERYSET** structure in the *lpqsResults* buffer. The client should continue to call this function until it returns **WSA\_E\_NOMORE**, indicating that all of **WSAQUERYSET** has been returned.

```
INT WSAlookupServiceNext (
    HANDLE          hLookup,
    DWORD           dwControlFlags,
    LPDWORD         lpdwBufferLength,
    LPWSAQUERYSET  lpqsResults
);
```

## Parameters

### *hLookup*

[in] Handle returned from the previous call to **WSALookupServiceBegin**.

### *dwControlFlags*

[in] Flags to control the next operation. Currently only LUP\_FLUSHPREVIOUS is defined as a means to cope with a result set which is too large. If an application does not wish to (or cannot) supply a large enough buffer, setting LUP\_FLUSHPREVIOUS instructs the provider to discard the last result set—which was too large—and move on to the next set for this call.

### *lpdwBufferLength*

[in/out] On input, the number of bytes contained in the buffer pointed to by *lpqsResults*. On output, if the function fails and the error is WSAEFAULT, then it contains the minimum number of bytes to pass for the *lpqsResults* to retrieve the record.

### *lpqsResults*

[out] Pointer to a block of memory, which will contain one result set in a **WSAQUERYSET** structure on return.

## Return Values

The return value is zero if the operation was successful. Otherwise, the value **SOCKET\_ERROR** is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSA_E_NO_MORE	There is no more data available. In Windows Sockets version 2, conflicting error codes are defined for WSAENOMORE (10102) and WSA_E_NO_MORE (10110). The error code WSAENOMORE will be removed in a future version and only WSA_E_NO_MORE will remain. For Windows Sockets version 2, however, applications should check for both WSAENOMORE and WSA_E_NO_MORE for the widest possible compatibility with name-space providers that use either one.
WSA_E_CANCELLED	A call to <b>WSALookupServiceEnd</b> was made while this call was still processing. The call has been canceled. The data in the <i>lpqsResults</i> buffer is undefined. In Windows Sockets version 2, conflicting error codes are defined for WSAECANCELLED (10103) and WSA_E_CANCELLED (10111). The error code WSAECANCELLED will be removed in a future version and only WSA_E_CANCELLED will remain. For Windows Sockets version 2, however, applications should check for both WSAECANCELLED and WSA_E_CANCELLED for the widest possible compatibility with name space providers that use either one.

(continued)



*(continued)*

Error code	Meaning
WSAEFAULT	The <i>IpqsResults</i> buffer was too small to contain a <b>WSAQUERYSET</b> set.
WSAEINVAL	One or more required parameters were invalid or missing.
WSA_INVALID_HANDLE	The specified Lookup handle is invalid.
WSANOTINITIALIZED	The <i>Ws2_32.dll</i> has not been initialized. The application must first call <b>WSAStartup</b> before calling any Windows Sockets functions.
WSANO_DATA	The name was found in the database, but no data matching the given restrictions was located.
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

### Remarks

The *dwControlFlags* parameter specified in this function and the ones specified at the time of **WSALookupServiceBegin** are treated as restrictions for the purpose of combination. The restrictions are combined between the ones at **WSALookupServiceBegin** time and the ones at **WSALookupServiceNext** time. Therefore the flags at **WSALookupServiceNext** can never increase the amount of data returned beyond what was requested at **WSALookupServiceBegin**, although it is *not* an error to specify more or fewer flags. The flags specified at a given **WSALookupServiceNext** apply only to that call.

The *dwControlFlags* LUP\_FLUSHPREVIOUS and LUP\_RES\_SERVICE are exceptions to the combined restrictions rule (because they are behavior flags instead of restriction flags). If either of these flags are used in **WSALookupServiceNext** they have their defined effect regardless of the setting of the same flags at **WSALookupServiceBegin**.

For example, if LUP\_RETURN\_VERSION is specified at **WSALookupServiceBegin** the service provider retrieves records including the version. If LUP\_RETURN\_VERSION is NOT specified at **WSALookupServiceNext**, the returned information does not include the version, even though it was available. No error is generated.

Also for example, if LUP\_RETURN\_BLOB is NOT specified at **WSALookupServiceBegin** but is specified at **WSALookupServiceNext**, the returned information does not include the private data. No error is generated.

### Query Results

The following table describes how the query results are represented in the **WSAQUERYSET** structure.

WSAQUERYSET member name	Result interpretation
<b>dwSize</b>	Will be set to <code>sizeof(WSAQUERYSET)</code> . This is used as a versioning mechanism.
<b>dwOutputFlags</b>	<code>RESULT_IS_ALIAS</code> flag indicates this is an alias result.
<b>LpszServiceInstanceName</b>	Referenced string contains service name.
<b>LpServiceClassId</b>	The GUID corresponding to the service class.
<b>LpVersion</b>	References version number of the particular service instance.
<b>LpszComment</b>	Optional comment string supplied by service instance.
<b>DwNameSpace</b>	Name space in which the service instance was found.
<b>LpNSProviderId</b>	Identifies the specific name space provider that supplied this query result.
<b>LpszContext</b>	Specifies the context point in a hierarchical name space at which the service is located.
<b>DwNumberOfProtocols</b>	Undefined for results.
<b>IpafpProtocols</b>	Undefined for results, all needed protocol information is in the <b>CSADDR_INFO</b> structures.
<b>LpszQueryString</b>	When <i>dwControlFlags</i> includes <code>LUP_RETURN_QUERY_STRING</code> , this parameter returns the unparsed remainder of the <i>LpszServiceInstanceName</i> specified in the original query. For example, in a name space that identifies services by hierarchical names that specify a host name and a file path within that host, the address returned might be the host address and the unparsed remainder might be the file path. If the <i>LpszServiceInstanceName</i> is fully parsed and <code>LUP_RETURN_QUERY_STRING</code> is used, this parameter is <code>NULL</code> or points to a zero-length string.
<b>DwNumberOfCsAddrs</b>	Indicates the number of elements in the array of <b>CSADDR_INFO</b> structures.
<b>Lpcsabuffer</b>	A pointer to an array of <b>CSADDR_INFO</b> structures, with one complete transport address contained within each element.
<b>LpBlob</b>	(Optional) This is a pointer to a provider-specific entity.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT/2000.

**+** See Also

**WSALookupServiceBegin, WSALookupServiceEnd**

## WSANTohl

The Windows Sockets **WSANTohl** function converts a **u\_long** from network byte order to host byte order.

```
int WSANTohl (
    SOCKET          s,
    u_long          netlong,
    u_long FAR     *lphostlong
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*netlong*

[in] 32-bit number in network byte order.

*lphostlong*

[out] Pointer to a 32-bit number in host byte order.

### Return Values

If no error occurs, **WSANTohl** returns zero. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lphostlong</i> parameter is not completely contained in a valid part of the user address space.

### Remarks

The **WSANTohl** function takes a 32-bit number in the network byte order associated with socket *s* and returns a 32-bit number pointed to by the *lphostlong* parameter in host byte order.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

**+** See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **htonl**, **htons**, **ntohl**, **ntohs**, **WSAhtonl**, **WSAhtons**, **WSANtohs**

## WSANtohs

The Windows Sockets **WSANtohs** function converts a **u\_short** from network byte order to host byte order.

```
int WSANtohs (
    SOCKET          s,
    u_short         netshort,
    u_short FAR    *lphostshort
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*netshort*

[in] 16-bit number in network byte order.

*lphostshort*

[out] Pointer to a 16-bit number in host byte order.

### Return Values

If no error occurs, **WSANtohs** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

### Error Codes

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lphostshort</i> parameter is not completely contained in a valid part of the user address space.

## Remarks

The **WSANTohs** function takes a 16-bit number in the network byte order associated with socket *s* and returns a 16-bit number pointed to by the *lphostshort* parameter in host byte order.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **htonl**, **htons**, **ntohl**, **ntohs**, **WSAHtonl**, **WSANTohl**, **WSAHtons**

# WSAProviderConfigChange

The Windows Sockets **WSAProviderConfigChange** function notifies the application when the provider configuration is changed.

```
int WINAPI
WSAProviderConfigChange (
    LPHANDLE          lpNotificationHandle,
    LPWSAOVERLAPPED  lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

## Parameters

### *lpNotificationHandle*

[in/out] Pointer to notification handle. If the notification handle is set to NULL (the handle value not the pointer itself), this function returns a notification handle in the location pointed to by *lpNotificationHandle*.

### *lpOverlapped*

[in] Pointer to a **WSAOVERLAPPED** structure.

### *lpCompletionRoutine*

[in] Pointer to the completion routine called when the provider change notification is received.

## Return Values

If no error occurs the **WSAProviderConfigChange** returns 0. Otherwise, a value of **SOCKET\_ERROR** is returned and a specific error code may be retrieved by calling **WSAGetLastError**. The error code **WSA\_IO\_PENDING** indicates that the overlapped operation has been successfully initiated and that completion (and thus change event) will be indicated at a later time.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to complete the operation.
WSA_INVALID_HANDLE	Value pointed by <i>IpNotificationHandle</i> parameter is not a valid notification handle.
WSAEOPNOTSUPP	Current operating system environment does not support provider installation or removal without restart.

### Remarks

The **WSAProviderConfigChange** function notifies the application of provider (both transport and name-space) installation or removal in Win32 operating environments that support such configuration change without requiring a restart. When called for the first time (*IpNotificationHandle* parameter points to NULL handle), this function completes immediately and returns notification handle in the location pointed by *IpNotificationHandle* that can be used in subsequent calls to receive notifications of provider installation and removal. The second and any subsequent calls only complete when provider information changes since the time the call was made. It is expected (but not required) that that application uses overlapped I/O on second and subsequent calls to **WSAProviderConfigChange**, in which case the call will return immediately and application will be notified of provider configuration changes using the completion mechanism chosen through specified overlapped completion parameters.

Notification handle returned by **WSAProviderConfigChange** is like any regular operating system handle that should be closed (when no longer needed) using Win32 **CloseHandle** call.

The following sequence of actions can be used to guarantee that application always has current protocol configuration information:

- Call **WSAProviderConfigChange**
- Call **WSAEnumProtocols** and/or **WSAEnumNameSpaceProviders**
- Whenever **WSAProviderConfigChange** notifies application of provider configuration change (through blocking or overlapped I/O), the whole sequence of actions should be repeated.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws\_32.lib.

## + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSAEnumProtocols**, **WSAEnumNameSpaceProviders**

# WSARecv

The Windows Sockets **WSARecv** function receives data from a connected socket.

```
int WSARecv (
    SOCKET                s,
    LPWSABUF             lpBuffers,
    DWORD                dwBufferCount,
    LPDWORD              lpNumberOfBytesRecvd,
    LPDWORD              lpFlags,
    LPWSAOVERLAPPED     lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

## Parameters

*s*

[in] Descriptor identifying a connected socket.

*lpBuffers*

[in/out] Pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer.

*dwBufferCount*

[in] Number of **WSABUF** structures in the *lpBuffers* array.

*lpNumberOfBytesRecvd*

[out] Pointer to the number of bytes received by this call if the receive operation completes immediately.

*lpFlags*

[in/out] Pointer to flags.

*lpOverlapped*

[in] Pointer to a **WSAOVERLAPPED** structure (ignored for nonoverlapped sockets).

*lpCompletionRoutine*

[in] Pointer to the completion routine called when the receive operation has been completed (ignored for nonoverlapped sockets).

## Return Values

If no error occurs and the receive operation has completed immediately, **WSARecv** returns zero. In this case, the completion routine will have already been scheduled to be called once the calling thread is in the alertable state. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling

**WSAGetLastError.** The error code `WSA_IO_PENDING` indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

Error code	Meaning
<code>WSANOTINITIALISED</code>	A successful <b>WSAStartup</b> call must occur before using this function.
<code>WSAENETDOWN</code>	The network subsystem has failed.
<code>WSAENOTCONN</code>	The socket is not connected.
<code>WSAEINTR</code>	The (blocking) call was canceled through <b>WSACancelBlockingCall</b> .
<code>WSAEINPROGRESS</code>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<code>WSAENETRESET</code>	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
<code>WSAENOTSOCK</code>	The descriptor is not a socket.
<code>WSAEFAULT</code>	The <i>lpBuffers</i> parameter is not completely contained in a valid part of the user address space.
<code>WSAEOPNOTSUPP</code>	<code>MSG_OOB</code> was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
<code>WSAESHUTDOWN</code>	The socket has been shut down; it is not possible to call <b>WSARecv</b> on a socket after <b>shutdown</b> has been invoked with <i>how</i> set to <code>SD_RECEIVE</code> or <code>SD_BOTH</code> .
<code>WSAEWOULDBLOCK</code>	Overlapped sockets: there are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the receive operation cannot be completed immediately.
<code>WSAEMSGSIZE</code>	The message was too large to fit into the specified buffer and (for unreliable protocols only) any trailing portion of the message that did not fit into the buffer has been discarded.
<code>WSAEINVAL</code>	The socket has not been bound (for example, with <b>bind</b> ).
<code>WSAECONNABORTED</code>	The virtual circuit was terminated due to a time-out or other failure.
<code>WSAECONNRESET</code>	The virtual circuit was reset by the remote side.
<code>WSAEDISCON</code>	Socket <i>s</i> is message oriented and the virtual circuit was gracefully closed by the remote side.

(continued)



*(continued)*

Error code	Meaning
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket.

### Remarks

The **WSARecv** function provides functionality over and above the standard **recv** function in three important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped **recv** operations.
- It allows multiple receive buffers to be specified making it applicable to the scatter/gather type of I/O.
- The *IpFlags* parameter is both an input and an output parameter, allowing applications to sense the output state of the MSG\_PARTIAL flag bit. However, the MSG\_PARTIAL flag bit is not supported by all protocols.

The **WSARecv** function is used on connected sockets or bound connectionless sockets specified by the *s* parameter and is used to read incoming data. The socket's local address must be known. For server applications, this is usually done explicitly through **bind** or implicitly through **accept** or **WSAAccept**. Explicit binding is discouraged for client applications. For client applications the socket can become bound implicitly to a local address through **connect**, **WSAConnect**, **sendto**, **WSASendTo**, or **WSAJoinLeaf**.

For connected, connectionless sockets, this function restricts the addresses from which received messages are accepted. The function only returns messages from the remote address specified in the connection. Messages from other addresses are (silently) discarded.

For overlapped sockets, **WSARecv** is used to post one or more buffers into which incoming data will be placed as it becomes available, after which the application-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or **WSAGetOverlappedResult**.

If both *IpOverlapped* and *IpCompletionRoutine* are NULL, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the blocking semantics are identical to that of the standard **recv** function and the *IpOverlapped* and *IpCompletionRoutine* parameters are ignored. Any data that has already been received and buffered by the transport will be copied into the supplied user buffers. In the case of a blocking socket with no data currently having been received and buffered by the transport, the call will block until data is received.

Windows Socket 2 does not define any standard blocking time-out mechanism for this function. For protocols acting as byte-stream protocols the stack tries to return as much data as possible subject to the supplied buffer space and amount of received data available. However, receipt of a single byte is sufficient to unblock the caller. There is no guarantee that more than a single byte will be returned. For protocols acting as message-oriented, a full message is required to unblock the caller.

Whether or not a protocol is acting as byte stream is determined by the setting of `XP1_MESSAGE_ORIENTED` and `XP1_PSEUDO_STREAM` in its **WSAPROTOCOL\_INFO** structure and the setting of the `MSG_PARTIAL` flag passed in to this function (for protocols that support it). The relevant combinations are summarized in the following table, (an asterisk "\*" indicates that the setting of this bit does not matter in this case).

<code>XP1_MESSAGE_ORIENTED</code>	<code>XP1_PSEUDO_STREAM</code>	<code>MSG_PARTIAL</code>	Acts as
not set	*	*	byte stream
*	set	*	byte stream
set	not set	set	byte stream
set	not set	not set	message oriented

The supplied buffers are filled in the order in which they appear in the array pointed to by *lpBuffers*, and the buffers are packed so that no holes are created.

The array of **WSABUF** structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For byte stream-style sockets (for example, type **SOCK\_STREAM**), incoming data is placed into the buffers until the buffers are filled, the connection is closed, or the internally buffered data is exhausted. Regardless of whether or not the incoming data fills all the buffers, the completion indication occurs for overlapped sockets.

For message-oriented sockets (for example, type **SOCK\_DGRAM**), an incoming message is placed into the supplied buffers up to the total size of the buffers supplied, and the completion indication occurs for overlapped sockets. If the message is larger than the buffers supplied, the buffers are filled with the first part of the message. If the `MSG_PARTIAL` feature is supported by the underlying service provider, the `MSG_PARTIAL` flag is set in *lpFlags* and subsequent receive operations will retrieve the rest of the message. If `MSG_PARTIAL` is not supported but the protocol is reliable, **WSARecv** generates the error `WSAEMSGSIZE` and a subsequent receive operation with a larger buffer can be used to retrieve the entire message. Otherwise, (that is, the protocol is unreliable and does not support `MSG_PARTIAL`), the excess data is lost, and **WSARecv** generates the error `WSAEMSGSIZE`.

For connection-oriented sockets, **WSARecv** can indicate the graceful termination of the virtual circuit in one of two ways that depend on whether the socket is byte stream or message oriented. For byte streams, zero bytes having been read (as indicated by a zero return value to indicate success, and *lpNumberOfBytesRecv* value of zero) indicates graceful closure and that no more bytes will ever be read. For message-oriented sockets, where a zero byte message is often allowable, a failure with an error code of WSAEDISCON is used to indicate graceful closure. In any case a return error code of WSAECONNRESET indicates an abortive close has occurred.

The *lpFlags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *lpFlags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

Value	Meaning
MSG_PEEK	Peeks at the incoming data. The data is copied into the buffer but is not removed from the input queue. This flag is valid only for nonoverlapped sockets.
MSG_OOB	Processes OOB data. (See section <i>DECnet Out-Of-band data</i> for a discussion of this topic.)
MSG_PARTIAL	This flag is for message-oriented sockets only. On output, indicates that the data supplied is a portion of the message transmitted by the sender. Remaining portions of the message will be supplied in subsequent receive operations. A subsequent receive operation with MSG_PARTIAL flag cleared indicates end of sender's message.  As an input parameter, this flag indicates that the receive operation should complete even if only part of a message has been received by the service provider.

For message-oriented sockets, the MSG\_PARTIAL bit is set in the *lpFlags* parameter if a partial message is received. If a complete message is received, MSG\_PARTIAL is cleared in *lpFlags*. In the case of delayed completion, the value pointed to by *lpFlags* is not updated. When completion has been indicated, the application should call **WSAGetOverlappedResult** and examine the flags indicated by the *lpdwFlags* parameter.

### Overlapped Socket I/O

If an overlapped operation completes immediately, **WSARecv** returns a value of zero and the *lpNumberOfBytesRecv* parameter is updated with the number of bytes received and the flag bits indicated by the *lpFlags* parameter are also updated. If the overlapped operation is successfully initiated and will complete later, **WSARecv** returns SOCKET\_ERROR and indicates error code WSA\_IO\_PENDING. In this case, *lpNumberOfBytesRecv* and *lpFlags* are not updated. When the overlapped operation completes, the amount of data transferred is indicated either through the *cbTransferred*

parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **WSAGetOverlappedResult**. Flag values are obtained by examining the *lpdwFlags* parameter of **WSAGetOverlappedResult**.

The **WSARecv** function can be called from within the completion routine of a previous **WSARecv**, **WSARecvFrom**, **WSASend** or **WSASendTo** function. For a given socket, I/O completion routines will *not* be nested. For a given socket, I/O completion routines will *not* be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate **WSAOVERLAPPED** structure.

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents** or **WSAGetOverlappedResult** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to TRUE. In this case the usage of the **hEvent** parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will *not* be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents** with the *fAlertable* parameter set to TRUE is invoked.

The transport providers allow an application to invoke send and receive operations from within the context of the socket I/O completion routine, and guarantee that, for a given socket, I/O completion routines will *not* be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is as follows:

```
void CALLBACK CompletionROUTINE (  
    IN DWORD          dwError,  
    IN DWORD          cbTransferred,  
    IN LPWSAOVERLAPPED lpOverlapped,  
    IN DWORD          dwFlags  
);
```

**CompletionRoutine** is a placeholder for an application-defined or library-defined function name. The *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* parameter specifies the number of

bytes received. The *dwFlags* parameter contains information that would have appeared in *lpFlags* if the receive operation had completed immediately. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. When using **WSAWaitForMultipleEvents**, all waiting completion routines are called before the alertable thread's wait is satisfied with a return code of `WSA_IO_COMPLETION`. The completion routines can be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order in which they are supplied.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSACloseEvent**, **WSACreateEvent**, **WSAGetOverlappedResult**, **WSASocket**, **WSAWaitForMultipleEvents**

---

## WSARecvDisconnect

The Windows Sockets **WSARecvDisconnect** function terminates reception on a socket, and retrieves the disconnect data if the socket is connection oriented.

```
int WSARecvDisconnect (
    SOCKET      s,
    LPWSABUF   lpInboundDisconnectData
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*lpInboundDisconnectData*

[out] Pointer to the incoming disconnect data.

### Return Values

If no error occurs, **WSARecvDisconnect** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

---

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The buffer referenced by the parameter <i>lpInboundDisconnectData</i> is too small.
WSAENOPROTOOPT	The disconnect data is not supported by the indicated protocol family.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.

### Remarks

The **WSARecvDisconnect** function is used on connection-oriented sockets to disable reception and retrieve any incoming disconnect data from the remote party. This is equivalent to a shutdown (`SD_RECV`), except that **WSASendDisconnect** also allows receipt of disconnect data (in protocols that support it).

After this function has been successfully issued, subsequent receives on the socket will be disallowed. Calling **WSARecvDisconnect** has no effect on the lower protocol layers. For TCP sockets, if there is still data queued on the socket waiting to be received, or data arrives subsequently, the connection is reset, since the data cannot be delivered to the user. For UDP, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated.

To successfully receive incoming disconnect data, an application must use other mechanisms to determine that the circuit has been closed. For example, an application needs to receive an `FD_CLOSE` notification, to receive a zero return value, or to receive a `WSAEDISCON` or `WSAECONNRESET` error code from **recv/WSARecv**.

The **WSARecvDisconnect** function does not close the socket, and resources attached to the socket will *not* be freed until **closesocket** is invoked.

The **WSARecvDisconnect** function does not block regardless of the `SO_LINGER` setting on the socket.

An application should not rely on being able to reuse a socket after it has been disconnected using **WSARecvDisconnect**. In particular, a Windows Sockets provider is not required to support the use of **connect/WSAConnect** on such a socket.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

**connect, socket**

---

## WSARecvEx

The Windows Sockets **WSARecvEx** function is identical to the **recv** function, except that the *flags* parameter is an [in, out] parameter. When a partial message is received while using datagram protocol, the MSG\_PARTIAL bit is set in the *flags* parameter on return from the function.

---

**Note** The Windows Sockets **WSARecvEx** function is a Microsoft-specific extension to the Windows Sockets specification. For more information, see *Microsoft Extensions and Windows Sockets 2*.

---

```
int PASCAL FAR WSARecvEx (  
    SOCKET      s,  
    char FAR   *buf,  
    int        len,  
    int        *flags  
);
```

### Parameters

*s*

[in] Descriptor identifying a connected socket.

*buf*

[out] Buffer for the incoming data.

*len*

[in] Length of *buf*.

*flags*

[in/out] Indicator specifying whether the message is fully or partially received for datagram sockets.

## Return Values

If no error occurs, **WSARecvEx** returns the number of bytes received. If the connection has been closed, it returns zero. Additionally, if a partial message was received, the `MSG_PARTIAL` bit is set in the *flags* parameter. If a complete message was received, `MSG_PARTIAL` is not set in *flags*.

Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

---

**Important** For a stream oriented-transport protocol, `MSG_PARTIAL` is never set on return from **WSARecvEx**. This function behaves identically to the Windows Sockets `recv` function for stream-transport protocols.

---

Error code	Meaning
<code>WSANOTINITIALISED</code>	A successful <b>WSAStartup</b> call must occur before using this function.
<code>WSAENETDOWN</code>	The network subsystem has failed.
<code>WSAEFAULT</code>	The <i>buf</i> parameter is not completely contained in a valid part of the user address space.
<code>WSAENOTCONN</code>	The socket is not connected.
<code>WSAEINTR</code>	The (blocking) call was canceled through <b>WSACancelBlockingCall</b> .
<code>WSAEINPROGRESS</code>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<code>WSAENETRESET</code>	The connection has been broken due to the remote host resetting.
<code>WSAENOTSOCK</code>	The descriptor is not a socket.
<code>WSAEOPNOTSUPP</code>	<code>MSG_OOB</code> was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
<code>WSAESHUTDOWN</code>	The socket has been shut down; it is not possible to use <b>WSARecvEx</b> on a socket after <b>shutdown</b> has been invoked with <i>how</i> set to <code>SD_RECEIVE</code> or <code>SD_BOTH</code> .
<code>WSAEWOULDBLOCK</code>	The socket is marked as nonblocking and the receive operation would block.
<code>WSAEINVAL</code>	The socket has not been bound with <b>bind</b> , or an unknown flag was specified, or <code>MSG_OOB</code> was specified for a socket with <code>SO_OOBINLINE</code> enabled or (for byte stream sockets only) <i>len</i> was zero or negative.

(continued)



(continued)

Error code	Meaning
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
WSAETIMEDOUT	The connection has been dropped because of a network failure or because the peer system failed to respond.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer usable. On a UPD-datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.

### Remarks

The **WSARecvEx** function that is part of the Microsoft implementation of Windows Sockets 2 is similar to the more common **recv** function except that the *flags* parameter is an in-out parameter, not just an in parameter. The additional out parameter is used to indicate whether a partial or complete message is received when a message-oriented protocol is being used.

The **WSARecvEx** and **recv** function identically for stream-oriented protocols.

Making the *flags* parameter an in-out parameter accommodates two common situations in which a partial message will be received:

- When the application's data buffer size is smaller than the message size and the message coincidentally arrives in two pieces.
- When the message is rather large and must arrive in several pieces.

The MSG\_PARTIAL bit is set in the *flags* parameter on return from **WSARecvEx** when a partial message was received. If a complete message was received, MSG\_PARTIAL is not set in *flags*.

The Windows Sockets **recv** function is different than **WSARecvEx** in that the **recv** function always receives a single message for each call for message-oriented transport protocols. The **recv** function does not have a means to indicate to the application that the data received is only a partial message. An application must build its own protocol for checking whether a message is partial or complete by checking for the error code WSAEMSGSIZE after each call to **recv**. When the application buffer is smaller than the data being sent, as much of the message as will fit is copied into the user's buffer and **recv** returns with the error code WSAEMSGSIZE. A subsequent call to **recv** will get the next part of the message.

Applications written for message-oriented transport protocols should be coded for this possibility if message sizing is not guaranteed by the application's data transfer protocol. An application can use **recv** and manage the protocol itself. Alternatively, an application can use **WSARecvEx** and check that the `MSG_PARTIAL` bit is set in the *flags* parameter.

The **WSARecvEx** function provides the developer with a more effective way of checking whether a message received is partial or complete when a very large message arrives incrementally. For example, if an application sends a one-megabyte message, the transport protocol must break up the message in order to send it over the physical network. It is theoretically possible for the transport protocol on the receiving side to buffer all the data in the message, but this would be quite expensive in terms of resources. Instead, **WSARecvEx** can be used, minimizing overhead and eliminating the need for an application-based protocol.

#### ! Requirements

**Version:** Requires Windows Sockets 1.1. A Microsoft-specific extension. Not supported on Windows 95.

**Header:** Declared in `Mswsock.h`.

**Library:** Use `Mswsock.lib`.

#### + See Also

**recvfrom**, **select**, **send**, **socket**, **WSAAsyncSelect**

## WSARecvFrom

The Windows Sockets **WSARecvFrom** function receives a datagram and stores the source address.

```
int WSARecvFrom (
    SOCKET                s,
    LPWSABUF             lpBuffers,
    DWORD                dwBufferCount,
    LPDWORD              lpNumberOfBytesRecvd,
    LPDWORD              lpFlags,
    struct sockaddr FAR *lpFrom,
    LPINT                lpFromlen,
    LPWSAOVERLAPPED     lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*lpBuffers*

[in/out] Pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer.

*dwBufferCount*

[in] Number of **WSABUF** structures in the *lpBuffers* array.

*lpNumberOfBytesRecvd*

[out] Pointer to the number of bytes received by this call if the **recv** operation completes immediately.

*lpFlags*

[in/out] Pointer to flags.

*lpFrom*

[out] Optional pointer to a buffer that will hold the source address upon the completion of the overlapped operation.

*lpFromlen*

[in/out] Pointer to the size of the *from* buffer, required only if *lpFrom* is specified.

*lpOverlapped*

[in] Pointer to a **WSAOVERLAPPED** structure (ignored for nonoverlapped sockets).

*lpCompletionRoutine*

[in] Pointer to the completion routine called when the **recv** operation has been completed (ignored for nonoverlapped sockets).

**Return Values**

If no error occurs and the receive operation has completed immediately, **WSARecvFrom** returns zero. In this case, the completion routine will have already been scheduled to be called once the calling thread is in the alertable state. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**. The error code **WSA\_IO\_PENDING** indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

<b>Error code</b>	<b>Meaning</b>
<b>WSANOTINITIALISED</b>	A successful <b>WSAStartup</b> call must occur before using this function.
<b>WSAENETDOWN</b>	The network subsystem has failed.
<b>WSAEFAULT</b>	The <i>lpBuffers</i> , <i>lpFlags</i> , <i>lpFrom</i> , <i>lpNumberOfBytesRecvd</i> , <i>lpFromlen</i> , <i>lpOverlapped</i> , or <i>lpCompletionRoutine</i> argument is not totally contained in a valid part of the user address space: the <i>lpFrom</i> buffer was too small to accommodate the peer address.
<b>WSAEINTR</b>	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
<b>WSAEINPROGRESS</b>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.

Error code	Meaning
WSAEINVAL	The socket has not been bound (with <b>bind</b> , for example).
WSAEISCONN	The socket is connected. This function is not permitted with a connected socket, whether the socket is connection-oriented or connectionless.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to <b>WSARecvFrom</b> on a socket after <b>shutdown</b> has been invoked with <i>how</i> set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: There are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the receive operation cannot be completed immediately.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and (for unreliable protocols only) any trailing portion of the message that did not fit into the buffer has been discarded.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer useable. On a UDP-datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEDISCON	Socket <i>s</i> is message oriented and the virtual circuit was gracefully closed by the remote side.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket.

### Remarks

The **WSARecvFrom** function provides functionality over and above the standard **recvfrom** function in three important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped receive operations.
- It allows multiple receive buffers to be specified making it applicable to the scatter/gather type of I/O.

- The *IpFlags* parameter is both an input and an output parameter, allowing applications to sense the output state of the MSG\_PARTIAL flag bit. Note however, that the MSG\_PARTIAL flag bit is not supported by all protocols.

The **WSARecvFrom** function is used primarily on a connectionless socket specified by *s*. The socket's local address must be known. For server applications, this is usually done explicitly through **bind**. Explicit binding is discouraged for client applications. For client applications using this function the socket can become bound implicitly to a local address through **sendto**, **WSASendTo**, or **WSAJoinLeaf**.

For overlapped sockets, this function is used to post one or more buffers into which incoming data will be placed as it becomes available on a (possibly connected) socket, after which the application-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or **WSAGetOverlappedResult**. Also, the values indicated by *IpFrom* and *IpFromlen* are not updated until completion is itself indicated. Applications must not use or disturb these values until they have been updated, therefore the application must not use automatic (that is, stack-based) variables for these parameters.

If both *IpOverlapped* and *IpCompletionRoutine* are NULL, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the blocking semantics are identical to that of the standard **WSARecv** function and the *IpOverlapped* and *IpCompletionRoutine* parameters are ignored. Any data that has already been received and buffered by the transport will be copied into the supplied user buffers. For the case of a blocking socket with no data currently having been received and buffered by the transport, the call will block until data is received.

The supplied buffers are filled in the order in which they appear in the array indicated by *IpBuffers*, and the buffers are packed so that no holes are created.

The array of **WSABUF** structures pointed to by the *IpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For connectionless socket types, the address from which the data originated is copied to the buffer indicated by *IpFrom*. The value pointed to by *IpFromlen* is initialized to the size of this buffer, and is modified on completion to indicate the actual size of the address stored there. As noted previously for overlapped sockets, the *IpFrom* and *IpFromlen* parameters are not updated until after the overlapped I/O has completed. The memory pointed to by these parameters must, therefore, remain available to the service provider and cannot be allocated on the application's stack frame. The *IpFrom* and *IpFromlen* parameters are ignored for connection-oriented sockets.

For byte stream-style sockets (for example, type **SOCK\_STREAM**), incoming data is placed into the buffers until:

- The buffers are filled.
- The connection is closed.
- The internally buffered data is exhausted.

Regardless of whether or not the incoming data fills all the buffers, the completion indication occurs for overlapped sockets. For message-oriented sockets, an incoming message is placed into the supplied buffers up to the total size of the buffers supplied, and the completion indication occurs for overlapped sockets. If the message is larger than the buffers supplied, the buffers are filled with the first part of the message. If the `MSG_PARTIAL` feature is supported by the underlying service provider, the `MSG_PARTIAL` flag is set in `lpFlags` and subsequent receive operation(s) will retrieve the rest of the message. If `MSG_PARTIAL` is not supported but the protocol is reliable, **WSARecvFrom** generates the error `WSAEMSGSIZE` and a subsequent receive operation with a larger buffer can be used to retrieve the entire message. Otherwise, (that is, the protocol is unreliable and does not support `MSG_PARTIAL`), the excess data is lost, and **WSARecvFrom** generates the error `WSAEMSGSIZE`.

The `lpFlags` parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the `lpFlags` parameter. The latter is constructed by using the bitwise OR operator with any of any of the following values.

Value	Meaning
<code>MSG_PEEK</code>	Peeks at the incoming data. The data is copied into the buffer but is not removed from the input queue. This flag is valid only for nonoverlapped sockets.
<code>MSG_OOB</code>	Processes OOB data. (See DECnet Out-Of-band data for more information.)
<code>MSG_PARTIAL</code>	This flag is for message-oriented sockets only. On output, this flag indicates that the data supplied is a portion of the message transmitted by the sender. Remaining portions of the message will be supplied in subsequent receive operations. A subsequent receive operation with <code>MSG_PARTIAL</code> flag cleared indicates the end of the sender's message.  As an input parameter, this flag indicates that the receive operation should complete even if only part of a message has been received by the service provider.

For message-oriented sockets, the `MSG_PARTIAL` bit is set in the `lpFlags` parameter if a partial message is received. If a complete message is received, `MSG_PARTIAL` is cleared in `lpFlags`. In the case of delayed completion, the value pointed to by `lpFlags` is not updated. When completion has been indicated the application should call **WSAGetOverlappedResult** and examine the flags pointed to by the `lpdwFlags` parameter.

## Overlapped Socket I/O

If an overlapped operation completes immediately, **WSARecvFrom** returns a value of zero and the *lpNumberOfBytesRecv*d parameter is updated with the number of bytes received and the flag bits pointed by the *lpFlags* parameter are also updated. If the overlapped operation is successfully initiated and will complete later, **WSARecvFrom** returns **SOCKET\_ERROR** and indicates error code **WSA\_IO\_PENDING**. In this case, *lpNumberOfBytesRecv*d and *lpFlags* is not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **WSAGetOverlappedResult**. Flag values are obtained either through the *dwFlags* parameter of the completion routine, or by examining the *lpdwFlags* parameter of **WSAGetOverlappedResult**.

The **WSARecvFrom** function can be called from within the completion routine of a previous **WSARecv**, **WSARecvFrom**, **WSASend**, or **WSASendTo** function. For a given socket, I/O completion routines will *not* be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate **WSAOVERLAPPED** structure.

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents** or **WSAGetOverlappedResult** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to TRUE. In this case the usage of the *hEvent* parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will *not* be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents** with the *fAlertable* parameter set to TRUE is invoked.

The transport providers allow an application to invoke send and receive operations from within the context of the socket I/O completion routine, and guarantee that, for a given socket, I/O completion routines will *not* be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is on the following page.

```
void CALLBACK CompletionROUTINE (  
    IN DWORD          dwError,  
    IN DWORD          cbTransferred,  
    IN LPWSAOVERLAPPED lpOverlapped,  
    IN DWORD          dwFlags  
);
```

The **CompletionRoutine** is a placeholder for an application-defined or library-defined function name. The *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* specifies the number of bytes received. The *dwFlags* parameter contains information that would have appeared in *lpFlags* if the receive operation had completed immediately. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. When using **WSAWaitForMultipleEvents**, all waiting completion routines are called before the alertable thread's wait is satisfied with a return code of `WSA_IO_COMPLETION`. The completion routines can be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order they are supplied.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSACloseEvent**, **WSACreateEvent**, **WSAGetOverlappedResult**, **WSASocket**, **WSAWaitForMultipleEvents**

---

## WSARemoveServiceClass

The Windows Sockets **WSARemoveServiceClass** function permanently removes from the registry service class schema.

```
INT WSARemoveServiceClass (  
    LPGUID lpServiceClassId  
);
```

### Parameters

*lpServiceClassId*

[in] Pointer to the GUID for the service class you want to remove.



## Return Values

The return value is zero if the operation was successful. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
<code>WSATYPE_NOT_FOUND</code>	The specified class was not found.
<code>WSAEACCES</code>	The calling routine does not have sufficient privileges to remove the Service.
<code>WSANOTINITIALIZED</code>	The <code>Ws2_32.dll</code> has not been initialized. The application must first call <b>WSAStartup</b> before calling any Windows Sockets functions.
<code>WSAEINVAL</code>	The specified GUID was not valid.
<code>WSA NOT ENOUGH MEMORY</code>	There was insufficient memory to perform the operation

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

## WSAResetEvent

The Windows Sockets **WSAResetEvent** function resets the state of the specified event object to nonsignaled.

```
BOOL WSAResetEvent (
    WSAEVENT hEvent
);
```

### Parameters

*hEvent*

[in] Handle that identifies an open event object handle.

### Return Values

If the **WSAResetEvent** function succeeds, the return value is `TRUE`. If the function fails, the return value is `FALSE`. To get extended error information, call **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_INVALID_HANDLE	The <i>hEvent</i> parameter is not a valid event object handle.

### Remarks

The **WSAResetEvent** function is used to set the state of the event object to nonsignaled.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSACloseEvent**, **WSACreateEvent**, **WSASetEvent**

## WSASend

The Windows Sockets **WSASend** function sends data on a connected socket.

```
int WSASend (
    SOCKET          s,
    LPWSABUF       lpBuffers,
    DWORD          dwBufferCount,
    LPDWORD        lpNumberOfBytesSent,
    DWORD          dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

### Parameters

**s**

[in] Descriptor identifying a connected socket.

*lpBuffers*

[in] Pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer. This array must remain valid for the duration of the send operation.

*dwBufferCount*

[in] Number of **WSABUF** structures in the *lpBuffers* array.

*lpNumberOfBytesSent*

[out] Pointer to the number of bytes sent by this call if the I/O operation completes immediately.

*dwFlags*

[in] Flags used to modify the behavior of the **WSASend** function call. See **Using dwFlags** in the **Remarks** section for more information.

*lpOverlapped*

[in] Pointer to a **WSAOVERLAPPED** structure. This parameter is ignored for nonoverlapped sockets.

*lpCompletionRoutine*

[in] Pointer to the completion routine called when the send operation has been completed. This parameter is ignored for nonoverlapped sockets.

## Return Values

If no error occurs and the send operation has completed immediately, **WSASend** returns zero. In this case, the completion routine will have already been scheduled to be called once the calling thread is in the alertable state. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**. The error code **WSA\_IO\_PENDING** indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>lpBuffers</i> , <i>lpNumberOfBytesSent</i> , <i>lpOverlapped</i> , <i>lpCompletionRoutine</i> argument is not totally contained in a valid part of the user address space.

Error code	Meaning
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOBUFS	The Windows Sockets provider reports a buffer deadlock.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, MSG_PARTIAL is not supported, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to <b>WSASend</b> on a socket after <b>shutdown</b> has been invoked with how set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: There are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the send operation cannot be completed immediately.
WSAEMSGSIZE	The socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	The socket has not been bound with <b>bind</b> or the socket is not created with the overlapped flag.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket, or the execution of the <b>SIO_FLUSH</b> command in <b>WSAIoctl</b> .

### Remarks

The **WSASend** function provides functionality over and above the standard **send** function in two important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped send operations.
- It allows multiple send buffers to be specified making it applicable to the scatter/gather type of I/O.

The **WSASend** function is used to write outgoing data from one or more buffers on a connection-oriented socket specified by *s*. It can also be used, however, on connectionless sockets that have a stipulated default peer address established through the **connect** or **WSAConnect** function.

For overlapped sockets (created using **WSASocket** with flag `WSA_FLAG_OVERLAPPED`) sending information uses overlapped I/O, unless both *lpOverlapped* and *lpCompletionRoutine* are NULL. In that case, the socket is treated as a nonoverlapped socket. A completion indication will occur, invoking the completion of a routine or setting of an event object, when the supplied buffer(s) have been consumed by the transport. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or **WSAGetOverlappedResult**.

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the last two parameters (*lpOverlapped*, *lpCompletionRoutine*) are ignored and **WSASend** adopts the same blocking semantics as **send**. Data is copied from the supplied buffer(s) into the transport's buffer. If the socket is nonblocking and stream oriented, and there is not sufficient space in the transport's buffer, **WSASend** will return with only part of the application's buffers having been consumed. Given the same buffer situation and a blocking socket, **WSASend** will block until all of the application's buffer contents have been consumed.

The array of **WSABUF** structures pointed to by the *lpBuffers* parameter is transient. If this operation is completed in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For message-oriented sockets, care must be taken not to exceed the maximum message size of the underlying provider, which can be obtained by getting the value of socket option `SO_MAX_MSG_SIZE`. If the data is too long to pass atomically through the underlying protocol the error `WSAEMSGSIZE` is returned, and no data is transmitted.

---

**Note** The successful completion of a **WSASend** does not indicate that the data was successfully delivered.

---

### Using *dwFlags*

The *dwFlags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *dwFlags* parameter. The latter is constructed by using the bitwise OR operator with any of any of the following values.

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Send OOB data on a stream-style socket such as <b>SOCK_STREAM</b> only. (See DECnet Out-Of-band data for a discussion of this topic.)
MSG_PARTIAL	Specifies that <i>lpBuffers</i> only contains a partial message. Note that the error code WSAEOPNOTSUPP will be returned by transports that do not support partial message transmissions.

### Overlapped Socket I/O

If an overlapped operation completes immediately, **WSASend** returns a value of zero and the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent. If the overlapped operation is successfully initiated and will complete later, **WSASend** returns SOCKET\_ERROR and indicates error code WSA\_IO\_PENDING. In this case, *lpNumberOfBytesSent* is not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **WSAGetOverlappedResult**.

The **WSASend** function can be called from within the completion routine of a previous **WSARecv**, **WSARecvFrom**, **WSASend**, or **WSASendTo** function. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate **WSAOVERLAPPED** structure.

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents** or **WSAGetOverlappedResult** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to TRUE. In this case the usage of the *hEvent* parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will *not* be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents** with the *fAlertable* parameter set to TRUE is invoked.

The transport providers allow an application to invoke send and receive operations from within the context of the socket I/O completion routine, and guarantee that, for a given socket, I/O completion routines will *not* be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is as follows:

```
void CALLBACK CompletionROUTINE(  
    IN  DWORD          dwError,  
    IN  DWORD          cbTransferred,  
    IN  LPWSAOVERLAPPED lpOverlapped,  
    IN  DWORD          dwFlags  
);
```

The **CompletionRoutine** function is a placeholder for an application-defined or library-defined function name. The *dwError* parameter specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes sent. Currently there are no flag values defined and *dwFlags* will be zero. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. All waiting completion routines are called before the alertable thread's wait is satisfied with a return code of `WSA_IO_COMPLETION`. The completion routines can be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be sent in the same order they are supplied.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSACloseEvent**, **WSACreateEvent**, **WSAGetOverlappedResult**, **WSASocket**, **WSAWaitForMultipleEvents**

---

## WSASendDisconnect

The Windows Sockets **WSASendDisconnect** function initiates termination of the connection for the socket and sends disconnect data.

```
int WSASendDisconnect (  
    SOCKET          s,  
    LPWSABUF       lpOutboundDisconnectData  
);
```

## Parameters

*s*

[in] Descriptor identifying a socket.

*lpOutboundDisconnectData*

[in] Pointer to the outgoing disconnect data.

## Return Values

If no error occurs, **WSASendDisconnect** returns zero. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOPROTOOPT	The parameter <i>lpOutboundDisconnectData</i> is not NULL, and the disconnect data is not supported by the service provider.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lpOutboundDisconnectData</i> parameter is not completely contained in a valid part of the user address space.

## Remarks

The **WSASendDisconnect** function is used on connection-oriented sockets to disable transmission and to initiate termination of the connection along with the transmission of disconnect data, if any. This is equivalent to a shutdown (**SD\_SEND**), except that **WSASendDisconnect** also allows sending disconnect data (in protocols that support it).

After this function has been successfully issued, subsequent sends are disallowed.

The *lpOutboundDisconnectData* parameter, if not NULL, points to a buffer containing the outgoing disconnect data to be sent to the remote party for retrieval by using **WSARecvDisconnect**.

The **WSASendDisconnect** function does not close the socket, and resources attached to the socket will not be freed until **closesocket** is invoked.



The **WSASendDisconnect** function does not block regardless of the `SO_LINGER` setting on the socket.

An application should not rely on being able to reuse a socket after calling **WSASendDisconnect**. In particular, a Windows Sockets provider is not required to support the use of **connect/WSAConnect** on such a socket.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

### + See Also

**connect, socket**

## WSASendTo

The Windows Sockets **WSASendTo** function sends data to a specific destination, using overlapped I/O where applicable.

```
int WSASendTo (
    SOCKET                s,
    LPWSABUF             lpBuffers,
    DWORD                dwBufferCount,
    LPDWORD              lpNumberOfBytesSent,
    DWORD                dwFlags,
    const struct sockaddr FAR *lpTo,
    int                  iToLen,
    LPWSAOVERLAPPED     lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

### Parameters

*s*

[in] Descriptor identifying a (possibly connected) socket.

*lpBuffers*

[in] Pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer. This array must remain valid for the duration of the send operation.

*dwBufferCount*

[in] Number of **WSABUF** structures in the *lpBuffers* array.

*lpNumberOfBytesSent*

[out] Pointer to the number of bytes sent by this call if the I/O operation completes immediately.

*dwFlags*

[in] Indicator specifying the way in which the call is made.

*lpTo*

[in] Optional pointer to the address of the target socket.

*iToLen*

[in] Size of the address in *lpTo*.

*lpOverlapped*

[in] A pointer to a **WSAOVERLAPPED** structure (ignored for nonoverlapped sockets).

*lpCompletionRoutine*

[in] Pointer to the completion routine called when the send operation has been completed (ignored for nonoverlapped sockets).

## Return Values

If no error occurs and the send operation has completed immediately, **WSASendTo** returns zero. In this case, the completion routine will have already been scheduled to be called once the calling thread is in the alertable state. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**. The error code **WSA\_IO\_PENDING** indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>lpBuffers</i> , <i>lpTo</i> , <i>lpOverlapped</i> , <i>lpNumberOfBytesSent</i> , or <i>lpCompletionRoutine</i> parameters are not part of the user address space, or the <i>lpTo</i> argument is too small.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOBUFS	The Windows Sockets provider reports a buffer deadlock.

(continued)

*(continued)*

Error code	Meaning
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, MSG_PARTIAL is not supported, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to <b>WSASendTo</b> on a socket after <b>shutdown</b> has been invoked with <b>how</b> set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: there are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the send operation cannot be completed immediately.
WSAEMSGSIZE	The socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	The socket has not been bound with <b>bind</b> , or the socket is not created with the overlapped flag.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.
WSAEADDRNOTAVAIL	The remote address is not a valid address (such as ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDRREQ	A destination address is required.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket, or the execution of the SIO_FLUSH command in <b>WSAIoctl</b> .

### Remarks

The **WSASendTo** function provides functionality over and above the standard **sendto** function in two important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped send operations.
- It allows multiple send buffers to be specified making it applicable to the scatter/gather type of I/O.

The **WSASendTo** function is normally used on a connectionless socket specified by *s* to send a datagram contained in one or more buffers to a specific peer socket identified by the *IpTo* parameter. Even if the connectionless socket has been previously connected using the **connect** function to a specific address, *IpTo* overrides the destination address for that particular datagram only. On a connection-oriented socket, the *IpTo* and *iToLen* parameters are ignored; in this case, the **WSASendTo** is equivalent to **WSASend**.

For overlapped sockets (created using **WSASocket** with flag `WSA_FLAG_OVERLAPPED`) sending data uses overlapped I/O, unless both *IpOverlapped* and *IpCompletionRoutine* are NULL in which case the socket is treated as a nonoverlapped socket. A completion indication will occur (invoking the completion routine or setting of an event object) when the supplied buffer(s) have been consumed by the transport. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or **WSAGetOverlappedResult**.

If both *IpOverlapped* and *IpCompletionRoutine* are NULL, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the last two parameters (*IpOverlapped*, *IpCompletionRoutine*) are ignored and **WSASendTo** adopts the same blocking semantics as **send**. Data is copied from the supplied buffer(s) into the transport's buffer. If the socket is nonblocking and stream oriented, and there is not sufficient space in the transport's buffer, **WSASendTo** returns with only part of the application's buffers having been consumed. Given the same buffer situation and a blocking socket, **WSASendTo** will block until all of the application's buffer contents have been consumed.

The array of **WSABUF** structures indicated by the *IpBuffers* parameter is transient. If this operation is completed in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For message-oriented sockets, care must be taken not to exceed the maximum message size of the underlying transport, which can be obtained by getting the value of socket option `SO_MAX_MSG_SIZE`. If the data is too long to pass atomically through the underlying protocol the error `WSAEMSGSIZE` is returned, and no data is transmitted.

The successful completion of a **WSASendTo** does not indicate that the data was successfully delivered.

The *dwFlags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *dwFlags* parameter. The latter is constructed by using the bitwise OR operator with any of any of the following values.

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Socket service provider may choose to ignore this flag.
MSG_OOB	Send OOB data (stream-style socket such as <b>SOCK_STREAM</b> only).
MSG_PARTIAL	Specifies that <i>lpBuffers</i> only contains a partial message. Note that the error code <b>WSAEOPNOTSUPP</b> will be returned by transports that do not support partial message transmissions.

### Overlapped Socket I/O

If an overlapped operation completes immediately, **WSASendTo** returns a value of zero and the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent. If the overlapped operation is successfully initiated and will complete later, **WSASendTo** returns **SOCKET\_ERROR** and indicates error code **WSA\_IO\_PENDING**. In this case, *lpNumberOfBytesSent* is not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **WSAGetOverlappedResult**.

The **WSASendTo** function can be called from within the completion routine of a previous **WSARecv**, **WSARecvFrom**, **WSASend**, or **WSASendTo** function. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate **WSAOVERLAPPED** structure.

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents** or **WSAGetOverlappedResult** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to TRUE. In this case the usage of the *hEvent* parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will *not* be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents** with the *fAlertable* parameter set to TRUE is invoked.

Transport providers allow an application to invoke send and receive operations from within the context of the socket I/O completion routine, and guarantee that, for a given socket, I/O completion routines will *not* be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is as follows:

```
void CALLBACK CompletionROUTINE (  
    IN DWORD          dwError,  
    IN DWORD          cbTransferred,  
    IN LPWSAOVERLAPPED lpOverlapped,  
    IN DWORD          dwFlags  
);
```

The **CompletionRoutine** function is a placeholder for an application-defined or library-defined function name. The *dwError* parameter specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* parameter specifies the number of bytes sent. Currently there are no flag values defined and *dwFlags* will be zero. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. All waiting completion routines are called before the alertable thread's wait is satisfied with a return code of `WSA_IO_COMPLETION`. The completion routines can be called in any order, not necessarily in the same order in which the overlapped operations are completed. However, the posted buffers are guaranteed to be sent in the same order they are supplied.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSACloseEvent**, **WSACreateEvent**, **WSAGetOverlappedResult**, **WSASocket**, **WSAWaitForMultipleEvents**

---

## WSASetBlockingHook

This function has been removed in compliance with the Windows Sockets 2 specification, revision 2.2.0.

The function is not exported directly by the `Ws2_32.dll`, and Windows Sockets 2 applications should not use this function. Windows Sockets 1.1 applications that call this function are still supported through the `Winsock.dll` and `Wsock32.dll`.

Blocking hooks are generally used to keep a single-threaded GUI application responsive during calls to blocking functions. Instead of using blocking hooks, an application should use a separate thread (separate from the main GUI thread) for network activity.

## WSASetEvent

The Windows Sockets **WSASetEvent** function sets the state of the specified event object to signaled.

```
BOOL WSASetEvent (
    WSAEVENT  hEvent
);
```

### Parameters

*hEvent*

[in] Handle that identifies an open event object.

### Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_INVALID_HANDLE	The <i>hEvent</i> parameter is not a valid event object handle.

### Remarks

The **WSASetEvent** function sets the state of the event object to be signaled.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSACloseEvent**, **WSACreateEvent**, **WSAResetEvent**

## WSASetLastError

The Windows Sockets **WSASetLastError** function sets the error code that can be retrieved through the **WSAGetLastError** function.

```
void WSASetLastError (  
    int    iError  
);
```

### Parameters

*iError*

[in] Integer that specifies the error code to be returned by a subsequent **WSAGetLastError** call.

### Return Values

This function generates no return values.

#### Error code

#### Meaning

WSANOTINITIALISED

A successful **WSAStartup** call must occur before using this function.

### Remarks

The **WSASetLastError** function allows an application to set the error code to be returned by a subsequent **WSAGetLastError** call for the current thread. Note that any subsequent Windows Sockets routine called by the application will override the error code as set by this routine.

The error code set by **WSASetLastError** is different from the error code reset by calling the function **getsockopt** with **SO\_ERROR**.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in **Winsock2.h**.

**Library:** Use **Ws2\_32.lib**.

**+** See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **getsockopt**, **WSAGetLastError**



## WSASetService

The Windows Sockets **WSASetService** function registers or removes from the registry a service instance within one or more name spaces. This function can be used to affect a specific name space provider, all providers associated with a specific name space, or all providers across all name spaces.

```
INT WSASetService (
    LPWSAQUERYSET      lpqsRegInfo,
    WSAESETSERVICEOP  essOperation,
    DWORD              dwControlFlags
);
```

### Parameters

*lpqsRegInfo*

[in] Pointer to the service information for registration or deregistration.

*essOperation*

[in] Enumeration whose values include the following.

Value	Description
RNRSERVICE_REGISTER	Register the service. For SAP, this means sending out a periodic broadcast. This is an NOP for the DNS name space. For persistent data stores, this means updating the address information.
RNRSERVICE_DEREGISTER	Remove the service from the registry. For SAP, this means stop sending out the periodic broadcast. This is an NOP for the DNS name space. For persistent data stores this means deleting address information.
RNRSERVICE_DELETE	Delete the service from dynamic name and persistent spaces. For services represented by multiple <b>CSADDR_INFO</b> structures (using the SERVICE_MULTIPLE flag), only the supplied address will be deleted, and this must match exactly the corresponding <b>CSADDR_INFO</b> structure that was supplied when the service was registered.

*dwControlFlags*

[in] Meaning of *dwControlFlags* is dependent on the following values.

Flag	Meaning
SERVICE_MULTIPLE	Controls scope of operation. When clear, service addresses are managed as a group. A register or removal from the registry invalidates all existing addresses before adding the given address set. When set, the action is only performed on the given address set. A register does not invalidate existing addresses and a removal from the registry only invalidates the given set of addresses.

The available values for *essOperation* and *dwControlFlags* combine to give meanings as shown in the following table.

Operation	Flags	Service already exists	Service does not exist
RNRSERVICE_REGISTER	None	Overwrites the object. Uses only addresses specified. Object is REGISTERED.	Creates a new object. Uses only addresses specified. Object is REGISTERED.
RNRSERVICE_REGISTER	SERVICE_MULTIPLE	Update object. Adds new addresses to existing set. Object is REGISTERED.	Creates a new object. Uses all addresses specified. Object is REGISTERED.
RNRSERVICE_DEREGISTER	None	Removes all addresses, but does not remove object from name space. Object is removed from the registry.	WSASERVICE_NOT_FOUND
RNRSERVICE_DEREGISTER	SERVICE_MULTIPLE	Updates object. Removes only addresses that are specified. Only marks object as DEREGISTERED if no addresses present. Does not remove from the name space.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	None	Removes object from the name space.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	SERVICE_MULTIPLE	Removes only addresses that are specified. Only removes object from the name space if no addresses remain.	WSASERVICE_NOT_FOUND

### Return Values

The return value for **WSASetService** is zero if the operation was successful. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSAEACCES	The calling routine does not have sufficient privileges to install the Service.
WSAEINVAL	One or more required parameters were invalid or missing.
WSANOTINITIALIZED	The <code>WinSxS.dll</code> has not been initialized. The application must first call <b>WSAStartup</b> before calling any Windows Sockets functions.

(continued)

*(continued)*

<b>Error code</b>	<b>Meaning</b>
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.
WSASERVICE NOT FOUND	No such service is known. The service cannot be found in the specified name space.

**Remarks**

SERVICE\_MULTIPLE lets an application manage its addresses independently. This is useful when the application wants to manage its protocols individually or when the service resides on more than one machine. For instance, when a service uses more than one protocol, it may find that one listening socket aborts but the others remain operational. In this case, the service could remove the aborted address from the registry without affecting the other addresses.

When using SERVICE\_MULTIPLE, an application must not let stale addresses remain in the object. This can happen if the application aborts without issuing a DEREGISTER request. When a service registers, it should store its addresses. On its next invocation, the service should explicitly remove these old stale addresses from the registry before registering new addresses.

**Service Properties**

The following table describes how service property data is represented in a **WSAQUERYSET** structure. Fields labeled as *(Optional)* can be supplied with a NULL pointer.

<b>WSAQUERYSET member name</b>	<b>Service property description</b>
<b>Field Name</b>	Service Property Description
<b>dwSize</b>	Must be set to sizeof ( <b>WSAQUERYSET</b> ). This is a versioning mechanism.
<b>dwOutputFlags</b>	Not applicable and ignored.
<b>LpszServiceInstanceName</b>	Referenced string contains the service instance name.
<b>LpServiceClassId</b>	The GUID corresponding to this service class.
<b>LpVersion</b>	<i>(Optional)</i> Supplies service instance version number.
<b>LpszComment</b>	<i>(Optional)</i> An optional comment string.
<b>DwNameSpace</b>	See table that follows.
<b>LpNSProviderId</b>	See table that follows.
<b>LpszContext</b>	<i>(Optional)</i> Specifies the starting point of the query in a hierarchical name space.
<b>DwNumberOfProtocols</b>	Ignored.

WSAQUERYSET member name	Service property description
<b>LpafpProtocols</b>	Ignored.
<b>LpszQueryString</b>	Ignored.
<b>DwNumberOfCsAddrs</b>	The number of elements in the array of <b>CSADDR_INFO</b> structures referenced by <i>lpcsaBuffer</i> .
<b>LpcsaBuffer</b>	A pointer to an array of <b>CSADDR_INFO</b> structures that contain the address(es) that the service is listening on.
<b>LpBlob</b>	(Optional) This is a pointer to a provider-specific entity.

As illustrated in the following, the combination of the *dwNameSpace* and *IpNSProviderId* parameters determine that name space providers are affected by this function.

<i>DwNameSpace</i>	<i>IpNSProviderId</i>	Scope of impact
Ignored	Non-NULL	The specified name-space provider.
A valid name space identifier	NULL	All name-space providers that support the indicated name space.
NS_ALL	NULL	All name-space providers.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT/2000.

## WSASocket

The Windows Sockets **WSASocket** function creates a socket that is bound to a specific transport-service provider.

```
SOCKET WSASocket (
    int                af,
    int                type,
    int                protocol,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    GROUP              g,
    DWORD              dwFlags
);
```

## Parameters

*af*

[in] Address family specification.

*type*

[in] Type specification for the new socket.

*protocol*

[in] Protocol to be used with the socket that is specific to the indicated address family.

*lpProtocolInfo*

[in] Pointer to a **WSAPROTOCOL\_INFO** structure that defines the characteristics of the socket to be created.

*g*

[in] Reserved.

*dwFlags*

[in] Flag that specifies the socket attribute.

## Return Values

If no error occurs, **WSASocket** returns a descriptor referencing the new socket. Otherwise, a value of **INVALID\_SOCKET** is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

---

**Note** This error code description is Microsoft-specific.

---

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	The specified protocol is not supported.
WSAEPROTOTYPE	The specified protocol is the wrong type for this socket.
WSAESOCKTNOSUPPORT	The specified socket type is not supported in this address family.

Error code	Meaning
WSAEINVAL	This value is true for <i>any</i> of the following conditions. <ul style="list-style-type: none"> <li>• The parameter <i>g</i> specified is not valid.</li> <li>• The <b>WSAPROTOCOL_INFO</b> structure that <i>lpProtocolInfo</i> points to is incomplete, the contents are invalid or the <b>WSAPROTOCOL_INFO</b> structure has already been used in an earlier duplicate socket operation.</li> <li>• The values specified for members of the socket triple <i>&lt;af, type, and protocol&gt;</i> are individually supported, but the given combination is not.</li> </ul>
WSAEFAULT	<i>lpProtocolInfo</i> argument is not in a valid part of the process address space.
WSAINVALIDPROVIDER	The service provider returned a version other than 2.2.
WSAINVALIDPROCTABLE	The service provider returned an invalid or incomplete procedure table to the <b>WSPStartup</b> .

### Remarks

The **WSASocket** function causes a socket descriptor and any related resources to be allocated and associated with a transport-service provider. By default, the socket will *not* have an overlapped attribute. If *lpProtocolInfo* is NULL, the *Ws2\_32.dll* uses the first three parameters (*af, type, protocol*) to determine which service provider is used by selecting the first transport provider able to support the stipulated address family, socket type, and protocol values. If the *lpProtocolInfo* is not NULL, the socket will be bound to the provider associated with the indicated **WSAPROTOCOL\_INFO** structure. In this instance, the application can supply the manifest constant **FROM\_PROTOCOL\_INFO** as the value for any of *af, type, or protocol*. This indicates that the corresponding values from the indicated **WSAPROTOCOL\_INFO** structure (*iAddressFamily, iSocketType, iProtocol*) are to be assumed. In any case, the values supplied for *af, type, and protocol* are supplied unmodified to the transport-service provider.

When selecting a protocol and its supporting service provider based on *af, type, and protocol*, this procedure will only choose a base protocol or a protocol chain, not a protocol layer by itself. Unchained protocol layers are not considered to have partial matches on *type* or *af*, either. That is, they do not lead to an error code of **WSAEAFNOSUPPORT** or **WSAEPROTONOSUPPORT**, if no suitable protocol is found.

---

**Note** The manifest constant **AF\_UNSPEC** continues to be defined in the header file but its use is *strongly discouraged*, as this can cause ambiguity in interpreting the value of the *protocol* parameter.

---

The *dwFlags* parameter can be used to specify the attributes of the socket by using the bitwise OR operator with any of the following flags.

Flag	Meaning
WSA_FLAG_OVERLAPPED	This flag causes an overlapped socket to be created. Overlapped sockets can utilize <b>WSASend</b> , <b>WSASendTo</b> , <b>WSARecv</b> , <b>WSARecvFrom</b> , and <b>WSAIoctl</b> for overlapped I/O operations, which allow multiple operations to be initiated and in progress simultaneously. All functions that allow overlapped operation ( <b>WSASend</b> , <b>WSARecv</b> , <b>WSASendTo</b> , <b>WSARecvFrom</b> , <b>WSAIoctl</b> ) also support nonoverlapped usage on an overlapped socket if the values for parameters related to overlapped operations are NULL.
WSA_FLAG_MULTIPPOINT_C_ROOT	Indicates that the socket created will be a c_root in a multipoint session. Only allowed if a rooted control plane is indicated in the protocol's <b>WSAPROTOCOL_INFO</b> structure. Refer to <i>Multipoint</i> and <i>Multicast Semantics</i> for additional information.
WSA_FLAG_MULTIPPOINT_C_LEAF	Indicates that the socket created will be a c_leaf in a multicast session. Only allowed if XP1_SUPPORT_MULTIPPOINT is indicated in the protocol's <b>WSAPROTOCOL_INFO</b> structure. Refer to <i>Multipoint</i> and <i>Multicast Semantics</i> for additional information.
WSA_FLAG_MULTIPPOINT_D_ROOT	Indicates that the socket created will be a d_root in a multipoint session. Only allowed if a rooted data plane is indicated in the protocol's <b>WSAPROTOCOL_INFO</b> structure. Refer to <i>Multipoint</i> and <i>Multicast Semantics</i> for additional information.
WSA_FLAG_MULTIPPOINT_D_LEAF	Indicates that the socket created will be a d_leaf in a multipoint session. Only allowed if XP1_SUPPORT_MULTIPPOINT is indicated in the protocol's <b>WSAPROTOCOL_INFO</b> structure. Refer to <i>Multipoint</i> and <i>Multicast Semantics</i> for additional information.

---

**Important** For multipoint sockets, exactly one of WSA\_FLAG\_MULTIPPOINT\_C\_ROOT or WSA\_FLAG\_MULTIPPOINT\_C\_LEAF *must* be specified, and exactly one of WSA\_FLAG\_MULTIPPOINT\_D\_ROOT or WSA\_FLAG\_MULTIPPOINT\_D\_LEAF *must* be specified. Refer to *Multipoint* and *Multicast Semantics* for additional information.

---

Connection-oriented sockets such as **SOCK\_STREAM** provide full-duplex connections, and must be in a connected state before any data can be sent or received on them. A connection to another socket is created with a **connect/WSAConnect** call. Once connected, data can be transferred using **send/WSASend** and **recv/WSARecv** calls. When a session has been completed, a **closesocket** must be performed.

The communications protocols used to implement a reliable, connection-oriented socket ensure that data is not lost or duplicated. If data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and subsequent calls will fail with the error code set to **WSAETIMEDOUT**.

Connectionless, message-oriented sockets allow sending and receiving of datagrams to and from arbitrary peers using **sendto/WSASendTo** and **recvfrom/WSARecvFrom**. If such a socket is connected to a specific peer, datagrams can be sent to that peer using **send/WSASend** and can be received from (only) this peer using **recv/WSARecv**.

Support for sockets with type **RAW** is not required, but service providers are encouraged to support raw sockets whenever possible.

### Shared Sockets

When a special **WSAPROTOCOL\_INFO** structure (obtained through the **WSADuplicateSocket** function and used to create additional descriptors for a shared socket) is passed as an input parameter to **WSASocket**, the *g* and *dwFlags* parameters are *ignored*. Such a **WSAPROTOCOL\_INFO** structure may only be used once, otherwise the error code **WSAEINVAL** will result.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

**Unicode:** Implemented as Unicode and ANSI versions on all platforms.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **accept**, **bind**, **connect**, **getsockname**, **getsockopt**, **ioctlsocket**, **listen**, **recv**, **recvfrom**, **select**, **send**, **sendto**, **setsockopt**, **shutdown**

## WSAStartup

The Windows Sockets **WSAStartup** function initiates use of `Ws2_32.dll` by a process.

```
int WSAStartup (
    WORD          wVersionRequested,
    LPWSADATA     lpWSADATA
);
```

### Parameters

*wVersionRequested*

[in] Highest version of Windows Sockets support that the caller can use. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

*lpWSADATA*

[out] Pointer to the **WSADATA** data structure that is to receive details of the Windows Sockets implementation.



## Return Values

The **WSAStartup** function returns zero if successful. Otherwise, it returns one of the error codes listed in the following.

An application cannot call **WSAGetLastError** to determine the error code as is normally done in Windows Sockets if **WSAStartup** fails. The `Ws2_32.dll` will *not* have been loaded in the case of a failure so the client data area where the last error information is stored could not be established.

Error code	Meaning
WSASYSNOTREADY	Indicates that the underlying network subsystem is not ready for network communication.
WSAVERNOTSUPPORTED	The version of Windows Sockets support requested is not provided by this particular Windows Sockets implementation.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 operation is in progress.
WSAEPROCLIM	Limit on the number of tasks supported by the Windows Sockets implementation has been reached.
WSAEFAULT	The <i>lpWSAData</i> is not a valid pointer.

## Remarks

The **WSAStartup** function *must* be the first Windows Sockets function called by an application or DLL. It allows an application or DLL to specify the version of Windows Sockets required and retrieve details of the specific Windows Sockets implementation. The application or DLL can only issue further Windows Sockets functions after successfully calling **WSAStartup**.

In order to support future Windows Sockets implementations and applications that can have functionality differences from the current version of Windows Sockets, a negotiation takes place in **WSAStartup**. The caller of **WSAStartup** and the `Ws2_32.dll` indicate to each other the highest version that they can support, and each confirms that the other's highest version is acceptable. Upon entry to **WSAStartup**, the `Ws2_32.dll` examines the version requested by the application. If this version is equal to or higher than the lowest version supported by the DLL, the call succeeds and the DLL returns in *wHighVersion* the highest version it supports and in *wVersion* the minimum of its high version and *wVersionRequested*. The `Ws2_32.dll` then assumes that the application will use *wVersion*. If the *wVersion* parameter of the **WSADATA** structure is unacceptable to the caller, it should call **WSACleanup** and either search for another `Ws2_32.dll` or fail to initialize.

It is legal and possible for an application written to this version of the specification to successfully negotiate a higher version number version. In that case, the application is only guaranteed access to higher-version functionality that fits within the syntax defined

in this version, such as new ioctl codes and new behavior of existing functions. New functions may be inaccessible. To get full access to the new syntax of a future version, the application must fully conform to that future version, such as compiling against a new header file, linking to a new library, or other special cases.

This negotiation allows both a `Ws2_32.dll` and a Windows Sockets application to support a range of Windows Sockets versions. An application can use `Ws2_32.dll` if there is any overlap in the version ranges. The following table shows how **WSAStartup** works with different applications and `Ws2_32.dll` versions.

App versions	DLL versions	<i>wVersion requested</i>	<i>wVersion</i>	<i>wHigh version</i>	End result
1.1	1.1	1.1	1.1	1.1	use 1.1
1.0 1.1	1.0	1.1	1.0	1.0	use 1.0
1.0	1.0 1.1	1.0	1.0	1.1	use 1.0
1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1	1.0	1.1	1.0	1.0	Application fails
1.0	1.1	1.0	---	---	WSAVERNOT SUPPORTED
1.0 1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1 2.0	1.1	2.0	1.1	1.1	use 1.1
2.0	2.0	2.0	2.0	2.0	use 2.0

### Example

The following code fragment demonstrates how an application that supports only version 2.2 of Windows Sockets makes a **WSAStartup** call:

```
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD( 2, 2 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    /* Tell the user that we could not find a usable */
    /* WinSock DLL.                               */
    return;
}

/* Confirm that the WinSock DLL supports 2.2.*/
/* Note that if the DLL supports versions greater */
/* than 2.2 in addition to 2.2, it will still return */
```

(continued)

(continued)

```
/* 2.2 in wVersion since that is the version we      */
/* requested.                                         */

if ( LOBYTE( wsaData.wVersion ) != 2 ||
      HIBYTE( wsaData.wVersion ) != 2 ) {
    /* Tell the user that we could not find a usable */
    /* WinSock DLL.                                  */
    WSACleanup( );
    return;
}

/* The WinSock DLL is acceptable. Proceed. */
```

Once an application or DLL has made a successful **WSAStartup** call, it can proceed to make other Windows Sockets calls as needed. When it has finished using the services of the `Ws2_32.dll`, the application or DLL must call **WSACleanup** to allow the `Ws2_32.dll` to free any resources for the application.

Details of the actual Windows Sockets implementation are described in the **WSADATA** structure.

An application or DLL can call **WSAStartup** more than once if it needs to obtain the **WSADATA** structure information more than once. On each such call the application can specify any version number supported by the DLL.

An application must call one **WSACleanup** call for every successful **WSAStartup** call to allow third-party DLLs to make use of a `Ws2_32.dll` on behalf of an application. This means, for example, that if an application calls **WSAStartup** three times, it must call **WSACleanup** three times. The first two calls to **WSACleanup** do nothing except decrement an internal counter; the final **WSACleanup** call for the task does all necessary resource deallocation for the task.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Library:** Use `Ws2_32.lib`.

#### + See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **send**, **sendto**, **WSACleanup**

# WSAStringToAddress

The Windows Sockets **WSAStringToAddress** function converts a numeric string to a **SOCKADDR** structure, suitable for passing to Windows Sockets routines that take such a structure.

```

INT WSAStringToAddress (
    LPTSTR           AddressString,
    INT              AddressFamily,
    LPWSA_PROTOCOL_INFO  IpProtocolInfo,
    LPSOCKADDR       IpAddress,
    LPINT            IpAddressLength
);

```

## Parameters

### *AddressString*

[in] Pointer to the zero-terminated human-readable numeric string to convert.

### *AddressFamily*

[in] Address family to which the string belongs.

### *IpProtocolInfo*

[in] (optional) The **WSA\_PROTOCOL\_INFO** structure associated with the provider to be used. If this is NULL, the call is routed to the provider of the first protocol supporting the indicated **AddressFamily**.

### *IpAddress*

[out] Buffer that is filled with a single **SOCKADDR**.

### *IpAddressLength*

[in/out] Length of the Address buffer. Returns the size of the resultant **SOCKADDR** structure. If the supplied buffer is not large enough, the function fails with a specific error of **WSAEFAULT** and this parameter is updated with the required size in bytes.

## Return Values

The return value for **WSAStringToAddress** is zero if the operation was successful. Otherwise, the value **SOCKET\_ERROR** is returned, and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSAEFAULT	The specified Address buffer is too small. Passes in a larger buffer.
WSAEINVAL	Unable to translate the string into a <b>SOCKADDR</b> . See the following Remarks section for more information.

(continued)

*(continued)*

Error code	Meaning
WSANOTINITIALIZED	The Ws2_32.dll has not been initialized. The application must first call <b>WSAStartup</b> before calling any Windows Socket functions.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

**Remarks**

The **WSAStringToAddress** function converts alphanumeric address to **SOCKADDR** structures. **WSAStringToAddress** is the protocol independent equivalent of the BSD **inet\_ntoa** function.

Any missing components of the address will be defaulted to a reasonable value, if possible. For example, a missing port number will default to zero. If the caller wants the translation to be done by a particular provider, it should supply the corresponding **WSAPROTOCOL\_INFO** structure in the *lpProtocolInfo* parameter.

The **WSAStringToAddress** function fails (and returns **WSAEINVAL**) if the **sin\_family** member of the **SOCKADDR\_IN** structure, which is passed in the *lpAddress* parameter in the form of a **SOCKADDR** structure, is not set to **AF\_INET**.

**! Requirements**

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT/2000.

---

## WSAUnhookBlockingHook

This function has been removed in compliance with the Windows Sockets 2 specification, revision 2.2.0.

The function is not exported directly by the Ws2\_32.dll, and Windows Sockets 2 applications should not use this function. Windows Sockets 1.1 applications that call this function are still supported through the Winsock.dll and Wsock32.dll.

Blocking hooks are generally used to keep a single-threaded GUI application responsive during calls to blocking functions. Instead of using blocking hooks, an application should use a separate thread (separate from the main GUI thread) for network activity.

# WSAWaitForMultipleEvents

The Windows Sockets **WSAWaitForMultipleEvents** function returns either when one or all of the specified event objects are in the signaled state, or when the time-out interval expires.

```
DWORD WSAWaitForMultipleEvents (  
    DWORD                cEvents,  
    const WSAEVENT FAR  *lphEvents,  
    BOOL                 fWaitAll,  
    DWORD                dwTimeout,  
    BOOL                 fAlertable  
);
```

## Parameters

### *cEvents*

[in] Indicator specifying the number of event object handles in the array pointed to by *lphEvents*. The maximum number of event object handles is `WSA_MAXIMUM_WAIT_EVENTS`. One or more events must be specified.

### *lphEvents*

[in] Pointer to an array of event object handles.

### *fWaitAll*

[in] Indicator specifying the wait type. If `TRUE`, the function returns when all event objects in the *lphEvents* array are signaled at the same time. If `FALSE`, the function returns when any one of the event objects is signaled. In the latter case, the return value indicates the event object whose state caused the function to return.

### *dwTimeout*

[in] Indicator specifying the time-out interval, in milliseconds. The function returns if the interval expires, even if conditions specified by the *fWaitAll* parameter are not satisfied. If *dwTimeout* is zero, the function tests the state of the specified event objects and returns immediately. If *dwTimeout* is `WSA_INFINITE`, the function's time-out interval never expires.

### *fAlertable*

[in] Indicator specifying whether the function returns when the system queues an I/O completion routine for execution by the calling thread. If `TRUE`, the completion routine is executed and the function returns. If `FALSE`, the completion routine is not executed when the function returns.

## Return Values

If the **WSAWaitForMultipleEvents** function succeeds, the return value indicates the event object that caused the function to return.

If the function fails, the return value is `WSA_WAIT_FAILED`. To get extended error information, call **WSAGetLastError**.

The return value upon success is one of the following values.


Value	Meaning
WSA_WAIT_EVENT_0 to (WSA_WAIT_EVENT_0 + cEvents - 1)	If <i>fWaitAll</i> is TRUE, the return value indicates that the state of all specified event objects is signaled. If <i>fWaitAll</i> is FALSE, the return value minus WSA_WAIT_EVENT_0 indicates the <i>lphEvents</i> array index of the object that satisfied the wait.
WAIT_IO_COMPLETION	One or more I/O completion routines are queued for execution.
WSA_WAIT_TIMEOUT	The time-out interval elapsed and the conditions specified by the <i>fWaitAll</i> parameter are not satisfied.
Error code	Meaning
WSANOTINITIALISED	A successful <b>WSAStartup</b> call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to complete the operation.
WSA_INVALID_HANDLE	One or more of the values in the <i>lphEvents</i> array is not a valid event object handle.
WSA_INVALID_PARAMETER	The <i>cEvents</i> parameter does not contain a valid handle count.

### Remarks

The **WSAWaitForMultipleEvents** function returns when any one or all of the specified objects are in the signaled state, or when the time-out interval elapses. This function is also used to perform an alertable wait by setting the parameter *fAlertable* to be TRUE. This enables the function to return when the system queues an I/O completion routine to be executed by the calling thread.

When *fWaitAll* is TRUE, the function's wait condition is satisfied only when the state of all objects is signaled at the same time. The function does not modify the state of the specified objects until all objects are simultaneously signaled.


Applications that simply need to enter an alertable wait state without waiting for any event objects to be signaled should use the Win32 **SleepEx** function.

 Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

 See Also

Windows Sockets Programming Considerations Overview, Microsoft Windows-Specific Extension Functions, **WSACloseEvent**, **WSACreateEvent**





## CHAPTER 9

# Winsock 2 Structures and Enumerations

## Windows Sockets Structures in the API

This chapter lists the structures used with Windows Sockets 2.

### AFPROTOCOLS

The Windows Sockets **AFPROTOCOLS** structure supplies a list of protocols to which application programmers can constrain queries. The **AFPROTOCOLS** structure is used for query purposes only.

```
typedef struct _AFPROTOCOLS {  
    INT iAddressFamily;  
    INT iProtocol;  
} AFPROTOCOLS, *PAFPROTOCOLS, *LPAFPROTOCOLS;
```

#### Members

##### **iAddressFamily**

Address family to which the query is to be constrained.

##### **iProtocol**

Protocol to which the query is to be constrained.

#### Remarks

The members of the **AFPROTOCOLS** structure are a functional pair, and only have meaning when used together, as protocol values have meaning only within the context of an address family.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

#### + See Also

**WSAQuerySet**, **WSALookupServiceBegin**, **NSPLookupServiceBegin**

# BLOB

A Windows Sockets **BLOB** structure, derived from Binary Large Object, contains information about a block of data.

```
typedef struct _BLOB {
    ULONG    cbSize;
    BYTE     *pBlobData;
} BLOB;
```

## Members

### cbSize

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

### pBlobData

Pointer to a block of data.

## Remarks

The structure name **BLOB** comes from the acronym BLOB, which stands for Binary Large Object.

This structure does not describe the nature of the data pointed to by **pBlobData**.

---

**Note** Windows Sockets defines a similar **BLOB** structure in *Wtypes.h*. Using both header files in the same source code file creates redefinition compile-time errors.

---

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later. Not supported on Windows 95.

**Header:** Declared in *Wtypes.h*.

### + See Also

**SERVICE\_INFO**

---

# CSADDR\_INFO

The Windows Sockets **CSADDR\_INFO** structure contains Windows Sockets address information for a network service or name space provider. The **GetAddressByName** function obtains Windows Sockets address information using **CSADDR\_INFO** structures.

```
typedef struct _CSADDR_INFO {
    SOCKET_ADDRESS    LocalAddr;
    SOCKET_ADDRESS    RemoteAddr;
```

```

INT         iSocketType;
INT         iProtocol;
} CSADDR_INFO;

```

## Members

### LocalAddr

Specifies a Windows Sockets local address.

In a client application, pass this address to the **bind** function to obtain access to a network service.

In a network service, pass this address to the **bind** function so that the service is bound to the appropriate local address.

### RemoteAddr

Specifies a Windows Sockets remote address. There are several uses for this remote address:

- You can use this remote address to connect to the service through the **connect** function. This is useful if an application performs **send/receive** operations that involve connection-oriented protocols.
- You can use this remote address with the **sendto** function when you are communicating over a connectionless (datagram) protocol. If you are using a connectionless protocol, such as UDP, **sendto** is typically the way you pass data to the remote system.

### iSocketType

Specifies the type of the Windows socket. The following socket types are defined in Winsock.h.

Value	Socket type
SOCK_STREAM	Stream. This is a protocol that sends data as a stream of bytes, with no message boundaries.
SOCK_DGRAM	Datagram. This is a connectionless protocol. There is no virtual circuit setup. There are typically no reliability guarantees. Services use <b>recvfrom</b> to obtain datagrams. The <b>listen</b> and <b>accept</b> functions do not work with datagrams.
SOCK_RDM	Reliably-Delivered Message. This is a protocol that preserves message boundaries in data.
SOCK_SEQPACKET	Sequenced packet stream. This is a protocol that is essentially the same as SOCK_RDM.

### iProtocol

Specifies a value to pass as the protocol parameter to the **socket** function to open a socket for this service.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Nspapi.h.

**+** See Also

**bind**, **connect**, **GetAddressByName**, **recv**, **send**, **sendto**

---

## fd\_set

The Windows Sockets **fd\_set** structure is used by various Windows Sockets functions and service providers, such as the **select** function, to place sockets into a “set” for various purposes, such as testing a given socket for readability using the *readfds* parameter of the **select** function.

```
typedef struct fd_set {
    u_int    fd_count;           // how many are SET?
    SOCKET   fd_array[FD_SETSIZE]; // an array of SOCKETS
} fd_set;
```

### Members

**fd\_count**

Number of sockets in the set.

**fd\_array**

Array of sockets that are in the set.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**+** See Also

**select**, **WSAAsyncSelect**, **WSAEventSelect**

---

## FLOWSPEC

The **FLOWSPEC** structure is defined in the Quality of Service (QOS) section of the SDK.

## hostent

The Windows Sockets **hostent** structure is used by functions to store information about a given host, such as host name, IP address, and so forth. An application should never attempt to modify this structure or to free any of its components. Furthermore, only one copy of the **hostent** structure is allocated per thread, and an application should therefore copy any information that it needs before issuing any other Windows Sockets API calls.

```
struct hostent {
    char FAR *      h_name;
    char FAR * FAR * h_aliases;
    short          h_addrtype;
    short          h_length;
    char FAR * FAR * h_addr_list;
};
```

### Members

#### **h\_name**

Official name of the host (PC). If using the DNS or similar resolution system, it is the Fully Qualified Domain Name (FQDN) that caused the server to return a reply. If using a local hosts file, it is the first entry after the IP address.

#### **h\_aliases**

Null-terminated array of alternate names.

#### **h\_addrtype**

Type of address being returned.

#### **h\_length**

Length of each address, in bytes.

#### **h\_addr\_list**

Null-terminated list of addresses for the host. Addresses are returned in network byte order. The macro **h\_addr** is defined to be **h\_addr\_list[0]** for compatibility with older software.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

### + See Also

**gethostbyaddr**

## in\_addr

The Windows Sockets **in\_addr** structure represents a host by its Internet address.

```

struct in_addr {
    union {
        struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
        struct { u_short s_w1,s_w2; } S_un_w;
        u_long S_addr;
    } S_un;
};

```

## Members

### S\_un\_b

Address of the host formatted as four u\_chars.

### S\_un\_w

Address of the host formatted as two u\_shorts.

### S\_addr

Address of the host formatted as a u\_long.

## ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

## + See Also

**inet\_addr, inet\_ntoa, SOCKADDR**

# linger

The Windows Sockets **linger** structure maintains information about a specific socket that specifies how that socket should behave when data is queued to be sent and the **closesocket** function is called on the socket.

```

struct linger {
    u_short l_onoff;
    u_short l_linger;
};

```

## Members

### l\_onoff

Specifies whether a socket should remain open for a specified amount of time after a **closesocket** function call to enable queued data to be sent.

### l\_linger

Enabling SO\_LINGER also disables SO\_DONTLINGER, and vice versa. Note that if SO\_DONTLINGER is DISABLED (that is, SO\_LINGER is ENABLED) then no time-out value is specified. In this case, the time-out used is implementation dependent.

If a previous time-out has been established for a socket (by enabling `SO_LINGER`), this time-out value should be reinstated by the service provider.

### Remarks

To enable `SO_LINGER`, the application should set `I_onoff` to a nonzero value, set `I_linger` to zero or the desired time-out (in seconds), and call the `setsockopt` function. To specify `SO_DONTLINGER` (that is, disable `SO_LINGER`) `I_onoff` should be set to zero and `setsockopt` should be called. Note that enabling `SO_LINGER` with a nonzero time-out on a nonblocking socket is not recommended.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

#### + See Also

`setsockopt`, `getsockopt`, `closesocket`

## NS\_SERVICE\_INFO

The Windows Sockets `NS_SERVICE_INFO` structure contains information about a network service or a network service type in the context of a specified name space, or a set of default name spaces.

```
typedef struct _NS_SERVICE_INFO {
    DWORD          dwNameSpace;
    SERVICE_INFO   ServiceInfo;
} NS_SERVICE_INFO;
```

### Members

#### dwNameSpace

Specifies the name space or a set of default name spaces to which this service information applies.

Use one of the following constant values to specify a name space.

Value	Name space
<code>NS_DEFAULT</code>	A set of default name spaces. The set of default name spaces typically includes all the name spaces installed on the system. System administrators, however, can exclude particular name spaces from the set.
<code>NS_DNS</code>	The Domain Name System used in the Internet to resolve the name of the host.

*(continued)*



*(continued)*

Value	Name space
NS_MS	The Microsoft name space.
NS_NDS	The NetWare 4 provider.
NS_NETBT	The NetBIOS over TCP/IP layer. The operating system registers their computer names with NetBIOS. This name space is used to convert a computer name to an IP address that uses this registration.
NS_NIS	
NS_SAP	The NetWare Service Advertising Protocol. This can access the Netware bindery, if appropriate. NS_SAP is a dynamic name space that enables the registration of services.
NS_STDA	
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\hosts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and lookup value in the cache of host to IP address mappings.
NS_WINS	The Windows Internet Name System (WINS) name space.
NS_X500	The X.500 directory service name space.

**ServiceInfo**

A **SERVICE\_INFO** structure that contains information about a network service or network service type.

**! Requirements**

**Version:** Requires Windows Sockets 1.1 or later. Not supported on Windows 95.

**Header:** Declared in Nspapi.h.

**Unicode:** Declared as Unicode and ANSI structures.

**+ See Also****SERVICE\_INFO**

## PROTOCOL\_INFO

The Windows Sockets **PROTOCOL\_INFO** structure contains information about a protocol.

```
typedef struct _PROTOCOL_INFO {
    DWORD    dwServiceFlags;
    INT      iAddressFamily;
```

```

INT      iMaxSockAddr;
INT      iMinSockAddr;
INT      iSocketType;
INT      iProtocol;
DWORD    dwMessageSize;
LPTSTR   lpProtocol;
} PROTOCOL_INFO;

```

## Members

### dwServiceFlags

A set of bit flags that specifies the services provided by the protocol. One or more of the following bit flags may be set.

Value	Meaning
XP_CONNECTIONLESS	If this flag is set, the protocol provides connectionless (datagram) service. If this flag is clear, the protocol provides connection-oriented data transfer.
XP_GUARANTEED_DELIVERY	If this flag is set, the protocol guarantees that all data sent will reach the intended destination. If this flag is clear, there is no such guarantee.
XP_GUARANTEED_ORDER	If this flag is set, the protocol guarantees that data will arrive in the order in which it was sent. Note that this characteristic does not guarantee delivery of the data, <i>only</i> its order. If this flag is clear, the order of data sent is not guaranteed.
XP_MESSAGE_ORIENTED	If this flag is set, the protocol is message-oriented. A message-oriented protocol honors message boundaries. If this flag is clear, the protocol is stream oriented, and the concept of message boundaries is irrelevant.
XP_PSEUDO_STREAM	If this flag is set, the protocol is a message-oriented protocol that ignores message boundaries for all receive operations.  This optional capability is useful when you do not want the protocol to frame messages. An application that requires stream-oriented characteristics can open a socket with type <b>SOCK_STREAM</b> for transport protocols that support this functionality, regardless of the value of <b>iSocketType</b> .

(continued)

(continued)

Value	Meaning
XP_GRACEFUL_CLOSE	If this flag is set, the protocol supports two-phase close operations, also known as graceful close operations. If this flag is clear, the protocol supports only abortive close operations.
XP_EXPEDITED_DATA	If this flag is set, the protocol supports expedited data, also known as urgent data.
XP_CONNECT_DATA	If this flag is set, the protocol supports connect data.
XP_DISCONNECT_DATA	If this flag is set, the protocol supports disconnect data.
XP_SUPPORTS_BROADCAST	If this flag is set, the protocol supports a broadcast mechanism.
XP_SUPPORTS_MULTICAST	If this flag is set, the protocol supports a multicast mechanism.
XP_BANDWIDTH_ALLOCATION	If this flag is set, the protocol supports a mechanism for allocating a guaranteed bandwidth to an application.
XP_FRAGMENTATION	If this flag is set, the protocol supports message fragmentation; physical network MTU is hidden from applications.
XP_ENCRYPTS	If this flag is set, the protocol supports data encryption.

### iAddressFamily

Value to pass as the *af* parameter when the **socket** function is called to open a socket for the protocol. This address family value uniquely defines the structure of protocol addresses, also known as **sockaddr** structures, used by the protocol.

### iMaxSockAddr

Maximum length of a socket address supported by the protocol.

### iMinSockAddr

Minimum length of a socket address supported by the protocol.

### iSocketType

Value to pass as the *type* parameter when the **socket** function is called to open a socket for the protocol.

Note that if XP\_PSEUDO\_STREAM is set in **dwServiceFlags**, the application can specify SOCK\_STREAM as the *type* parameter to **socket**, regardless of the value of **iSocketType**.

### iProtocol

Value to pass as the *protocol* parameter when the **socket** function is called to open a socket for the protocol.

**dwMessageSize**

Maximum message size supported by the protocol. This is the maximum size of a message that can be sent from or received by the host. For protocols that do not support message framing, the actual maximum size of a message that can be sent to a given address may be less than this value.

The following special message size values are defined.

Value	Meaning
0	The protocol is stream-oriented; the concept of message size is not relevant.
0xFFFFFFFF	The protocol is message-oriented, but there is no maximum message size.

**IpProtocol**

Points to a zero-terminated string that supplies a name for the protocol; for example, "SPX2."

**! Requirements**

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Nspapi.h.

**Unicode:** Declared as Unicode and ANSI structures.

**+ See Also**

**EnumProtocols, socket**

## protoent

The Windows Sockets **protoent** structure contains the name and protocol numbers that correspond to a given protocol name. Applications must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and therefore, the application should copy any information it needs before issuing any other Windows Sockets function calls.

```
struct protoent {
    char FAR *    p_name;
    char FAR * FAR * p_aliases;
    short        p_proto;
};
```

**Members****p\_name**

Official name of the protocol.

**p\_aliases**

Null-terminated array of alternate names.

**p\_proto**

Protocol number, in host byte order.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

---

## QOS

The **QOS** structure (implemented in Windows 2000 as the **QOS** structure) is defined in the *Quality of Service (QOS) API reference* later in this book.

---

## servent

The Windows Sockets **servent** structure is used to store or return the name and service number for a given service name.

```
struct servent {
    char FAR *      s_name;
    char FAR * FAR * s_aliases;
    short          s_port;
    char FAR *      s_proto;
};
```

**Members****s\_name**

Official name of the service.

**s\_aliases**

Null-terminated array of alternate names.

**s\_port**

Port number at which the service can be contacted. Port numbers are returned in network byte order.

**s\_proto**

Name of the protocol to use when contacting the service.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**+** See Also

**getservbyname**

## SERVICE\_ADDRESS

The Windows Sockets **SERVICE\_ADDRESS** structure contains address information for a service. The structure can accommodate many types of Interprocess Communications (IPC) mechanisms and their address forms, including Remote Procedure Calls (RPC), named pipes, and sockets.

```
typedef struct _SERVICE_ADDRESS {
    DWORD    dwAddressType;
    DWORD    dwAddressFlags;
    DWORD    dwAddressLength;
    DWORD    dwPrincipalLength;
    BYTE     *lpAddress;
    BYTE     *lpPrincipal;
} SERVICE_ADDRESS;
```

### Members

#### **dwAddressType**

Address family to which the socket address pointed to by **lpAddress** belongs.

#### **dwAddressFlags**

Set of bit flags that specify properties of the address. The following bit flags are defined.

Value	Meaning
SERVICE_ADDRESS_FLAG_RPC_CN	If this bit flag is set, the service supports connection-oriented RPC over this transport protocol.
SERVICE_ADDRESS_FLAG_RPC_DG	If this bit flag is set, the service supports datagram-oriented RPC over this transport protocol.
SERVICE_ADDRESS_FLAG_RPC_NB	If this bit flag is set, the service supports NetBIOS RPC over this transport protocol.

#### **dwAddressLength**

Size, in bytes, of the address.

#### **dwPrincipalLength**

Reserved for future use. Must be zero.

#### **lpAddress**

Pointer to a socket address of the appropriate type.

#### **lpPrincipal**

Reserved for future use. It must be null.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later. Not supported on Windows 95.

**Header:** Declared in Nspapi.h.

**+** See Also

**SERVICE\_ADDRESSES, SERVICE\_INFO**

---

## SERVICE\_ADDRESSES

The Windows Sockets **SERVICE\_ADDRESSES** structure contains an array of **SERVICE\_ADDRESS** data structures.

```
typedef struct _SERVICE_ADDRESSES {
    DWORD          dwAddressCount;
    SERVICE_ADDRESS Addresses[1];
} SERVICE_ADDRESSES;
```

### Members

**dwAddressCount**

Specifies the number of **SERVICE\_ADDRESS** structures in the **Addresses** array

**Addresses**

An array of **SERVICE\_ADDRESS** data structures. Each **SERVICE\_ADDRESS** structure contains information about a network service address.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later. Not supported on Windows 95.

**Header:** Declared in Nspapi.h.

**+** See Also

**SERVICE\_ADDRESS, SERVICE\_INFO**

---

## SERVICE\_INFO

The Windows Sockets **SERVICE\_INFO** structure contains information about a network service or a network service type.

```
typedef struct _SERVICE_INFO {
    LPGUID lpServiceType;
    LPTSTR lpServiceName;
    LPTSTR lpComment;
```

```

LPTSTR  lpLocale;
DWORD   dwDisplayHint;
DWORD   dwVersion;
DWORD   dwTime;
LPTSTR  lpMachineName;
LPSERVICE_ADDRESSES lpServiceAddress;
BLOB    ServiceSpecificInfo;
} SERVICE_INFO;

```

## Members

### IpServiceType

Pointer to a GUID that is the type of the network service.

### IpServiceName

Pointer to a zero-terminated string that is the name of the network service.

If you are calling the **SetService** function with the **dwNameSpace** parameter set to **NS\_DEFAULT**, the network service name must be a common name. A common name is what the network service is commonly known as. An example of a common name for a network service is “My SQL Server”.

If you are calling the **SetService** function with the **dwNameSpace** parameter set to a specific service name, the network service name can be a common name or a distinguished name. A distinguished name distinguishes the service to a unique location with a directory service. An example of a distinguished name for a network service is “MS\\SYS\\NT\\DEV\\My SQL Server”.

### IpComment

Pointer to a zero-terminated string that is a comment or description for the network service. For example, “Used for development upgrades.”

### IpLocale

Pointer to a zero-terminated string that contains locale information.

### dwDisplayHint

Specifies a hint as to how to display the network service in a network browsing user interface. This can be one of the following values.

Value	Meaning
RESOURCE_DISPLAYTYPE_ DOMAIN	Displays the network service as a domain.
RESOURCE_DISPLAYTYPE_ FILE	Displays the network service as a file.
RESOURCE_DISPLAYTYPE_ GENERIC	The method used to display the object does not matter.
RESOURCE_DISPLAYTYPE_ GROUP	Displays the network service as a group.

(continued)



(continued)

Value	Meaning
RESOURCE_DISPLAYTYPE_SERVER	Displays the network service as a server.
RESOURCE_DISPLAYTYPE_SHARE	Displays the network service as a sharepoint.
RESOURCE_DISPLAYTYPE_TREE	Displays the network service as a tree.

### dwVersion

Version information for the network service. The high word of this value specifies a major version number. The low word of this value specifies a minor version number.

### dwTime

Reserved for future use. Must be set to zero.

### lpMachineName

Pointer to a zero-terminated string that is the name of the computer on which the network service is running.

### lpServiceAddress

Pointer to a **SERVICE\_ADDRESSES** structure that contains an array of **SERVICE\_ADDRESS** structures. Each **SERVICE\_ADDRESS** structure contains information about a network service address.

A network service can call the **getsockname** function to determine the local address of the system.

### ServiceSpecificInfo

A **BLOB** structure that specifies service-defined information.

---

**Note** In general, the data pointed to by the **BLOB** structure's **pBlobData** member must not contain any pointers. That is because only the network service knows the format of the data; copying the data without such knowledge would lead to pointer invalidation. If the data pointed to by **pBlobData** contains variable-sized elements, offsets from **pBlobData** can be used to indicate the location of those elements. There is one exception to this general rule: when **pBlobData** points to a **SERVICE\_TYPE\_INFO\_ABS** structure. This is possible because both the **SERVICE\_TYPE\_INFO\_ABS** structure, and any **SERVICE\_TYPE\_VALUE\_ABS** structures it contains are predefined, and thus their formats are known to the operating system.

---

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later. Not supported on Windows 95.

**Header:** Declared in Nspapi.h.

**Unicode:** Declared as Unicode and ANSI structures.

**+** See Also

**BLOB**, **GetService**, **NS\_SERVICE\_INFO**, **SetService**, **SERVICE\_ADDRESS**, **SERVICE\_ADDRESSES**, **SERVICE\_TYPE\_INFO\_ABS**, **SERVICE\_TYPE\_VALUE\_ABS**

## SERVICE\_TYPE\_INFO\_ABS

The Windows Sockets **SERVICE\_TYPE\_INFO\_ABS** structure contains information about a network service type. You use a **SERVICE\_TYPE\_INFO\_ABS** structure to add a network service type to a name space.

```
typedef struct _SERVICE_TYPE_INFO_ABS {
    LPTSTR          lpTypeName;
    DWORD           dwValueCount;
    SERVICE_TYPE_VALUE_ABS Values[1];
} SERVICE_TYPE_INFO_ABS;
```

### Members

#### lpTypeName

Pointer to a zero-terminated string that is the name of the network service type. This name is the same in all name spaces, and is used by the **GetTypeByName** and **GetNameByType** functions.

#### dwValueCount

Number of **SERVICE\_TYPE\_VALUE\_ABS** structures in the **Values** member array that follows **dwValueCount**.

#### Values[1]

Array of **SERVICE\_TYPE\_VALUE\_ABS** structures.

Each of these structures contains information about a service type value that the operating system or network service may need when an instance of this network service type is registered with a name space.

The information in these structures may be specific to a name-space. For example, if a network service uses the SAP name space, but does not have a **GUID** that contains the SAP identifier (SAPID), it defines the SAPID in a **SERVICE\_TYPE\_VALUE\_ABS** structure.

### Remarks

When you use the **SetService** function to add a network service type to a name space, the **SERVICE\_TYPE\_INFO\_ABS** structure is passed as the **ServiceSpecificInfo BLOB** member of a **SERVICE\_INFO** structure. Although the **ServiceSpecificInfo** member generally should not contain pointers, an exception is made in the case of the **SERVICE\_TYPE\_INFO\_ABS** and **SERVICE\_TYPE\_VALUE\_ABS** structures.

**!** Requirements

**Version:** Requires Windows Sockets 1.1 or later. Not supported on Windows 95.

**Header:** Declared in Nspapi.h.

**Unicode:** Declared as Unicode and ANSI structures.

**+** See Also

SetService, SERVICE\_INFO, SERVICE\_TYPE\_VALUE\_ABS

---

## SERVICE\_TYPE\_VALUE\_ABS

The Windows Sockets **SERVICE\_TYPE\_VALUE\_ABS** structure contains information about a network-service type value. This information may be specific to a name space.

```
typedef struct _SERVICE_TYPE_VALUE_ABS {
    DWORD    dwNameSpace;
    DWORD    dwValueType;
    DWORD    dwValueSize;
    LPTSTR   lpValueName;
    PVOID    lpValue;
} SERVICE_TYPE_VALUE_ABS;
```

### Members

#### dwNameSpace

Specifies the name space, or a set of default name spaces, for which the network service type value is intended. Name-space providers will look only at values intended for their name space.

Use one of the following constants to specify a name space.

Value	Name space
NS_DEFAULT	A set of default name spaces. The function queries each name space within this set. The set of default name spaces typically includes all the name spaces installed on the system. System administrators, however, can exclude particular name spaces from the set. NS_DEFAULT is the value that most applications should use for <b>dwNameSpace</b> .
NS_DNS	The Domain Name System used in the Internet for host name resolution.

Value	Name space
NS_NETBT	The NetBIOS over TCP/IP layer. All Windows NT/Windows 2000 systems register their computer names with NetBIOS. This name space is used to convert a computer name to an IP address that uses this registration. Note that NS_NETBT may access a WINS server to perform the resolution.
NS_SAP	The Netware Service Advertising Protocol. This may access the Netware bindery if appropriate. NS_SAP is a dynamic name space that allows registration of services.
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\hosts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and looks up host names and IP addresses in cache of host to IP address mappings.

**dwValueType**

Type of the value data. Specify one of the following types.

Value	Meaning
REG_BINARY	Binary data in any form.
REG_DWORD	A 32-bit number.
REG_MULTI_SZ	An array of null-terminated strings, terminated by two null characters.
REG_SZ	A null-terminated string.

**dwValueSize**

Size of the value data, in bytes. In the case of REG\_SZ and REG\_MULTI\_SZ string data, the terminating characters are counted as part of the size.

**lpValueName**

Pointer to a zero-terminated string that is the name of the value. This name is specific to a name space.

Several commonly used value name strings are associated with defined constants. These name strings include the following.

Constant	Name string
SERVICE_TYPE_VALUE_SAPID	"SapId"
SERVICE_TYPE_VALUE_CONN	"ConnectionOriented"
SERVICE_TYPE_VALUE_TCPPOINT	"TcpPort"
SERVICE_TYPE_VALUE_UDPPOINT	"UdpPort"

**lpValue**

Pointer to the value data.

## Remarks

When you use the **SetService** function to add a network service type to a name space, a **SERVICE\_TYPE\_INFO\_ABS** structure is passed as the **ServiceSpecificInfo BLOB** member of a **SERVICE\_INFO** structure. Although the **ServiceSpecificInfo** member generally should not contain pointers, an exception is made in the case of the **SERVICE\_TYPE\_INFO\_ABS** and **SERVICE\_TYPE\_VALUE\_ABS** structures.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later. Not supported on Windows 95.

**Header:** Declared in Nspapi.h.

**Unicode:** Declared as Unicode and ANSI structures.

### + See Also

**SetService**, **SERVICE\_INFO**, **SERVICE\_TYPE\_INFO\_ABS**

## sockaddr

The Windows Sockets **sockaddr** structure varies depending on the protocol selected. Except for the *sa\_family* parameter, **sockaddr** contents are expressed in network byte order.

```
struct sockaddr {
    u_short    sa_family;
    char      sa_data[14];
};
```

In Windows Sockets 2, the *name* parameter is not strictly interpreted as a pointer to a **sockaddr** structure. It is presented in this manner for Windows Sockets compatibility. The actual structure is interpreted differently in the context of different address families. The only requirements are that the first *u\_short* is the address family and the total size of the memory buffer in bytes is *namelen*.

The structure below is used with TCP/IP. Other protocols use similar structures.

```
struct sockaddr_in {
    short    sin_family;
    u_short  sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8];
};
```

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

## SOCKADDR\_IRDA

The Windows Sockets **SOCKADDR\_IRDA** structure is used in conjunction with IrDA socket operations, defined by address family **AF\_IRDA**.

```
typedef struct _SOCKADDR_IRDA
{
    u_short    irdaAddressFamily;
    u_char     irdaDeviceID[4];
    char       irdaServiceName[25];
} SOCKADDR_IRDA, *PSOCKADDR_IRDA, FAR *LPSOCKADDR_IRDA;
```

### Members

#### **irdaAddressFamily**

Address family. This member is always **AF\_IRDA**.

#### **irdaDeviceID**

Device identifier (ID) of the IrDA device to which the client wants to issue the **connect** function call. Ignored by server applications.

#### **irdaServiceName**

Well-known service name associated with a server application. Specified by servers during their **bind** function call.

### Remarks

Client applications make use of each field in the **SOCKADDR\_IRDA** structure. The **irdaDeviceID** member is obtained by a previous discovery operation performed by making a **setsockopt**(**IRLMP\_ENUMDEVICES**) function call. For more information on performing a discovery operation, see the *Notes for IrDA Sockets* section in the *Remarks* section of **setsockopt**.

The **irdaServiceName** member is filled with the well-known value that the server application specified in its **bind** function call.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in **Af\_irda.h**.

#### + See Also

**getsockopt**, **setsockopt**, **bind**, **connect**

---

## SOCKET\_ADDRESS

The **SOCKET\_ADDRESS** structure stores protocol-specific address information.

```
typedef struct _SOCKET_ADDRESS {
    LPSOCKADDR    IpSockaddr ;
    INT           iSockaddrLength ;
} SOCKET_ADDRESS, *PSOCKET_ADDRESS;
```

## Members

### IpSockaddr

Pointer to a socket address

### iSockaddrLength

Length of the socket address, in bytes.

## ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

## + See Also

**WSAIoctl, WSPIoctl**

---

# timeval

The Windows Sockets **timeval** structure is used to specify time values. It is associated with the BSD file Time.h.

```
struct timeval {
    long    tv_sec;        // seconds
    long    tv_usec;      // and microseconds
};
```

## Members

### tv\_sec

Time value, in seconds.

### tv\_usec

Time value, in microseconds.

## ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

## + See Also

**linger, select**

## TRANSMIT\_FILE\_BUFFERS

The Windows Sockets **TRANSMIT\_FILE\_BUFFERS** structure specifies data to be transmitted before and after file data during a **TransmitFile** function file transfer operation.

```
typedef struct _TRANSMIT_FILE_BUFFERS {
    PVOID   Head;
    DWORD   HeadLength;
    PVOID   Tail;
    DWORD   TailLength;
} TRANSMIT_FILE_BUFFERS;
```

### Members

#### Head

Pointer to a buffer that contains data to be transmitted before the file data is transmitted.

#### HeadLength

Number of bytes in the buffer pointed to by **Head** that are to be transmitted.

#### Tail

Pointer to a buffer that contains data to be transmitted after the file data is transmitted.

#### TailLength

Number of bytes of data in the buffer pointed to by the **Tail** member that are to be transmitted.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later. Not supported on Windows 95.

**Header:** Declared in Winsock.h.

### + See Also

**TransmitFile**

---

## WSABUF

The Windows Sockets **WSABUF** structure enables the creation or manipulation of a data buffer.

```
typedef struct __WSABUF {
    u_long   len;
    char FAR *buf;
} WSABUF, FAR * LPWSABUF;
```



## Members

### len

Length of the buffer.

### buf

Pointer to the buffer.

## ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

# WSADATA

The members of the Windows Sockets **WSADATA** structure are:

```
typedef struct WSADATA {
    WORD          wVersion;
    WORD          wHighVersion;
    char          szDescription[WSADESCRIPTION_LEN+1];
    char          szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short iMaxSockets;
    unsigned short iMaxUdpDg;
    char FAR *    lpVendorInfo;
} WSADATA, *LPWSADATA;
```

## Members

### wVersion

Version of the Windows Sockets specification that the *Ws2\_32.dll* expects the caller to use.

### wHighVersion

Highest version of the Windows Sockets specification that this *.dll* can support (also encoded as above). Normally this is the same as *wVersion*.

### szDescription

Null-terminated ASCII string into which the *Ws2\_32.dll* copies a description of the Windows Sockets implementation. The text (up to 256 characters in length) can contain any characters except control and formatting characters: the most likely use that an application can put this to is to display it (possibly truncated) in a status message.

### szSystemStatus

Null-terminated ASCII string into which the *WSs2\_32.dll* copies relevant status or configuration information. The *Ws2\_32.dll* should use this parameter only if the information might be useful to the user or support staff: it should not be considered as an extension of the *szDescription* parameter.

### **iMaxSockets**

Retained for backward compatibility, but should be ignored for Windows Sockets version 2 and later, as no single value can be appropriate for all underlying service providers.

### **iMaxUdpDg**

Ignored for Windows Sockets version 2 and onward. **iMaxUdpDg** is retained for compatibility with Windows Sockets specification 1.1, but should not be used when developing new applications. For the actual maximum message size specific to a particular Windows Sockets service provider and socket type, applications should use **getsockopt** to retrieve the value of option **SO\_MAX\_MSG\_SIZE** after a socket has been created.

### **IpVendorInfo**

Ignored for Windows Sockets version 2 and onward. It is retained for compatibility with Windows Sockets specification 1.1. Applications needing to access vendor-specific configuration information should use **getsockopt** to retrieve the value of option **PVD\_CONFIG**. The definition of this value (if utilized) is beyond the scope of this specification.

---

**Note** An application should ignore the **iMaxSockets**, **iMaxUdpDg**, and **IpVendorInfo** members in **WSADATA** if the value in **wVersion** after a successful call to **WSAStartup** is at least 2. This is because the architecture of Windows Sockets has been changed in version 2 to support multiple providers, and **WSADATA** no longer applies to a single vendor's stack. Two new socket options are introduced to supply provider-specific information: **SO\_MAX\_MSG\_SIZE** (replaces the **iMaxUdpDg** element) and **PVD\_CONFIG** (allows any other provider-specific configuration to occur).

---

### **!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

---

## **WSANAMESPACE\_INFO**

The Windows Sockets **WSANAMESPACE\_INFO** structure contains all registration information for a name space provider.

```
typedef struct _WSANAMESPACE_INFO {
    GUID                NSProviderId;
    DWORD               dwNameSpace;
    BOOL                fActive;
    DWORD               dwVersion;
    LPTSTR              lpszIdentifier;
} WSANAMESPACE_INFO, *PWSANAMESPACE_INFO;
```

## Members

### **NSProviderId**

Unique identifier for this name-space provider.

### **dwNameSpace**

Name space supported by this implementation of the provider.

### **fActive**

If TRUE, indicates that this provider is active. If FALSE, the provider is inactive and is not accessible for queries, even if the query specifically references this provider.

### **dwVersion**

Name space–version identifier.

### **lpszIdentifier**

Display string for the provider.

## ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Unicode:** Declared as Unicode and ANSI structures.

---

# WSANETWORKEVENTS

The Windows Sockets **WSANETWORKEVENTS** structure is used to store a socket's internal information about network events.

```
typedef struct _WSANETWORKEVENTS {
    long    lNetworkEvents;
    int     iErrorCodes[FD_MAX_EVENTS];
} WSANETWORKEVENTS, *LPWSANETWORKEVENTS;
```

## Members

### **lNetworkEvents**

Indicates which of the FD\_XXX network events have occurred.

### **iErrorCodes**

An array that contains any associated error codes, with an array index that corresponds to the position of event bits in **lNetworkEvents**. The identifiers FD\_READ\_BIT, FD\_WRITE\_BIT and other can be used to index the **iErrorCodes** array.

## ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

 See Also

**WSAEnumNetworkEvents, WSAEventSelect**

## WSAOVERLAPPED

The Windows Sockets **WSAOVERLAPPED** structure provides a communication medium between the initiation of an overlapped I/O operation and its subsequent completion. The **WSAOVERLAPPED** structure is designed to be compatible with the Win32 **OVERLAPPED** structure:

```
typedef struct _WSAOVERLAPPED {
    DWORD      Internal;
    DWORD      InternalHigh;
    DWORD      Offset;
    DWORD      OffsetHigh;
    WSAEVENT   hEvent;
} WSAOVERLAPPED, *LPWSAOVERLAPPED;
```

### Members

#### Internal

Reserved for internal use. The Internal member is used internally by the entity that implements overlapped I/O. For service providers that create sockets as installable file system (IFS) handles, this parameter is used by the underlying operating system. Other service providers (non-IFS providers) are free to use this parameter as necessary.

#### InternalHigh

Reserved. Used internally by the entity that implements overlapped I/O. For service providers that create sockets as IFS handles, this parameter is used by the underlying operating system. NonIFS providers are free to use this parameter as necessary.

#### Offset

Reserved for use by service providers.

#### OffsetHigh

Reserved for use by service providers.

#### hEvent

If an overlapped I/O operation is issued without an I/O completion routine (*lpCompletionRoutine* is null), then this parameter should either contain a valid handle to a **WSAEVENT** object or be null. If *lpCompletionRoutine* is non-null then applications are free to use this parameter as necessary.

 Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**+** See Also

**WSASend, WSARecv, WSAGetOverlappedResult**

## WSAPROTOCOL\_INFO

The Windows Sockets **WSAPROTOCOL\_INFO** structure is used to store or retrieve complete information for a given protocol.

```
typedef struct _WSAPROTOCOL_INFO {
    DWORD          dwServiceFlags1;
    DWORD          dwServiceFlags2;
    DWORD          dwServiceFlags3;
    DWORD          dwServiceFlags4;
    DWORD          dwProviderFlags;
    GUID           ProviderId;
    DWORD          dwCatalogEntryId;
    WSAPROTOCOLCHAIN ProtocolChain;
    int            iVersion;
    int            iAddressFamily;
    int            iMaxSockAddr;
    int            iMinSockAddr;
    int            iSocketType;
    int            iProtocol;
    int            iProtocolMaxOffset;
    int            iNetworkByteOrder;
    int            iSecurityScheme;
    DWORD          dwMessageSize;
    DWORD          dwProviderReserved;
    TCHAR          szProtocol[WSAPROTOCOL_LEN+1];
} WSAPROTOCOL_INFO, *LPWSAPROTOCOL_INFO;
```

### Members

#### **dwServiceFlags1**

Bitmask describing the services provided by the protocol. The following values are possible:

#### **XP1\_CONNECTIONLESS**

Provides connectionless (datagram) service. If not set, the protocol supports connection-oriented data transfer.

#### **XP1\_GUARANTEED\_DELIVERY**

Guarantees that all data sent will reach the intended destination.

**XP1\_GUARANTEED\_ORDER**

Guarantees that data only arrives in the order in which it was sent and that it is not duplicated. This characteristic does not necessarily mean that the data is always delivered, but that any data that is delivered is delivered in the order in which it was sent.

**XP1\_MESSAGE\_ORIENTED**

Honors message boundaries—as opposed to a stream-oriented protocol where there is no concept of message boundaries.

**XP1\_PSEUDO\_STREAM**

A message-oriented protocol, but message boundaries are ignored for all receipts. This is convenient when an application does not desire message framing to be done by the protocol.

**XP1\_GRACEFUL\_CLOSE**

Supports two-phase (graceful) close. If not set, only abortive closes are performed.

**XP1\_EXPEDITED\_DATA**

Supports expedited (urgent) data.

**XP1\_CONNECT\_DATA**

Supports connect data.

**XP1\_DISCONNECT\_DATA**

Supports disconnect data.

**XP1\_INTERRUPT**

Bit is reserved.

**XP1\_SUPPORT\_BROADCAST**

Supports a broadcast mechanism.

**XP1\_SUPPORT\_MULTIPOINT**

Supports a multipoint or multicast mechanism. Control and data plane attributes are indicated below.

**XP1\_MULTIPOINT\_CONTROL\_PLANE**

Indicates whether the control plane is rooted (value = 1) or nonrooted (value = 0).

**XP1\_MULTIPOINT\_DATA\_PLANE**

Indicates whether the data plane is rooted (value = 1) or nonrooted (value = 0).

**XP1\_QOS\_SUPPORTED**

Supports quality of service requests.

**XP1\_UNI\_SEND**

Protocol is unidirectional in the send direction.

**XP1\_UNI\_RECV**

Protocol is unidirectional in the recv direction.

**XP1\_IFS\_HANDLES**

Socket descriptors returned by the provider are operating system Installable File System (IFS) handles.

**XP1\_PARTIAL\_MESSAGE**

The MSG\_PARTIAL flag is supported in **WSASend** and **WSASendTo**.

Note that only one of **XP1\_UNI\_SEND** or **XP1\_UNI\_RECV** may be set. If a protocol can be unidirectional in either direction, two **WSAPROTOCOL\_INFOW** structures should be used. When neither bit is set, the protocol is considered to be bidirectional.

**dwServiceFlags2**

Reserved for additional protocol-attribute definitions.

**dwServiceFlags3**

Reserved for additional protocol-attribute definitions.

**dwServiceFlags4**

Reserved for additional protocol-attribute definitions.

**dwProviderFlags**

Provides information about how this protocol is represented in the protocol catalog. The following flag values are possible:

**PFL\_MULTIPLE\_PROTO\_ENTRIES**

Indicates that this is one of two or more entries for a single protocol (from a given provider) which is capable of implementing multiple behaviors. An example of this is SPX which, on the receiving side, can behave either as a message-oriented or a stream-oriented protocol.

**PFL\_RECOMMENDED\_PROTO\_ENTRY**

Indicates that this is the recommended or most frequently used entry for a protocol that is capable of implementing multiple behaviors.

**PFL\_HIDDEN**

Set by a provider to indicate to the *Ws2\_32.dll* that this protocol should not be returned in the result buffer generated by **WSAEnumProtocols**. Obviously, a Windows Sockets 2 application should never see an entry with this bit set.

**PFL\_MATCHES\_PROTOCOL\_ZERO**

Indicates that a value of zero in the *protocol* parameter of **socket** or **WSASocket** matches this protocol entry.

**ProviderId**

Globally unique identifier assigned to the provider by the service provider vendor. This value is useful for instances where more than one service provider is able to implement a particular protocol. An application may use the **dwProviderId** value to distinguish between providers that might otherwise be indistinguishable.

**dwCatalogEntryId**

Unique identifier assigned by the *WS2\_32.DLL* for each **WSAPROTOCOL\_INFOW** structure.

**WSAPROTOCOLCHAIN** ProtocolChain;

If the length of the chain is 0, this **WSAPROTOCOL\_INFOW** entry represents a layered protocol which has Windows Sockets 2 SPI as both its top and bottom edges. If the length of the chain equals 1, this entry represents a base protocol whose Catalog Entry identifier is in the **dwCatalogEntryId** member of the **WSAPROTOCOL\_INFOW** structure. If the length of the chain is larger than 1, this entry represents a protocol chain which consists of one or more layered protocols on top of a base protocol. The corresponding Catalog Entry identifiers are in the ProtocolChain.ChainEntries array starting with the layered protocol at the top (the zero element in the ProtocolChain.ChainEntries array) and ending with the base protocol. Refer to the Windows Sockets 2 Service Provider Interface specification for more information on protocol chains.

**iVersion**

Protocol version identifier.

**iAddressFamily**

Value to pass as the address family parameter to the **socket/WSASocket** function in order to open a socket for this protocol. This value also uniquely defines the structure of protocol addresses **SOCKADDRs** used by the protocol.

**iMaxSockAddr**

Maximum address length.

**iMinSockAddr**

Minimum address length.

**iSocketType**

Value to pass as the socket type parameter to the **socket** function in order to open a socket for this protocol.

**iProtocol**

Value to pass as the protocol parameter to the **socket** function in order to open a socket for this protocol.

**iProtocolMaxOffset**

Maximum value that may be added to **iProtocol** when supplying a value for the *protocol* parameter to **socket** and **WSASocket**. Not all protocols allow a range of values. When this is the case **iProtocolMaxOffset** is zero.

**iNetworkByteOrder**

Currently these values are manifest constants (**BIGENDIAN** and **LITTLEENDIAN**) that indicate either big-endian or little-endian with the values 0 and 1 respectively.

**iSecurityScheme**

Indicates the type of security scheme employed (if any). A value of **SECURITY\_PROTOCOL\_NONE** is used for protocols that do not incorporate security provisions.



**dwMessageSize**

Maximum message size supported by the protocol. This is the maximum size that can be sent from any of the host's local interfaces. For protocols that do not support message framing, the actual maximum that can be sent to a given address may be less. There is no standard provision to determine the maximum inbound message size. The following special values are defined:

0

The protocol is stream-oriented and hence the concept of message size is not relevant.

0x1

The maximum outbound (send) message size is dependent on the underlying network MTU (maximum sized transmission unit) and hence cannot be known until after a socket is bound. Applications should use **getsockopt** to retrieve the value of `SO_MAX_MSG_SIZE` after the socket has been bound to a local address.

0xFFFFFFFF

The protocol is message-oriented, but there is no maximum limit to the size of messages that may be transmitted.

**dwProviderReserved**

Reserved for use by service providers.

**szProtocol**

Array of characters that contains a human-readable name identifying the protocol, for example "SPX2". The maximum number of characters allowed is `WSAPROTOCOL_LEN`, which is defined to be 255.

**! Requirements**

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Winsock2.h`.

**Unicode:** Declared as Unicode and ANSI structures.

**+ See Also**

**WSAEnumProtocols**, **WSASend**, **WSASendTo**, **getsockopt**, **socket**

## WSAPROTOCOLCHAIN

The Windows Sockets **WSAPROTOCOLCHAIN** structure contains a counted list of Catalog Entry identifiers that comprise a protocol chain. This structure is defined as follows:

```
typedef struct _WSAPROTOCOLCHAIN {
    int ChainLen;
    DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, *LPWSAPROTOCOLCHAIN;
```

## Members

### ChainLen

Length of the chain. The following settings apply:

Setting **ChainLen** to zero indicates a layered protocol

Setting **ChainLen** to one indicates a base protocol

Setting **ChainLen** to greater than one indicates a protocol chain

### ChainEntries

Array of protocol chain entries.

## Remarks

If the length of the chain is larger than 1, this structure represents a protocol chain which consists of one or more layered protocols on top of a base protocol. The corresponding Catalog Entry IDs are in the ProtocolChain.ChainEntries array starting with the layered protocol at the top (the zeroth element in the ProtocolChain.ChainEntries array) and ending with the base protocol. Refer to Windows Sockets 2 Service Provider Interface for more information on protocol chains.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

### + See Also

**WSAEnumProtocols**

# WSAQUERYSET

The Windows Sockets **WSAQUERYSET** structure provides relevant information about a given service, including service class ID, service name, applicable name-space identifier and protocol information, as well as a set of transport addresses at which the service listens.

```
typedef struct _WSAQuerySet {
    DWORD           dwSize;
    LPTSTR         lpszServiceInstanceName;
    LPGUID         lpServiceClassId;
    LPWSAVERSION  lpVersion;
    LPTSTR         lpszComment;
    DWORD          dwNameSpace;
    LPGUID         lpNSProviderId;
    LPTSTR         lpszContext;
```

*(continued)*

(continued)

```

DWORD          dwNumberOfProtocols;
LPAFPROTOCOLS  lpafpProtocols;
LPTSTR         lpzQueryString;
DWORD          dwNumberOfCsAddrs;
LPCSADDR_INFO  lpcsaBuffer;
DWORD          dwOutputFlags;
LPBLOB         lpBlob;
} WSAQUERYSET, *PWSAQUERYSETW;

```

## Members

### dwSize

Must be set to sizeof(**WSAQUERYSET**). This is a versioning mechanism.

### dwOutputFlags

Ignored for queries.

### lpzServiceInstanceName

(Optional) Referenced string contains service name. The semantics for using wildcards within the string are not defined, but can be supported by certain name space providers.

### IpServiceClassId

(Required) The GUID corresponding to the service class.

### IpVersion

(Optional) References desired version number and provides version comparison semantics (that is, version must match exactly, or version must be not less than the value supplied).

### lpzComment

Ignored for queries.

### dwNameSpace

Identifier of a single name space in which to constrain the search, or NS\_ALL to include all name spaces.

### IpNSProviderId

(Optional) References the GUID of a specific name-space provider, and limits the query to this provider only.

### lpzContext

(Optional) Specifies the starting point of the query in a hierarchical name space.

### dwNumberOfProtocols

Size of the protocol constraint array, can be zero.

### lpafpProtocols

(Optional) References an array of **AFPROTOCOLS** structure. Only services that utilize these protocols will be returned.

### lpzQueryString

(Optional) Some name spaces (such as Whois++) support enriched SQL-like queries that are contained in a simple text string. This parameter is used to specify that string.

**dwNumberOfCsAddrs**

Ignored for queries.

**IpcsaBuffer**

Ignored for queries.

**IpBlob**

(Optional) This is a pointer to a provider-specific entity.

**Remarks**

In most instances, applications interested in only a particular transport protocol should constrain their query by address family and protocol rather than by name space. This would allow an application that needs to locate a TCP/IP service, for example, to have its query processed by all available name spaces such as the local hosts file, DNS, and NIS.

**! Requirements**

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Unicode:** Declared as Unicode and ANSI structures.

**+ See Also**

**WSAQuerySet, WSASetService, WSALookupServiceBegin, WSALookupServiceNext**

## WSASERVICECLASSINFO

The Windows Sockets **WSASERVICECLASSINFO** structure contains information about a specified service class. For each service class in Windows Sockets 2, there is a single **WSASERVICECLASSINFO** structure.

```
typedef struct _WSAServiceClassInfo {
    LPGUID          IpServiceClassId;
    LPTSTR          IpszServiceClassName;
    DWORD           dwCount;
    LPWSANSCONFIG  IpClassInfos;
} WSASERVICECLASSINFO, *PWSASERVICECLASSINFO;
```

**Members****IpServiceClassId**

Unique Identifier (GUID) for the service class.

**IpszServiceClassName**

Well known associated with the service class.

**dwCount**

Number of entries in **IpClassInfos**.

**IpClassInfos**

Array of **WSANSCLASSINFOW** structures that contains information about the service class.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Winsock2.h.

**Unicode:** Declared as Unicode and ANSI structures.

**+** See Also

**NSPGetServiceClassInfo, NSPLookupServiceBegin**

---

## WSATHREADID

The Windows Sockets **WSATHREADID** structure enables a provider to identify a thread on which asynchronous procedure calls (APCs) can be queued using the **WPUQueueApc** function.

```
typedef struct _WSATHREADID {
    HANDLE    ThreadHandle;
    DWORD     Reserved;
} WSATHREADID, *LPWSATHREADID;
```

**Members****ThreadHandle**

Handle to the thread ID.

**Reserved**

Reserved.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

**+** See Also

**WPUQueueApc, WSPIoctl, WSPSend, WSPRecv**

## Windows Sockets Enumeration in the API

The following enumeration is used in Windows Sockets:

- **WSAECOMPARATOR**

The following enumeration is obsolete:

- **GUARANTEE**

---

## GUARANTEE

The Windows Sockets **GUARANTEE** enumeration is no longer used. For information regarding the Windows 2000 implementation of Quality of Service, and the associated API, refer to Chapters 13 through 17 later in this volume.

---

## WSAECOMPARATOR

The Windows Sockets **WSAECOMPARATOR** enumeration type is used for version comparison semantics in Windows Sockets 2.

```
typedef enum _WSAEcomparator {
    COMP_EQUAL = 0,
    COMP_NOTLESS
} WSAECOMPARATOR, *PWSAECOMPARATOR;
```

Enumerator Value	Meaning
COMP_EQUAL	Used for determining whether version values are equal.
COMP_NOTLESS	Used for determining whether a version value is no less than a specified value.

### ! Requirements

**Version:** Requires Windows Sockets 1.1 or later.

**Header:** Declared in Winsock2.h.

### + See Also

**WSALookupServiceBegin**, **WSAQuerySet**, **NSPLookupServiceBegin**, **NSPLookupServiceNext**, **WSASetService**, **NSPSetService**



---

## CHAPTER 10

# Winsock 2 SPI Overview

## Welcome to Windows Sockets 2 SPI

This chapter describes the Windows Sockets 2 Service Provider Interface (SPI). It consists, primarily, of information from the Windows Sockets 2 SPI specification, but also includes additional information. The information in this document is not presented in exactly the same way as specification.

## Using the SPI Document

This document provides the online material needed to create Windows Sockets service or transport providers for Windows operating systems, using the Microsoft implementation of Windows Sockets 2. It is intended as a reference tool and outlines the functions of the Windows Sockets SPI.

You should be familiar with Win32 programming concepts and the Windows Sockets API to make the best use of this document. Thus, you may want to refer to other references that provide a more systematic guide to writing Windows Sockets applications.

---

**Note** This documentation is intended for service provider developers, also known as transports. If you are developing a Windows Sockets 2 application, see chapters 6 through 9 in this volume.

---

## Overview of the Windows Sockets 2 SPI

Windows Sockets 2 utilizes the sockets paradigm that was first popularized by Berkeley Software Distribution (BSD) UNIX. It was later adapted for Microsoft Windows in the Windows Sockets 1.1.

One of the primary goals of Windows Sockets 2 has been to provide a protocol-independent interface fully capable of supporting the emerging networking capabilities, such as real-time multimedia communications.

Windows Sockets 2 is an interface, not a protocol. As an interface, it is used to discover and utilize the communications capabilities of any number of underlying transport protocols. Because it is not a protocol, it does not in any way affect the bits on the wire, and does not need to be utilized on both ends of a communications link.



Windows Sockets programming previously centered around TCP/IP. Some of the programming practices that worked with TCP/IP do not work with every protocol. As a result, the Windows Sockets 2 API added new functions where necessary that, in turn, must be implemented in the underlying service provider.

Windows Sockets 2 has changed its architecture to provide easier access to multiple transport protocols. Following the Windows Open System Architecture (WOSA) model, Windows Sockets 2 now defines a standard SPI between the application programming interface (API), with its functions exported from `Ws2_32.dll`, and the protocol stacks. Consequently, Windows Sockets 2 support is not limited to TCP/IP protocol stacks as is the case for Windows Sockets 1.1. For more information, see *Windows Sockets 2 Architectural Overview*.

There are new challenges in developing Windows Sockets 2 applications. When sockets only supported TCP/IP, a developer could create an application that supported only two socket types: connectionless and connection-oriented. Connectionless protocols used `SOCK_DGRAM` sockets and connection-oriented protocols used `SOCK_STREAM` sockets. Now, these are just two of the many new socket types. Additionally, developers can no longer rely on socket type to describe all the essential attributes of a transport protocol.

The service provider is created by implementing the functions defined in this document. This is a new challenge differing significantly from Windows Sockets 1.1.

## Windows Sockets 2 SPI Features

The new Windows Sockets 2 extends functionality in a number of areas.

Features	Description
Access to protocols other than TCP/IP	Allows an application to use the familiar socket interface to achieve simultaneous access to a number of installed transport protocols.
Overlapped I/O with scatter/gather	Incorporates the overlapped paradigm for socket I/O and incorporates scatter/gather capabilities as well, following the model established in Win32 environments.
Protocol-independent name resolution facilities:	Includes a standardized set of functions for querying and working with the myriad of name resolution domains that exist today (for example DNS, SAP, and X.500).
Protocol-independent multicast and multipoint:	Windows Sockets 2 applications discover what type of multipoint or multicast capabilities a transport provides and use these facilities in a generic manner.

Features	Description
Quality of Service (QOS)	Establishes conventions applications use to negotiate required service levels for parameters such as bandwidth and latency. Other QOS-related enhancements mechanisms for network-specific QOS extensions. QOS implementation in Windows is explained in detail in its own section under Networking Services in the Platform SDK.
Other frequently requested extensions	Incorporates shared sockets and conditional acceptance; exchange of user data at connection setup/teardown time; and protocol-specific extension mechanisms.

## Microsoft Extensions and the Windows Sockets 2 SPI

The Windows Sockets 2 specification defines an extension mechanism that exposes advanced transport functionality to application programs. See the *SPI: Function Extension Mechanism in the SPI* section. The following Microsoft-specific extensions that were added to Windows Sockets 1.1 are also available to Windows Sockets 2 applications:

- **AcceptEx**
- **GetAcceptExSockaddrs**
- **TransmitFile**
- **WSARecvEx**

These functions are not exported from the `Ws2_32.dll`; they are exported from `Mswsock.dll`.

An application written to use the Microsoft-specific extensions to Windows Sockets will not run correctly over a Windows Sockets service provider that does not support those extensions.

## Socket Handles for the Windows Sockets 2 SPI

A socket handle can optionally be a file handle In Windows Sockets 2. It is possible to use socket handles with **ReadFile**, **WriteFile**, **ReadFileEx**, **WriteFileEx**, **DuplicateHandle**, and other Win32 functions. Not all transport service providers will support this option. For an application to run over the widest possible number of service providers, it should not assume that socket handles are file handles.

Windows Sockets 2 has expanded certain functions used for transferring data between sockets using handles. The functions offer advantages specific to sockets for transferring data and include **WSARecv**, **WSASend**, and **WSADuplicateSocket**.

## Windows Sockets 2 Architectural Overview

This chapter provides an overview of the Windows Sockets 2 architecture. It describes and illustrates the relationships between applications, the Windows Sockets 2 DLL and Windows Sockets service providers. A high-level view of the division of responsibilities between the Windows Sockets 2 DLL and the Windows Sockets service providers is also provided.

### Windows Sockets 2 as a WOSA Component

The Windows Sockets 2 network transport and name resolution services are provided as a Windows Open Services Architecture (WOSA) component. They consist of both an application programming interface (API) used by applications and service provider interfaces (SPIs) implemented by service providers. This document defines the service provider interfaces for data transport and name resolution. While it is designed to be a stand-alone reference for the implementers of Windows Sockets 2 service providers, developers are strongly encouraged to obtain and become familiar with the *Windows Sockets 2 Application Programming Interface* as well.

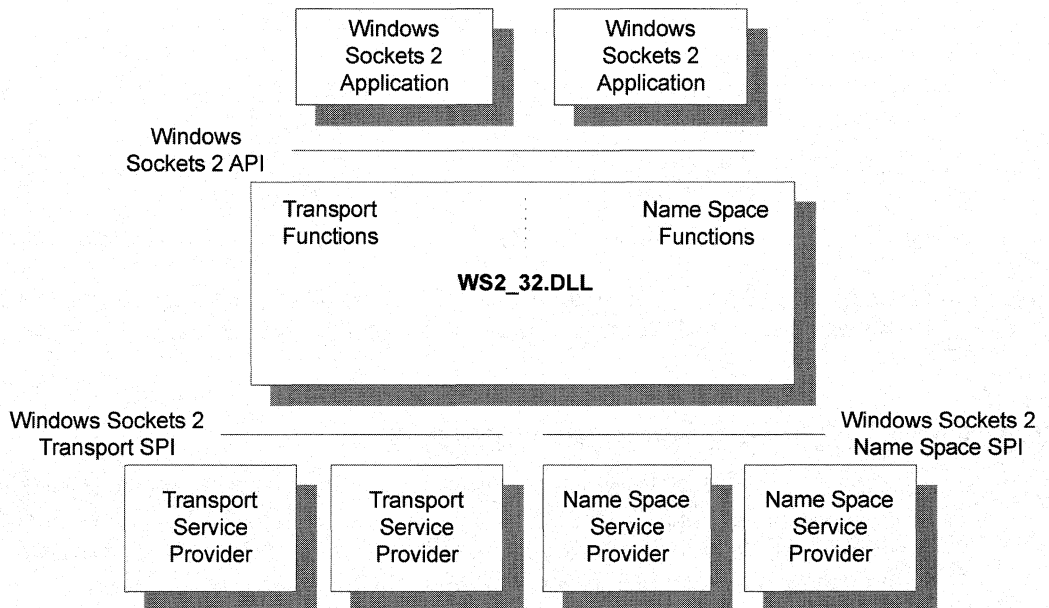
WOSA provides a common set of interfaces for connecting front-end applications with back-end services. The front-end application and back-end services need not speak each other's language in order to communicate as long as they both know how to talk to their respective WOSA interfaces. As a result, WOSA allows application developers and vendors of back-end services to mix and match applications and services to build solutions that shield programmers and users from the underlying complexity of the system. WOSA defines an abstraction layer to heterogeneous computing resources through the WOSA set of APIs. Because this set of APIs is extensible, new services and their corresponding APIs can be added as needed. Applications written to the WOSA APIs have access not only to the various computing environments supported today, but also to all additional environments as they become available. Moreover, applications don't have to be modified in any way to enjoy this support.

Each service recognized by WOSA also has a set of interfaces that service-provider vendors use to take advantage of the seamless interoperability that WOSA provides. To provide transparent access for applications, each implementation of a particular WOSA service simply needs to support the functions defined by its service provider interface.

Like most WOSA components, Windows Sockets 2 uses a Windows Dynamic-Link Library (DLL) that allows applications and service providers software components to be bound together at runtime. In this way, applications are able to connect to services dynamically. An application needs to know only the definition of the interface, not its implementation.

## Windows Sockets 2 DLLs

Windows Sockets network services follow the WOSA model, meaning that there exists a Windows Sockets Application Programming Interface (API), which is the application programmer's access to network services, Windows Sockets Service Provider Interfaces (SPIs) which are implemented by transport service providers and name resolution service provider vendors, and `Ws2_32.dll`. The Windows Sockets 2 WOSA-compliant architecture is illustrated in Figure 10-1.



**Figure 10-1: Winsock 2 WOSA-Compliant Architecture.**

The SPI is intended to be used within all 32-bit implementations and versions of Microsoft® Windows® including Windows® NT® and Windows® 95®.

## Function Interface Model

Windows Sockets transport and name space service providers are DLLs with a single *exported* procedure entry point for the service provider initialization function **WSPStartup** or **NSPStartup**, respectively. All other service provider functions are made accessible to the `Ws2_32.dll` through the service provider's dispatch table. Service provider DLL's are loaded into memory by the `Ws2_32.dll` only when needed, and are unloaded when their services are no longer required.

The SPI also defines several circumstances in which a transport service provider calls up into the `Ws2_32.dll` (upcalls) to obtain DLL support services. The transport service provider DLL is given the `Ws2_32.dll`'s upcall dispatch table through the *UpcallTable* parameter to **WSPStartup**.

Service providers should have their file extension changed from “DLL” to “.WSP” or “.NSP”. This requirement is not strict. A service provider will still operate with the Ws2\_32.dll with any file extension.

## Naming Conventions

The Windows Sockets SPI uses the following function prefix naming convention.

Prefix	Meaning	Description
WSP	Windows Sockets Service Provider	Transport service provider entry points
WPU	Windows Sockets Provider Upcall	Ws2_32.dll entry points for service providers
WSC	Windows Sockets Configuration	WS2_32.dll entry points for installation applets
NSP	Name Space Provider	Name space provider entry points

As described above, these entry points are *not* exported (with the exception of **WSPStartup** and **NSPStartup**), but are accessed through an exchange of dispatch tables.

## Windows Sockets 2 Service Providers

As shown in the figure, Windows Sockets 2 Architecture, there are two basic types of service providers: transport providers and name space providers. Examples of transport providers include protocol stacks such as TCP/IP or IPX/SPX, while an example of a name space provider would be an interface to the Internet’s Domain Naming System (DNS). Separate sections of the service provider interface specification apply to each type of service provider.

Transport and name space service providers must be registered with the Ws2\_32.dll at the time they are installed. This registration need only be done once for each provider as the necessary information is retained in persistent storage.

### Transport Service Providers

A given transport service provider supports one or more protocols. For example, a TCP/IP provider would supply (as a minimum) the TCP and UDP protocols, while an IPX/SPX provider might supply IPX, SPX, and SPX II. Each protocol supported by a particular provider is described by a **WSAPROTOCOL\_INFOW** structure, and the total set of such structures can be thought of as the catalog of installed protocols. Applications can retrieve the contents of this catalog (see **WSAEnumProtocols**), and by examining the available **WSAPROTOCOL\_INFOW** structures, discover the communications attributes associated with each protocol.

## Layered Protocols and Protocol Chains in the SPI

Windows Sockets 2 accommodates the notion of a layered protocol. A layered protocol is one that implements only higher level communications functions, while relying on an underlying transport stack for the actual exchange of data with a remote endpoint. An example of such a layered protocol would be a security layer that adds protocol to the connection establishment process in order to perform authentication and to establish a mutually agreed upon encryption scheme. Such a security protocol would generally require the services of an underlying reliable transport protocol such as TCP or SPX. The term *base protocol* refers to a protocol such as TCP or SPX which is fully capable of performing data communications with a remote endpoint, and the term *layered protocol* is used to describe a protocol that cannot stand alone. A *protocol chain* would then be defined as one or more layered protocols strung together and anchored by a base protocol.

This stringing together of layered protocols and base protocols into chains can be accomplished by arranging for the layered protocols to support the Windows Sockets 2 SPI at both their upper and lower edges. A special **WSAPROTOCOL\_INFOW** structure is created, which refers to the protocol chain as a whole, and which describes the explicit order in which the layered protocols are joined. (See Figure 10-2.)

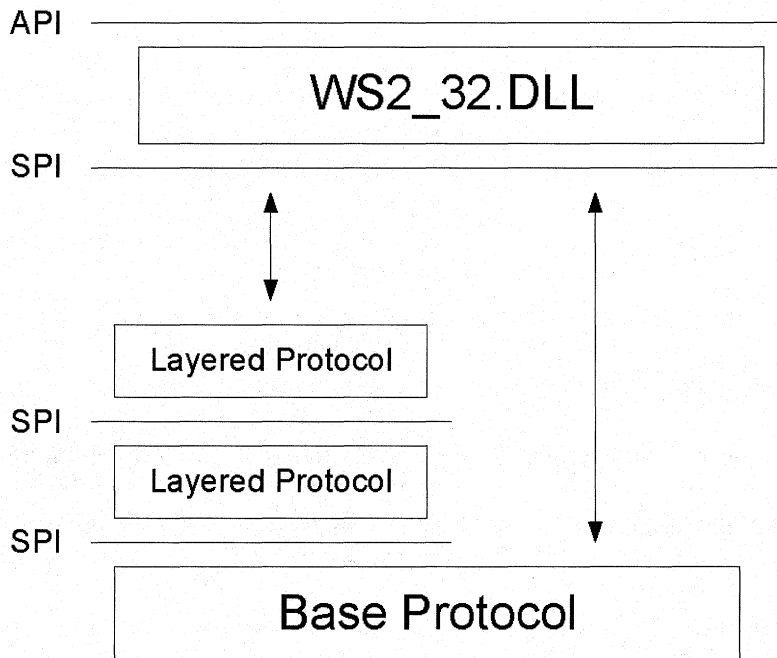


Figure 10-2: Joining Layered Protocols.

## Namespace Service Providers

A namespace provider implements an interface mapping between the Windows Sockets 2 name space SPI and the native programmatic interface of an existing name service such as DNS, X.500, Netware® Directory Services (NDS), etc. While a name space provider supports exactly one name space, it is possible for multiple providers for a given name space to be installed. It is also possible for a single DLL to create an instance of multiple different name space providers. As name space providers are installed, a catalog of **WSANAMESPACE\_INFO** structures is maintained. An application may use **WSAEnumNameSpaceProviders** to discover which name spaces are supported on a machine. Refer to *Section Name Resolution Service Provider Requirements* for detailed information.

## Legacy GetXbyY Service Providers

Windows Sockets 2 fully supports the TCP/IP-specific name resolution facilities found in Windows Sockets version 1.1. It does this by including the set of GetXbyY functions in the SPI. However, the treatment of this set of functions is somewhat different from the rest of the SPI functions. The GetXbyY functions appearing in the SPI are prefaced with GETXBYYSP\_, and are summarized as follows:

### Berkeley Style Functions

SPI function name	Description
<b>GETXBYYSP_gethostbyaddr</b>	Supply a <b>hostent</b> structure for the specified host address.
<b>GETXBYYSP_gethostbyname</b>	Supply a <b>hostent</b> structure for the specified host name.
<b>GETXBYYSP_getprotobyname</b>	Supply a <b>protoent</b> structure for the specified protocol name.
<b>GETXBYYSP_getprotobynumber</b>	Supply a <b>protoent</b> structure for the specified protocol number.
<b>GETXBYYSP_getservbyname</b>	Supply a <b>servent</b> structure for the specified service name.
<b>GETXBYYSP_getservbyport</b>	Supply a <b>servent</b> structure for the service at the specified port.
<b>GETXBYYSP_gethostname</b>	Return the standard host name for the local machine.

## Async Style Functions

SPI function name	Description
<b>GETXBYYSP_WSAAsyncGetHostByAddr</b>	Supply a <b>hostent</b> structure for the specified host address.
<b>GETXBYYSP_WSAAsyncGetHostByName</b>	Supply a <b>hostent</b> structure for the specified host name.
<b>GETXBYYSP_WSAAsyncGetProtoByName</b>	Supply a <b>protoent</b> structure for the specified protocol name.
<b>GETXBYYSP_WSAAsyncGetProtoByNumber</b>	Supply a <b>protoent</b> structure for the specified protocol number.
<b>GETXBYYSP_WSAAsyncGetServByName</b>	Supply a <b>servent</b> structure for the specified service name.
<b>GETXBYYSP_WSAAsyncGetServByPort</b>	Supply a <b>servent</b> structure for the service at the specified port.
<b>GETXBYYSP_WSACancelAsyncRequest</b>	Cancel an asynchronous GetXbyY operation.

The syntax and semantics of these GetXbyY functions are exactly the same as is documented in the Windows Sockets 2 API Specification and are, therefore, not repeated in this document.

The Windows Sockets 2 DLL allows exactly one service provider to offer these services. Therefore, there is no need to include pointers to these functions in the procedure table received from service providers at startup. In 32-bit environments the path to the DLL that implements these functions is retrieved from the registry key named:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\WinSock2\
Parameters\GetXbyYLibraryPath
```

---

**Warning** How the path is obtained in 16-bit environments has not yet been determined.

---

## Built-In Default GetXbyY Service Provider

A default GetXbyY service provider is integrated into the standard Windows Sockets 2 run-time components. This default provider implements all of the above functions, thus it is not required for these functions to be implemented by any name space provider. However, a name space provider is free to provide any or all of these functions (and thus override the defaults) by simply storing the string which is the path to the DLL that implements these functions in the indicated registry key. Any of the GetXbyY functions not exported by the named provider DLL will be supplied through the built-in defaults. Note, however, that if a provider elects to supply any of the async version of the GetXbyY functions, he should supply all of the async functions so that the cancel operation will work appropriately.



For 32-bit environments, the current implementation of the default GetXbyY service provider resides within the Microsoft Wsock32.dll. Depending on how the TCP/IP settings have been established through Control Panel, name resolution will occur using either DNS or local host files. When DNS is used, the default GetXbyY service provider uses standard Windows Sockets 1.1 API calls to communicate with the DNS server. These transactions will occur using whatever TCP/IP stack is configured as the default TCP/IP stack. Two special cases however, deserve special mention.

The default implementation of **GETXBYYSP\_gethostname** obtains the local host name from the registry. This will correspond to the name assigned to "My Computer". The default implementation of **GETXBYYSP\_gethostbyname** and **GETXBYYSP\_WSAAsyncGetHostByName** always compares the supplied host name with the local host name. If they match, the default implementation uses a private interface to probe the Microsoft TCP/IP stack in order to discover its local IP address. Thus, in order to be completely independent of the Microsoft TCP/IP stack, a name space provider *must* implement both **GETXBYYSP\_gethostbyname** and **GETXBYYSP\_WSAAsyncGetHostByName**.

## Windows Sockets 2 Identifiers

A Windows Sockets 2 clearinghouse has been established for service provider vendors to obtain unique identifiers for new address families, socket types, and protocols. FTP and world-wide web servers are used to supply current identifier/value mappings, and email is used to request allocation of new ones. At the time of this writing the world-wide web URL for the Windows Sockets 2 Identifier Clearinghouse is:

*<http://www.stardust.com/winsock/>*

## Data Transport Providers

The following sections apply only to data transport service providers and the data transport portion of the SPI.

### Transport Division of Responsibilities Between DLL and Service Providers

This section provides an overview of the division of responsibility between the Ws2\_32.dll and transport service providers.

#### Ws2\_32.dll Functionality for Transport

The major task of the data transport portion of the Ws2\_32.dll is to serve as a sort of traffic manager between service providers and applications. Consider several different service providers interacting with the same application. Each service provider interacts strictly with the Ws2\_32.dll. The Ws2\_32.dll takes care of:

- Selecting an appropriate service provider for creating sockets based on a protocol description.

- Forwarding application procedure calls involving a socket to the appropriate service provider that created or controls that socket. Service providers are unaware that any of this is happening.

They do not need to be concerned about the details of cooperating with one another or even the existence of other service providers. By abstracting the service providers into a consistent DLL interface and performing this automatic traffic routing function, the `Ws2_32.dll` allows applications to interact with a variety of providers without requiring the applications to be aware of the divisions between providers, where different providers are installed.

The `Ws2_32.dll` relies on the parameters of the API socket creation functions (**socket** and **WSASocket**) to determine which service provider to utilize. The selected transport service provider will be invoked through the **WSPSocket** function. In the case of the **socket** function, the `Ws2_32.dll` finds the first entry in the set of installed **WSAPROTOCOL\_INFOW** structures that matches the values supplied in the tuple formed by the (*address family, socket type, protocol*) parameters. To preserve backward compatibility, the `Ws2_32.dll` treats the value of zero for either *address family* or *socket type* as a wildcard value. The value of zero for *protocol* is not considered a wildcard value by the `Ws2_32.dll` unless such behavior is indicated for a particular protocol by having the `PFL_MATCHES_PROTOCOL_ZERO` flag set in the **WSAPROTOCOL\_INFOW** structure.

For the **WSASocket** function, if `NULL` is supplied for *lpProtocolInfo*, the behavior is exactly as just described for **socket**. If a **WSAPROTOCOL\_INFO** structure is referenced, however, the `Ws2_32.dll` does not perform any matching function but immediately relays the socket creation request to the transport service provider associated with the indicated **WSAPROTOCOL\_INFO** structure. The values for the (*address family, socket type, protocol*) tuple are supplied intact to the service provider in the **WSPSocket** function. Service providers are free to ignore or pay attention to the values of the (*address family, socket type, protocol*) parameters as is appropriate, but they must *not* indicate an error condition when the value of either *address family* or *socket type* is zero. In addition, service providers must not indicate an error condition when the manifest constant `FROM_PROTOCOL_INFO` is contained in any of the (*address family, socket type, protocol*) parameters. This value simply indicates that the application wishes to use the values found in the corresponding parameters of the **WSAPROTOCOL\_INFO** structure: (*iAddressFamily, iSocketType, iProtocol*).

As part of socket creation a service provider informs the `Ws2_32.dll` about the association between itself and the new socket by means of parameters passed to **WPUCreateSocketHandle** or **WPUModifyIFSHandle**. The `Ws2_32.dll` keeps track of this association between socket handles and particular service providers. Whenever an application interface function that refers to a socket handle is called, the `Ws2_32.dll` looks up the association and calls the corresponding service provider interface function of the appropriate service provider.

In addition to its major traffic routing service, the `Ws2_32.dll` provides a number of other services such as protocol enumeration, socket descriptor management (allocation, deallocation, and context value association) for nonfile-system service providers, blocking hook management on a per-thread basis, byte-swapping utilities, queuing of Asynchronous Procedure Calls (APCs) to facilitate invocation of I/O completion routines, and version negotiation between applications and the `Ws2_32.dll`, as well as between the `Ws2_32.dll` and service providers.

### Transport Service Provider Functionality

Service providers implement the actual transport protocol which includes such functions as setting up connections, transferring data, exercising flow control and error control, etc. The `Ws2_32.dll` has no knowledge about how requests to service providers are realized; this is up to the service provider implementation. The implementation of such functions may differ greatly from one provider to another. Service providers hide the implementation-specific details of how network operations are accomplished.

Transport service providers can be broadly divided into two categories: those whose socket descriptors are real file system handles (and are hereafter referred to as Installable File System (IFS) providers; the remainder are referred to as non-IFS providers. The `Ws2_32.dll` always passes the transport service provider's socket descriptor on up to the Windows Sockets application, so applications are free to take advantage of socket descriptors that are file system handles if they so choose.

To summarize: Service providers implement the low-level network-specific protocols. The `Ws2_32.dll` provides the medium-level traffic management that interconnects these transport protocols with applications. Applications in turn provide the policy of how these traffic streams and network-specific operations are used to accomplish the functions desired by the user.

### Transport Mapping Between API and SPI Functions

The Windows Sockets Transport SPI is similar to the Windows Sockets API in that all of the basic socket functions appear. When a new Windows Sockets 2 version of a function and the original Windows Sockets 1.1 version of a function both exist in the API, only the new version will show up in the SPI. For example:

- **connect** and **WSAConnect** both map to **WSPConnect**
- **accept** and **WSAAccept** map to **WSPAccept**
- **socket** and **WSASocket** map to **WSPSocket**

Other API functions that are collapsed into the enhanced versions in SPI include:

- **send**
- **sendto**
- **recv**
- **recvfrom**
- **ioctlsocket**

Support functions like **htonl**, **htons**, **ntohl**, and **ntohs** are implemented in the `Ws2_32.dll`, and are not passed down to service providers. The same holds true for the WSA versions of these functions.

Windows Sockets service provider enumeration and the blocking hook-related functions are realized in the `Ws2_32.dll`, thus **WSAEnumProtocols**, **WSAIsBlocking**, **WSASetBlockingHook**, and **WSAUnhookBlockingHook** do not appear as SPI functions.

Since error codes are returned along with SPI functions, equivalents of **WSAGetLastError** and **WSASetLastError** are not needed in the SPI.

The event object manipulation and wait functions including **WSACreateEvent**, **WSACloseEvent**, **WSASetEvent**, **WSAResetEvent**, and **WSAWaitForMultipleEvents** are mapped directly to native OS services and thus are not present in the SPI.

All the TCP/IP-specific name conversion and resolution functions in Windows Sockets 1.1 such as **getxbyy**, **WSAAsyncGetXByY** and **WSACancelAsyncRequest**, as well as **gethostname** are implemented within the `Ws2_32.dll` in terms of the new name resolution facilities. See Windows Sockets 1.1 Compatible Name Resolution for TCP/IP for details. Conversion functions such as **inet\_addr** and **inet\_ntoa** are implemented within the `Ws2_32.dll`.

## Function Extension Mechanism in the SPI

Since the Windows Sockets DLL itself is no longer supplied by each individual stack vendor, it is no longer possible for a stack vendor to offer extended functionality by just adding entry points to the Windows Sockets DLL. To overcome this limitation, Windows Sockets 2 takes advantage of the new **WSAIoctl** function to accommodate service providers who wish to offer provider-specific functionality extensions. This mechanism presupposes, of course, that an application is aware of a particular extension and understands both the semantics and syntax involved. Such information would typically be supplied by the service provider vendor.

In order to invoke an extension function, the application must first ask for a pointer to the desired function. This is done through the **WSAIoctl** function using the `SIO_GET_EXTENSION_FUNCTION_POINTER` command code. The input buffer to the **WSAIoctl** function contains an identifier for the desired extension function and the output buffer will contain the function pointer itself. The application can then invoke the extension function directly without passing through the `Ws2_32.dll`.

The identifiers assigned to extension functions are globally unique identifiers (GUIDs) that are allocated by service provider vendors. Vendors who create extension functions are urged to publish full details about the function including the syntax of the function prototype. This makes it possible for common and/or popular extension functions to be offered by more than one service provider. An application can obtain the function pointer and use the function without needing to know anything about the particular service provider that implements the function.

## Transport Configuration and Installation

In order for a transport protocol to be accessible through Windows Sockets it must be properly installed on the system and registered with Windows Sockets. When a transport service provider is installed by invoking a vendor's installation program, configuration information must be added to a configuration database to give the *Ws2\_32.dll* required information regarding the service provider. The *Ws2\_32.dll* exports an installation function, **WSCInstallProvider** for the vendor's installation program to supply the relevant information about the to-be-installed service provider, for example, the name and path to the service provider DLL and a list of **WSAPROTOCOL\_INFOW** structures that this provider can support. Symmetrically, the *Ws2\_32.dll* also provides a function, **WSCDeinstallProvider**, for a vendor's deinstallation program to remove all the relevant information from the configuration database maintained by the *Ws2\_32.dll*. The exact location and format of this configuration information is private to the *Ws2\_32.dll*, and can only be manipulated by the above-mentioned functions.

The order in which transport service providers are initially installed governs the order in which they are enumerated through **WSCEnumProtocols** at the service provider interface, or through **WSAEnumProtocols** at the application interface. More importantly, this order also governs the order in which protocols and service providers are considered when a client requests creation of a socket based on its address family, type, and protocol identifier. Windows Sockets 2 includes an applet called *Sporder.exe* that allows the catalog of installed protocols to be re-ordered interactively after protocols have already been installed. Windows Sockets 2 also includes an auxiliary DLL, *Sporder.dll*, that exports a procedural interface for re-ordering protocols. This procedural interface is described in *Service Provider Ordering*.

### Installing Layered Protocols and Protocol Chains

The **WSAPROTOCOL\_INFO** structure supplied with each protocol to be installed indicates whether the protocol is a base protocol, layered protocol, or protocol chain. The value of the *ProtocolChain.ChainLen* parameter is interpreted as shown in the following table.

Value	Meaning
0	Layered protocol.
1	Base protocol (or chain with only one component).
>1	Protocol chain.

Installation of protocol chains can only occur after successful installation of all of the constituent components (base protocols and layered protocols). The **WSAPROTOCOL\_INFOW** structure for a protocol chain uses the *ProtocolChain* parameter to describe the length of the chain and the identity of each component. The individual protocols that make up a chain are listed in order in the *ProtocolChain.ChainEntries* array, with the zeroth element of the array corresponding to the first layered provider. Protocols are identified by their *CatalogEntryID* values, which

are assigned by the `Ws2_32.dll` at protocol installation time, and can be found in the `WSAPROTOCOL_INFOW` structure for each protocol.

The values for the remaining parameters in the protocol chain's `WSAPROTOCOL_INFOW` structure should be chosen to reflect the attributes and identifiers that best characterize the protocol chain as a whole. When selecting these values, developers should bear in mind that communications over protocol chains can only occur when both endpoints have compatible protocol chains installed, and that applications must be able to recognize the corresponding `WSAPROTOCOL_INFO` structure.

When a base protocol is installed, it is not necessary to make any entries in the `ProtocolChain.ChainEntries` array. It is implicitly understood that the sole component of this chain is already identified in the `CatalogEntryID` parameter of the same `WSAPROTOCOL_INFO` structure. Also note that protocol chains may not include multiple instances of the same layered protocol.

## Name Resolution Providers

The following sections apply only to name resolution service providers and the name resolution portion of the SPI.

### Name Resolution Model for the SPI

In developing a protocol-independent client/server application, there are two basic requirements that exist with respect to name resolution and registration:

- The ability of the server half of the application (hereafter referred to as a service) to register its existence within (or become accessible to) one or more name spaces.
- The ability of the client application to find the service within a name space and obtain the required transport protocol and addressing information.

For those accustomed to developing TCP/IP based applications, this may seem to involve little more than looking up a host address and then using an agreed upon port number. Other networking schemes, however, allow the location of the service, the protocol used for the service, and other attributes to be discovered at run-time. To accommodate the broad diversity of capabilities found in existing name services, the Windows Sockets 2 interface adopts the model described below.

A *namespace* refers to some capability to associate (as a minimum) the protocol and addressing attributes of a network service with one or more human-friendly names. Many name spaces are currently in wide use including the Internet's Domain Name System (DNS), Netware Directory Services (NDS), X.500, etc. These name spaces vary widely in how they are organized and implemented. Some of their properties are particularly important to understand from the perspective of Windows Sockets name resolution.

## Types of Namespaces in the SPI

There are three different types of namespaces in which a service could be registered:

- Dynamic
- Static
- Persistent

Dynamic namespaces allow services to register with the name space on the fly, and for clients to discover the available services at run time. Dynamic name spaces frequently rely on broadcasts to indicate the continued availability of a network service. Examples of dynamic name spaces include the SAP name space used within a Netware environment and the NBP name space used by Appletalk®.

Static name spaces require all of the services to be registered ahead of time, that is, when the name space is created. The DNS is an example of a static name space. Although there is a programmatic way to resolve names, there is no programmatic way to register names.

Persistent name spaces allow services to register with the name space on the fly. Unlike dynamic name spaces however, persistent name spaces retain the registration information in nonvolatile storage where it remains until such time as the service requests that it be removed. Persistent name spaces are typified by directory services such as X.500 and the NDS (Netware Directory Service). These environments allow the adding, deleting, and modification of service properties. In addition, the service object representing the service within the directory service could have a variety of attributes associated with the service. The most important attribute for client applications is the service's addressing information.

## Namespace Organization in the SPI

Many namespaces are arranged hierarchically. Some, such as X.500 and NDS, allow unlimited nesting. Others allow services to be combined into a single level of hierarchy or group. This is typically referred to as a workgroup. When constructing a query, it is often necessary to establish a context point within a namespace hierarchy from which the search will begin.

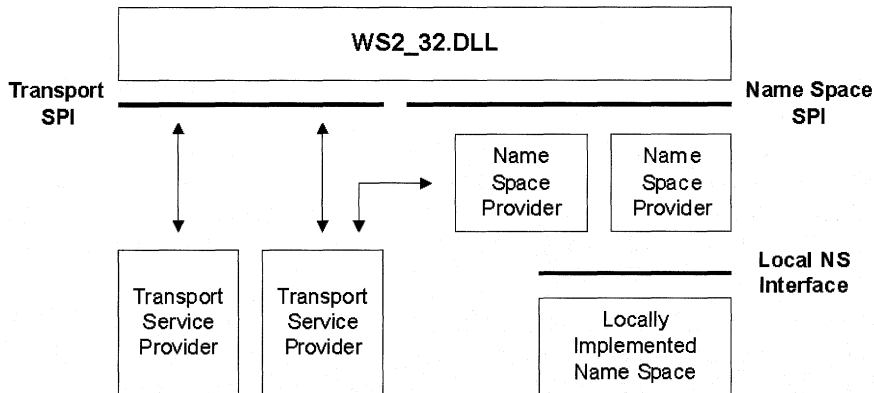
## Namespace Provider Architecture in the SPI

Naturally, the programmatic interfaces used to query the various types of name spaces and to register information within a namespace (if supported) differ widely. A *namespace provider* is a locally-resident piece of software that knows how to map between the Windows Sockets namespace SPI and some existing namespace (which could be implemented locally or accessed through the network). This is shown in Figure 10-3.

---

**Note** Note that it is possible for a given namespace, say DNS, to have more than one namespace provider installed on a given machine.

---



**Figure 10-3: Namespace Provider.**

As mentioned above, the generic term *service* refers to the server-half of a client/server application. In Windows Sockets, a service is associated with a *service class*, and each instance of a particular service has a *service name* which must be unique within the service class. Examples of service classes include FTP Server, SQL Server, XYZ Corp. Employee Info Server, etc.

As the example attempts to illustrate, some service classes are well known while others are very unique and specific to a particular vertical application. In either case, every service class is represented by both a class name and a class identifier. The class name does not necessarily need to be unique, but the class identifier must be. Globally Unique Identifiers (GUIDs) are used to represent service class IDs. For well-known services, class names, and class identifiers (GUIDs) have been pre-allocated, and macros are available to convert between, for example, TCP port numbers and the corresponding class identifier GUIDs. For other services, the developer chooses the class name and uses the `Uuidgen.exe` utility to generate a GUID for the class identifier.

The notion of a service class exists to allow a set of attributes to be established that are held in common by all instances of a particular service. This set of attributes is supplied to Windows Sockets at the time the service class is defined, and is referred to as the service class schema information. The `Ws2_32.dll` in turn relays this information to all active name space providers. When an instance of a service is installed and made available on a host machine, its service name is used to distinguish this particular instance from others that may be known to the name space.

Note that the installation of a service class only needs to occur on machines where the service executes, not on all of the clients which may utilize the service. Where possible, the `Ws2_32.dll` will provide service class schema information to a name space provider at the time an instance of a service is to be registered or a service query is initiated. The `Ws2_32.dll` does not, of course, store this information itself, but attempts to retrieve it from a name space provider that has indicated its ability to supply this data. Since there is no guarantee that the `Ws2_32.dll` will be able to supply the service class schema, name space providers that need this information must have a fallback mechanism to obtain it through name space-specific means.



The Internet's Domain Name System does not have a well-defined means to store service class schema information. As a result, DNS name space providers will only be able to accommodate well-known TCP/IP services for which a service class GUID has been preallocated. In practice, this is not a serious limitation since service class GUIDs have been preallocated for the entire set of TCP and UDP ports, and macros are available to retrieve the GUID associated with any TCP or UDP port. Thus, all of the familiar services such as ftp, telnet, whois, etc. are well supported. When querying for these services, by convention the host name of the target machine is the service instance name.

Continuing with our service class example, instance names of the ftp service may be "alder.intel.com" or "rhino.microsoft.com" while an instance of the XYZ Corp. Employee Info Server might be named "XYZ Corp. Employee Info Server Version 3.5". In the first two cases, the combination of the service class GUID for ftp and the machine name (supplied as the service instance name) uniquely identify the desired service. In the third case, the host name where the service resides can be discovered at service query time, so the service instance name does not need to include a host name.

## **Name Resolution Division of Responsibilities Between DLL and Service Providers**

The following paragraphs describe how the Ws2\_32.dll and the name space providers cooperate together to implement the name resolution services supported by the Windows Sockets 2 API.

### **Ws2\_32.dll Functionality for Name Resolution**

The Ws2\_32.dll manages the registration and demand loading of individual name space provider DLLs. It also is responsible for routing name space operations from a Windows Sockets 2 application to the appropriate set of name space providers. This mapping is governed by the value of name space and service provider identifier parameters that are found in individual API functions. As a general rule, when a specific name space provider is referenced, the operation is only routed to an identified provider. If the name space provider identifier is NULL but a particular name space is referenced, the operation is routed to all name space providers that support the identified name space. If the name space provider identifier is NULL and the name space identifier is given as NS\_ALL, then the operation is routed to all active name space providers.

As part of starting a query to one or more service providers, the Ws2\_32.dll allocates an object to keep track of the ongoing state of the query. An opaque handle representing this object is returned to the application that started the query. The application supplies this handle as a parameter each time it repetitively calls an application interface function to retrieve the next unit of data resulting from the query.

In response to these application interface procedure calls, the Ws2\_32.dll uses the information it stores in the object to make corresponding calls to the name space providers involved in the query. The Ws2\_32.dll updates the information in its object as

each successive application interface call occurs so that the corresponding calls to name space providers progress appropriately through all of the name space providers involved in the query.

### **Name Space Provider Functionality**

Each name space provider is responsible for mapping the set of functions appearing in the Windows Sockets 2 name resolution SPI to the appropriate transactions with the supported name space. In some cases, this is primarily a matter of mapping the SPI to whatever native interface exists for the name space. In others, the name space provider must conduct transactions with the name space provider over the network. Some name space providers will do this by making calls to the Windows Sockets API, others will use private interfaces to associated transport stacks.

### **Name Resolution Mapping Between API and SPI Functions**

The installation of service classes, registration of service instances and basic query operations all map fairly directly from the API to the SPI. The

**WSAGetServiceClassNameByClassId** function does not have a corresponding function in the SPI, as this function is implemented in `Ws2_32.dll` by making a call to **NSPGetServiceClassInfo**.

The helper functions **WSAAddressToString** and **WSAStringToAddress** are mapped to the corresponding functions in the transport API, as only a transport provider will necessarily know how to perform the translation on a **SOCKADDR** structure.

### **Name Resolution Configuration and Installation**

In order for a name space provider to be accessible through Windows Sockets it must be properly installed on the system and registered with Windows Sockets. When a name space provider is installed by invoking a vendor's installation program, configuration information must be added to a configuration database to give the `Ws2_32.dll` required information regarding the service provider. The `Ws2_32.dll` exports

**WSCInstallNameSpace** for the vendor's installation program to use. This function is used to supply relevant information about the to-be-installed service provider. This information includes:

- Provider Name - A string representing the provider for display in Control Panel
- Provider Version - The version of this provider
- Provider Path - A path name to the provider DLL
- Name Space - The name space supported by the provider
- Provider GUID - A unique, vendor-supplied number representing this provider/name space combination. This is used as a key for all subsequent references to this provider, and for uninstall. These values are created using the `Uuidgen.exe` utility.

- Stores all flag - a flag indicating whether this name space provider will be responsible for retaining all service class schema information for all service classes. If such a provider exists, the `Ws2_32.dll` does not need to query each individual name space provider for this information.

Symmetrically, the `Ws2_32.dll` also provides a function, **WSCUninstallNameSpace**, for a vendor's deinstallation program to remove all the relevant information from the configuration database. The exact location and format of this configuration information is private to the `Ws2_32.dll`, and can only be manipulated by the above-mentioned functions.

At any point, an NSP is considered to be either active or inactive, with this setting controlled through the **WSCEnableNSProvider** function. Name space providers that are inactive continue to show up when enumerated through **WSAEnumNameSpaceProviders**, but the `Ws2_32.dll` will not route any query or service registration operations to these providers. This capability can be useful in situations where more than one of the installed name space providers can support a given name space.

When multiple name space providers are referenced in a single API function, the order in which the order in which the queries and registration operations are routed to name space providers is unspecified. The order is unrelated to the order in which name space providers are installed. There are two ways to control which name space providers are used to resolve a name query. First, the name space configuration function, **WSCEnableNSProvider** can be used to enable and disable name spaces in a machine-wide, persistent way. Second, applications can direct an individual query to a particular provider by specifying that provider's identifying GUID as part of the query.

## Windows Sockets 2 Transport Provider Requirements

The following sections provide a description of each of the functional areas which transport service providers are required to implement. Where appropriate, implementation considerations and guidelines are also provided.

### Service Provider Activation

The following sections describe the sequence of events involved in bringing a transport service provider DLL into memory, initializing it, and eventually, de-initializing it.

#### Initialization

The `Ws2_32.dll` loads the service provider's interface DLL into the system by using the standard Microsoft® Windows® dynamic library loading mechanisms, and initializes it by calling **WSPStartup**. This is usually triggered by an application calling either **socket** or **WSASocket** in order to create a new socket that is to be associated with a service provider whose interface DLL is not currently loaded into memory. The path to each

service provider's interface DLL is stored by the `Ws2_32.dll` at the time the service provider is being installed. See section *Configuration and Installation* for more information.

Over time, different versions may exist for the Windows Sockets 2 DLLs, applications, and service providers. New versions may define new features, new parameters to data structures and bit parameters, etc. Version numbers therefore indicate how to interpret various data structures.

To allow optimal mixing and matching of different versions of applications, versions of the `Ws2_32.dll` itself, and versions of service providers by different vendors, the SPI provides a version negotiation mechanism for use between the `Ws2_32.dll` and the service providers. This version negotiation is handled by **WSPStartup**. Basically, the `Ws2_32.dll` passes to the service provider the highest version numbers with which it is compatible. The service provider compares this with its own supported range of version numbers. If these ranges overlap, the service provider returns a value within the overlapping portion of the range as the result of the negotiation. Usually, this should be the highest possible value. If the ranges do not overlap, the two parties are incompatible and the function returns an error.

**WSPStartup** must be called at least once by each client process, and may be called multiple times by `Ws2_32.dll` or other entities. A matching **WSPCleanup** must be called for each successful **WSPStartup** call. The service provider should maintain a reference count on a per-process basis. On each **WSPStartup** call, the caller may specify any version number supported by the SP DLL.

A service provider must store the pointer to the client's upcall dispatch table that is received as a **WSPStartup** parameter on a per-process basis. If a given process calls **WSPStartup** multiple times, the service provider must use only the most recently supplied dispatch table pointer.

As part of the service provider initialization process The `Ws2_32.dll` retrieves the service provider's dispatch table through the `lpProcTable` parameter in order to obtain entry points to the rest of the SPI functions specified in this document.

Using a dispatch table (as opposed to the usual DLL mechanisms for accessing entry points) serves two purposes:

- It is more convenient for the `Ws2_32.dll` since a single call can be made to discover the entire set of entry points.
- It enables layered service providers formed into protocol chains to operate more efficiently.

### Initializing Protocol Chains

At the time the **WSAPROTOCOL\_INFOW** structure for a protocol chain is installed, the path to the first layered provider in the chain is also specified. When a protocol chain is initialized, the `Ws2_32.dll` uses this path to load the provider DLL and then invokes **WSPStartup**. Since **WSPStartup** includes a pointer to the chain's **WSAPROTOCOL\_INFOW** structure as one of its parameters, layered providers can

determine what type of chain they are being initialized into, and the identity of the next lower layer in the chain. A layered provider would then in turn load the next protocol provider in the chain and initialize it with a call to **WSPStartup**, and so forth. Whenever the next lower layer is another layered provider, the chain's **WSAPROTOCOL\_INFOW** structure must be referenced in the **WSPStartup** call. When the next lower layer is a base protocol (signifying the end of the chain), the chain's **WSAPROTOCOL\_INFOW** structure is no longer propagated downward. Instead, the current layer must reference a **WSAPROTOCOL\_INFOW** structure that corresponds to the protocol that the base provider should use. Thus, the base provider has no notion of being involved in a protocol chain.

The dispatch table provided by any given layered provider will, in many instances, duplicate the entry points of an underlying provider. The layered provider would only insert its own entry points for functions that it needed to be directly involved in. Note, however, that it is imperative that a layered provider **not** modify the contents of the upcall table that it received when calling **WSPStartup** on the next lower layer in a protocol chain. These upcalls *must* be made directly to the Windows Sockets 2 DLL.

## Cleanup

The `Ws2_32.dll` (and layered protocols) will call **WSPCleanup** once for each invocation of **WSPStartup**. On each invocation, **WSPCleanup** should decrement a per-process reference counter, and when the counter reaches zero the service provider must prepare itself to be unloaded from memory. The first order of business is to finish transmitting any unsent data on sockets that are configured for a graceful close. Thereafter, any and all resources held by the provider are to be freed. The service provider must be left in a state where it can be immediately re-initialized by a call to **WSPStartup**.

## Error Reporting and Parameter Validation

The scheme for error reporting differs between the SPI and API interfaces. Windows Sockets service providers report errors along with the function returning, as opposed to the per-thread based approach utilized in the API. The `Ws2_32.dll` uses the service provider's per-function error code to update the per-thread error value that is obtained through the **WSAGetLastError** API function. Service providers are still required, however, to maintain the per-socket based error which can be retrieved through the **SO\_ERROR** socket option.

The `Ws2_32.dll` performs parameter validation only on function calls that are implemented entirely within itself. Service providers are responsible for performing all of their own parameter validation.

## Byte Ordering Assumptions

A service provider should treat all **SOCKADDR** components exclusive of the address family parameter as if they are in the network byte order, and the address family parameter as in the local machine's byte order. It is the Windows Sockets application's

responsibility to make sure that addresses contained in **SOCKADDR** structures are properly arranged. The Windows Sockets API provides a number of conversion routines to simplify this task. Currently these routines understand conversion between the local host's natural byte order and either big-endian or little-endian network byte ordering. The Windows Sockets architecture can support other byte ordering schemes in the future.

## Socket Creation and Descriptor Management

The following sections describe aspects of creating new sockets and the allocation of socket descriptors.

### Descriptor Allocation

While Windows Sockets service providers are encouraged to implement sockets as installable file system (IFS) objects, the Windows Sockets architecture also accommodates service providers whose socket handles are not IFS objects. Providers with IFS handles indicate this through the **XP1\_IFS\_HANDLES** attribute bit in the **WSAPROTOCOL\_INFOW** structure. (Note: the **XP1\_IFS\_HANDLES** attribute bit was not included in release 2.0.8 of the API specification, but has since been added through the errata mechanism.) Windows Sockets SPI clients may take advantage of providers whose socket descriptors are IFS handles by using these descriptors with standard Win32 I/O facilities, such as **ReadFile** and **WriteFile**.

Whenever an IFS provider creates a new socket descriptor, it is *mandatory* that the provider call **WPUModifyIFSHandle** prior to supplying the new handle to a Windows Sockets SPI client. This function takes a provider identifier and a proposed IFS handle from the provider as input and returns a (possibly) modified handle. The IFS provider must supply only the modified handle to its client, and all requests from the client will reference only this modified handle. The modified handle is guaranteed to be indistinguishable from the proposed handle as far as the operating system is concerned. Thus in most instances, the service provider will simply choose to use only the modified handle in all of its internal processing. The purpose of this modification function is to allow the **Ws2\_32.dll** to greatly streamline the process of identifying the service provider associated with a given socket.

Providers that do not return IFS handles *must* obtain a valid handle from the **Ws2\_32.dll** via the **WPUCreateSocketHandle** call. The nonIFS provider must offer only a Windows Sockets 2.dll-supplied handle to its client, and all requests from the client will reference only these handles. As a convenience to service provider implementers, one of the input parameters supplied by a provider in **WPUCreateSocketHandle** is a **DWORD** context value. The **Ws2\_32.dll** associates this context value with the allocated socket handle and allows a service provider to retrieve the context value at any time through the **WPUQuerySocketHandleContext** call. A typical use for this context value would be to store a pointer to a service provider maintained data structure used to store socket state information.

## Socket Attribute Flags and Modes

There are several socket attributes which can be indicated through the *flags* parameter in **WSPSocket**. The `WSA_FLAG_OVERLAPPED` flag indicates that a socket will be used for overlapped I/O operations. Support of this attribute is mandatory for all service providers. See *Overlapped I/O* for more information. Note that creating a socket with the overlapped attribute has no impact on whether a socket is currently in the blocking or nonblocking mode. Sockets created with the overlapped attribute may be used to perform overlapped I/O, and doing so does not change the blocking mode of a socket. Since overlapped I/O operations do not block, the blocking mode of a socket is irrelevant for these operations.

Four additional attribute flags are used when creating sockets that are to be used for multipoint and/or multicast operations, and support for these attributes is optional. Providers that support multipoint attributes indicate this through the `XP1_SUPPORT_MULTIPOINT` bit in their respective **WSAPROTOCOL\_INFOW** structures. See **WSPSocket** and section *Protocol-Independent Multicast and Multipoint in the API* for the definition and usage of each of these flags. Only sockets that are created with multipoint-related attributes can be used in the **WSPJoinLeaf** function for creating multipoint sessions.

A socket is in one of two modes, blocking and nonblocking, at any time. Sockets are created in the blocking mode by default, and can be changed to nonblocking mode by calling **WSPAsyncSelect**, **WSPEventSelect**, or **WSPIoctl**. A socket can be switched back to blocking mode by using **WSPIoctl** if no **WSPAsyncSelect** or **WSPEventSelect** is active. The mode for a socket only affects those functions which may block, and does not affect overlapped I/O operations. See section *Blocking Operations* for more information.

## Closing Sockets

**WSPCloseSocket** releases the socket descriptor so that any pending operations in any threads of the same process will be aborted, and any further reference to it will fail. Note that a reference count should be employed for shared sockets, and only if this is the last reference to an underlying socket, should the information associated with this socket be discarded, provided graceful close is not requested (that is, `SO_DONTLINGER` is not set). In the case of `SO_DONTLINGER` being set, any data queued for transmission will be sent, if possible, before information associated with the socket is released. See **WSPCloseSocket** for more information.

For nonIFS service providers, **WPUCloseSocketHandle** must be invoked to release the socket handle back to the Windows Sockets 2 DLL.

## Blocking Operations

The notion of blocking in a Windows environment has historically been a very important one. In Windows Sockets 1.1 environments, blocking calls were discouraged since they tend to disable ongoing interaction with the Windowing system, and since they employ a

pseudo blocking technique which, for a variety of reasons, does not always work as intended. However, in preemptively scheduled Win32 environments such as Windows 95/98® and Windows NT®/Windows® 2000, blocking calls make much more sense, can be implemented by native operating system services, and are in fact a generally preferred mechanism. The Windows Sockets 2 API no longer supports pseudo blocking, but because the Windows Sockets 1.1 compatibility shims must continue to emulate this behavior, service providers must support this as described below.

## Pseudo vs. True Blocking

In 16 Windows environments, true blocking is not supported by the OS, thus a blocking operation that cannot be completed immediately is handled as follows:

- The service provider initiates the operation, and then enters a loop during which it dispatches any Windows messages (yielding the processor to another thread if necessary)
- It then checks for the completion of the Windows Sockets function.
- If the function has completed, or if **WSPCancelBlockingCall** has been invoked, the loop is terminated and the blocking function completes with an appropriate result.

This is what is meant by the term pseudo blocking, and the loop referred to above is known as the default blocking hook.

## Blocking Hook

Although this mechanism is sufficient for simple applications, it cannot support the complex message-dispatching requirements of more advanced applications such as those using the Multiple Document Interface (MDI) model. For such applications, a thread-specific blocking hook may be installed by the application. This will be called by the service provider instead of the default blocking hook described above. A service provider must retrieve a pointer to the per-thread blocking hook from the `Ws2_32.dll` by calling **WPUQueryBlockingCallback**. If the application has not installed its own blocking hook a pointer to the default blocking hook function will be returned.

A Windows Sockets service provider cannot assume that an application-supplied blocking hook allows message processing to continue as the default blocking hook does. Some applications cannot tolerate the possibility of reentrant messages while a blocking operation is outstanding. Such an application's blocking hook function would simply return `FALSE`. If a service provider depends on messages for its internal operation, it may execute **PeekMessage**(`hMyWnd...`) before executing the application's blocking hook so that it can get its own messages without affecting the rest of the system.

There is no default blocking hook installed in preemptive multithreaded versions of Windows. This is because other processes will not be blocked if a single application is waiting for an operation to complete (and hence not calling **PeekMessage** or **GetMessage** which causes the application to yield the processor in nonpreemptive Windows). When the service provider calls **WPUQueryBlockingCallback** a null pointer will be returned indicating that the provider is to use native OS blocking functions.



However, in order to preserve backward compatibility, an application-supplied blocking hook can still be installed on a per-thread basis in 32 bit versions of Windows.

The Windows Sockets service provider calls the blocking hook only if all of the following are true: the routine is one which is defined as being able to block, the specified socket is a blocking socket, and the request cannot be completed immediately. If only nonblocking sockets and **WSPAsyncSelect/WSPEventSelect** instead of **WSPSelect** are used, then the blocking hook will never be called.

---

**Important** If, during the time pseudo blocking is being used to block a thread, a Windows message is received for the thread, there is a risk that the thread will attempt to issue another Windows Sockets call. Because of the difficulty of managing this condition safely, the Windows Sockets 1.1 specification disallowed this behavior. It is not permissible for a given thread to make multiple nested Windows Sockets function calls. Only one outstanding function call is allowed for a particular thread. Any nested Windows Sockets function calls fail with the error **WSAEINPROGRESS**. It should be emphasized that this restriction applies to both blocking and nonblocking operations, but only in Windows Sockets 1.1 environments. There are a few exceptions to this rule, including two functions that allow an application to determine whether a pseudo blocking operation is in fact in progress, and to cancel such an operation if need be. These are described below.

---

## Canceling Blocking Operations

A thread may, at any time, call **WSAIsBlocking** to determine whether or not a blocking call is currently in progress. (This function is implemented within the Windows Sockets 1.1 compatibility shims and hence has no SPI counterpart.) Clearly this is only possible when pseudo blocking, as opposed to true blocking, is being employed by the service provider. When necessary, **WSPCancelBlockingCall** may be called at any time to cancel any current pseudo blocking operation.

## Event Objects in the Windows Sockets 2 SPI

Event objects are introduced in Windows Sockets 2 as a general synchronization mechanism between Windows Sockets 2 service providers and applications. They are used for a number of purposes including indicating the completion of overlapped operations and the occurrence of one or more network events.

### Creating Event Objects

The **Ws2\_32.dll** provides facilities for event object creation to both applications and service providers, although in most instances event objects will be created by applications. Event object services are made available to Windows Sockets service providers through **WPUCreateEvent** simply as a convenience mechanism for any internal processing that may benefit from same. Note that the event object handle is only valid in the context of the calling process. In Win32 environments the realization of event objects is through the native event services provided by the operating system.

## Using Event Objects

Windows Sockets event objects are fairly simple constructs which can be created and closed, set and cleared, waited upon and polled. The general usage model is for clients to create an event object and supply the handle as a parameter (or as a component of a parameter structure) in functions such as **WSPSend** and **WSPEventSelect**. When the nominated condition has occurred, the service provider uses the handle to set the event object by calling **WPUSetEvent**. Meanwhile, the Windows Sockets SPI client may either block and wait or poll until the event object becomes set (or as it is sometimes called: signaled). The client may subsequently clear or reset the event object and use it again.

## Destroying Event Objects

The entity which creates an event object (application or service provider) is responsible for destroying it when it is no longer required. Service providers do this through **WPUCloseEvent**.

## Notification of Network Events

One of the most important responsibilities of a data transport service provider is that of providing indications to the client when certain network events have occurred. The list of defined network events consists of the following:

- **FD\_CONNECT** - A connection to a remote host or to a multicast session has been completed
- **FD\_ACCEPT** - A remote host is making a connection request
- **FD\_READ** - Network data has arrived which is available to be read
- **FD\_WRITE** - Space has become available in the service provider's buffers so that additional data may now be sent
- **FD\_OOB** - Out of band data is available to be read
- **FD\_CLOSE** - The remote host has closed down the connection
- **FD\_QOS** - A change has occurred in negotiated QOS levels
- **FD\_GROUP\_QOS** - Reserved.
- **FD\_ROUTING\_INTERFACE\_CHANGE** – A local interface that should be used to reach the destination specified in **SIO\_ROUTING\_INTERFACE\_CHANGE** IOCLT has changed
- **FD\_ADDRESS\_LIST\_CHANGE** – The list of local addresses to which application can bind has changed

The set of network events enumerated above is sometimes referred to as the **FD\_XXX** events. Indication of the occurrence of one or more of such network events may be given in a number of ways depending on how the client requests for notification.

## Selects

In the original BSD sockets interface the **select** call was the standard (and only) means to obtain network event indications. For each socket, information on read, write, or error status can be polled or waited on. See **WSPSelect** for more information. Note that network event FD\_QOS and cannot be obtained by this approach.

## Windows Messages

Windows Sockets 1.1 introduced the async-select mechanism in order to provide network event indications in a manner that did not involve either polling or blocking. The **WSPAsyncSelect** function is used to register an interest in one or more network events as listed above, and supply a window handle to be used for notification. When a nominated network event occurs, a client-specified Windows message is sent to the indicated window. The service provider must use the **WPUPostMessage** function to accomplish this.

In Win32 environments, this may not be the most efficient notification mechanism, and is inconvenient for daemons and services that don't want to open any windows. The event object signaling mechanism described below is introduced to solve this problem.

## Event Object Signaling

**WSPEventSelect** behaves exactly like **WSPAsyncSelect** except that, rather than cause a Windows message to be sent on the occurrence of any nominated FD\_XXX network event (for example, FD\_READ, FD\_WRITE, etc.), a designated event object is set.

Also, the fact that a particular FD\_XXX network event has occurred is remembered by the service provider. This is needed since the occurrence of any and all nominated network events will result in a single event object becoming signaled. A call to **WSPEnumNetworkEvents** causes the current contents of the network event memory to be copied to a client-supplied buffer and the network event memory to be cleared. The client may also designate a particular event object to be cleared atomically along with the network event memory.

## Socket Groups in the Windows Sockets 2 SPI

All use of Socket Groups is reserved.

### Socket Group Operations

All use of Socket Groups is reserved.

### Required Socket Grouping Behavior

All use of Socket Groups is reserved.

### Recommended Socket Grouping Behavior

All use of Socket Groups is reserved.

## Quality of Service in the Windows Sockets 2 SPI

Quality of Service is implemented in Windows 2000 through various Windows 2000 QOS components. For details and implementation guidelines, see the *QOS chapters* later in this book.

## Socket Connections on Connection-Oriented Protocols

The following paragraphs describe the semantics applicable to socket connections over connection-oriented protocols.

### Binding to a Local Address

Before a socket can be used to set up a connection or receive a connection request, it needs to be bound to a local address. This could be done explicitly by calling **WSPBind**, or implicitly by **WSPConnect** if the socket is unbound when this function is called.

### Protocol Basics: Listen, Connect, Accept

The basic operations involved with establishing a socket connection can be most conveniently explained in terms of the client-server paradigm.

The server side will first create a socket, bind it to a well known local address (so that the client can find it), and put the socket in listening mode, through **WSPListen**, in order to prepare for any incoming connection requests and to specify the length of the connection backlog queue. Service providers hold pending connection requests in a backlog queue until they are acted upon by the server or are withdrawn (due to time-out) by the client. A service provider may silently ignore requests to extend the size of the backlog queue beyond a provider-defined upper limit.

At this point, if a blocking socket is being used, the server side may immediately call **WSPAccept** which will block until a connection request is pending. Conversely, the server may also use one of the network event indication mechanisms discussed previously to arrange for notification of incoming connection requests. Depending on the notification mechanism selected, the provider will either issue a Windows message or signal an event object when connection requests arrive. See section *Notification of Network Events* for how to register for the `FD_ACCEPT` network event.

The client side will create an appropriate socket, and initiate the connection by calling **WSPConnect**, specifying the known server address. Clients usually do not perform an explicit **bind** operation prior to initiating a connection, allowing the service provider to perform an implicit bind on their behalf. If the socket is in blocking mode, **WSPConnect** will block until the server has received and acted upon the connection request (or until a time-out occurs). Otherwise, the client should use one of the network event indication mechanisms discussed previously to arrange for notification of a new connection being established. Depending on the notification mechanism selected, the provider will indicate this either through a Windows message or by signaling an event object.

When the server side invokes **WSPAccept**, the service provider calls the application-supplied condition function, using function parameters to pass into the server information from the top entry in the connection request backlog queue. This information includes such things as address of connecting host, any available user data, and any available QOS information. Using this information the server's condition function determines whether to accept the request, reject the request, or defer making a decision until later. This decision is indicated through the return value of the condition function. See section *Notification of Network Events* for how to register for the FD\_CONNECT network event.

If the server decides to accept the request, the provider must create a new socket with all of the same attributes and event registrations as the listening socket. The original socket remains in the listening state so that subsequent connection requests can be received. Through output parameters of the condition function, the server may also supply any response user data and assign QOS parameters (assuming that these operations are supported by the service provider).

## Determining Local and Remote Names

**WSPGetSockName** is used to retrieve the local address for bound sockets. This is especially useful when a **WSPConnect** call has been made without doing a **WSPBind** first; **WSPGetSockName** provides the only means to determine the local association which has been set implicitly by the provider.

After a connection has been set up, **WSPGetPeerName** can be used to determine the address of the peer to which a socket is connected.

## Enhanced Functionality at Connect Time

Windows Sockets 2 offers an expanded set of operations that can occur coincident to establishing a socket connection. The service provider requirements for implementing these features are described below.

### Conditional Acceptance

As described previously, **WSPAccept** invokes a client-supplied condition function that uses input parameters to supply information about the pending connection request. This information can be used by the client to accept or reject a connection request based on caller information such as caller identifier, QOS, etc. If the condition function returns CF\_ACCEPT, a new socket is created with the same properties as the listening socket, and a handle to the new socket is returned. If the condition function returns CF\_REJECT, the connection request should be rejected. If the condition function returns CF\_DEFER, the accept/reject decision cannot be made immediately, and the service provider must leave the connection request on the backlog queue. The client must call **WSPAccept** again, when it is ready to make a decision, and arrange for the condition function to return either CF\_ACCEPT or CF\_REJECT. While a deferred connection request is at the top of the backlog queue, the service provider does not issue any further indications for pending connection requests.

## Exchanging User Data at Connect Time

Some protocols allow a small amount of user data to be exchanged at connect time. If such data has been received from the connecting host, it is placed in a service provider buffer, and a pointer to this buffer along with a length value are supplied to the Windows Sockets SPI client through input parameters to the **WSPAccept** condition function. If the Windows Sockets SPI client has response data to return to the connecting host, it may copy this into a buffer that is supplied by the service provider. A pointer to this buffer and an integer indicating buffer size are also supplied as condition function input parameters (if supported by the protocol).

## Establishing Socket Groups

All use of Group Sockets is reserved.

## Connection Shutdown

The following paragraphs describe operations incident to shutting down an established socket connection.

### Initiating Shutdown Sequence

A socket connection can be taken down in one of several ways. **WSPShutdown** (with *how* equal to `SD_SEND` or `SD_BOTH`), and **WSPSendDisconnect** may be used to initiate a graceful connection shutdown. **WSPCloseSocket** can be used to initiate either a graceful or abortive shutdown, depending on the linger options invoked by the closing a socket. See below for more information about graceful and abortive shutdowns, and linger options.

### Indicating Remote Shutdown

Service providers indicate connection teardown that is initiated by the remote party through the `FD_CLOSE` network event. Graceful shutdown is also be indicated through **WSPRecv** when the number of bytes read is 0 for byte-stream protocols, or through a return error code of `WSAEDISCON` for message-oriented protocols. In any case, a **WSPRecv** return error code of `WSAECONNRESET` indicates an abortive shutdown.

### Exchanging User Data at Shutdown Time

At connection teardown time, it is also possible (for protocols that support this) to exchange user data between the endpoints. The end that initiates the teardown can call **WSPSendDisconnect** to indicate that no more data is to be sent and cause the connection teardown sequence to be initiated. For certain protocols, part of this teardown sequence is the delivery of disconnect data from the teardown initiator. After receiving notice that the remote end has initiated the teardown sequence (typically through the `FD_CLOSE` indication), the **WSPRecvDisconnect** function may be called to receive the disconnect data (if any).

To illustrate how disconnect data might be used, consider the following scenario. The client half of a client/server application is responsible for terminating a socket connection. Coincident with the termination it provides (through disconnect data) the total number of transactions it processed with the server. The server in turn responds with the cumulative grand total of transactions that it has processed with all clients. The sequence of calls and indications might occur as shown in the following table.

Client side	Server side
(1) Invokes <b>WSPSendDisconnect</b> to conclude session and supply transaction total.	(2) Gets <code>FD_CLOSE</code> , or <b>WSPRecv</b> with a return value of zero or <code>WSAEDISCON</code> indicating graceful shutdown in progress.
	(3) Invokes <b>WSPRecvDisconnect</b> to get client's transaction total.
	(4) Computes cumulative grand total of all transactions.
	(5) Invokes <b>WSPSendDisconnect</b> to transmit grand total.
(6) Receives <code>FD_CLOSE</code> indication.	(5a) Invokes <code>WSPClosesocket</code>
(7) Invokes <b>WSPRecvDisconnect</b> to receive and store cumulative grand total of transactions.	
	(8) Invokes <code>WSPClosesocket</code>

Step (5a) must follow step (5), but has no timing relationship with steps (6), (7), or (8).

### Graceful Shutdown, Linger Options, and Socket Closure in the SPI

It is important to distinguish between shutting down a socket connection and closing a socket. Shutting down a socket connection involves an exchange of protocol messages between the two endpoints, which is hereafter referred to as a shutdown sequence. Two general classes of shutdown sequences are defined: graceful and abortive. In a graceful shutdown sequence, any data that has been queued but not yet transmitted can be sent prior to the connection being closed. In an abortive shutdown, any unsent data is lost. The occurrence of a shutdown sequence (graceful or abortive) can also be used to provide an `FD_CLOSE` indication to the associated applications signifying that a shutdown is in progress. Closing a socket, on the other hand, causes the socket handle to become deallocated so that the application can no longer reference or use the socket in any manner.

In Windows Sockets, both the **WSPShutdown** function, and the **WSPSendDisconnect** function can be used to initiate a shutdown sequence, while the **WSPCloseSocket** function is used to deallocate socket handles and free up any associated resources.

Some amount of confusion arises, however, from the fact that the **WSPCloseSocket** function will implicitly cause a shutdown sequence to occur if it has not already happened. In fact, it has become a rather common programming practice to rely on this feature and use **WSPCloseSocket** to both initiate the shutdown sequence and deallocate the socket handle.

To facilitate this usage, the sockets interface provides for controls through the socket option mechanism that allows the programmer to indicate whether the implicit shutdown sequence should be graceful or abortive, and also whether the **WSPCloseSocket** function should linger (that is, not complete immediately) to allow time for a graceful shutdown sequence to complete.

By establishing appropriate values for the socket options `SO_LINGER` and `SO_DONTLINGER`, the following types of behavior can be obtained with the **WSPCloseSocket** function.

- Abortive shutdown sequence, immediate return from **WSPCloseSocket**.
- Graceful shutdown, delay return until either shutdown sequence completes or a specified time interval elapses. If the time interval expires before the graceful shutdown sequence completes, an abortive shutdown sequence occurs and **WSPCloseSocket** returns.
- Graceful shutdown, return immediately, and allow the shutdown sequence to complete in the background. This is the default behavior. Note, however, that the application has no way of knowing when (or whether) the graceful shutdown sequence completes.

One technique that can be used to minimize the chance of problems occurring during connection teardown is not to rely on an implicit shutdown being initiated by **WSPCloseSocket**. Instead, one of the two explicit shutdown functions (**WSPShutdown** or **WSPSendDisconnect**) are used. This in turn will cause an `FD_CLOSE` indication to be received by the peer application indicating that all pending data has been received. To illustrate this, the following table shows the functions that would be invoked by the client and server components of an application, where the client is responsible for initiating a graceful shutdown.

#### Client side

#### Server side

(1) Invokes **WSPShutdown** (`s`, `SD_SEND`) to signal end of session and that client has no more data to send.

(2) Receives `FD_CLOSE`, indicating graceful shutdown in progress and that all data has been received.

(3) Sends any remaining response data.

*(continued)*



(continued)

### Client side

### Server side

(5a) Gets `FD_READ` and invoke `recv` to get any response data sent by server.

(4) Invokes **WSPShutdown**(*s*, *SD\_SEND*) to indicate server has no more data to send.

(5) Receives `FD_CLOSE` indication.

(4a) Invokes **WSPCloseSocket**

(6) Invokes **WSPCloseSocket**

The timing sequence is maintained from step (1) to step (6) between the client and the server, except for steps (4a) and (5a) which only have local timing significance in the sense that step (5) follows step (5a) on the client side while step (4a) follows step (4) on the server side, with no timing relationship with the remote party.

## Socket Connections on Connectionless Protocols

The following sections describe the semantics of using connect operations on connectionless protocols such as UDP and IPX.

### Connecting to a Default Peer

For a socket bound to a connectionless protocol, the operation performed by **WSPConnect** is merely to establish a default destination address so that the socket may be used with subsequent connection-oriented send and receive operations (**WSPSend** and **WSPRecv**). Any datagrams received from an address other than the destination address specified will be discarded.

### Reconnecting and Disconnecting

The default destination may be changed by simply calling **WSPConnect** again, even if the socket is already connected. Any datagrams queued for receipt are discarded if the new address is different from the address specified in a previous **WSPConnect**.

If the address supplied is all zeroes, the socket will be disconnected—the default remote address will be indeterminate, so **WSPSend** and **WSPRecv** calls will return the error code `WSAENOTCONN`, although **WSPSendTo** and **WSPRecvFrom** may still be used.

### Using Sendto While Connected

**WSPSendTo** will always deliver the data to the specified address, even though a designated peer for the sending socket has been established in **WSPConnect**.

## Socket I/O

There are three primary ways of doing I/O in Windows Sockets 2:

- Blocking I/O.
- Nonblocking I/O along with asynchronous notification of network events.
- Overlapped I/O with completion indication.

We describe each method in the following sections. Blocking I/O is the default behavior, nonblocking mode can be used on any socket that is placed into nonblocking mode, and overlapped I/O can only occur on sockets that are created with the overlapped attribute. It is also interesting to note that the two calls for sending: **WSPSend** and **WSPSendTo** and the two calls for receiving: **WSPRecv** and **WSPRecvFrom** each implement all three methods of I/O. Service providers determine how to perform the I/O operation based on socket modes and attributes and the input parameter values.

## Blocking Input/Output

The simplest form of I/O in Windows Sockets 2 is blocking I/O. As mentioned in section *Socket Attribute Flags and Modes*, sockets are created in blocking mode by default. Any I/O operation with a blocking socket will not return until the operation has been fully completed. Thus, any thread can only execute one I/O operation at a time. For example, if a thread issues a receive operation and no data is currently available, the thread will block until data becomes available and is placed into the thread's buffer. Although this is simple, it is not necessarily the most efficient way to do I/O in all versions of Windows. In Win16 environments for example, blocking is strongly discouraged due to the nonpreemptive nature of the operating system. (see section *Pseudo vs. True Blocking* for more information).

## Nonblocking Input/Output

If a socket is in a nonblocking mode, any I/O operation must either complete immediately or return error code **WSAEWOULDBLOCK** indicating that the operation cannot be finished right away. In the latter case, a mechanism is needed to discover when it is appropriate to try the operation again with the expectation that the operation will succeed. A set of network events has been defined for this purpose. These events can be polled or waited on by using **WSPSelect**, or they can be registered for asynchronous delivery by calling **WSPAsyncSelect** or **WSPEventSelect**. (see section *Notification of Network Events* for more information)

## Overlapped Input/Output

Windows Sockets 2 introduces overlapped I/O and requires that all transport providers support this capability. Overlapped I/O can be performed only on sockets that were created through the **WSPSocket** function with the **WSA\_FLAG\_OVERLAPPED** flag set, and follow the model established in Win32.

For receiving, a client uses **WSPRecv** or **WSPRecvFrom** to supply buffers into which data is to be received. If one or more buffers are posted prior to the time when data has been received by the network, it is possible that data will be placed into the user's buffers immediately as it arrives and thereby avoid the copy operation that would otherwise occur. If data is already present when receive buffers are posted, it is copied immediately into the user's buffers. If data arrives when no receive buffers have been posted by the application, the service provider resorts to the synchronous style of operation where the incoming data is buffered internally until such time as the client issues a receive call and thereby supplies a buffer into which the data may be copied.

An exception to this would be if the application used **WSPSetSockOpt** to set the size of the receive buffer to zero. In this instance, reliable protocols would only allow data to be received when application buffers had been posted, and data on unreliable protocols would be lost.

On the sending side, clients use **WSPSend** or **WSPSendTo** to supply pointers to filled buffers and then agree not to disturb the buffers in any way until the network has consumed the buffer's contents.

Overlapped send and receive calls return immediately. A return value of zero indicates that the I/O operation completed immediately and that the corresponding completion indication has already occurred. That is, the associated event object has been signaled, or the completion routine has been queued through **WPUQueueApc**. A return value of **SOCKET\_ERROR** coupled with an error code of **WSA\_IO\_PENDING** indicates that the overlapped operation has been successfully initiated and that a subsequent indication will be provided when send buffers have been consumed or when receive buffers are filled. Any other error code indicates that the overlapped operation was not successfully initiated and that no completion indication will be forthcoming.

Both send and receive operations can be overlapped. The receive functions may be invoked multiple times to post receive buffers in preparation for incoming data, and the send functions may be invoked multiple times to queue up multiple buffers to be sent. Note that while a series of overlapped send buffers will be sent in the order supplied, the corresponding completion indications may occur in a different order. Likewise, on the receiving side, buffers will be filled in the order they are supplied, but completion indications may occur in a different order.

The deferred completion feature of overlapped I/O is also available for **WSPIoctl**.

### Delivering Completion Indications

Service providers have two ways to indicate overlapped completion: setting a client-specified event object, or invoking a client-specified completion routine. In both cases a data structure, **WSAOVERLAPPED**, is associated with each overlapped operation. This structure is allocated by the client and used by it to indicate which event object (if any) is to be set when completion occurs. The **WSAOVERLAPPED** structure may be used by the service provider as a place to store a handle to the results (for example, number of bytes transferred, updated flags, error codes, etc.) for a particular overlapped operation. To obtain these results clients must invoke **WSPGetOverlappedResult**, passing in a pointer to the corresponding overlapped structure.

If event based completion indication is selected for a particular overlapped I/O request, the **WSPGetOverlappedResult** routine may itself be used by clients to either poll or wait for completion of the overlapped operation. If completion-routine-based completion indication is selected for a particular overlapped I/O request, only the polling option of **WSPGetOverlappedResult** is available. A client may also use other means to wait (such as using **WSAWaitForMultipleEvents**) until the corresponding event object has

been signaled or the specified completion routine has been invoked by the service provider. Once completion has been indicated, the client may invoke **WSPGetOverlappedResult**, with the expectation that the call will complete immediately.

### Invoking Socket I/O Completion Routines

If the *lpCompletionRoutine* parameter to an overlapped operation is not NULL, it is the service provider's responsibility to arrange for invocation of the client-specified completion routine when the overlapped operation completes. Since the completion routine must be executed in the context of the same thread that initiated the overlapped operation, it cannot be invoked directly from the service provider. The *Ws2\_32.dll* offers an asynchronous procedure call (APC) mechanism to facilitate invocation of completion routines.

A service provider arranges for a function to be executed in the proper thread by calling **WPUQueueApc**. This function can be called from any process and thread context, even a context different from the thread and process that was used to initiate the overlapped operation.

**WPUQueueApc** takes as input parameters a pointer to a **WSATHREADID** structure, a pointer to an APC function to be invoked, and a 32-bit context value that is subsequently passed to the APC function. Service providers are always supplied with a pointer to the proper **WSATHREADID** structure through the *lpThreadId* parameter to the overlapped function. The provider should store the **WSATHREADID** structure locally and supply a pointer to this copy of the **WSATHREADID** structure as an input parameter to **WPUQueueApc**. Once the **WPUQueueApc** function returns, the provider can dispose of its copy of the **WSATHREADID**.

The procedure **WPUQueueApc** simply enqueues sufficient information to call the indicated APC function with the given parameters, but does not call it. When the target thread enters an alertable wait state, this information is dequeued and a call is made to the APC function in that target thread and process context. Because the APC mechanism supports only a single 32-bit context value, the APC function cannot itself be the client-specified completion routine, which involves more parameters. The service provider must instead supply a pointer to its own APC function which uses the supplied context value to access the needed result information for the overlapped operation, and then invokes the client-specified completion routine.

For service providers where a user-mode component implements overlapped I/O, a typical usage of the APC mechanism is as follows:

- When the I/O operation completes, the provider allocates a small buffer and packs it with a pointer to the client-supplied completion procedure and parameter values to pass to the procedure.
- It queues an APC, specifying the pointer to the buffer as the context value and its own intermediate procedure as the target procedure.
- When the target thread eventually enters alertable wait state, the service provider's intermediate procedure is called in the proper thread context.

- The intermediate procedure simply unpacks parameters, deallocates the buffer, and calls the client-supplied completion procedure.
- For service providers where a kernel-mode component implements overlapped I/O, a typical implementation is similar, except that the implementation would use standard kernel interfaces to enqueue the APC.

Description of the relevant kernel interfaces is outside the scope of the Windows Sockets 2 specification.

---

**Important** Service providers must allow Windows Sockets 2 clients to invoke send and receive operations from within the context of the socket I/O completion routine, and guarantee that, for a given socket, I/O completion routines will not be nested.

---

Under some circumstances, a layered service provider may need to initiate and complete overlapped operations from within an internal worker thread. In this case, a **WSATHREADID** would not be available from an incoming function call. The service provider interface provides an upcall, **WPUOpenCurrentThread**, to obtain a **WSATHREADID** for the current thread. When this **WSATHREADID** is no longer needed, its resources should be returned by calling **WPUCloseThread**.

### Summary of Overlapped Completion Indication Mechanisms in the SPI

The following table summarizes the completion semantics for an overlapped socket, showing the various combination of *IpOverlapped*, *hEvent*, and *IpCompletionRoutine*:

<b>IpOverlapped</b>	<b>hEvent</b>	<b>IpCompletionRoutine</b>	<b>Completion indication</b>
NULL	Not applicable	Ignored	Operation completes synchronously, that is, it behaves as if it were a nonoverlapped socket.
!NULL	NULL	NULL	Operation completes overlapped, but there is no Windows Sockets 2-supported completion mechanism. The completion port mechanism (if supported) may be used in this case, otherwise there will be no completion notification.

<b>IpOverlapped</b>	<b>hEvent</b>	<b>IpCompletionRoutine</b>	<b>Completion indication</b>
!NULL	!NULL	NULL	Operation completes overlapped, notification by signaling event object.
!NULL	Ignored	NULL	Operation completes overlapped, notification by scheduling completion routine.

## Support for Scatter/Gather Input/Output in the SPI

The **WSPSend**, **WSPSendTo**, **WSPRecv**, and **WSPRecvFrom** routines all take an array of client buffers as input parameters and thus may be used for scatter/gather (or vectored) I/O. This can be very useful in instances where portions of each message being transmitted consist of one or more fixed length header components in addition to a message body. Such header components need not be concatenated into a single contiguous buffer prior to sending. Likewise on receiving, the header components can be automatically split off into separate buffers, leaving the message body pure.

Utilizing lists of buffers instead of a single buffer does not change the boundaries that apply to receive operations. For message-oriented protocols, a receive operation completes whenever a single message has been received, regardless of how many or few of the supplied buffers were used. Likewise for stream-oriented protocols, a receive completes when an unspecified quantity of bytes arrives over the network, not necessarily when all of the supplied buffers are full.

## Out-of-Band Data in the SPI

The service providers which support the out-of-band data (OOB) abstraction for the stream-style sockets must adhere to the semantics in this section. We will describe OOB data handling in a protocol-independent manner. Please refer to the *Windows Sockets 2 Protocol-Specific Annex* (a separate document) for a discussion of OOB data implemented using urgent data in TCP/IP service providers. In the following, the use of **WSPRecv** also implies **WSPRecvFrom**.

### Protocol Independent–Out-of-Band Data in the SPI

OOB data is a logically independent transmission channel associated with each pair of connected stream sockets. OOB data may be delivered to the user independently of normal data. The abstraction defines that the OOB data facilities must support the reliable delivery of at least one OOB data block at a time. This data block may contain at least one byte of data, and at least one OOB data block may be pending delivery to the user at any one time. For communications protocols which support in-band signaling (that is, TCP, where the urgent data is delivered in sequence with the normal data), the system normally extracts the OOB data from the normal data stream and stores it separately (leaving a gap in the normal data stream). This allows users to choose between receiving the OOB data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to peek at OOB data.

A user can determine if there is any OOB data waiting to be read using the **WSIoctl(SIOCATMARK)** function. For protocols where the concept of the position of the OOB data block within the normal data stream is meaningful (that is, TCP), a Windows Sockets service provider will maintain a conceptual marker indicating the position of the last byte of OOB data within the normal data stream. This is not necessary for the implementation of the **WSIoctl (SIOCATMARK)** functionality—the presence or absence of OOB data is all that is required.

For protocols where the concept of the position of the OOB data block within the normal data stream is meaningful an application may prefer to process out-of-band data inline, as part of the normal data stream. This is achieved by setting the socket option **SO\_OOBINLINE** (see section **WSIoctl**). For other protocols where the OOB data blocks are truly independent of the normal data stream, attempting to set **SO\_OOBINLINE** will result in an error. An application can use the **SIOCATMARK WSIoctl** command to determine whether there is any unread OOB data preceding the mark. For example, it might use this to resynchronize with its peer by ensuring that all data up to the mark in the data stream is discarded when appropriate.

With **SO\_OOBINLINE** disabled (by default):

- The Windows Sockets service provider notifies a client of an **FD\_OOB** event, if the client registered for notification with **WSPAsyncSelect**, in exactly the same way **FD\_READ** is used to notify of the presence of normal data. That is, **FD\_OOB** is posted when OOB data arrives and there was no OOB data previously queued, and also when data is read using the **MSG\_OOB** flag, and some OOB data remains to be read after the read operation has returned. **FD\_READ** messages are not posted for OOB data.
- The Windows Sockets service provider returns from **WSPSelect** with the appropriate *exceptfds* socket set if OOB data is queued on the socket.
- The client can call **WSPRecv** with **MSG\_OOB** to read the urgent data block at any time. The block of OOB data jumps the queue.
- The client can call **WSPRecv** without **MSG\_OOB** to read the normal data stream. The OOB data block will not appear in the data stream with normal data. If OOB data remains after any call to **WSPRecv**, the service provider notifies the client with **FD\_OOB** or through *exceptfds* when using **WSPSelect**.
- For protocols where the OOB data has a position within the normal data stream, a single **WSPRecv** operation will not span that position. One **WSPRecv** will return the normal data before the mark, and a second **WSPRecv** is required to begin reading data after the mark.

With **SO\_OOBINLINE** enabled:

- **FD\_OOB** messages are *not* posted for OOB data—for the purpose of the **WSPSelect** and **WSPAsyncSelect** functions, OOB data is treated as normal data, and indicated by setting the socket in *readfds* or by sending an **FD\_READ** message respectively.

- The client may not call **WSPRecv** with the `MSG_OOB` flag set to read the OOB data block—the error code `WSAEINVAL` will be returned.
- The client can call **WSPRecv** without the `MSG_OOB` flag set. Any OOB data will be delivered in its correct order within the normal data stream. OOB data will never be mixed with normal data—there must be three read requests to get past the OOB data. The first returns the normal data prior to the OOB data block, the second returns the OOB data, the third returns the normal data following the OOB data. In other words, the OOB data block boundaries are preserved.

The **WSPAsyncSelect** routine is particularly well suited to handling notification of the presence of OOB data when `SO_OOBINLINE` is off.

## Shared Sockets in the SPI

Socket sharing between processes in Windows Sockets is implemented as follows. A source process calls **WSPDuplicateSocket** to obtain a special **WSAPROTOCOL\_INFOW** structure. It uses some interprocess communications (IPC) mechanism to pass the contents of this structure to a target process. The target process then uses the **WSAPROTOCOL\_INFOW** structure in a call to **WSPSocket**. The socket descriptor returned by this function will be an additional socket descriptor to an underlying socket which thus becomes shared.

It is the service provider's responsibility to perform whatever operations are needed in the source process context and to create a **WSAPROTOCOL\_INFOW** structure that will be recognized when it subsequently appears as a parameter to **WSPSocket** in the target processes' context. The *dwProviderReserved* parameter of the **WSAPROTOCOL\_INFOW** structure is available for the service provider's use, and may be used to store any useful context information, including a duplicated handle.

This mechanism is designed to be appropriate for both single-threaded versions of Windows (such as Windows 3.1) and preemptive multithreaded versions of Windows (such as Windows 95 and Windows NT/Windows 2000). Note however, that sockets may be shared amongst threads in a given process without using the **WSPDuplicateSocket** function, since a socket descriptor is valid in all of a process' threads.

As is described in section Descriptor Allocation, when new socket descriptors are allocated IFS providers must call **WPUModifyIFSHandle** and nonIFS providers must call **WPUCreateSocketHandle**.

One possible scenario for establishing and using a shared socket in a handoff mode is illustrated here.



Source process	IPC	Destination process
1) <b>WSPSocket</b> , <b>WSPConnect</b>		
2) Requests target process identifier.	⇒	
		3) Receives process identifier request and responds.
4) Receives process identifier.	⇐	
5) Calls <b>WSPDuplicateSocket</b> to get a special <b>WSAPROTOCOL_INFOW</b> structure.		
6) Sends <b>WSAPROTOCOL_INFOW</b> structure to target.	⇒	
		7) Receives <b>WSAPROTOCOL_INFOW</b> structure.
		8) Calls <b>WSPSocket</b> to create shared socket descriptor.
10) <b>WSPClosesocket</b>		9) Uses shared socket for data exchange.

## Multiple Handles to a Single Socket

Since what are duplicated are the socket descriptors and not the underlying sockets, all of the states associated with a socket are held in common across all the descriptors. For example a **WSPSetSockOpt** operation performed using one descriptor is subsequently visible using a **WSPGetSockOpt** from any or all descriptors.

Notification on shared sockets is subject to the usual constraints of **WSPAsyncSelect** and **WSPEventSelect**. Issuing either of these calls using any of the shared descriptors cancels any previous event registration for the socket, regardless of which descriptor was used to make that registration. Thus, for example, it would not be possible to have process A receive **FD\_READ** events and process B receive **FD\_WRITE** events. For situations when such tight coordination is required, it is suggested that developers consider using threads instead of separate processes.

## Reference Counting

A process may call **WSPCloseSocket** on a duplicated socket and the descriptor will become deallocated. The underlying socket, however, will remain open until **WSPCloseSocket** is called on the last remaining descriptor.

## Precedence Guidelines

The two (or more) descriptors that reference a shared socket may be used independently as far as I/O is concerned. However, the Windows Sockets interface does not implement any type of access control, so it is up to the processes involved to coordinate their operations on a shared socket. A typical use for shared sockets is to have one process that is responsible for creating sockets and establishing connections hand off sockets to other processes which are responsible for information exchange.

The general guideline for supporting multiple outstanding operations on shared sockets is that a service provider is encouraged to honor all simultaneous operations on shared sockets, especially the most recent operation on a socket object. If need be, this may occur at the expense of aborting some of the previous but still pending operations, which will return `WSAEINTR` in this case.

## Protocol-Independent Multicast and Multipoint in the SPI

Just as Windows Sockets 2 allows the basic data transport capabilities of numerous transport protocols to be accessed in a generic manner, it also provides a generic way to use multipoint and multicast capabilities of transports that implement these features. To simplify, the term multipoint is used hereafter to refer to both multicast and multipoint communications.

Current multipoint implementations (for example, IP multicast, ST-II, T.120, ATM UNI, etc.) vary widely with respect to how nodes join a multipoint session, whether a particular node is designated as a central or root node, and whether data is exchanged between all nodes or only between a root node and the various leaf nodes. The Windows Sockets 2 **WSAPROTOCOL\_INFOW** structure is used to declare the various multipoint attributes of a protocol. By examining these attributes the programmer will know what conventions to follow in using the applicable Windows Sockets 2 functions to set up, use, and tear down multipoint sessions.

The features of Windows Sockets 2 that support multicast can be summarized as follows:

- Three attribute bits in the **WSAPROTOCOL\_INFOW** structure.
- Four flags defined for the *dwFlags* parameter of **WSPSocket**
- One function, **WSPJoinLeaf**, for adding leaf nodes into a multipoint session.
- Two **WSPIoctl** command codes for controlling multipoint loopback and establishing the scope for multicast transmissions. (The latter corresponds to the IP multicast time-to-live or TTL parameter.)

---

**Note** The inclusion of these multipoint features in Windows Sockets 2 does not preclude a service provider from also supporting an existing protocol-dependent interface, such as the Deering socket options for IP multicast.

---

## Multipoint Taxonomy and Glossary

The taxonomy described below first distinguishes the control plane that concerns itself with the way a multipoint session is established, from the data plane that deals with the transfer of data among session participants.

In the control plane, there are two distinct types of session establishment: *rooted* and *nonrooted*. In the case of rooted control, there exists a special participant, called *c\_root*, that is different from the rest of the members of this multipoint session, each of which is called a *c\_leaf*. The *c\_root* must remain present for the whole duration of the multipoint session, as the session will be broken up in the absence of the *c\_root*. The *c\_root* usually initiates the multipoint session by setting up the connection to a *c\_leaf*, or a number of *c\_leafs*. The *c\_root* may add more *c\_leafs*, or (in some cases) a *c\_leaf* can join the *c\_root* at a later time. Examples of the rooted control plane can be found in ATM and ST-II.

For a nonrooted control plane, all members belonging to a multipoint session are leaves, that is, no special participant acting as a *c\_root* exists. Each *c\_leaf* needs to add itself to a pre-existing multipoint session that either is always available (as in the case of an IP multicast address), or has been set up through some OOB mechanism which is outside the scope of this discussion (and hence not addressed in the proposed Windows Sockets extensions). Another way to look at this is that a *c\_root* still exists, but can be considered to be in the network cloud as opposed to one of the participants. Because a control root still exists, a nonrooted control plane could also be considered to be implicitly rooted. Examples of this kind of implicitly rooted multipoint schemes are: a teleconferencing bridge, the IP multicast system, a Multipoint Control Unit (MCU) in an H.320 video conference, etc.

In the data plane, there are also two types of data transfer styles: *rooted* and *nonrooted*. In a rooted data plane, a special participant called *d\_root* exists. Data transfer only occurs between the *d\_root* and the rest of the members of this multipoint session, each of which is referred to as a *d\_leaf*. The traffic could be unidirectional, or bidirectional. The data sent out from the *d\_root* will be duplicated (if required) and delivered to every *d\_leaf*, while the data from *d\_leafs* will only go to the *d\_root*. In the case of a rooted data plane, there is no traffic allowed among *d\_leafs*. An example of a protocol that is rooted in the data plane is ST-II.

In a nonrooted data plane, all the participants are equal in the sense that any data they send will be delivered to all the other participants in the same multipoint session. Likewise each *d\_leaf* node will be able to receive data from all other *d\_leafs*, and in some cases, from other nodes which are not participating in the multipoint session as well. No special *d\_root* node exists. IP-multicast is nonrooted in the data plane.

Note that the question of where data unit duplication occurs, or whether a shared single tree or multiple shortest-path trees are used for multipoint distribution are protocol issues. As such, they are irrelevant to the interface the client would use to perform multipoint communications. Therefore these issues are not addressed by the Windows Sockets interface.

The following table depicts the taxonomy described above and indicates how existing schemes fit into each of the categories. Note that there does not appear to be any multipoint schemes that employ a nonrooted control plane along with a rooted data plane.

	<b>Rooted control plane</b>	<b>Nonrooted (implicit rooted) control plane</b>
<b>Rooted data plane</b>	ATM, ST-II	No known examples
<b>Nonrooted data plane</b>	T.120	IP-multicast, H.320 (MCU)

## Multipoint Attributes in the **WSAPROTOCOL\_INFOW** Structure

Three attribute parameters are defined in the **WSAPROTOCOL\_INFOW** structure in order to distinguish the different schemes used in the control and data planes, respectively:

- **XP1\_SUPPORT\_MULTIPPOINT** with a value of 1 indicates this protocol entry supports multipoint communications, and that the following two parameters are meaningful.
- **XP1\_MULTIPPOINT\_CONTROL\_PLANE** indicates whether the control plane is rooted (value = 1) or nonrooted (value = 0).
- **XP1\_MULTIPPOINT\_DATA\_PLANE** indicates whether the data plane is rooted (value = 1) or nonrooted (value = 0).

Two **WSAPROTOCOL\_INFOW** entries would be present if a multipoint protocol supported both rooted and nonrooted data planes, one entry for each.

## Multipoint Socket Attributes

In some instances sockets joined to a multipoint session may have some behavioral differences from point-to-point sockets. For example, a **d\_leaf** socket in a rooted data plane scheme can only send information to the **d\_root** participant. This creates a need for the client to be able to indicate the intended use of a socket coincident with its creation. This is done through four multipoint attribute flags that can be set through the **dwFlags** parameter in **WSPSocket**:

- **WSA\_FLAG\_MULTIPPOINT\_C\_ROOT**, for the creation of a socket acting as a **c\_root**, and only allowed if a rooted control plane is indicated in the corresponding **WSAPROTOCOL\_INFOW** entry.
- **WSA\_FLAG\_MULTIPPOINT\_C\_LEAF**, for the creation of a socket acting as a **c\_leaf**, and only allowed if **XP1\_SUPPORT\_MULTIPPOINT** is indicated in the corresponding **WSAPROTOCOL\_INFOW** entry.
- **WSA\_FLAG\_MULTIPPOINT\_D\_ROOT**, for the creation of a socket acting as a **d\_root**, and only allowed if a rooted data plane is indicated in the corresponding **WSAPROTOCOL\_INFOW** entry.

- `WSA_FLAG_MULTIPOINT_D_LEAF`, for the creation of a socket acting as a `d_leaf`, and only allowed if `XP1_SUPPORT_MULTIPOINT` is indicated in the corresponding `WSAPROTOCOL_INFOW` entry.

When creating a multipoint socket, exactly one of the two control plane flags, and one of the two data plane flags must be set in `WSPSocket`'s `dwFlags` parameter. Thus, the four possibilities for creating multipoint sockets are: “`c_root/d_root`”, “`c_root/d_leaf`”, “`c_leaf/d_root`”, or “`c_leaf /d_leaf`”.

## **SIO\_MULTIPOINT\_LOOPBACK Ioctl**

When `d_leaf` sockets are used in a nonrooted data plane, it is generally desirable to be able to control whether traffic sent out is also received back on the same socket. The `SIO_MULTIPOINT_LOOPBACK` command code for `WSPIoctl` is used to enable or disable loopback of multipoint traffic.

## **SIO\_MULTICAST\_SCOPE Ioctl**

When multicasting is employed, it is usually necessary to specify the scope over which the multicast should occur. Here scope is defined to be the number of routed network segments to be covered. The `SIO_MULTICAST_SCOPE` command code for `WSPIoctl` is used to control this. A scope of zero would indicate that the multicast transmission would not be placed on the wire but could be disseminated across sockets within the local host. A scope value of one (the default) indicates that the transmission will be placed on the wire, but will not cross any routers. Higher scope values determine the number of routers that may be crossed. Note that this corresponds to the time-to-live (TTL) parameter in IP multicasting.

## **SPI Semantics for Joining Multipoint Leaves**

In the following, a multipoint socket is frequently characterized by describing its role in one of the two planes (control or data). It should be understood that this same socket has a role in the other plane, but this is not mentioned in order to keep the references short. For example a reference to a `c_root` socket could refer to either a `c_root/d_root` or a `c_root/d_leaf` socket.

In rooted control plane schemes, new leaf nodes are added to a multipoint session in one or both of two different ways. In the first method, the root uses `WSPJoinLeaf` to initiate a connection with a leaf node and invite it to become a participant. On the leaf node, the peer application must have created a `c_leaf` socket and used `WSPListen` to set it into listen mode. The leaf node will receive an `FD_ACCEPT` indication when invited to join the session, and signals its willingness to join by calling `WSPAccept`. The root application will receive an `FD_CONNECT` indication when the join operation has been completed.

In the second method, the roles are essentially reversed. The root client creates a `c_root` socket and sets it into listen mode. A leaf node wishing to join the session creates a `c_leaf` socket and uses `WSPJoinLeaf` to initiate the connection and request admittance.

The root client receives `FD_ACCEPT` when an incoming admittance request arrives, and admits the leaf node by calling **WSPAccept**. The leaf node receives `FD_CONNECT` when it has been admitted.

In a nonrooted control plane, where all nodes are `c_leafs`, the **WSPJoinLeaf** function is used to initiate the inclusion of a node into an existing multipoint session. An `FD_CONNECT` indication is provided when the join has been completed and the returned socket descriptor is useable in the multipoint session. In the case of IP multicast, this would correspond to the `IP_ADD_MEMBERSHIP` socket option.

There are, therefore, three instances where a client would use **WSPJoinLeaf**:

- Acting as a multipoint root and inviting a new leaf to join the session.
- Acting as a leaf making an admittance request to a rooted multipoint session.
- Acting as a leaf seeking admittance to a nonrooted multipoint session (for example, IP multicast.)

## Using WSPJoinLeaf

As mentioned previously, **WSPJoinLeaf** is used to join a leaf node into a multipoint session. **WSPJoinLeaf** has the same parameters and semantics as **WSPConnect** except that it returns a socket descriptor (as in **WSPAccept**), and it has an additional *dwFlags* parameter. The *dwFlags* parameter is used to indicate whether the socket will be acting only as a sender, only as a receiver, or both. Only multipoint sockets may be used for input parameter *s* in this function. If the multipoint socket is in the nonblocking mode, the returned socket descriptor will not be useable until after a corresponding `FD_CONNECT` indication has been received. A root application in a multipoint session may call **WSPJoinLeaf** one or more times in order to add a number of leaf nodes, however at most one multipoint connection request may be outstanding at a time.

The socket descriptor returned by **WSPJoinLeaf** is different depending on whether the input socket descriptor, *s*, is a `c_root` or a `c_leaf`. When used with a `c_root` socket, the *name* parameter designates a particular leaf node to be added and the returned socket descriptor is a `c_leaf` socket corresponding to the newly added leaf node. It is not intended to be used for exchange of multipoint data, but rather is used to receive network event indications (for example `FD_CLOSE`) for the connection that exists to the particular `c_leaf`. Some multipoint implementations may also allow this socket to be used for side chats between the root and an individual leaf node. An `FD_CLOSE` indication should be given for this socket if the corresponding leaf node calls **WSPCloseSocket** to drop out of the multipoint session. Symmetrically, invoking **WSPCloseSocket** on the `c_leaf` socket returned from **WSPJoinLeaf** will cause the socket in the corresponding leaf node to get `FD_CLOSE` notification.

When **WSPJoinLeaf** is invoked with a `c_leaf` socket, the *name* parameter contains the address of the root application (for a rooted control scheme) or an existing multipoint session (nonrooted control scheme), and the returned socket descriptor is the same as the input socket descriptor. In a rooted control scheme, the root client would put its `c_root` socket in the listening mode by calling **WSPListen**. The standard `FD_ACCEPT`

notification will be delivered when the leaf node requests to join itself to the multipoint session. The root client uses the usual **WSPAccept** function to admit the new leaf node. The value returned from **WSPAccept** is also a `c_leaf` socket descriptor just like those returned from **WSPJoinLeaf**. To accommodate multipoint schemes that allow both root-initiated and leaf-initiated joins, it is acceptable for a `c_root` socket that is already in listening mode to be used as in input to **WSPJoinLeaf**.

A multipoint root client is generally responsible for the orderly dismantling of a multipoint session. Such an application may use **WSPShutdown** or **WSPClosesocket** on a `c_root` socket to cause all of the associated `c_leaf` sockets, including those returned from **WSPJoinLeaf** and their corresponding `c_leaf` sockets in the remote leaf nodes, to get `FD_CLOSE` notification.

## Semantic Differences Between Multipoint Sockets and Regular Sockets in the SPI

In the control plane, there are some significant semantic differences between a `c_root` socket and a regular point-to-point socket:

- The `c_root` socket can be used in **WSPJoinLeaf** to join a new leaf.
- Placing a `c_root` socket into the listening mode (by calling **WSPListen**) does not preclude the `c_root` socket from being used in a call to **WSPJoinLeaf** to add a new leaf, or for sending and receiving multipoint data.
- The closing of a `c_root` socket will cause all the associated `c_leaf` sockets to get `FD_CLOSE` notification.

There are no semantic differences between a `c_leaf` socket and a regular socket in the control plane, except that the `c_leaf` socket can be used in **WSPJoinLeaf**, and the use of `c_leaf` socket in **WSPListen** indicates that only multipoint connection requests should be accepted.

In the data plane, the semantic differences between the `d_root` socket and a regular point-to-point socket are

- The data sent on the `d_root` socket will be delivered to all the leaves in the same multipoint session.
- The data received on the `d_root` socket may be from any of the leaves.

The `d_leaf` socket in the rooted data plane has no semantic difference from the regular socket, however, in the nonrooted data plane, the data sent on the `d_leaf` socket will go to all the other leaf nodes, and the data received could be from any other leaf nodes. As mentioned earlier, the information about whether the `d_leaf` socket is in a rooted or nonrooted data plane is contained in the corresponding **WSAPROTOCOL\_INFOW** structure for the socket.

## Socket Options and IOCTLs

The socket options for Windows Sockets 2 are summarized in the following table. More detailed information is provided in section 4 under **WSPGetSockOpt** and/or **WSPSetSockOpt**. There are other new protocol-specific socket options which can be found in the *Protocol-Specific Annex*.

A Windows Sockets service provider must recognize all of these options, and (for **WSPGetSockOpt**) return plausible values for each. The default value for each option is shown in the following table.

Value	Type	Meaning	Default	Note
<b>SO_ACCEPTCONN</b>	BOOL	Socket is listening.	FALSE unless a <b>WSPListen</b> has been performed.	
<b>SO_BROADCAST</b>	BOOL	Socket is configured for the transmission of broadcast messages.	FALSE	
<b>SO_DEBUG</b>	BOOL	Debugging is enabled.	FALSE	(i)
<b>SO_DONTLINGER</b>	BOOL	If true, the <b>SO_LINGER</b> option is disabled.	TRUE	
<b>SO_DONTROUTE</b>	BOOL	Routing is disabled. Not supported on ATM sockets (results in an error).	FALSE	(i)
<b>SO_ERROR</b>	int	Retrieves error status and clear.	0	
<b>SO_GROUP_ID</b>	GROUP	Reserved.	NULL	Get only
<b>SO_GROUP_PRIORITY</b>	int	Reserved.	0	
<b>SO_KEEPAIVE</b>	BOOL	Keepalives are being sent. Not supported on ATM sockets (results in an error).	FALSE	(i)
<b>SO_LINGER</b>	Structure linger	Returns the current linger options.	<code>l_onoff</code> is 0	

(continued)



*(continued)*

Value	Type	Meaning	Default	Note
<b>SO_MAX_MSG_SIZE</b>	int	Maximum outbound size of a message for message socket types. There is no provision to determine the maximum inbound message size. Has no meaning for stream-oriented sockets.	Implementation dependent	Get only
<b>SO_OOINLINE</b>	BOOL	OOB data is being received in the normal data stream.	FALSE	
<b>SO_PROTOCOL_INFOW</b>	structure <b>WSAPROTOCOL_INFOW</b>	Description of protocol information for the protocol that is bound to this socket.	Protocol dependent	Get only
<b>SO_RCVBUF</b>	int	Buffer size for receives.	Implementation dependent	(i)
<b>SO_REUSEADDR</b>	BOOL	The address to which this socket is bound can be used by others. Not applicable on ATM sockets.	FALSE	
<b>SO_SNDBUF</b>	int	Total per-socket buffer space reserved for sends. This is unrelated to <b>SO_MAX_MSG_SIZE</b> or the size of a TCP window.	Implementation dependent	(i)
<b>SO_TYPE</b>	int	The type of the socket (for example, <b>SOCK_STREAM</b> ).	As created through socket.	
<b>PVD_CONFIG</b>	char FAR *	An opaque data structure object containing configuration information of the service provider.	Implementation dependent	
<b>TCP_NODELAY</b>	BOOL	Disables the Nagle algorithm for send coalescing.	Implementation dependent	

(i) A service provider may silently ignore this option on **WSPSetSockOpt** and return a constant value for **WSPGetSockOpt**, or it may accept a value for **WSPSetSockOpt** and return the corresponding value in **WSPGetSockOpt** without using the value in any way.

## Summary of Socket ioctl Opcodes

The socket IOCTL opcodes for Windows Sockets 2 are summarized in the following table. More detailed information is provided in section 4 under **WSPIoctl**. There are other new protocol-specific IOCTL opcodes that can be found in the protocol-specific annex.

Opcode	Input type	Output type	Meaning
<b>FIONBIO</b>	Unsigned long	<Not used>	Enables or disables nonblocking mode on the socket.
<b>FIONREAD</b>	<Not used>	Unsigned long	Determines the amount of data that can be read atomically from the socket.
<b>SIOCATMARK</b>	<Not used>	BOOL	Determines whether or not all OOB data has been read.
<b>SIO_ASSOCIATE_HANDLE</b>	Companion API dependent	<Not used>	Associates the socket with the specified handle of a companion interface.
<b>SIO_ENABLE_CIRCULAR_QUEUEING</b>	<Not used>	<Not used>	Enables circular queuing.
<b>SIO_FIND_ROUTE</b>	<b>SOCKADDR</b> structure	<Not used>	Requests the route to the specified address to be discovered.
<b>SIO_FLUSH</b>	<Not used>	<Not used>	Discards current contents of the sending queue.
<b>SIO_GET_BROADCAST_ADDRESS</b>	<Not used>	<b>SOCKADDR</b> structure	Retrieves the protocol-specific broadcast address to be used in <b>WSPSendTo</b> .
<b>SIO_GET_QOS</b>	<Not used>	QOS	Retrieves current flow specification(s) for the socket.
<b>SIO_GET_GROUP_QOS</b>	<Not used>	QOS	Reserved.
<b>SIO_MULTIPOINT_LOOKBACK</b>	BOOL	<Not used>	Controls whether data sent in a multipoint session will also be received by the same socket on the local host.

*(continued)*

*(continued)*

<b>Opcode</b>	<b>Input type</b>	<b>Output type</b>	<b>Meaning</b>
<b>SIO_MULTICAST_SCOPE</b>	int	<Not used>	Specifies the scope over which multicast transmissions will occur.
<b>SIO_SET_QOS</b>	QOS	<Not used>	Establishes new flow specification(s) for the socket.
<b>SIO_SET_GROUP_QOS</b>	QOS	<Not used>	Reserved.
<b>SIO_TRANSLATE_HANDLE</b>	int	Companion-API dependent	Obtains a corresponding handle for socket <i>s</i> that is valid in the context of a companion interface.
<b>SIO_ROUTING_INTERFACE_QUERY</b>	<b>SOCKADDR</b>	<b>SOCKADDR</b>	Obtains the address of the local interface that should be used to send to the specified address.
<b>SIO_ROUTING_INTERFACE_CHANGE</b>	<b>SOCKADDR</b>	<Not used>	Requests notification of changes in information reported through <b>SIO_ROUTING_INTERFACE_QUERY</b> for the specified address.
<b>SIO_ADDRESS_LIST_QUERY</b>	<Not used>	<b>SOCKET_ADDRESS_LIST</b>	Obtains the list of addresses to which the application can bind.
<b>SIO_ADDRESS_LIST_CHANGE</b>	<Not used>	<Not used>	Requests notification of changes in information reported through <b>SIO_ADDRESS_LIST_QUERY</b>
<b>SIO_QUERY_PNP_TARGET_HANDLE</b>	<Not used>	<b>SOCKET</b>	Obtains socket descriptor of the next provider in the chain on which current socket depends in regards to PnP.

## Summary of SPI Functions

The SPI functions for Windows Sockets 2 are summarized in the following tables.

### Generic Data Transport Functions

This section lists the Data Transport functions exposed by `Ws2spi.h`.

Function	Description
<b>WSPAccept</b>	An incoming connection is acknowledged and associated with an immediately created socket. The original socket is returned to the listening state. This function also allows for conditional acceptance.
<b>WSPAsyncSelect</b>	Performs asynchronous version of WSPSelect.
<b>WSPBind</b>	Assigns a local name to an unnamed socket.
<b>WSPCancelBlockingCall</b>	Cancels an outstanding blocking Windows Sockets call.
<b>WSPCleanup</b>	Signs off from the underlying Windows Sockets service provider.
<b>WSPCloseSocket</b>	Removes a socket from the per-process object reference table. Only blocks if <b>SO_LINGER</b> is set with a nonzero time-out on a blocking socket.
<b>WSPConnect</b>	Initiates a connection on the specified socket. This function also allows for exchange of connect data and QOS specification.
<b>WSPDuplicateSocket</b>	Returns a <b>WSAPROTOCOL_INFOW</b> structure that can be used to create a new socket descriptor for a shared socket.
<b>WSPEnumNetworkEvents</b>	Discovers occurrences of network events.
<b>WSPEventSelect</b>	Associates network events with an event object.
<b>WSPGetOverlappedResult</b>	Gets completion status of overlapped operation.
<b>WSPGetPeerName</b>	Retrieves the name of the peer connected to the specified socket.
<b>WSPGetSockName</b>	Retrieves the local address to which the specified socket is bound.
<b>WSPGetSockOpt</b>	Retrieves options associated with the specified socket.
<b>WSPGetQOSByName</b>	Supplies QOS parameters based on a well-known service name.
<b>WSPIoctl</b>	Provides control for sockets.
<b>WSPJoinLeaf</b>	Joins a leaf node into a multipoint session.
<b>WSPListen</b>	Listens for incoming connections on a specified socket.
<b>WSPRecv</b>	Receives data from a connected or unconnected socket. This function accommodates scatter/gather I/O, overlapped sockets, and provides the flags parameter as IN/OUT.

*(continued)*

*(continued)*

<b>Function</b>	<b>Description</b>
<b>WSPRecvDisconnect</b>	Terminates reception on a socket, and retrieve the disconnect data if the socket is connection-oriented.
<b>WSPRecvFrom</b>	Receives data from either a connected or unconnected socket. This function accommodates scatter/gather I/O, overlapped sockets and provides the flags parameter as IN/OUT.
<b>WSPSelect</b>	Performs synchronous I/O multiplexing.
<b>WSPSend</b>	Sends data to a connected socket. This function also accommodates scatter/gather I/O and overlapped sockets.
<b>WSPSendDisconnect</b>	Initiates termination of a socket connection and optionally send disconnect data.
<b>WSPSendTo</b>	Sends data to either a connected or unconnected socket. This function also accommodates scatter/gather I/O and overlapped sockets.
<b>WSPSetSockOpt</b>	Stores options associated with the specified socket.
<b>WSPShutdown</b>	Shuts down part of a full-duplex connection.
<b>WSPSocket</b>	A socket creation function which takes a <b>WSAPROTOCOL_INFOW</b> structure as input and allows overlapped sockets to be created.
<b>WSPStartup</b>	Initializes the underlying Windows Sockets service provider.

## Upcalls Exposed by Windows Sockets 2 DLL

This section lists the upcalls that service providers may make into the Windows Sockets client. Service providers receive an upcall dispatch table as a parameter to **WSPStartup()**, and use entries in this table to make the upcalls. Therefore, a client does not need to export its WPU functions.

It is not mandatory that providers use all of these upcalls. The following table indicates which upcalls must be used and which are optional.

Function	Description	Status	Meaning
<b>WPUCloseEvent</b>	Closes an open event object handle.	Optional.	The provider may use an appropriate OS call instead.
<b>WPUCloseSocketHandle</b>	Closes a socket handle allocated by the Windows Sockets DLL.	Required.	The Ws2_32.dll needs to query and/or modify internal state information associated with the socket handle.
<b>WPUCloseThread</b>	Closes a thread ID for an internal service thread.		
<b>WPUCompleteOverlappedRequest</b>	Delivers overlapped I/O completion notification where the completion mechanism is something other than user mode APC.		
<b>WPUCreateEvent</b>	Creates a new event object.	Optional.	The provider may use an appropriate OS call instead.
<b>WPUCreateSocketHandle</b>	Creates a new socket handle for nonIFS providers.	Required for nonIFS providers.	The Ws2_32.dll needs to query and/or modify internal state information associated with the socket handle.
<b>WPUFDIsSet</b>	Checks the membership of the specified socket handle.	Optional.	This is just a convenience function that knows how to dig through <b>FD SET</b> structures. A provider may need to dig through these structures explicitly anyway.

*(continued)*

*(continued)*

<b>Function</b>	<b>Description</b>	<b>Status</b>	<b>Meaning</b>
<b>WPUGetProviderPath</b>	Retrieves the DLL path for the specified provider.	Required.	Only the Ws2_32.dll would know where an adjacent protocol layer (potentially from another vendor) has been installed.
<b>WPUModifyIFSHandle</b>	Receives a (possibly) modified IFS handle from the Windows Sockets DLL.	Required for IFS providers.	The Ws2_32.dll needs to query and/or modify internal state information associated with the socket handle.
<b>WPUPostMessage</b>	Performs the standard <b>PostMessage</b> function in a way that maintains backward compatibility.	Required.	Windows NT/2000 only. Windows 95 allows post message from kernel mode.
<b>WPUQueryBlockingCallback</b>	Returns a pointer to a thread's blocking hook function.	Required.	There is no corresponding OS functionality. Only the Ws2_32.dll has the information to accomplish this.
<b>WPUQuerySocketHandleContext</b>	Gets a socket's context value (nonIFS providers only).	Required for nonIFS providers.	The Ws2_32.dll needs to query and/or modify internal state information associated with the socket handle.
<b>WPUQueueApc</b>	Queues a user-mode APC to the specified thread.	Optional.	The <b>QueueUserApc</b> may also be used.

Function	Description	Status	Meaning
<b>WPUResetEvent</b>	Resets an event object.	Optional.	The provider may use an appropriate OS call instead.
<b>WPUSetEvent</b>	Sets an event object.	Optional.	The provider may use an appropriate OS call instead.

## Installation and Configuration Functions

The following functions are implemented in the `Ws2_32.dll`, and are intended to be used by applications that install Windows Sockets transport and name space service providers on a machine. These functions neither affect currently running applications, nor are the changes made by these functions visible to currently running applications.

For all providers, a provider identifier is a GUID which is generated by the developer of the provider (using the `Uuidgen.exe` utility) and supplied to `Ws2_32.dll`.

Function	Description
<b>WSCDeinstallProvider</b>	Removes a service provider from the registry.
<b>WSEnumProtocols</b>	Retrieves information about available transport protocols.
<b>WSCInstallProvider</b>	Registers a new service provider.

## Name Resolution Service Provider Requirements

The following sections provide a description of each of the functional areas that name space providers are required to implement. Where appropriate, implementation considerations and guidelines are also provided.

## Summary of Namespace Provider Functions

The namespace service provider functions can be grouped into five categories:

- Namespace provider configuration (and installation)
- Provider initialization
- Service installation
- Client queries
- Helper functions (and macros)

The following sections identify the functions in each category and briefly describe their intended use. Key data structures are also described.



## Namespace Provider Configuration and Installation

- **WSCInstallNameSpace**
- **WSCUnInstallNameSpace**
- **WSCEnableNSProvider**

As mentioned previously, the installation application for a namespace provider must call **WSCInstallNameSpace** to register with the Ws2\_32.dll and supply static configuration information. The Ws2\_32.dll uses this information to accomplish its routing function and in its implementation of **WSAEnumNameSpaceProviders**. The **WSCUnInstallNameSpace** function is used to remove a name space provider from the registry, and the **WSCEnableNSProvider** function is used to toggle a provider between the active and inactive states.

The results of these three operations are not visible to applications that are currently loaded and running. Only applications that begin executing after these operations have occurred will be affected by them.

This architecture explicitly supports the instantiation of multiple name space providers within a single DLL, however each such provider must have a unique name space provider identifier (GUID) allocated, and a separate call to **WSCInstallNameSpace** must occur for each instantiation. Such a provider can determine which instantiation is being invoked because the namespace provider (NSP) identifier appears as a parameter in every NSP function.

## Namespace Provider Initialization and Cleanup

- **NSPStartup**
- **NSPCleanup**

As is the case for the transport SPI, a namespace provider is initialized with a call to **NSPStartup** and is terminated with a final call to **NSPCleanup**. Calls to the startup function may be nested, but will be matched by corresponding calls to the cleanup function. A provider should employ reference counting to determine when the final call to **NSPCleanup** has occurred.

## Service Installation in the Windows Sockets 2 SPI

- **NSPInstallServiceClass**
- **NSPRemoveServiceClass**
- **NSPSetService**

When the required service class does not already exist, a namespace SPI client uses **NSPInstallServiceClass** to install a new service class by supplying a service class name, a GUID for the service class identifier, and a series of **WSANSCLASSINFO** structures. These structures are each specific to a particular name space, and supply common values such as recommended TCP port numbers or Netware SAP Identifiers.

A service class can be removed by calling **NSPRemoveServiceClass** and supplying the GUID corresponding to the class identifier.

Once a service class exists, specific instances of a service can be installed or removed via **NSPSetService**. This function takes a **WSAQUERYSET** structure as an input parameter along with an operation code and operation flags. The operation code indicates whether the service is being installed or removed. The **WSAQUERYSET** structure provides all of the relevant information about the service including service class identifier, service name (for this instance), applicable name space identifier and protocol information, and a set of transport addresses to which the service listens.

## Service Query

- **NSPLookupServiceBegin**
- **NSPLookupServiceNext**
- **NSPLookupServiceEnd**

A name service query involves a series of calls: **NSPLookupServiceBegin**, followed by one or more calls to **NSPLookupServiceNext** and ending with a call to **NSPLookupServiceEnd**. **NSPLookupServiceBegin** takes a **WSAQUERYSET** structure as input in order to define the query parameters along with a set of flags to provide additional control over the search operation. It returns a query handle which is used in the subsequent calls to **NSPLookupServiceNext** and **NSPLookupServiceEnd**.

The name space SPI client invokes **NSPLookupServiceNext** to obtain query results, with results supplied in an client-supplied **WSAQUERYSET** buffer. The client continues to call **NSPLookupServiceNext** until the error code **WSA\_E\_NO\_MORE** is returned indicating that all results have been retrieved. The search is then terminated by a call to **NSPLookupServiceEnd**. The **NSPLookupServiceEnd** function can also be used to cancel a currently pending **NSPLookupServiceNext** when called from another thread.

In Windows Sockets 2, conflicting error codes are defined for **WSAENOMORE** (10102) and **WSA\_E\_NO\_MORE** (10110). The error code **WSAENOMORE** will be removed in a future version and only **WSA\_E\_NO\_MORE** will remain. Name space providers should switch to using the **WSA\_E\_NO\_MORE** error code as soon as possible to maintain compatibility with the widest possible range of applications.

## Helper Functions in the SPI

- **NSPGetServiceClassInfo**

The **NSPGetServiceClassInfo** function retrieves service class schema information that has been retained by a name space provider. It is also used by the Windows Sockets 2 DLL in its implementation of **WSAGetServiceClassNameByClassId**.

The following macros from **Winsock2.h** are available and can aid in mapping between well known service classes and these name spaces.

SVCID_TCP(Port)	Given a port for TCP/IP or UDP/IP or the object type in the case of Netware,
SVCID_UDP(Port)	returns the GUID.
SVCID_NETWARE(Object Type)	
IS_SVCID_TCP(GUID)	Returns TRUE if the GUID is within the allowable range.
IS_SVCID_UDP(GUID)	
IS_SVCID_NETWARE(GUID)	
PORT_FROM_SVCID_TCP(GUID)	Returns the port or object type associated with the GUID.
PORT_FROM_SVCID_UDP(GUID)	
SAPID_FROM_SVCID_NETWARE(GUID)	

## Name Resolution Data Structures in the SPI

There are several important data structures that are used extensively throughout the name resolution functions. These are described below.

- Query-Related Data Structures in the SPI
- Service Class Data Structures in the SPI

### Query-Related Data Structures in the SPI

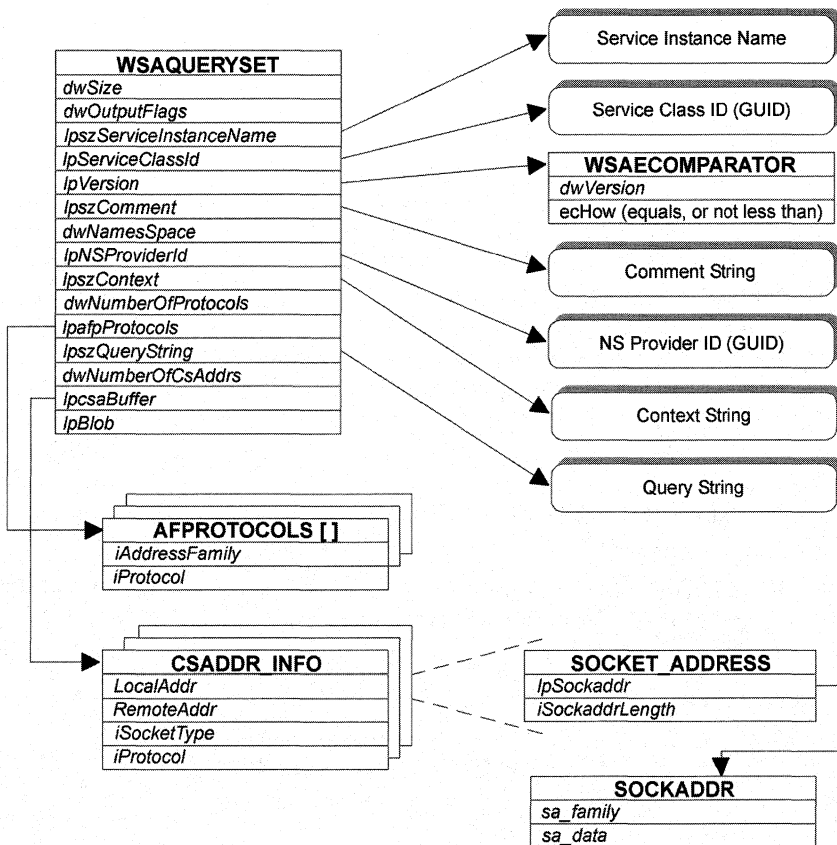
The **WSAQUERYSET** structure is used to form queries for **NSPLookupServiceBegin**, and used to deliver query results for **NSPLookupServiceNext**. It is a complex structure since it contains pointers to several other structures, some of which reference still other structures. Figure 10-4 shows the relationship between **WSAQUERYSET** and the structures it references.

Within the **WSAQUERYSET** structure, most of the members are self explanatory, but some deserve additional explanation. The *dwSize* will be filled in with `sizeof(WSAQUERYSET)`, and can be used by name space providers to detect and adapt to different versions of the **WSAQUERYSET** structure that may appear over time.

The **dwOutputFlags** member is used by a name space provider to provide additional information about query results. For details, see **NSPLookupServiceNext**.

The **WSAECOMPARATOR** structure referenced by *Ipversion* is used for both query constraint and results. For queries, the **dwVersion** member indicates the desired version of the service. The **ecHow** member is an enumerated type which specifies how the comparison will be made. The choices are **COMP\_EQUALS** which requires that an exact match in version occurs, or **COMP\_NOTLESS** which specifies that the service's version number be no less than the value of *dwVersion*.

The interpretation of **dwNameSpace** and *IpNSProviderId* depends upon how the structure is being used and is described further in the individual function descriptions that utilize this structure.



**Figure 10-4: The Data Structure and the Structures It References.**

The **lpszContext** member applies to hierarchical name spaces, and specifies the starting point of a query or the location within the hierarchy where the service resides. The general rules are:

- A value of NULL, blank ("") will start the search at the default context.
- A value of "\ " starts the search at the top of the name space.
- Any other value starts the search at the designated point.

Providers that do not support containment may return an error if anything other than "" or "\ " is specified. Providers that support limited containment, such as groups, should accept "", "\ ", or a designated point. Contexts are name space specific. If **dwNameSpace** is NS\_ALL, then only "" or "\ " should be passed as the context since these are recognized by all name spaces.

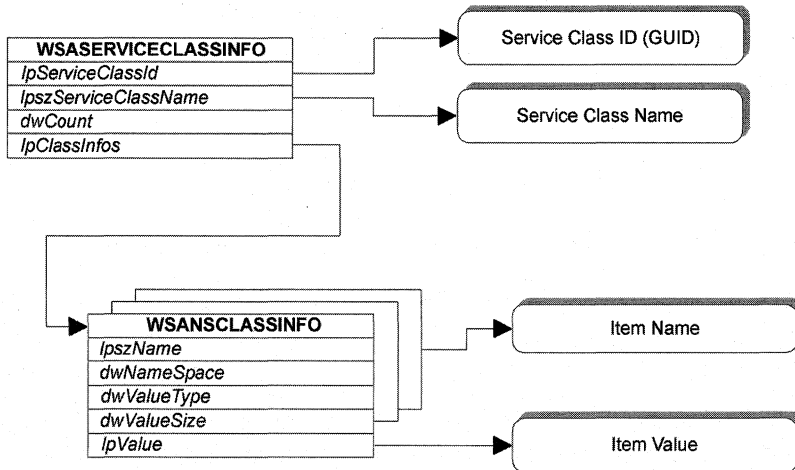
The **lpszQueryString** member is used to supply additional, name space-specific query information such as a string describing a well-known service and transport protocol name, as in "ftp/tcp".

The **AFPROTOCOLS** structure referenced by *lpafpProtocols* is used for query purposes only, and supplies a list of protocols to constrain the query. These protocols are represented as (address family, protocol) pairs, since protocol values only have meaning within the context of an address family.

The array of **CSADDR\_INFO** structure referenced by *lpcsaBuffer* contains all of the information needed for either a service to use in establishing a listen, or a client to use in establishing a connection to the service. The **LocalAddr** and **RemoteAddr** members both directly contain a **SOCKET\_ADDRESS** structure. A service would create a socket using the tuple (*LocalAddr.lpSockaddr->sa\_family*, *iSocketType*, *iProtocol*). It would bind the socket to a local address using *LocalAddr.lpSockaddr*, and *LocalAddr.lpSockaddrLength*. The client creates its socket with the tuple (*RemoteAddr.lpSockaddr->sa\_family*, *iSocketType*, *iProtocol*), and uses the combination of *RemoteAddr.lpSockaddr*, and *RemoteAddr.lpSockaddrLength* when making a remote connection.

### Service Class Data Structures in the SPI

When a new service class is installed, a **WSASERVICECLASSINFO** structure must be prepared and supplied. This structure also consists of substructures that contain a series of parameters that apply to specific name spaces. (See Figure 10-5.)



**Figure 10-5: Service Class Data Structures.**

For each service class, there is a single **WSASERVICECLASSINFO** structure. Within the **WSASERVICECLASSINFO** structure, the service class's unique identifier is contained in *IpServiceClassId*, and an associated display string is referenced by *lpServiceClassName*.

The *IpClassInfos* member in the **WSASERVICECLASSINFO** structure references an array of **WSANSCLASSINFO** structures, each of which supplies a named and typed parameter that applies to a specified name space. Examples of values for the **IpszName** member include: SAPID, TCPPORT, UDPPOINT, etc. These strings are generally specific to the name space identified in **dwNameSpace**. Typical values for *dwValueType* might be REG\_DWORD, REG\_SZ, etc. The **dwValueSize** member indicates the length of the data item pointed to by *IpValue*.

The entire collection of data represented in a **WSASERVICECLASSINFO** structure is provided to each name space provider via **NSPInstallServiceClass**. Each individual name space provider then sifts through the list of **WSANSCLASSINFO** structures and retain the information applicable to it. This architecture also envisions the future existence of a special name space provider that would retain all of the service class schema information for all of the name spaces. The *Ws2\_32.dll* would query this provider to obtain the **WSASERVICECLASSINFO** data needed to supply to name space providers when **NSPLookupServiceBegin** is invoked to initiate a query, and when **NSPSetService** is invoked to register a service. Name space provider should not rely on this capability for the time being, and should instead have a provider-specific means to obtain any needed service class schema information. In the absence of a provider that stores all service class schema for all name spaces, the *Ws2\_32.dll* will use **NSPGetServiceClassInfo** to obtain such information from each individual name space provider.

## Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 SPI

Windows Sockets 1.1 defined a number of routines that were used for name resolution with TCP/IP networks. These are customarily called the **getXbyY** functions and include the following.

- gethostname**
- gethostbyaddr**
- gethostbyname**
- getprotobyname**
- getprotobynumber**
- getservbyname**
- getservbyport**

Asynchronous versions of these functions were also defined.

- WSAAsyncGetHostByAddr**
- WSAAsyncGetHostByName**
- WSAAsyncGetProtoByName**
- WSAAsyncGetProtoByNumber**
- WSAAsyncGetServByName**
- WSAAsyncGetServByPort**

These functions are specific to TCP/IP networks and developers of protocol-independent applications are discouraged from continuing to utilize these transport-specific functions. However, to retain strict backward compatibility with Windows Sockets 1.1, all of the above functions continue to be supported as long as at least one name space provider is present that supports the AF\_INET address family.

The Ws2\_32.dll implements these compatibility functions in terms of the new, protocol-independent name resolution facilities using an appropriate sequence of **WSALookupServiceBegin**, **WSALookupServiceNext**, **WSALookupServiceEnd** function calls. The details of how the **getXbyY** functions are mapped to name resolution functions are provided below. The Ws2\_32.dll handles the differences between the asynchronous and synchronous versions of the **getXbyY** functions, so that only the implementation of the synchronous **getXbyY** functions are discussed.

### Basic Approach for getXbyY in the SPI

Most **getXbyY** functions are translated by Ws2\_32.dll to a **WSALookupServiceBegin**, **WSALookupServiceNext**, **WSALookupServiceEnd** sequence that uses one of a set of special GUIDs as the service class. These GUIDs identify the type of **getXbyY** operation that is being emulated. The query is constrained to those NSPs that support AF\_INET. Whenever a **getXbyY** function returns a **HOSTENT** or **SERVENT** structure, the Ws2\_32.dll will specify the LUP\_RETURN\_BLOB flag in **WSALookupServiceBegin** so that the desired information will be returned by the NSP. These structures must be modified slightly in that the pointers contained within must be replaced with offsets that are relative to the start of the blob's data. All values referenced by these pointer members must, of course, be completely contained within the blob, and all strings are ASCII.

### getprotobyname and getprotobynumber Functions in the SPI

These functions are implemented within Ws2\_32.dll by consulting a local protocols database. They do not result in any name resolution query.

### getservbyname and getservbyport Functions in the SPI

The **WSALookupServiceBegin** query uses SVCID\_INET\_SERVICEBYNAME as the service class GUID. The *lpzServiceInstanceName* parameter references a string which indicates the service name or service port, and (optionally) the service protocol. The formatting of the string is illustrated as ftp/tcp or 21/tcp or just ftp. The string is not case sensitive. The slash mark, if present, separates the protocol identifier from the preceding part of the string. The Ws2\_32.dll will specify LUP\_RETURN\_BLOB and the NSP will place a **SERVENT** structure in the blob (using offsets instead of pointers as described above). NSPs should honor these other LUP\_RETURN\_\* flags as well.

Flag	Description
LUP_RETURN_NAME	Returns the <b>s_name</b> member from <b>SERVENT</b> structure in <i>lpszServiceInstanceName</i> .
LUP_RETURN_TYPE	Returns canonical GUID in <i>lpServiceClassId</i> . It is understood that a service identified as ftp or 21 may be on another port according to locally established conventions. The <b>s_port</b> member of the <b>SERVENT</b> structure should indicate where the service can be contacted in the local environment. The canonical GUID returned when LUP_RETURN_TYPE is set should be one of the predefined GUIDs from <i>svcs.h</i> that corresponds to the port number indicated in the <b>SERVENT</b> structure.

### gethostbyname Function in the SPI

The **WSALookupServiceBegin** query uses **SVCID\_INET\_HOSTADDRBYNAME** as the service class GUID. The host name is supplied in *lpszServiceInstanceName*. The *Ws2\_32.dll* specifies **LUP\_RETURN\_BLOB** and the NSP places a **HOSTENT** struct in the blob (using offsets instead of pointers as described above). NSPs should honor these other **LUP\_RETURN\_\*** flags as well.

Flag	Description
LUP_RETURN_NAME	Returns the <b>h_name</b> member from <b>HOSTENT</b> structure in <i>lpszServiceInstanceName</i> .
LUP_RETURN_ADDR	Returns addressing information from <b>HOSTENT</b> in <b>CSADDR_INFO</b> structures, port information is defaulted to zero. Note that this routine does <i>not</i> resolve host names consisting of a dotted internet address.

### gethostbyaddr Function in the SPI

The **WSALookupServiceBegin** query uses **SVCID\_INET\_HOSTNAMEBYADDR** as the service class GUID. The host address is supplied in *lpszServiceInstanceName* as a dotted internet string (for example, 192.9.200.120). The *Ws2\_32.dll* specifies **LUP\_RETURN\_BLOB** and the NSP places a **HOSTENT** structure in the blob (using offsets instead of pointers as described above). NSPs should honor these other **LUP\_RETURN\_\*** flags as well.

Flag	Description
LUP_RETURN_NAME	Returns the <b>h_name</b> member from <b>HOSTENT</b> structure in <i>lpszServiceInstanceName</i> .
LUP_RETURN_ADDR	Returns addressing information from <b>HOSTENT</b> in <b>CSADDR_INFO</b> structures, port information is defaulted to zero.



## gethostname Function in the SPI

The **WSALookupServiceBegin** query uses **SVCID\_HOSTNAME** as the service class GUID. If *lpszServiceInstanceName* is NULL or references a NULL string (that is ""), the local host is to be resolved. Otherwise, a lookup on a specified host name shall occur. For the purposes of emulating **gethostname** the *Ws2\_32.dll* will specify a null pointer for *lpszServiceInstanceName*, and specify **LUP\_RETURN\_NAME** so that the host name is returned in the *lpszServiceInstanceName* parameter. If an application uses this query and specifies **LUP\_RETURN\_ADDR** then the host address will be provided in a **CSADDR\_INFO** structure. The **LUP\_RETURN\_BLOB** action is undefined for this query. Port information will be defaulted to zero unless the *lpszQueryString* references a service such as ftp, in which case the complete transport address of the indicated service will be supplied.

## Sample Code for a Service Provider

This section contains a source code sample that demonstrates how to implement the **GetXbyY** functions using the new, protocol-independent RNR functions. A developer should implement these functions as a starting point. To comply with the Windows Sockets specification, many more functions are needed.

---

**Important** The following code is not guaranteed to compile.

---

```
/**+
xbyrnr.cpp

Copyright (c) 1996 Microsoft Corporation
All rights reserved

GetXbyY emulation via new WinSock2 RNR. This source module shows
code that is built into the WinSock2 DLL (ws2_32.dll). It
demonstrates how the older GetXByY functions are mapped to the new
WSALookupServiceBegin, WSALookupServiceNext, WSALookupServiceEnd
functions.

This module is not guaranteed to compile. It is provided as source
code for RNR name-space service providers to understand what will
be coming down to their code in response to the traditional
GetXbyY calls.

At this time, only
    gethostname
    gethostbyname
    gethostbyaddr
    getservbyname
```

```
    getservbyport  
are implemented in this manner.
```

Warning: This code is preliminary, and may change before Windows Sockets 2 is released.

Warning: This is not provided as a template for either RNR applications or name space providers. This code is only intended to illustrate what happens in the WinSock2 DLL to map the GetXbyY calls to the new RNR APIs.

```
--*/  
  
#include "svcguid.h"  
  
//  
// Forward declares  
//  
  
LPBLOB  
getxyDataEnt(  
    PCHAR pResults,  
    DWORD dwSize,  
    LPSTR lpszName,  
    LPGUID lpType,  
    LPSTR * lppName  
);  
  
VOID  
FixList(PCHAR ** List, PCHAR Base);  
  
VOID  
UnpackHostEnt(struct hostent * hostent);  
  
VOID  
UnpackServEnt(struct servent * servent);  
  
GUID HostnameGuid = SVCID_INET_HOSTADDRBYNAME;  
GUID AddressGuid = SVCID_INET_HOSTADDRBYINETSTRING;  
GUID IANAGuid = SVCID_INET_SERVICEBYNAME;  
  
//  
// Utility to turn a list of offsets into a list of addresses. Used  
// to convert structures returned as BLOBs.
```

(continued)

(continued)

```
//
VOID
FixList(PCHAR ** List, PCHAR Base)
{
    if(*List)
    {
        PCHAR * Addr;

        Addr = *List = (PCHAR *) ( ((DWORD)*List + Base) );
        while(*Addr)
        {
            *Addr = (PCHAR)((DWORD)*Addr + Base);
            Addr++;
        }
    }
}

//
// Routine to convert a hostent returned in a BLOB to one with
// usable pointers. The structure is converted in-place.
//
VOID
UnpackHostEnt(struct hostent * hostent)
{
    PCHAR pch;

    pch = (PCHAR)hostent;

    if(hostent->h_name)
    {
        hostent->h_name = (PCHAR)((DWORD)hostent->h_name + pch);
    }
    FixList(&hostent->h_aliases, pch);
    FixList(&hostent->h_addr_list, pch);
}

//
// Routine to unpack a servent returned in a BLOB to one with
// usable pointers. The structure is converted in-place
//
VOID
```



```

UnpackServEnt(struct servent * servent)
{
    PCHAR pch;

    pch = (PCHAR)servent;

    FixList(&servent->s_aliases, pch);
    servent->s_name = (PCHAR)(DWORD(servent->s_name) + pch);
    servent->s_proto = (PCHAR)(DWORD(servent->s_proto) + pch);
}

struct hostent FAR * WSAAPI
gethostbyaddr(
    IN const char FAR * addr,
    IN int len,
    IN int type
)
/*++
Routine Description:

    Get host information corresponding to an address.

Arguments:

    addr - A pointer to an address in network byte order.

    len - The length of the address, which must be 4 for PF_INET
    addresses.

    type - The type of the address, which must be PF_INET.

Returns:

    If no error occurs, gethostbyaddr() returns a pointer to the
    hostent structure described above. Otherwise it returns a NULL
    pointer and a specific error code is stored with SetErrorCode().
--*/
{
    CHAR qbuf[100];
    struct hostent *ph;
    LPBLOB pBlob;
    PCHAR pResults;
    int err, ErrorCode;
    PDPROCESS Process;

```

*(continued)*

(continued)

```
PDTHREAD Thread;

err = PROLOG(&Process,
            &Thread,
            &ErrorCode);
if(err != NO_ERROR)
{
    SetLastError(ErrorCode);
    return(NULL);
}

pResults = new CHAR[RNR_BUFFER_SIZE];

if(!pResults)
{
    SetLastError(WSA_NOT_ENOUGH_MEMORY);
    return(NULL);
}

//
// NOTICE. Only handles current inet address forms.
//
(void)wsprintfA(qbuf, "%u.%u.%u.%u",
               ((unsigned)addr[0] & 0xff),
               ((unsigned)addr[1] & 0xff),
               ((unsigned)addr[2] & 0xff),
               ((unsigned)addr[3] & 0xff));

pBlob = getxyDataEnt(pResults,
                    RNR_BUFFER_SIZE,
                    qbuf,
                    &AddressGuid,
                    0);

if(pBlob)
{
    ph = (struct hostent *)Thread->CopyHostEnt(pBlob);
    if(ph)
    {
        UnpackHostEnt(ph);
    }
}
else
{
```



```

    ph = 0;
    if(GetLastError() == WSASERVICE_NOT_FOUND)
    {
        SetLastError(WSANO_ADDRESS);
    }
}
delete pResults;
return(ph);
} // gethostbyaddr

```

```

struct hostent FAR * WSAAPI
gethostbyname(
    IN const char FAR * name
)

```

/\*++

Routine Description:

Get host information corresponding to a hostname.

Arguments:

name - A pointer to the null terminated name of the host.

Returns:

If no error occurs, gethostbyname() returns a pointer to the hostent structure described above. Otherwise it returns a NULL pointer and a specific error code is stored with SetLastError().

--\*/

```

{
    struct hostent * hent;
    LPBLOB pBlob;
    PCHAR pResults;
    int err, ErrorCode;
    PDPROCESS Process;
    PDTHREAD Thread;
    CHAR szLocalName[200]; // for storing the local name. This
                          // is simply a big number assumed
                          // to be large enough. This is used
                          // only when the caller chooses not to
                          // provide a name.

    PCHAR pszName;

```

(continued)

(continued)

```
err = PROLOG(&Process,
            &Thread,
            &ErrorCode);
if(err != NO_ERROR)
{
    SetLastError(ErrorCode);
    return(NULL);
}

//
// A NULL input name means look for the local name. So,
// get it.
//
if(!name || !*name)
{
    if(gethostname(szLocalName, 200) != NO_ERROR)
    {
        return(NULL);
    }
    pszName = szLocalName;
}
else
{
    pszName = (PCHAR)name;
}

pResults = new CHAR[RNR_BUFFER_SIZE];

if(!pResults)
{
    SetLastError(WSA_NOT_ENOUGH_MEMORY);
    return(NULL);
}

pBlob = getxyDataEnt( pResults,
                    RNR_BUFFER_SIZE,
                    pszName,
                    &HostnameGuid,
                    0);

if(pBlob)
{
    hent = (struct hostent *)Thread->CopyHostEnt(pBlob);
    if(hent)
    {
```



```
        UnpackHostEnt(hent);
    }
}
else
{
    hent = 0;
    if(GetLastError() == WSASERVICE_NOT_FOUND)
    {
        SetLastError(WSAHOST_NOT_FOUND);
    }
}
delete pResults;
return(hent);
} // gethostbyname
```

```
int WSAAPI
gethostname(
    OUT char FAR * name,
    IN int namelen
)
```

/\*\*+

Routine Description:

Return the standard host name for the local machine.

Arguments:

name - A pointer to a buffer that will receive the host name.

namelen - The length of the buffer.

Returns:

Zero on success else SOCKET\_ERROR. The error code is stored with SetErrorCode().

---\*/

```
{
    PCHAR lpName;
    int err, ErrorCode;
    PDPROCESS Process;
    PDTHREAD Thread;
    PCHAR pResults;

    err = PROLOG(&Process,
```

(continued)



(continued)

```
        &Thread,
        &ErrorCode);
if(err != NO_ERROR)
{
    SetLastError(ErrorCode);
    return(SOCKET_ERROR);
}

pResults = new CHAR[RNR_BUFFER_SIZE];

if(!pResults)
{
    SetLastError(WSA_NOT_ENOUGH_MEMORY);
    return(SOCKET_ERROR);
}

if(getxyDataEnt(pResults,
               RNR_BUFFER_SIZE,
               NULL,
               &HostnameGuid,
               &lpName
               ))
{
    INT iSize = strlen(lpName) + 1;

    if(iSize <= namelen)
    {
        memcpy(name, lpName, iSize);
    }
    else
    {
        SetLastError(WSAEFAULT);
        err = SOCKET_ERROR;
    }
}
else
{
    err = SOCKET_ERROR; // assume SetLastError has been set
}
delete pResults;
return(err);
} // gethostname
```

```
struct servent FAR * WSAAPI
getservbyport(
    IN int port,
    IN const char FAR * proto
)
```

```
/*++
```

Routine Description:

Get service information corresponding to a port and protocol.

Arguments:

port - The port for a service, in network byte order.

proto - An optional pointer to a protocol name. If this is NULL, getservbyport() returns the first service entry for which the port matches the s\_port. Otherwise getservbyport() matches both the port and the proto.

Returns:

If no error occurs, getservbyport() returns a pointer to the servent structure described above. Otherwise it returns a NULL pointer and a specific error code is stored with SetLastError().

```
--*/
```

```
{
    PCHAR pszTemp;
    struct servent * sent;
    int err, ErrorCode;
    PDPROCESS Process;
    PDTHREAD Thread;
    LPBLOB pBlob;
    PCHAR pResults;

    err = PROLOG(&Process,
                &Thread,
                &ErrorCode);
    if(err != NO_ERROR)
    {
        SetLastError(ErrorCode);
        return(NULL);
    }
}
```

(continued)



(continued)

```
pResults = new CHAR[RNR_BUFFER_SIZE];

if(!pResults)
{
    SetLastError(WSA_NOT_ENOUGH_MEMORY);
    return(NULL);
}

if(!proto)
{
    proto = "";
}

//
// the 5 is the max number of digits in a port
//
pszTemp = new CHAR[strlen(proto) + 1 + 1 + 5];
wsprintfA(pszTemp, "%d/%s", (port & 0xffff), proto);
pBlob = getxyDataEnt(pResults,
                    RNR_BUFFER_SIZE,
                    pszTemp,
                    &IANAGuid,
                    0
                    );

delete pszTemp;
if(!pBlob)
{
    sent = 0;
    if(GetLastError() == WSATYPE_NOT_FOUND)
    {
        SetLastError(WSANO_DATA);
    }
}
else
{
    sent = (struct servent *)Thread->CopyServEnt(pBlob);
    if(sent)
    {
        UnpackServEnt(sent);
    }
}
delete pResults;
return(sent);
```

```
} // getservbyport

struct servent FAR * WSAAPI
getservbyname(
    IN const char FAR * name,
    IN const char FAR * proto
)
/*++
Routine Description:

    Get service information corresponding to a service name and
    protocol.

Arguments:

    name - A pointer to a null terminated service name.

    proto - An optional pointer to a null terminated protocol name. If
    this pointer is NULL, getservbyname() returns the first
    service entry for which the name matches the s_name or one
    of the s_aliases. Otherwise getservbyname() matches both
    the name and the proto.

Returns:

    If no error occurs, getservbyname() returns a pointer to the servent
    structure described above. Otherwise it returns a NULL pointer and a
    specific error code is stored with SetLastError().

--*/
{
    PCHAR pszTemp;
    struct servent * sent;
    int err, ErrorCode;
    PDPROCESS Process;
    PDTHREAD Thread;
    LPBLOB pBlob;
    PCHAR pResults;

    err = PROLOG(&Process,
                &Thread,
                &ErrorCode);
    if(err != NO_ERROR)
    {
        SetLastError(ErrorCode);
    }
}
```

(continued)



(continued)

```
        return(NULL);
    }

    pResults = new CHAR[RNR_BUFFER_SIZE];

    if(!pResults)
    {
        SetLastError(WSA_NOT_ENOUGH_MEMORY);
        return(NULL);
    }

    if(!proto)
    {
        proto = "";
    }

    pszTemp = new CHAR[strlen(name) + strlen(proto) + 1 + 1];
    wsprintfA(pszTemp, "%s/%s", name, proto);
    pBlob = getxyDataEnt(pResults,
                        RNR_BUFFER_SIZE,
                        pszTemp,
                        &IANAGuid,
                        0
                        );

    delete pszTemp;
    if(!pBlob)
    {
        sent = 0;
        if(GetLastError() == WSATYPE_NOT_FOUND)
        {
            SetLastError(WANO_DATA);
        }
    }
    else
    {
        sent = (struct servent *)Thread->CopyServEnt(pBlob);
        if(sent)
        {
            UnpackServEnt(sent);
        }
    }
    delete pResults;
    return(sent);
} // getservbyname

//
```

```
// Common routine for obtaining a xxxent buffer. Input is used to
// execute the WSALookup series of APIs.
//
// Args:
//   pResults -- a buffer supplied by the caller to be used in
//               the WASLookup calls. This should be as large as
//               the caller can afford to offer.
//   dwLength -- number of bytes in pResults
//   lpszName -- pointer to the service name. May be NULL
//   lpType   -- pointer to the service type . This should be one of
//               the SVCID_INET_xxxxx types. It may be anything
//               that produces a BLOB.
//   lppName  -- pointer to pointer where the resulting name pointer
//               is stored. May be NULL if the name is not needed.
//
// Returns:
//   0 -- No BLOB data was returned. In general, this means the
//        operation failed. Even if the WSALookupNext succeeded
//        and returned a name, the name will not be returned.
//   else -- a pointer to the BLOB.
//
//
// The protocol restrictions list for all emulation operations. This
// should limit the invoked providers to the set that know about
// hostents and servents. If not, then the special SVCID_INET GUIDs
// should take care of the remainder.
//
AFPROTOCOLS afp[2] = {
    {AF_INET, IPPROTO_UDP},
    {AF_INET, IPPROTO_TCP}
};

LPBLOB
getxyDataEnt(
    PCHAR pResults,
    DWORD dwLength,
    LPSTR lpszName,
    LPGUID lpType,
    LPSTR * lppName)
{
    PWSAQUERYSETA pwsaq = (PWSAQUERYSETA)pResults;
    int err;
    HANDLE hRnR;
```

(continued)



(continued)

```
LPBLOB pvRet = 0;
INT Err = 0;

//
// create the query
//
memset(pwsaq, 0, sizeof(*pwsaq));
pwsaq->dwSize = sizeof(*pwsaq);
pwsaq->lpszServiceInstanceName = lpszName;
pwsaq->lpszServiceClassId = lpType;
pwsaq->dwNameSpace = NS_ALL;
pwsaq->dwNumberOfProtocols = 2;
pwsaq->lpaafpProtocols = &afp[0];

err = WSALookupServiceBeginA(pwsaq,
                             LUP_RETURN_BLOB | LUP_RETURN_NAME,
                             &hRnR);

if(err == NO_ERROR)
{
    //
    // The query was accepted, so execute it via the Next call.
    //
    err = WSALookupServiceNextA(
        hRnR,
        0,
        &dwLength,
        pwsaq);

    //
    // if NO_ERROR was returned and a BLOB is present, this
    // worked, just return the requested information. Otherwise,
    // invent an error or capture the transmitted one.
    //

    if(err == NO_ERROR)
    {
        if(pvRet = pwsaq->lplpBlob)
        {
            if(lppName)
            {
                *lppName = pwsaq->lpszServiceInstanceName;
            }
        }
        else
        {

```

```
        err = WSANO_DATA;
    }
}
else
{
    //
    // WSALookupServiceEnd clobbers LastError so save
    // it before closing the handle.
    //

    err = GetLastError();
}
WSALookupServiceEnd(hRnR);

//
// if an error happened, stash the value in LastError
//

if(err != NO_ERROR)
{
    SetLastError(err);
}
}
return(pvRet);
}
```

## Additional Windows Sockets 2 SPI Concerns

This section contains information on service provider ordering and the Windows Sockets SPI header file, `Ws2spi.h`.

### Service Provider Ordering

The order in which transport service providers are initially installed governs the order in which they are enumerated through **WSCEnumProtocols** at the service provider interface, or through **WSAEnumProtocols** at the application interface. More importantly, this order also governs the order in which protocols and service providers are considered when a client requests creation of a socket based on its address family, type, and protocol identifier. Windows Sockets 2 includes an applet called `Sporder.exe` that allows the catalog of installed protocols to be re-ordered interactively after protocols have already been installed. Windows Sockets 2 also includes an auxiliary .dll, `Sporder.dll`, that exports a procedural interface for re-ordering protocols. This procedural interface consists of a single procedure called **WSCWriteProviderOrder**.



The interface definition may be imported into a C or C++ program by means of the include file Sporder.h. The entry point may be linked by means of the lib file Sporder.lib. The actual procedure specification is given in the following section.

## Windows Sockets SPI Header File - Ws2spi.h

The function prototypes and structures in this document can be found in Ws2spi.h.

New versions of Ws2spi.h will appear periodically as new identifiers are allocated by the Windows Sockets Identifier Clearinghouse. The clearinghouse can be reached through the world wide web:

*<http://www.stardust.com/winsock/>*

Developers are urged to stay current with successive revisions of Ws2spi.h as they are made available by the clearinghouse.

## CHAPTER 11

# Winsock 2 SPI Reference

## Winsock 2 SPI Reference

### NSPCleanup

The **NSPCleanup** function terminates the use of a particular Windows Sockets name space service provider.

```
int NSPCleanup (  
    LPGUID IpProviderId  
);
```

#### Parameters

*IpProviderId*

[in] Pointer to the GUID of the name-space provider that is to be terminated.

#### Return Values

If no error occurs, **NSPCleanup** returns a value of **NO\_ERROR** (zero). Otherwise, **SOCKET\_ERROR** (-1) is returned and the provider must set the appropriate error code using **SetLastError**.

#### Remarks

The **NSPCleanup** function is called when an application is finished using a Windows Sockets name space service provider. The **NSPCleanup** function deregisters a particular name-space provider and allows the transport service provider to free any of the name-space provider's allocated resources.

The **NSPStartup** function must be called successfully before using any name-space providers. It is permissible to make more than one **NSPStartup** call. However, for each **NSPStartup** call, a corresponding **NSPCleanup** call must also be issued. Only the final **NSPCleanup** for the service provider does the actual cleanup; the preceding calls simply decrement an internal reference count in the service provider.

This function should not return until the name space service provider DLL can be unloaded from memory.

## Error Codes

Error code	Meaning
WSAEINVAL	The <i>IpProviderId</i> does not specify a valid provider.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to perform this operation.

## NSPGetServiceClassInfo

The **NSPGetServiceClassInfo** function retrieves all the pertinent class information (schema) pertaining to the name-space provider. This call retrieves any name space-specific information that is common to all instances of the service, including connection information for SAP, or port information for SAP or TCP.

```
int NSPGetServiceClassInfo (
    LPGUID                IpProviderId,
    OUT LPDWORD           IpdwBufSize,
    OUT LPWSASERVICECLASSINFOW IpServiceClassInfo
);
```

### Parameters

#### *IpProviderId*

[in] Pointer to the GUID of the specific name-space provider from which the service class schema is to be retrieved.

#### *IpdwBufSize*

[in] Number of bytes contained in the buffer pointed to by *IpServiceClassInfo* on input. Alternately, if the function fails and the error is **WSAEFAULT**, *IpdwBufSize* contains the minimum number of bytes to pass for the *IpServiceClassInfo* to retrieve the record on output.

#### *IpServiceClassInfo*

[in] Returns service class to name space-specific mapping information. The *IpServiceClassId* parameter must be filled in to indicate which **WSASERVICECLASSINFOW** record should be returned.

### Return Values

If no error occurs, the **NSPGetServiceClassInfo** function returns **NO\_ERROR** (zero). Otherwise, **SOCKET\_ERROR** (-1) is returned and it must set the appropriate error code using **SetLastError**.

## Remarks

The `W2_32.dll` uses this function to implement the **WSAGetServiceClassNameByClassId** function, as well as to retrieve the name space specific information passed into the **NSPLookupServiceBegin** and **NSPSetService**.

## Error Codes

Error code	Meaning
WSAEACCES	Calling routine does not have sufficient privileges to access the information.
WSAEFAULT	The <i>lpServiceClass</i> buffer was too small to contain a <b>WSASERVICECLASSINFOW</b> .
WSAEINVAL	Specified service class identifier or name-space—provider identifier is invalid.
WSATYPE_NOT_FOUND	Specified class was not found in any of the name spaces.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to perform this operation.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

## NSPInstallServiceClass

The **NSPInstallServiceClass** function registers service class schema within the name-space providers.

The schema includes the class name, class identifier, and any name space-specific type information that is common to all instances of the service, such as SAP identifier or object identifier. A dynamic name-space provider is expected to store any class information associated with that name space. Other name-space providers should do whatever makes sense.

```
int NSPInstallServiceClass (
    LPGUID                lpProviderId,
    LPWSASERVICECLASSINFOW lpServiceClassInfo
);
```

## Parameters

*lpProviderId*

[in] Pointer to the GUID of the specific name-space provider that this service class schema is being registered in.

*IpServiceClassInfo*

[in] Contains service class schema information.

**Return Values**

The function should return NO\_ERROR (zero) if the routine succeeds. It should return SOCKET\_ERROR (-1) if the routine fails and it must set the appropriate error code using **SetLastError**.

**Remarks**

Name space providers are encouraged but not required to store information that is specific to the name space they support.

**Error Codes**

<b>Error code</b>	<b>Meaning</b>
WSAEACCES	Calling routine does not have sufficient privileges to perform this operation.
WSAEALREADY	Service class information has already been registered for this service class identifier. To modify service class information, first use <b>NSPRemoveServiceClass</b> , then reinstall with updated class information data.
WSAEINVAL	Service class identifier was invalid or improperly structured.
WSA_INVALID_PARAMETER	Name space provider cannot supply the requested class information.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to perform this operation.

**! Requirements**

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

---

## NSPLookupServiceBegin

The **WSALookupServiceBegin** function initiates a client query that is constrained by the information contained within a **WSAQUERYSET** structure.

**WSALookupServiceBegin** only returns a handle, which should be used by subsequent calls to **WSALookupServiceNext** to get the actual results. Since this operation can not be canceled, it should be implemented to execute quickly. While it is acceptable to initiate a network query, this function should not require a response in order to return successfully.

```
int NSPLookupServiceBegin (
    LPGUID                IpProviderId,
    LPWSAQUERYSETW        IpqsRestrictions,
    LPWSASERVICECLASSINFOW IpServiceClassInfo,
    DWORD                 dwControlFlags,
    LPHANDLE              IphLookup
);
```

### Parameters

#### *IpProviderId*

[in] Contains the specific provider identifier that should be used for the query.

#### *IpqsRestrictions*

[in] Contains the search criteria. See the following for more information.

#### *IpServiceClassInfo*

[in] **WSASERVICECLASSINFOW** structure that contains all the schema information for the service.

#### *dwControlFlags*

[in] Controls the depth of the search.

Value	Meaning
LUP_DEEP	Query deep as opposed to just the first level.
LUP_CONTAINERS	Return containers only.
LUP_NOCONTAINERS	Do not return any containers.
LUP_FLUSHCACHE	If the provider has been caching information, ignore the cache and go query the name space itself.
LUP_FLUSHPREVIOUS	Used as a value for the <i>dwControlFlags</i> argument in <b>NSPLookupServiceNext</b> . Setting this flag instructs the provider to discard the last result set, which was too large for the supplied buffer, and move on to the next result set.
LUP_NEAREST	If possible, return results in the order of distance. The measure of distance is provider specific.
LUP_RES_RESERVICE	Indicates whether prime response is in the remote or local part of <b>CSADDR_INFO</b> structure. The other part needs to be usable in either case.

(continued)

(continued)

Value	Meaning
LUP_RETURN_ALIASES	Any available alias information is to be returned in successive calls to <b>NSPLookupServiceNext</b> , and each alias returned will have the <b>RESULT_IS_ALIAS</b> flag set.
LUP_RETURN_NAME	Retrieves the name as <i>lpzServiceInstanceName</i> .
LUP_RETURN_TYPE	Retrieves the type as <i>lpServiceClassId</i> .
LUP_RETURN_WSAVERSION	Retrieves the version as <i>lpVersion</i> .
LUP_RETURN_COMMENT	Retrieves the comment as <i>lpzComment</i> .
LUP_RETURN_ADDR	Retrieves the addresses as <i>lpcsaBuffer</i> .
LUP_RETURN_BLOB	Retrieves the private data as <i>lpBlob</i> .
LUP_RETURN_ALL	Retrieves all the information.

#### *lphLookup*

[out] Handle to be used in subsequent calls to **NSPLookupServiceNext** in order to retrieve the results set.

### Return Values

The function should return **NO\_ERROR** (zero) if the routine succeeds. It should return **SOCKET\_ERROR** (-1) if the routine fails and it must set the appropriate error code using **SetLastError**.

### Remarks

If **LUP\_CONTAINERS** is specified in a call, all other restriction values should be avoided. If any are supplied, it is up to the name service provider to decide if it can support this restriction over the containers. If it cannot, it should return an error.

Some name service providers may have other means of finding containers. For example, containers can all be of some well-known type, or of a set of well-known types, and therefore a query restriction could be created for finding them. No matter what other means the name service provider has for locating containers, **LUP\_CONTAINERS** and **LUP\_NOCONTAINERS** take precedence. Hence, if a query restriction is given that includes containers, specifying **LUP\_NOCONTAINERS** will prevent the container items from being returned. Similarly, no matter the query restriction, if **LUP\_CONTAINERS** is given, only containers should be returned. If a name space does not support containers and **LUP\_CONTAINERS** is specified, it should simply return **WSANO\_DATA**.

The preferred method of obtaining the containers within another container, is the call:

```
dwStatus = NSPLookupServiceBegin(
    lpqsRestrictions,
    LUP_CONTAINERS,
    lphLookup);
```

followed by the requisite number of **NSPLookupServiceNext** calls. This will return all containers contained immediately within the starting context; that is, it is not a deep query. With this, one can map the address space structure by walking the hierarchy, perhaps enumerating the content of selected containers. Subsequent uses of **NSPLookupServiceBegin** use the containers returned from a previous call.

### Forming Queries

As mentioned above, a **WSAQUERYSET** structure is used as an input parameter to **NSPLookupServiceBegin** in order to qualify the query. The following table indicates how the **WSAQUERYSET** is used to construct a query. When a member is marked as *(Optional)* a NULL pointer can be supplied, indicating that the parameter will *not* be used as a search criteria. See *Query-Related Data Structures* for additional information.

<b>WSAQUERYSET member name</b>	<b>Query interpretation</b>
<b>dwSize</b>	Will be set to sizeof( <b>WSAQUERYSET</b> ). This is a versioning mechanism.
<b>dwOutputFlags</b>	Ignored for queries.
<b>lpzServiceInstanceName</b>	<i>(Optional)</i> Referenced string contains service name. The semantics for wildcarding within the string are not defined, but can be supported by certain name-space providers.
<b>IpServiceClassId</b>	<i>(Required)</i> GUID corresponding to the service class.
<b>IpVersion</b>	<i>(Optional)</i> References desired version number and provides version comparison semantics (that is, version must match exactly, or version must be not less than the value supplied).
<b>lpzComment</b>	Ignored for queries.
<b>dwNameSpace</b>	Identifier of a single name space in which to constrain the search, or NS_ALL to include all name spaces.
<b>IpNSProviderId</b>	<i>(Optional)</i> References the GUID of a specific name-space provider and limits the query to this provider only.
<b>lpzContext</b>	<i>(Optional)</i> Specifies the starting point of the query in a hierarchical name space.
<b>dwNumberOfProtocols</b>	Size of the protocol constraint array, can be zero.
<b>lpafpProtocols</b>	<i>(Optional)</i> References an array of <b>AFPROTOCOLS</b> structure. Only services that utilize these protocols will be returned. It is legal for the value AF_UNSPEC to appear as a protocol family value, signifying a wildcard. Name space providers may supply information on any service that uses the corresponding protocol, regardless of address family.
<b>lpzQueryString</b>	<i>(Optional)</i> Some name spaces (such as whois++) support enriched SQL-like queries that are contained in a simple text string. This parameter is used to specify that string.

*(continued)*



*(continued)***WSAQUERYSET member name      Query interpretation**


---

<b>dwNumberOfCsAddrs</b>	Ignored for queries.
<b>lpcsaBuffer</b>	Ignored for queries.
<b>lpBlob</b>	<i>(Optional)</i> Pointer to a provider-specific entity.

**Error Codes****Error code****Meaning**


---

WSAEINVAL	One or more parameters were invalid for this provider or missing.
WSANO_DATA	Name was found in the database but it does not have the correct associated data being resolved for.
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to perform this operation.

**! Requirements****Version:** Requires Windows Sockets 2.0.**Header:** Declared in Ws2spi.h.

## NSPLookupServiceEnd

The **NSPLookupServiceEnd** function is called to free the handle after previous calls to **NSPLookupServiceBegin** and **NSPLookupServiceNext**.

It is possible to receive an **NSPLookupServiceEnd** call on another thread while processing an **NSPLookupServiceNext**. This indicates that the client has canceled the request and the provider should close the handle and return from the **NSPLookupServiceNext** call as well, setting the last error to **WSA\_E\_CANCELLED**.

```
int NSPLookupServiceEnd (
    HANDLE    hLookup
);
```

**Parameters***hLookup*[in] Handle previously obtained by calling **NSPLookupServiceBegin**.

## Return Values

The function should return `NO_ERROR` (zero) if the routine succeeds. It should return `SOCKET_ERROR` (-1) if the routine fails and it must set the appropriate error code using `SetLastError`.

## Remarks

In Windows Sockets 2, conflicting error codes are defined for `WSAECANCELLED` (10103) and `WSA_E_CANCELLED` (10111). The error code `WSAECANCELLED` will be removed in a future version and only `WSA_E_CANCELLED` will remain. Name Space Providers should switch to using the `WSA_E_CANCELLED` error code as soon as possible to maintain compatibility with the widest possible range of applications.

## Error Codes

Error code	Meaning
<code>WSA_INVALID_HANDLE</code>	Handle is not valid.
<code>WSA_NOT_ENOUGH_MEMORY</code>	Not enough free memory available to perform this operation.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

## NSPLookupServiceNext

The `NSPLookupServiceNext` function is called after obtaining a handle from a previous call to `NSPLookupServiceBegin` in order to retrieve the requested service information.

The provider will pass back a `WSAQUERYSET` structure in the `lpqsResults` buffer. The client should continue to call this function until it returns `WSA_E_NOMORE`, indicating that all the `WSAQUERYSET` have been returned.

```
int NSPAPI WSALookupServiceNext (
    HANDLE          hLookup,
    DWORD          dwControlFlags,
    LPDWORD        lpdwBufferLength,
    LPWSAQUERYSET  lpqsResults
);
```

## Parameters

*hLookup*

[in] Handle returned from the previous call to `WSALookupServiceBegin`.

*dwControlFlags*

[in] Flags to control the next operation. Currently only LUP\_FLUSHPREVIOUS is defined as a means to cope with a result set that is too large. If an application does not wish to (or cannot) supply a large enough buffer, setting LUP\_FLUSHPREVIOUS instructs the provider to discard the last result set, which was too large, and move to the next set for this call.

*lpdwBufferLength*

[in/out] On input, the number of bytes contained in the buffer pointed to by *lpqsResults*. On output, if the function fails and the error is WSAEFAULT, then it contains the minimum number of bytes to pass for the *lpqsResults* to retrieve the record.

*lpqsResults*

[out] Pointer to a block of memory that will contain one result set in a **WSAQUERYSET** structure on return.

**Return Values**

The function should return NO\_ERROR (zero) if the routine succeeds. It should return SOCKET\_ERROR (-1) if the routine fails and it must set the appropriate error code using **SetLastError**.

**Remarks**

The *dwControlFlags* specified in this function and the ones specified at the time of **NSPLookupServiceBegin** are treated as “restrictions” for the purpose of combination. The restrictions are combined between the ones at **NSPLookupServiceBegin** time and the ones at **NSPLookupServiceNext** time. Therefore, the flags at **NSPLookupServiceNext** can never increase the amount of data returned beyond what was requested at **NSPLookupServiceBegin**, although it is *not* an error to specify more or less flags. The flags specified at a given **NSPLookupServiceNext** apply only to that call.

The *dwControlFlags* LUP\_FLUSHPREVIOUS and LUP\_RES\_SERVICE are exceptions to the combined restrictions rule (because they are behavior flags instead of restriction flags). If either of these flags is used in **NSPLookupServiceNext**, they have their defined effect regardless of the setting of the same flags at **NSPLookupServiceBegin**.

For example, if LUP\_RETURN\_VERSION is specified at **NSPLookupServiceBegin**, the service provider retrieves records including the version. If LUP\_RETURN\_VERSION is *not* specified at **NSPLookupServiceNext**, the returned information does not include the version, even though it was available. No error is generated.

Also for example, if LUP\_RETURN\_BLOB is *not* specified at **NSPLookupServiceBegin** but is specified at **NSPLookupServiceNext**, the returned information does not include the private data. No error is generated.

## Query Results

The following table describes how the query results are represented in the **WSAQUERYSET** structure. Refer to *Query-Related Data Structures* for additional information.

<b>WSAQUERYSET member name</b>	<b>Result interpretation</b>
<b>dwSize</b>	Will be set to <code>sizeof(WSAQUERYSET)</code> . This is used as a versioning mechanism.
<b>dwOutputFlags</b>	<code>RESULT_IS_ALIAS</code> flag indicates this is an alias result.
<b>lpzServiceInstanceName</b>	References the string that contains the service name.
<b>lpServiceClassId</b>	GUID corresponding to the service class.
<b>lpVersion</b>	References version number of the particular service instance.
<b>lpzComment</b>	Optional comment string supplied by service instance.
<b>dwNameSpace</b>	Name space in which the service instance was found.
<b>lpNSProviderId</b>	Identifies the specific name-space provider that supplied this query result.
<b>lpzContext</b>	Specifies the context point in a hierarchical name space at which the service is located.
<b>dwNumberOfProtocols</b>	Undefined for results.
<b>lpafpProtocols</b>	Undefined for results, all needed protocol information is in the <b>CSADDR_INFO</b> structures.
<b>lpzQueryString</b>	When <i>dwControlFlags</i> includes <code>LUP_RETURN_QUERY_STRING</code> , this member returns the unparsed remainder of the <i>lpzServiceInstanceName</i> specified in the original query. For example, in a name space that identifies services by hierarchical names that specify a host name and a file path within that host, the address returned might be the host address and the unparsed remainder might be the file path. If the <i>lpzServiceInstanceName</i> is fully parsed and <code>LUP_RETURN_QUERY_STRING</code> is used, this member is <code>NULL</code> or points to a zero-length string.
<b>dwNumberOfCsAddrs</b>	Indicates the number of elements in the array of <b>CSADDR_INFO</b> structures.
<b>lpcsaBuffer</b>	Pointer to an array of <b>CSADDR_INFO</b> structures, with one complete transport address contained within each element.
<b>lpBlob</b>	(Optional) Pointer to a provider-specific entity.

## Error Codes

Error code	Meaning
WSA_E_NO_MORE	<p>There is no more data available.</p> <p>In Windows Sockets 2, conflicting error codes are defined for WSAENOMORE (10102) and WSA_E_NO_MORE (10110).The error code WSAENOMORE will be removed in a future version and only WSA_E_NO_MORE will remain. Name space providers should switch to using the WSA_E_NO_MORE error code as soon as possible to maintain compatibility with the widest possible range of applications.</p>
WSA_E_CANCELLED	<p>Call to <b>NSPLookupServiceEnd</b> was made while this call was still processing. The call has been canceled. The data in the <i>lpqsResults</i> buffer is undefined.</p> <p>In Windows Sockets 2, conflicting error codes are defined for WSAECANCELLED (10103) and WSA_E_CANCELLED (10111).The error code WSAECANCELLED will be removed in a future version and only WSA_E_CANCELLED will remain. Name space providers should switch to using the WSA_E_CANCELLED error code as soon as possible to maintain compatibility with the widest possible range of applications.</p>
WSAEFAULT	The <i>lpqsResults</i> buffer was too small to contain a <b>WSAQUERYSET</b> set.
WSAEINVAL	One or more parameters were invalid or missing for this provider.
WSA_INVALID_HANDLE	Specified lookup handle is invalid.
WSANO_DATA	The name was found in the database but no data matching the given restrictions was located.
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to perform this operation.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

# NSPRemoveServiceClass

The **NSPRemoveServiceClass** function permanently removes a specified service class from the name space.

```
int NSPRemoveServiceClass (
    LPGUID lpProviderId,
    LPGUID lpServiceClassId
);
```

## Parameters

*lpProviderId*

[in] Pointer to the GUID of the specific name-space provider that this service class schema is to be removed from.

*lpServiceClassId*

[in] Pointer to the GUID for the service class to remove.

## Return Values

The function should return **NO\_ERROR** (zero) if the routine succeeds. It should return **SOCKET\_ERROR** (-1) if the routine fails and it must set the appropriate error code using **SetLastError**.

## Error Codes

Error code	Meaning
WSATYPE_NOT_FOUND	Specified class was not found in any of the name spaces.
WSAEACCES	Calling routine does not have sufficient privileges to remove the Service.
WSA_INVALID_PARAMETER	Specified GUID was not valid.
WSAEINVAL	Specified service class identifier GUID was not valid.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to perform this operation.

## ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

# NSPSetService

The **NSPSetService** function registers or deregisters a service instance within a name space.

```
int NSPSetService (
    LPGUID                lpProviderId,
    LPWSASERVICECLASSINFO lpServiceClassInfo,
    LPWSAQUERYSETW        lpqsRegInfo,
    WSAESETSERVICEOP     essOperation,
    DWORD                 dwControlFlags
);
```

## Parameters

### *lpProviderId*

[in] Pointer to the GUID of the specific name-space provider in which the service is being registered.

### *lpServiceClassInfo*

[in] Contains service class schema information.

### *lpqsRegInfo*

[in] Specifies property information to be updated upon registration.

### *essOperation*

[in] Enumeration whose values include:

#### RNRSERVICE\_REGISTER

Register the service. For SAP, this means sending out a periodic broadcast.

This is an NOP for the DNS name space. For persistent data stores this means updating the address information.

#### RNRSERVICE\_DEREGISTER

Deregister the service. For SAP, this means stop sending out the periodic broadcast.

This is an NOP for the DNS name space. For persistent data stores this means deleting address information.

#### RNRSERVICE\_DELETE

Delete the service from dynamic name and persistent spaces. For services represented by multiple **CSADDR\_INFO** structures (using the **SERVICE\_MULTIPLE** flag), only the supplied address will be deleted, and this must match exactly the corresponding **CSADD\_INFO** structure that was supplied when the service was registered.

### *dwControlFlags*

[in] Set of control flags whose values include:

#### SERVICE\_MULTIPLE

Controls scope of operation. A register or deregister invalidates all existing addresses before adding the given address set. When set, the action is only performed on the given address set. A register does not invalidate existing addresses and a deregister only invalidates the given set of addresses.

The available values for *essOperation* and *dwControlFlags* combine to give meanings as shown in the following table.

Operation	Flags	Service already exists	Service does not exist
RNRSERVICE_REGISTER	None	Overwrites the object. Uses only addresses specified. Object is REGISTERED.	Creates a new object. Uses only addresses specified. Object is REGISTERED.
RNRSERVICE_REGISTER	SERVICE_MULTIPLE	Updates object. Adds new addresses to existing set. Object is REGISTERED.	Creates a new object. Uses all addresses specified. Object is REGISTERED.
RNRSERVICE_DEREGISTER	None	Removes all addresses, but does not remove object from name space. Object is DEREGISTERED.	WSASERVICE_NOT_FOUND
RNRSERVICE_DEREGISTER	SERVICE_MULTIPLE	Updates object. Removes only addresses that are specified. Only mark object as DEREGISTERED if no addresses are present. Does not remove from the name space.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	None	Removes object from the name space.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	SERVICE_MULTIPLE	Removes only addresses that are specified. Only removes object from the name space if no addresses remain.	WSASERVICE_NOT_FOUND

### Return Values

The function should return `NO_ERROR` (zero) if the routine succeeds. It should return `SOCKET_ERROR` (-1) if the routine fails and it must set the appropriate error code using `SetLastError`.

### Remarks

`SERVICE_MULTIPLE` lets an application manage its addresses independently. This is useful when the application wants to manage its protocols individually or when the service resides on more than one machine. For instance, when a service uses more than one protocol, it may find that one listening socket aborts but the others remain operational. In this case, the service could deregister the aborted address without affecting the other addresses.



When using `SERVICE_MULTIPLE`, an application must not let stale addresses remain in the object. This can happen if the application aborts without issuing a `DEREGISTER` request. When a service registers, it should store its addresses. On its next invocation, the service should explicitly deregister these old stale addresses before registering new addresses.

### Service Properties

The following table describes how service property data is represented in a `WSAQUERYSET` structure. Members labeled as *(Optional)* can be supplied with a `NULL` pointer.

#### WSAQUERYSET

member name	Service property description
<b>dwSize</b>	Must be set to <code>sizeof(WSAQUERYSET)</code> . This is a versioning mechanism.
<b>DwOuputFlags</b>	Not applicable and ignored.
<b>LpszServiceInstanceName</b>	Referenced string contains the service instance name.
<b>LpServiceClassId</b>	GUID corresponding to this service class.
<b>LpVersion</b>	<i>(Optional)</i> Supplies service instance version number.
<b>LpszComment</b>	<i>(Optional)</i> An optional comment string.
<b>DwNameSpace</b>	Ignored for this operation.
<b>LpNSProviderId</b>	Ignored for this operation, provider identifier is contained in the <i>IpProviderId</i> parameter.
<b>LpszContext</b>	<i>(Optional)</i> Specifies the starting point of the query in a hierarchical name space.
<b>DwNumberOfProtocols</b>	Ignored.
<b>LpafpProtocols</b>	Ignored.
<b>LpszQueryString</b>	Ignored.
<b>DwNumberOfCsAddr</b>	Number of elements in the array of <code>CSADDR_INFO</code> structures referenced by <i>lpcsaBuffer</i> .
<b>LpcsaBuffer</b>	Pointer to an array of <code>CSADDR_INFO</code> structures that contain the address[es] that the service is listening on.
<b>LpBlob</b>	<i>(Optional)</i> Pointer to a provider-specific entity.

---

**Note** It is acceptable for the `iProtocol` member of the `CSADDR_INFO` structure to contain the manifest constant `IPROTOCOL_ANY`, signifying a wildcard value. The name-space provider should substitute a value that is reasonable for the given address family and socket type.

---

## Error Codes

Error code	Meaning
WSAEACCES	Calling routine does not have sufficient privileges to install the service.
WSAEINVAL	One or more parameters were invalid or missing for this provider.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to perform this operation.
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

## NSPStartup

The **NSPStartup** function retrieves the dynamic information about a provider, such as the list of the DLL entry points.

This function is called by the client upon initialization. **NSPStartup** and **NSPCleanup** must be called as pairs. All the NSP functions must be called from within an **NSPStartup/NSPCleanup** pair. WSC functions do not need to be called from within an **NSPStartup/NSPCleanup** pair either.

```
int NSPStartup (
    LPGUID          lpProviderId,
    LPNSP_ROUTINE  lpnspRoutines
);
```

### Parameters

*lpProviderId*

[in] Indicates the desired provider for which to return the entry points.

*lpnspRoutines*

[out] Pointer to all the provider entry points.

### Data Types

The following data types are needed for this call.

**NSP\_ROUTINE**

The **NSP\_ROUTINE** structure contains information regarding all the functions implemented by a given provider.

```

typedef struct _NSP_ROUTINE {
    DWORD      cbSize;
    DWORD      dwMajorVersion;
    DWORD      dwMinorVersion;
    INT ( *NSPCleanup) ;
    INT ( *NSPLookupServiceBegin) ;
    INT ( *NSPLookupServiceNext) ;
    INT ( *NSPLookupServiceEnd) ;
    INT ( *NSPSetService) ;
    INT ( *NSPInstallServiceClass) ;
    INT ( *NSPRemoveServiceClass) ;
    INT ( *NSPGetServiceClassInfo) ;
} NSP_ROUTINE, *PNSP_ROUTINE, *LPNSP_ROUTINE;

```

**cbSize**

Size of this structure.

**dwMajorVersion**

Major version of the service provider specification supported by this provider.

**dwMinorVersion**

Minor version of the service provider specification supported by this provider.

**\*NSP\*.\***

Pointers to the various NSP functions. Every entry must point to a valid function. If the provider does not implement this function, it should simply return WSAENOTIMPLEMENTED.

---

**Note** In the header file this structure contains complete prototypes for all the **NSP** pointers.

---

**Return Values**

The function should return NO\_ERROR (zero) if the routine succeeds. It should return SOCKET\_ERROR (-1) if the routine fails and it must set the appropriate error code using **SetLastError**.

**Error Codes****Error Code****Meaning**

WSAEINVAL

One or more parameters were invalid or missing for this provider.

WSA\_NOT\_ENOUGH\_MEMORY

Not enough free memory available to perform this operation.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

## WPUCloseEvent

The **WPUCloseEvent** function closes an open event object handle.

```
BOOL WPUCloseEvent (  
    WSAEVENT    hEvent,  
    LPINT       lpErrno  
);
```

### Parameters

*hEvent*

[in] Handle to an open event object.

*lpErrno*

[out] Pointer to the error code.

### Return Values

If the function succeeds, the return value is TRUE. Otherwise, the return value is FALSE and a specific error code is available in *lpErrno*.

### Error Codes

Error code	Meaning
WSA_INVALID_HANDLE	The <i>hEvent</i> is not a valid event object handle.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

**+** See Also

**WPUCreateEvent**

## WPUCloseSocketHandle

The **WPUCloseSocketHandle** function closes an existing socket handle.

```
int WPUCloseSocketHandle (
    SOCKET    s,
    LPINT     lpErrno
);
```

### Parameters

*s*

[in] Handle to socket created with **WPUCreateSocketHandle**.

*lpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, **WPUCreateSocketHandle** returns zero. Otherwise, it returns `SOCKET_ERROR`, and a specific error code is available in *lpErrno*.

### Remarks

The **WPUCloseSocketHandle** function closes an existing socket handle created by **WPUCreateSocketHandle**. This function removes the socket from `Ws2_32.dll`'s internal socket table. The owning service provider is responsible for releasing any resources associated with the socket.

### Error Codes

Error code	Meaning
WSAENOTSOCK	Descriptor is not a socket created by <b>WPUCreateSocketHandle</b> .

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

#### + See Also

**WPUCreateSocketHandle**

## WPUCloseThread

The **WPUCloseThread** function closes a thread opened with a call to **WPUOpenCurrentThread**.

```
INT WSPAPI WPUCloseThread (
    LPWSATHREADID lpThreadId,
    LPINT         lpErrno
);
```

## Parameters

### *lpThreadId*

[in] Pointer to a **WSATHREADID** structure that identifies the thread context. This structure must have been initialized by a previous call to **WPUOpenCurrentThread**.

### *lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WPUOpenCurrentThread** returns zero. Otherwise, it returns **SOCKET\_ERROR**, and a specific error code is available in *lpErrno*.

## Remarks

The **WPUCloseThread** function is used in a layered service provider to deallocate the resources that were initiated in a call by the **WPUOpenCurrentThread** function.

The **WSATHREADID** structure in the *lpThreadId* is the thread to deallocate.

Every call to **WPUOpenCurrentThread** must have a call to **WPUCloseThread**. These two functions are used when the overlapped functions, such as **WSPSend**, are called in a lower layer of the service provider than the current thread.

## Error Codes

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSPStartup</b> call must occur before using this function.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

### + See Also

**WPUCloseThread**, **WPUOpenCurrentThread**

# WPUCompleteOverlappedRequest

The **WPUCompleteOverlappedRequest** function performs overlapped I/O completion notification for overlapped I/O operations.

```
WSAEVENT WPUCompleteOverlappedRequest (
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
```

(continued)

(continued)

DWORD	<i>dwError</i> ,
DWORD	<i>cbTransferred</i> ,
LPINT	<i>lpErrno</i>
);	

## Parameters

*s*

[in] Service provider socket created by **WPUCreateSocketHandle**.

*lpOverlapped*

[in] Pointer to a **WSAOVERLAPPED** structure associated with the overlapped I/O operation whose completion is to be notified.

*dwError*

[in] Completion status of the overlapped I/O operation whose completion is to be notified.

*cbTransferred*

[in] Number of bytes transferred to or from client buffers (the direction of the transfer depends on the send or receive nature of the overlapped I/O operation whose completion is to be notified).

*lpErrno*

[out] Pointer to the error code resulting from execution of this function.

## Return Values

If no error occurs, **WPUCompleteOverlappedRequest** returns zero and notifies completion of the overlapped I/O operation according to the mechanism selected by the client (signals an event found in the **WSAOVERLAPPED** structure referenced by *lpOverlapped* and/or queues a completion status report to the completion port associated with the socket if a completion port is associated). Otherwise, **WPUCompleteOverlappedRequest** returns **SOCKET\_ERROR**, and a specific error code is available in *lpErrno*.

## Remarks

The **WPUCompleteOverlappedRequest** function performs overlapped I/O completion notification for overlapped I/O operations where the client-specified completion mechanism is something other than user mode—asynchronous procedure call (APC). This function can only be used for socket handles created by **WPUCreateSocketHandle**.

---

**Note** This function is different from other functions with the **WPU** prefix in that it is not accessed through the upcall table. Instead, it is exported directly by *Ws2\_32.dll*. Service providers that need to call this function should link with *WS2\_32.lib* or use appropriate operating system functions such as **LoadLibrary** and **GetProcAddress** to retrieve the function pointer.

---

The function **WPUCompleteOverlappedRequest** is used by service providers that do not implement Installable File System (IFS) functionality directly for the socket handles they expose. It performs completion notification for any overlapped I/O request for which the specified completion notification is other than a user-mode APC.

**WPUCompleteOverlappedRequest** is supported only for the socket handles created by **WPUCreateSocketHandle** and not for sockets created by a service provider directly.

If the client selects a user-mode APC as the notification method, the service provider should use **WPUQueueApc** or another appropriate operating system function to perform the completion notification. If user-mode APC is not selected by the client, a service provider that does not implement IFS functionality directly cannot determine whether or not the client has associated a completion port with the socket handle. Thus, it cannot determine whether the completion notification method should be queuing a completion status record to a completion port or signaling an event found in the

**WSAOVERLAPPED** structure. The Windows Socket 2 architecture keeps track of any completion port association with a socket created by **WPUCreateSocketHandle** and can make a correct decision between completion port-based notification or event-based notification.

When **WPUCompleteOverlappedRequest** queues a completion indication, it sets the **InternalHigh** member of the **WSAOVERLAPPED** structure to the count of bytes transferred. Then, it sets the **Internal** member to some OS-dependent value other than the special value **WSS\_OPERATION\_IN\_PROGRESS**. There may be some slight delay after **WPUCompleteOverlappedRequest** returns before these values appear, since processing may occur asynchronously. However, the **InternalHigh** value (byte count) is guaranteed to be set by the time **Internal** is set.

**WPUCompleteOverlappedRequest** is available both on Windows® 95/98 and Windows NT®/Windows® 2000. It works as stated (performs the completion notification as requested by the client) whether or not the socket handle has been associated with a completion port.

### Interaction with **WSPGetOverlappedResult**

The behavior of **WPUCompleteOverlappedRequest** places some constraints on how a service provider implements **WSPGetOverlappedResult** since only the **Offset** and **OffsetHigh** members of the **WSAOVERLAPPED** structure are exclusively controlled by the service provider, yet three values (byte count, flags, and error) must be retrieved from the structure by **WSPGetOverlappedResult**. A service provider may accomplish this any way it chooses as long as it interacts with the behavior of **WPUCompleteOverlappedRequest** properly. However, a typical implementation is as follows:

- At the start of overlapped processing, the service provider sets **Internal** to **WSS\_OPERATION\_IN\_PROGRESS**.



- When the I/O operation has been completed, the provider sets **OffsetHigh** to the Windows Socket 2 error code resulting from the operation, sets **Offset** to the flags resulting from the I/O operation, and calls **WPUCompleteOverlappedRequest**, passing the transfer byte count as one of the parameters. **WPUCompleteOverlappedRequest** eventually sets **InternalHigh** to the transfer byte count, then sets **Internal** to a value other than **WSS\_OPERATION\_IN\_PROGRESS**.
- When **WSPGetOverlappedResult** is called, the service provider checks **Internal**. If it is **WSS\_OPERATION\_IN\_PROGRESS**, the provider waits on the event handle in the **hEvent** member or returns an error, based on the setting of the **FWAIT** flag of **WSPGetOverlappedResult**. If not in progress, or after completion of waiting, the provider returns the values from **InternalHigh**, **OffsetHigh**, and **Offset** as the transfer count, operation result error code, and flags respectively.

### Error Codes

Error code	Meaning
WSAEINVAL	Socket is not a socket created by <b>WPUCreateSocketHandle</b> .

**+** See Also

**WSPGetOverlappedResult**, **WPUCreateSocketHandle**, **WPUQueueApc**

## WPUCreateEvent

The **WPUCreateEvent** function creates a new event object.

```
WSAEVENT WPUCreateEvent (
    LPINT lpErrno
);
```

### Parameters

*lpErrno*  
[out] Pointer to the error code.

### Return Values

If no error occurs, **WPUCreateEvent** function returns the handle of the event object.

Otherwise, the return value is **WSA\_INVALID\_EVENT** and a specific error code is available in *lpErrno*.

### Remarks

The event object created by this function is manual reset with an initial state of nonsignaled. If a Win32 service provider wants auto reset events, it can call the Win32 **CreateEvent** function directly. For more information, see **CreateEvent**.

## Error Codes

Error code	Meaning
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to create the event object.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

### + See Also

**WPUCloseEvent**

# WPUCreateSocketHandle

The **WPUCreateSocketHandle** function creates a new socket handle.

```
SOCKET WPUCreateSocketHandle (  
    DWORD    dwCatalogEntryId,  
    DWORD    dwContext,  
    LPINT    lpErrno  
);
```

## Parameters

*dwCatalogEntryId*

[in] Descriptor identifying the calling service provider.

*dwContext*

[in] Context value to associate with the new socket handle.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WPUCreateSocketHandle** returns the new socket handle. Otherwise, it returns `INVALID_SOCKET`, and a specific error code is available in *lpErrno*.

## Remarks

The **WPUCreateSocketHandle** function creates a new socket handle for the specified provider. The handles created by **WPUCreateSocketHandle** are indistinguishable from true file system handles. This is significant in two respects. First, the Windows Socket 2 architecture takes care of redirecting the file system functions **ReadFile** and **WriteFile** to this service provider's **WSPRecv** and **WSPSend** functions, respectively. Second,

in operating systems that support completion ports, the Windows Sockets 2 architecture supports associating a completion port with the socket handle and using it to report overlapped I/O completion.

---

**Note** That the mechanism for redirecting **ReadFile** and **WriteFile** necessarily involves a user-to-kernel transition to get to the redirector, followed by a kernel-to-user transition to get to **WSPRecv** or **WSPSend**. On return, these transitions are retraced in reverse. This can be a significant performance penalty. Any service provider that uses **WPUCreateSocketHandle** to create its socket handles should not set **XP1\_IFS\_HANDLES** in its **WSAPROTOCOL\_INFOW** structure. Clients should take the absence of **XP1\_IFS\_HANDLES** as guidance to avoid the use of **ReadFile** and **WriteFile**.

There is no exceptional performance penalty for using the completion port mechanism with socket handles created with **WPUCreateSocketHandle**. A service provider should use **WPUCompleteOverlappedRequest** to announce completion of overlapped I/O operations that may involve a completion port. Clients may freely use operating system functions to create, associate, and use a completion port for completion notification (for example, **CreateloCompletionPort**, **GetQueuedCompletionStatus**, see relevant OS documentation for details). Note that completion ports are not integrated with the other asynchronous notification mechanisms offered by Windows Sockets 2. That is, a client can do a multiple-wait that includes multiple events and completion callbacks, but there is no predefined way for the multiple-wait to include completion ports.

---

### Layered Service Provider Considerations

This procedure is of particular interest to Layered Service Providers. A layered service provider may use this procedure, instead of **WPUModifyIFSHandle** to create the socket handles it exposes to its client. The advantage of using this procedure is that all I/O requests involving the socket can be guaranteed to go through this service provider. This is true even if the client assumes that the sockets are file system handles and calls the file system functions **ReadFile** and **WriteFile** (although it would pay a performance penalty for this assumption).

The guarantee that all I/O goes through this layer is a requirement for layers that need to process the I/O stream either before or after the actual I/O operation. Creating socket handles using **WPUCreateSocketHandle** and specifying an appropriate service provider interface procedure dispatch table at the time of **WSPStartup** makes sure the layer has the chance to get involved in starting each I/O operation. When the client requests overlapped I/O operations, this service provider layer will usually have to arrange to get into the path of I/O completion notification as well.

To see why this is true, consider what happens if the client associates a completion port with the socket handle for the purpose of overlapped I/O completion notification. The port is associated with the socket handle exposed by this layer, not the next layer's socket handle. There is no way for this layer to determine if a completion port has been

associated or what the port is. When this layer calls the next layer's I/O operation, it uses the next layer's socket handle. The next layer's socket handle will *not* have the same completion port association. The client's expected completion-port notification will *not* happen without some extra help.

The usual way a layered service provider takes care of this is to substitute a different overlapped I/O structure and different overlapped I/O parameters when invoking an I/O operation in the next layer. The substitute overlapped I/O structure references the client's stored overlapped structure and parameters. The invocation of the next layer sets up a callback notification. When the callback notification occurs, this layer performs any post-processing desired, retrieves the overlapped I/O information it stored on behalf of the client, discards the substitute structures, and forwards an appropriate completion notification to the client.

## Error Codes

Error code	Meaning
WSAENOBUFFS	Not enough buffers available, too many sockets.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

### + See Also

**WPUCloseSocketHandle**, **WPUQuerySocketHandleContext**, **WPUModifyIFSHandle**, **WPUCompleteOverlappedRequest**

## WPUFDIsSet

The **WPUFDIsSet** function checks the membership of the specified socket handle.

```
int WPUFDIsSet (
    SOCKET          s,
    FD_SET FAR     *set
);
```

### Parameters

*s*

[in] Descriptor identifying the socket.

*set*

[in] Set to check for the membership of socket *s*.

## Return Values

If no error occurs, a value of nonzero is returned denoting that socket *s* is a member of the *set*. Otherwise, the return value is zero.

**+** See Also

**WSPSelect**

---

# WPUGetProviderPath

The **WPUGetProviderPath** function retrieves the DLL path for the specified provider.

```
int WPUGetProviderPath (
    LPGUID    IpProviderId,
    LPWSTR    IpszProviderDllPath,
    LPINT     IpProviderDllPathLen,
    LPINT     IpErrno
);
```

## Parameters

*IpProviderId*

[in] Locally unique identifier of the provider. This must be a value obtained by using **WSCEnumProtocols**.

*IpszProviderDllPath*

[out] Pointer to a buffer containing a string that identifies the provider DLL's path. This path is a null-terminated string and any embedded environment strings (such as %SystemRoot%) have *not* been expanded.

*IpProviderDllPathLen*

[in/out] Size of the buffer pointed to by *IpszProviderDllPath*.

*IpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WPUGetProviderPath** returns zero. Otherwise, it returns **SOCKET\_ERROR**, and a specific error code is available in *IpErrno*.

## Remarks

The **WPUGetProviderPath** function retrieves the DLL path for the specified provider. The DLL path is null-terminated and can contain embedded environment strings (such as %SystemRoot%). Thus, the string should be expanded prior to being used with **LoadLibrary**. For more information, see **LoadLibrary**.

## Error Codes

Error code	Meaning
WSAEINVAL	The <i>IpProviderId</i> does not specify a valid provider.
WSAEFAULT	Either <i>IpszProviderDllPath</i> or <i>IpErrno</i> is not in a valid part of the user address space, or <i>IpProviderDllPathLen</i> is too small.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

### + See Also

**WSCInstallProvider**, **WSEnumProtocols**

## WPUModifyIFSHandle

The **WPUModifyIFSHandle** function receives a (possibly) modified IFS handle from *Ws2\_32.dll*.

```
SOCKET WPUModifyIFSHandle (
    DWORD      dwCatalogEntryId,
    SOCKET     ProposedHandle,
    LPINT      IpErrno
);
```

### Parameters

*dwCatalogEntryId*

[in] Descriptor identifying the calling service provider.

*ProposedHandle*

[in] IFS handle allocated by the provider.

*IpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, **WPUModifyIFSHandle** returns the modified socket handle. Otherwise, it returns `INVALID_SOCKET`, and a specific error code is available in *IpErrno*.

## Remarks

The **WPUModifyIFSHandle** handle allows the `Ws2_32.dll` to streamline its internal operations by returning a possibly modified version of the supplied IFS handle. It is guaranteed that the returned handle is indistinguishable from the proposed handle as far as the operating system is concerned. IFS providers *must* call this function before returning any newly created socket descriptor to a service provider. The service provider will only use the modified handle for any subsequent socket operations. This routine is only used by IFS providers whose socket descriptors are real IFS handles.

## Layered Service Provider Considerations

This procedure may also be used by a layered provider that is layered on top of a base IFS provider and wants to expose the base provider's socket handles as its own instead of creating them with a **WPUCreateSocketHandle** call. A layered provider that wishes to "pass through" the IFS socket handles it receives from the next layer in the chain can call **WPUModifyIFSHandle**, passing its own catalog entry ID as *dwCatalogEntryId*. This informs the Windows Sockets DLL that this layer, and not the next layer, should be the target for SPI calls involving that socket handle.

There are several limitations a layered provider should observe if it takes this approach.

- The provider should expose base provider entry points for **WSPSend** and **WSPRecv** in the procedure dispatch table it returns at the time of **WSPStartup** to make sure the Windows Sockets SPI client's access to these functions is as efficient as possible.
- The provider cannot rely on its **WSPSend** and **WSPRecv** functions being invoked for all I/O, particularly in the case of the I/O system functions **ReadFile** and **WriteFile**. These functions would bypass the layered provider and invoke the base IFS provider's implementation directly even if the layered provider puts its own entry points for these functions into the procedure dispatch table.
- The provider cannot rely on any ability to post-process overlapped I/O using **WSPSend**, **WSPSendTo**, **WSPRecv**, **WSPRecvFrom**, or **WSPIoctl**. Post-processing notification may happen through completion ports and bypass the layered provider entirely. A layered provider has no way to determine that a completion port was used or determine what port it is. The layered provider has no way to insert itself into the notification sequence.
- The provider should pass through all overlapped I/O requests directly to the base provider using the original overlapped parameters (for example, the **WSAOVERLAPPED** structure and completion routine pointer). The provider should expose the base provider entry point for **WSPGetOverlappedResult**. Since some overlapped I/O requests can bypass the layered provider completely, the layered provider cannot reliably mark **WSAOVERLAPPED** structures to determine which ones it can report results for, and which ones would have to be passed through to the underlying provider's **WSPGetOverlappedResult**. This effectively means that **WSPIoctl** has to be a pass-through operation to the underlying provider.

## Error Codes

Error code	Meaning
WSAEINVAL	Proposed handle is invalid.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

### + See Also

**WPUCreateSocketHandle**

---

# WPUOpenCurrentThread

The **WPUOpenCurrentThread** function opens a handle to the current thread that can be used with overlapped functions in a layered service provider. This is intended to be used by layered service providers that wish to initiate overlapped I/O from nonapplication threads.

```
INT WPUOpenCurrentThread (  
    LPWSATHREADID  lpThreadId,  
    LPINT          lpErrno  
);
```

## Parameters

*lpThreadId*

[out] Pointer to a **WSATHREADID** structure that can then be passed to an overlapped function.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WPUOpenCurrentThread** returns the zero. Otherwise, it returns `SOCKET_ERROR`, and a specific error code is available in *lpErrno*.

## Remarks

The **WPUOpenCurrentThread** function provides a pointer to a **WSATHREADID** structure that can then be passed to an overlapped function such as **WSPSend** or **WSPRecv**. Layered service providers that use a private thread in one of the upper layers will use **WPUOpenCurrentThread** to pass a **WSATHREADID** pointer to the lower layer that is administering overlapped functions.



Overlapped functions such as **WSPSend** and **WSPRecv** can then be used in the same way as a regular service provider.

Every call to **WPUOpenCurrentThread** must have a corresponding call to **WPUCloseThread**.

## Error Codes

Error code	Meaning
WSANOTINITIALISED	A successful <b>WSPStartup</b> call must occur before using this function.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

### + See Also

**WPUCloseThread**, **WSPSend**, **WSPRecv**, **WPUOpenCurrentThread**

## WPUPostMessage

The **WPUPostMessage** function performs the standard Win32 **PostMessage** function in a way that maintains backward compatibility with older versions of *Wsock32.dll*.

```

BOOL WPUPostMessage (
    HWND      hWnd,
    UINT      Msg,
    WPARAM    wParam,
    LPARAM    lParam
);

```

### Parameters

*hWnd*

[in] Handle to the window that will receive the message.

*Msg*

[in] Message that will be posted.

*wParam*

[in] First parameter containing additional message-specific information.

*lParam*

[in] Second parameter containing additional message-specific information.

## Return Values

If no error occurs, **WPUPostMessage** returns the TRUE value. Otherwise, the FALSE value is returned.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

## WPUQueryBlockingCallback

The **WPUQueryBlockingCallback** function returns a pointer to a callback function the service provider should invoke periodically while servicing blocking operations.

```
int WPUQueryBlockingCallback (  
    DWORD                dwCatalogEntryId,  
    LPBLOCKINGCALLBACK FAR *lpfpfnCallback,  
    LPDWORD              lpdwContext,  
    LPINT                lpErrno  
);
```

### Parameters

*dwCatalogEntryId*

[in] Descriptor identifying the calling service provider.

*lpfpfnCallback*

[out] Pointer that receives the blocking callback function.

*lpdwContext*

[out] Pointer that receives a context value that the service provider must pass into the blocking callback.

*lpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, **WPUQueryBlockingCallback** returns zero and stores a pointer to a blocking callback function in *lpfnCallback* and an associated context value in *lpdwContext*. Otherwise, it returns SOCKET\_ERROR, and a specific error code is available in *lpErrno*.

### Remarks

The **WPUQueryBlockingCallback** function returns a pointer to a callback function in *lpfnCallback* to be invoked periodically during blocking operations. This function also returns a context value in *lpdwContext* to be passed into the blocking callback.

Under Win32, this function can return NULL in *lpfnCallback*, indicating that no user defined-blocking hook is installed. In this case, the service provider should use the native Win32 synchronization objects to implement blocking.

**LPBLOCKINGCALLBACK** is defined as follows:

```
typedef BOOL ( CALLBACK FAR * LPBLOCKINGCALLBACK )( DWORD dwContext );
```

The blocking callback will return TRUE if the service provider is to continue waiting for the blocking operation to complete. It will return FALSE if the blocking operation has been canceled with the **WSPCancelBlockingCall**.

Any missing components of the address will default to a reasonable value if possible. For example, a missing port number will default to zero.

## Error Codes

Error code	Meaning
WSAEFAULT	The <i>lpfnCallback</i> or the <i>lpdwContext</i> argument is not a valid part of the process address space.
WSAEINVAL	The <i>dwCatalogEntryId</i> is invalid.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

### + See Also

**WSPCancelBlockingCall**

# WPUQuerySocketHandleContext

The **WPUQuerySocketHandleContext** function queries the context value associated with the specified socket handle.

```
int WPUQuerySocketHandleContext (
    SOCKET      s,
    LPDWORD     lpContext,
    LPINT       lpErrno
);
```

## Parameters

*s*

[in] Descriptor identifying the socket whose context is to be queried.

*lpContext*

[out] Pointer that will receive the context value.

*lpErrno*

[out] Pointer to the error code.

**Return Values**

If no error occurs, **WPUQuerySocketHandleContext** returns zero and stores the current context value in *lpContext*. Otherwise, it returns `SOCKET_ERROR`, and a specific error code is available in *lpErrno*.

**Remarks**

The **WPUQuerySocketHandleContext** function queries the current context value associated with the specified socket handle. Service providers typically use this function to retrieve a pointer to provider-specific data associated with the socket. For example, a service provider can use the socket context to store a pointer to a structure containing the socket's state, local and remote transport addresses, and event objects for signaling network events.

This function is only used by non-IFS providers since IFS providers are not able to supply a context value.

**Error Codes**

Error code	Meaning
WSAENOTSOCK	Descriptor is not a socket created by <b>WPUCreateSocketHandle</b> .

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

**+** See Also

**WPUCreateSocketHandle**

## WPUQueueApc

The **WPUQueueApc** function queues a user mode-asynchronous procedure call (APC) to the specified thread in order to facilitate invocation of overlapped I/O completion routines.

```
int WPUQueueApc(
    LPWSATHREADID lpThreadId,
```

(continued)

(continued)

```
LPWSAUSERAPC    lpfnUserApc,
DWORD           dwContext,
LPINT           lpErrno
);
```

## Parameters

*lpThreadId*

[in] Pointer to a **WSATHREADID** structure that identifies the thread context. A pointer to this structure is supplied to the service provider by the *Ws2\_32.dll* as an input parameter to an overlapped operation. The provider should store the **WSATHREADID** structure locally and provide a pointer to this local store. The local copy of **WSATHREADID** is no longer needed once **WPUQueueApc** returns.

*lpfnUserApc*

[in] Points to the APC function to be called.

*dwContext*

[in] 32-bit context value that is subsequently supplied as an input parameter to the APC function.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WPUQueueApc** returns zero and queues the completion routine for the specified thread. Otherwise, it returns **SOCKET\_ERROR**, and a specific error code is available in *lpErrno*.

## Remarks

This function queues an APC function against the specified thread. Under Win32, this will be done using a user mode—asynchronous procedure call (APC). The APC will only execute when the specified thread is blocked in an alertable wait. For Win16, a callback will be made directly. In Win16 environments, this call is safe for use within an interrupt context.

**LPWSAUSERAPC** is defined as follows:

```
typedef void ( CALLBACK FAR * LPWSAUSERAPC )( DWORD dwContext );
```

Because the APC mechanism supports only a single 32-bit context value, *lpfnUserApc* itself cannot be the client specified—completion routine, which involves more parameters. The service provider must instead supply a pointer to its own APC function that uses the supplied *dwContext* value to access the needed result information for the overlapped operation, and then invokes the client specified—completion routine.

For service providers where a user-mode component implements overlapped I/O, a typical usage of the APC mechanism is as follows.

1. When the I/O operation completes, the provider allocates a small buffer and packs it with a pointer to the client-supplied completion procedure and parameter values to pass to the procedure.
2. It queues an APC, specifying the pointer to the buffer as the *dwContext* value and its own intermediate procedure as the target procedure *lpfnUserApc*.
3. When the target thread eventually enters alertable wait state, the service provider's intermediate procedure is called in the proper thread context.
4. The intermediate procedure simply unpacks parameters, deallocates the buffer, and calls the client-supplied completion procedure.

## Error Codes

Error code	Meaning
WSAEFAULT	The <i>dwThreadId</i> does not specify a valid thread.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

### + See Also

**WSPSend, WSPSendTo, WSPRecv, WSPRecvFrom, WSPIoctl**

## WPUResetEvent

The **WPUResetEvent** function resets the state of the specified event object to nonsignaled. In Win16 environments, this call is safe for use within interrupt context.

```

BOOL WPUResetEvent(
    WSAEVENT    hEvent,
    LPINT       lpErrno
);

```

### Parameters

*hEvent*

[in] Handle that identifies an open event object.

*lpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, the **WPUResetEvent** function returns the value TRUE. Otherwise, FALSE is returned, and a specific error code is available in *lpErrno*.

## Error Codes

Error code	Meaning
WSA_INVALID_HANDLE	The <i>hEvent</i> is not a valid event object handle.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

### + See Also

**WPUCreateEvent**, **WPUSetEvent**, **WPUCloseEvent**

## WPUSetEvent

The **WPUSetEvent** function sets the state of the specified event object to signaled. In Win16 environments, this call is safe for use within interrupt context.

```
BOOL WPUSetEvent (
    WSAEVENT  hEvent,
    LPINT     lpErrno
);
```

### Parameters

*hEvent*

[in] Handle that identifies an open event object.

*lpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, the **WPUSetEvent** function returns the value TRUE. Otherwise, FALSE is returned, and a specific error code is available in *lpErrno*.

## Error Codes

Error code	Meaning
ERROR_INVALID_HANDLE	The <i>hEvent</i> is not a valid event object handle.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

**+** See Also**WPUCreateEvent, WPUResetEvent, WPUCloseEvent**

## WSCDeinstallProvider

The **WSCDeinstallProvider** function removes the specified transport provider from the system configuration database.

```
int WSCDeinstallProvider (
    LPGUID    IpProviderId,
    LPINT     IpErrno
);
```

### Parameters

*IpProviderId*

[in] Globally unique identifier of the provider to deinstall. This value is stored within each **WSAPROTOCOL\_INFOW** structure.

*IpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, **WSCDeinstallProvider** returns zero. Otherwise, it returns **SOCKET\_ERROR**, and a specific error code is available in *IpErrno*.

### Remarks

The **WSCDeinstallProvider** function removes the common Windows Sockets 2 configuration information for the specified provider. After this routine completes successfully, the configuration information stored in the registry will be changed. However, any *Ws2\_32.dll* instances currently in memory will not be able to see this change.

The caller of this function must remove any additional files or service provider-specific configuration information that is needed to completely de-install the service provider.

### Error Codes

Error code	Meaning
WSAEINVAL	The <i>IpProviderId</i> does not specify a valid provider.
WSAEFAULT	The <i>IpErrno</i> is not in a valid part of the user address space.



**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

**WSCInstallProvider, WSCEnumProtocols**

---

## WSCEnableNSProvider

The **WSCEnableNSProvider** function changes the state of a given name-space provider. It is intended to give the end user the ability to change the state of the name-space providers through Control Panel.

```
int WSCEnableNSProvider (  
    LPGUID    IpProviderId,  
    BOOL      fEnable  
);
```

### Parameters

*IpProviderId*

[in] Locally unique identifier for this provider.

*fEnable*

[in] Boolean value that, if TRUE, the provider is set to the active state. If FALSE, the provider is disabled and will not be available for query operations or service registration.

### Return Values

If no error occurs, the **WSCEnableNSProvider** function returns NO\_ERROR (zero). Otherwise, it returns SOCKET\_ERROR and must set the appropriate error code using **SetLastError**.

### Remarks

The name space configuration functions do not affect applications that are already running. Newly installed name-space providers will *not* be visible to applications nor will the changes in a name-space provider's activation state. Applications launched after the call to **WSCEnableNSProvider** will see the changes.

The **WSCEnableNSProvider** function is intended to be used by the Control Panel to change the state of the providers. An ISV should not just blindly de-activate another ISV's provider in order to activate its own. The choice should be left to the user.

## Error Codes

Error code	Meaning
WSAEINVAL	The specified name-space—provider identifier is invalid.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

**Library:** Use `Ws2_32.lib`.

## WSEnumProtocols

The **WSEnumProtocols** function retrieves information about available transport protocols.

```
int WSEnumProtocols (
    LPINT          lpiProtocols,
    LPWSAPROTOCOL_INFOW lpProtocolBuffer,
    LPDWORD       lpdwBufferLength,
    LPINT         lpErrno
);
```

### Parameters

#### *lpiProtocols*

[in] Null-terminated array of *iProtocol* values. This parameter is optional; if *lpiProtocols* is NULL, information on all available protocols is returned. Otherwise, information is retrieved only for those protocols listed in the array.

#### *lpProtocolBuffer*

[out] Buffer that is filled with **WSAPROTOCOL\_INFOW** structures.

#### *lpdwBufferLength*

[in/out] On input, the count of bytes in the *lpProtocolBuffer* buffer passed to **WSEnumProtocols**. On output, the minimum buffer size that can be passed to **WSEnumProtocols** to retrieve all the requested information.

#### *lpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, **WSEnumProtocols** returns the number of protocols to be reported on. Otherwise, a value of `SOCKET_ERROR` is returned and a specific error code is available in *lpErrno*.

## Remarks

This function is used to discover information about the collection of transport protocols installed on the local machine. This function differs from its API counterpart (**WSAEnumProtocols**) in that **WSAPROTOCOL\_INFOW** structures for all installed protocols, including layered protocols, can be obtained. (**WSAEnumProtocols** only returns information on base protocols and protocol chains.) The *lpiProtocols* parameter can be used as a filter to constrain the amount of information provided. Typically, a NULL pointer is supplied so the function will return information on all available transport protocols.

A **WSAPROTOCOL\_INFOW** structure is provided in the buffer pointed to by *lpProtocolBuffer* for each requested protocol. If the supplied buffer is not large enough (as indicated by the input value of *lpdwBufferLength*), the value pointed to by *lpdwBufferLength* will be updated to indicate the required buffer size. The Windows Sockets SPI client should then obtain a large enough buffer and call this function again. The **WSCEnumProtocols** function cannot enumerate over multiple calls; the passed-in buffer must be large enough to hold all expected entries in order for the function to succeed. This reduces the complexity of the function and should not pose a problem because the number of protocols loaded on a machine is typically small.

The order in which the **WSAPROTOCOL\_INFOW** structures appear in the buffer coincides with the order in which the protocol entries were registered by the service provider with the *Ws2\_32.dll*, or with any subsequent reordering that may have occurred through the Windows Sockets applet supplied for establishing default transport providers.

## Error Codes

Error code	Meaning
WSAEFAULT	One of more of the arguments is not in a valid part of the user address space.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAENOBUFFS	Buffer length was too small to receive all the relevant <b>WSAPROTOCOL_INFOW</b> structures and associated information. Pass in a buffer at least as large as the value returned in <i>lpdwBufferLength</i> .

## ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

**Library:** Use *Ws2\_32.lib*.

# WSCGetProviderPath

The **WSCGetProviderPath** function retrieves the DLL path for the specified provider.

```
int WSCGetProviderPath(
    LPGUID    IpProviderId,
    LPWSTR    IpszProviderDllPath,
    LPINT     IpProviderDllPathLen,
    LPINT     IpErrno
);
```

## Parameters

*IpProviderId*

[in] Locally unique identifier of the provider. This value is obtained by using **WSCEnumProtocols**.

*IpszProviderDllPath*

[out] Pointer to a buffer into which the provider DLL's path string is returned. The path is a null-terminated string and any embedded environment strings, such as %SystemRoot%, have not been expanded.

*IpProviderDllPathLen*

[in/out] Size of the buffer pointed to by the *IpszProviderDllPath* parameter.

*IpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WSCGetProviderPath** returns zero. Otherwise, it returns **SOCKET\_ERROR**. The specific error code is available in *IpErrno*.

## Remarks

The **WSCGetProviderPath** function retrieves the DLL path for the specified provider. The DLL path can contain embedded environment strings, such as %SystemRoot%, and thus should be expanded prior to being used with the Win32 **LoadLibrary** function. For more information, see **LoadLibrary**.

## Error Codes

Error code	Meaning
WSAEINVAL	The <i>IpProviderId</i> parameter does not specify a valid provider.
WSAEFAULT	The <i>IpszProviderDllPath</i> or <i>IpErrno</i> parameter is not in a valid part of the user address space, or <i>IpProviderDllPathLen</i> is too small.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

**Library:** Use Ws2\_32.lib.

**+** See Also

**WSCInstallProvider, WSCEnumProtocols**

---

## WSCInstallNameSpace

The **WSCInstallNameSpace** function installs a name-space provider. For providers that are able to support multiple names spaces, this function must be called once for every name space supported, and a unique provider identifier must be supplied each time.

```
int WSCInstallNameSpace (  
    LPWSTR    lpszIdentifier,  
    LPWSTR    lpszPathName,  
    DWORD     dwNameSpace,  
    DWORD     dwVersion,  
    LPGUID    lpProviderId  
);
```

### Parameters

***lpszIdentifier***

[in] Pointer to a string that identifies the provider for use in the GUID.

***lpszPathName***

[in] Pointer to a string that contains the path to the provider's DLL image. The string observes the usual rules for path resolution: this path can contain embedded environment strings (such as %SystemRoot%). Such environment strings are expanded whenever the Ws2\_32.dll needs to subsequently load the provider DLL on behalf of an application. After any embedded environment strings are expanded, the Ws2\_32.dll passes the resulting string into the **LoadLibrary** function to load the provider into memory. For more information, see **LoadLibrary**.

***dwNameSpace***

[in] Descriptor that specifies the name space supported by this provider.

***dwVersion***

[in] Descriptor that specifies the version number of the provider.

***lpProviderId***

[in] Unique identifier for this provider. This GUID should be generated by Uuidgen.exe.

## Return Values

If no error occurs, the **WSCInstallNameSpace** function returns `NO_ERROR` (zero). Otherwise, it returns `SOCKET_ERROR` if the function fails, and it must set the appropriate error code using **SetLastError**.

## Remarks

The name space configuration functions do not affect applications that are already running. Newly installed name-space providers will *not* be visible to applications nor will the changes in a name-space provider's activation state. Applications launched after the call to **WSCInstallNameSpace** will see the changes.

## Error Codes

Error code	Meaning
WSAEACCESS	Calling routine does not have sufficient privileges to install a name space.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

**Library:** Use `Ws2_32.lib`.

# WSCInstallProvider

The **WSCInstallProvider** function installs the specified transport provider into the system configuration database.

```
int WSCInstallProvider (
    const LPGUID                lpProviderId,
    const LPWSTR                lpszProviderDllPath,
    const LPWSAPROTOCOL_INFOW  lpProtocolInfoList,
    DWORD                      dwNumberOfEntries,
    LPINT                       lpErrno
);
```

## Parameters

*lpProviderId*

[in] Pointer to a provider-selected, globally unique identifier (GUID).

*lpszProviderDllPath*

[in] Pointer to a string containing the load path to the provider's DLL. This string observes the usual rules for path resolution and can contain embedded environment strings (such as `%SystemRoot%`). Such environment strings are expanded whenever the `Ws2_32.dll` needs to subsequently load the provider DLL on behalf of an

application. After any embedded environment strings are expanded, the `Ws2_32.dll` passes the resulting string into the **LoadLibrary** function to load the provider into memory. For more information, see **LoadLibrary**.

*lpProtocolInfoList*

[in] Points to an array of **WSAPROTOCOL\_INFOW** structures. Each structure defines a protocol, `address_family`, and `socket_type` supported by the provider.

*dwNumberOfEntries*

[in] Contains the number of entries in the *lpProtocolInfoList* array.

*lpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, **WSCInstallProvider** returns zero. Otherwise, it returns `SOCKET_ERROR`, and a specific error code is available in *lpErrno*.

### Remarks

This routine creates the necessary common Windows Sockets 2 configuration information for the specified provider. It is applicable to base protocols, layered protocols, and protocol chains. After this routine completes successfully, the protocol information provided in *lpProtocolInfoList* will be returned by the **WSAEnumProtocols**. Note that in Win32 environments, only instances of the `Ws2_32.dll` created after a successful completion of this function will include the new entries in **WSAEnumProtocols**.

Any file installation or service provider-specific configuration must be performed by the caller.

### Error Codes

Error code	Meaning
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

**Library:** Use `Ws2_32.lib`.

#### + See Also

**WSCDeinstallProvider**, **WSEnumProtocols**

## WSCUninstallNameSpace

The **WSCUninstallNameSpace** function uninstalls the indicated name-space provider.

```
int WSCUninstallNameSpace (  
    LPGUID    IpProviderId  
);
```

### Parameters

*IpProviderId*

[in] Unique identifier for this provider.

### Return Values

If no error occurs, **WSCUninstallNameSpace** returns NO\_ERROR (zero). Otherwise, it returns SOCKET\_ERROR and must set the appropriate error code using **SetLastError**.

### Remarks

The name space configuration functions do not affect applications that are already running. Newly installed name-space providers will not be visible to applications nor will the changes in a name-space provider's activation state. Applications launched after the call to **WSCUninstallNameSpace** will see the changes.

### Error Codes

Error code	Meaning
WSAEINVAL	Specified name-space—provider identifier is invalid.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

**Library:** Use Ws2\_32.lib.

## WSCWriteProviderOrder

The **WSCWriteProviderOrder** function is used to reorder the available transport providers. The order of the protocols determines the priority of a protocol when being enumerated or selected for us.

```
int WSCWriteProviderOrder (  
    LPDWORD    IpwdCatalogEntryId,  
    DWORD      dwNumberOfEntries  
);
```



## Parameters

### *lpwdCatalogEntryId*

[in] Array of **CatalogEntryId** elements found in the **WSAPROTOCOL\_INFO** structure. The order of the **CatalogEntryId** elements is the new priority ordering for the protocols.

### *dwNumberOfEntries*

[in] Number of elements in the *lpwdCatalogEntryId* array.

## Return Values

The function returns **ERROR\_SUCCESS** (zero) if the routine is successful. Otherwise, it returns a specific error code.

## Remarks

The order in which transport service providers are initially installed governs the order in which they are enumerated through **WSCEnumProtocols** at the service provider interface, or through **WSAEnumProtocols** at the application interface. More importantly, this order also governs the order in which protocols and service providers are considered when a client requests creation of a socket based on its address family, type, and protocol identifier.

Windows Sockets 2 includes an application called Sporder.exe that allows the catalog of installed protocols to be reordered interactively after protocols have already been installed. Windows Sockets 2 also includes an auxiliary DLL, Sporder.dll that exports this procedural interface for reordering protocols. This interface can be imported by linking with Sporder.lib.

Here are scenarios in which the **WSCWriteProviderOrder** function can fail.

- The *dwNumberOfEntries* parameter is not equal to the number of registered service providers.
- The *lpwdCatalogEntryId* contains an invalid catalog identifier.
- The *lpwdCatalogEntryId* does not contain all valid catalog identifiers exactly one time.
- The routine is not able to access the registry for some reason (for example, inadequate user permissions).
- Another process (or thread) is currently calling the function.

## Error Codes

Error code	Meaning
WSAEINVAL	Input parameters were bad, no action was taken.
ERROR_BUSY	Routine is being called by another thread or process.
(other)	Routine may return any registry error code.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Sporder.h.

**Library:** Included as a resource in Sporder.dll.

## WSPAccept

The **WSPAccept** function conditionally accepts a connection based on the return value of a condition function.

```
SOCKET WSPAccept (  
    SOCKET                s,  
    struct sockaddr FAR  *addr,  
    LPINT                addrLen,  
    LPCONDITIONPROC     lpfnCondition,  
    DWORD                dwCallbackData,  
    LPINT                lpErrno  
);
```

### Parameters

*s*

[in] Descriptor identifying a socket that is listening for connections after a **WSPListen**.

*addr*

[out] Optional pointer to a buffer that receives the address of the connecting entity, as known to the service provider. The exact format of the *addr* argument is determined by the address family established when the socket was created.

*addrLen*

[in/out] Optional pointer to an integer that contains the length of the *addr* parameter.

*lpfnCondition*

[in] Procedure instance address of an optional-condition function furnished by Windows Sockets. This function is used in the accept or reject decision based on the caller information passed in as parameters.

*dwCallbackData*

[in] Callback data to be passed back to the Windows Socket 2 client as the value of the *dwCallbackData* parameter of the condition function. This parameter is not interpreted by the service provider.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WSPAccept** returns a value of type `SOCKET` that is a descriptor for the accepted socket. Otherwise, a value of `INVALID_SOCKET` is returned, and a specific error code is available in *lpErrno*.

## Remarks

The **WSPAccept** function extracts the first connection on the queue of pending connections on socket *s*, and checks it against the condition function, provided the condition function is specified (that is, not `NULL`). The condition function must be executed in the same thread as this routine. If the condition function returns `CF_ACCEPT`, **WSPAccept** creates a new socket.

Newly created sockets have the same properties as the socket *s*, including network events registered with **WSPAsyncSelect** or with **WSPEventSelect**. As described in *Descriptor Allocation*, when new socket descriptors are allocated, IFS providers must call **WPUModifyIFSHandle** and non-IFS providers must call **WPUCreateSocketHandle**.

If the condition function returns `CF_REJECT`, **WSPAccept** rejects the connection request. If the application's accept/reject decision cannot be made immediately, the condition function will return `CF_DEFER` to indicate that no decision has been made. No action about this connection request is to be taken by the service provider. When the application is ready to take action on the connection request, it will invoke **WSPAccept** again and return either `CF_ACCEPT` or `CF_REJECT` as a return value from the condition function.

For sockets that are in the (default) blocking mode, if no pending connections are present on the queue, **WSPAccept** blocks the caller until a connection is present. For sockets in nonblocking mode, if this function is called when no pending connections are present on the queue, **WSPAccept** returns the error code `WSAEWOULDBLOCK`. The accepted socket cannot be used to accept more connections. The original socket remains open.

The argument *addr* is a result parameter that is filled with the address of the connecting entity, as known to the service provider. The exact format of the *addr* parameter is determined by the address family in which the communication is occurring. The *addrlen* is a value-result parameter; it will initially contain the amount of space pointed to by *addr*. On return, it must contain the actual length (in bytes) of the address returned by the service provider. This call is used with connection-oriented socket types such as **SOCK\_STREAM**. If *addr* and/or *addrlen* are equal to `NULL`, then no information about the remote address of the accepted socket is returned. Otherwise, these two parameters shall be filled in regardless of whether the condition function is specified or what it returns.

The prototype of the condition function is as follows.

```

int CALLBACK
ConditionFunc(
    IN LPWSABUF    lpCallerId,
    IN LPWSABUF    lpCallerData,
    IN OUT LPQOS   lpSQOS,
    IN OUT LPQOS   lpGQOS,
    IN LPWSABUF    lpCalleeId,
    IN LPWSABUF    lpCalleeData,
    OUT GROUP FAR  *g,
    IN DWORD       dwCallbackData
);

```

The *lpCallerId* and *lpCallerData* are value parameters that must contain the address of the connecting entity and any user data that was sent along with the connection request. If no caller identifier or caller data is available, the corresponding parameter will be NULL. Many network protocols do not support connect-time caller data. Most conventional network protocols can be expected to support caller identifier information at connection-request time. The *buf* part of the **WSABUF** pointed to by *lpCallerId* points to a **SOCKADDR**. The **SOCKADDR** is interpreted according to its address family (typically by casting the **SOCKADDR** to some type specific to the address family).

The *lpSQOS* parameter references the flow specifications for socket *s* specified by the caller, one for each direction, followed by any additional provider-specific parameters. The sending or receiving flow specification values will be ignored as appropriate for any unidirectional sockets. A NULL value for *lpSQOS* indicates that there is no caller-supplied QOS and that no negotiation is possible. A non-NULL *lpSQOS* pointer indicates that a QOS negotiation is to occur or that the provider is prepared to accept the QOS request without negotiation.

The *lpCalleeId* is a value parameter that contains the local address of the connected entity. The *buf* part of the **WSABUF** pointed to by *lpCalleeId* points to a **SOCKADDR**. The **SOCKADDR** is interpreted according to its address family (typically by casting the **SOCKADDR** to some type specific to the address family).

The *lpCalleeData* is a result parameter used by the condition function to supply user data back to the connecting entity. The storage for this data must be provided by the service provider. The *lpCalleeData->len* initially contains the length of the buffer allocated by the service provider and pointed to by *lpCalleeData->buf*. A value of zero means passing user data back to the caller is not supported. The condition function will copy up to *lpCalleeData->len* bytes of data into *lpCalleeData->buf*, and then update *lpCalleeData->len* to indicate the actual number of bytes transferred. If no user data is to be passed back to the caller, the condition function will set *lpCalleeData->len* to zero. The format of all address and user data is specific to the address family to which the socket belongs.

The *dwCallbackData* parameter value passed to the condition function is the value passed as the *dwCallbackData* parameter in the original **WSPAccept** call. This value is interpreted only by the Windows Sockets 2 client. This allows a client to pass some

context information from the **WSPAccept** call site through to the condition function, which provides the condition function with any additional information required to determine whether to accept the connection. A typical usage is to pass a (suitably cast) pointer to a data structure containing references to application-defined objects with which this socket is associated.

## Error Codes

Error code	Meaning
WSAECONNREFUSED	Connection request was forcefully rejected as indicated in the return value of the condition function (CF_REJECT).
WSAENETDOWN	Network subsystem has failed.
WSAEFAULT	The <i>addrLen</i> argument is too small or the <i>lpfnCondition</i> is not part of the user address space.
WSAEINTR	(Blocking) call was canceled through <b>WSPCancelBlockingCall</b> .
WSAEINPROGRESS	Blocking Windows Sockets call is in progress.
WSAEINVAL	<b>WSPListen</b> was not invoked prior to <b>WSPAccept</b> , parameter <i>g</i> specified in the condition function is not a valid value, the return value of the condition function is not a valid one, or any case where the specified socket is in an invalid state.
WSAEMFILE	Queue is nonempty upon entry to <b>WSPAccept</b> and there are no socket descriptors available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	Descriptor is not a socket.
WSAEOPNOTSUPP	Referenced socket is not a type that supports connection-oriented service.
WSATRY_AGAIN	Acceptance of the connection request was deferred as indicated in the return value of the condition function (CF_DEFER).
WSAEWOULDBLOCK	Socket is marked as nonblocking and no connections are present to be accepted.
WSAEACCES	Connection request that was offered has timed out or been withdrawn.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

**+** See Also

**WSPAccept**, **WSPBind**, **WSPConnect**, **WSPGetSockOpt**, **WSPListen**, **WSPSelect**, **WSPSocket**, **WSPAsyncSelect**, **WSPEventSelect**

## WSPAddressToString

The **WSPAddressToString** function converts all components of a **SOCKADDR** structure into a human readable-numeric string representation of the address. This is used mainly for display purposes.

```
int WSPAddressToString (
    LPSOCKADDR          IpsaAddress,
    DWORD               dwAddressLength,
    LPWSA_PROTOCOL_INFO IpProtocolInfo,
    LPWSTR              lpszAddressString,
    LPDWORD             lpdwAddressStringLength,
    LPINT               lpErrno
);
```

### Parameters

*IpsaAddress*

[in] Points to a **SOCKADDR** structure to translate into a string.

*dwAddressLength*

[in] Length of the address **SOCKADDR**.

*IpProtocolInfo*

[in] (required) **WSA\_PROTOCOL\_INFO** structure associated with the provider that will do the translation.

*lpszAddressString*

[out] Buffer that receives the human readable—address string.

*lpdwAddressStringLength*

[in/out] Length of the *AddressString* buffer. Returns the length of the string actually copied into the buffer. If the supplied buffer is not large enough, the function fails with a specific error of **WSAEFAULT** and this parameter is updated with the required size in bytes.

*lpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, **WSPAddressToString** returns zero. Otherwise, it returns **SOCKET\_ERROR**, and a specific error code is available in *lpErrno*.

## Layered Service Provider Considerations

A layered service provider supplies an implementation of this function, but it is also a client of this function if and when it calls **WSPAddressToString** of the next layer in the protocol chain. Some special considerations apply to the *IpProtocolInfo* parameter as it is propagated down through the layers of the protocol chain.

If the next layer in the protocol chain is another layer, then, when the next layer's **WSPAddressToString** is called, this layer must pass to the next layer a *IpProtocolInfo* parameter that references the same unmodified **WSAPROTOCOL\_INFO** structure with the same unmodified chain information. However, if the next layer is the base protocol (that is, the last element in the chain), this layer performs a substitution when calling the base provider's **WSPAddressToString**. In this case, the base provider's **WSAPROTOCOL\_INFO** structure should be referenced by the *IpProtocolInfo* parameter. One vital benefit of this policy is that base service providers do not have to be aware of protocol chains.

This same propagation policy applies when propagating a **WSAPROTOCOL\_INFO** structure through a layered sequence of other functions such as **WSPDuplicateSocket**, **WSPStartup**, **WSPSocket**, or **WSPStringToAddress**.

## Error Codes

Error code	Meaning
WSAEFAULT	Specified AddressString buffer is too small. Pass in a larger buffer.
WSA_EINVAL	Specified Address is not a valid socket address, or its address family is not supported by the provider, or the specified <i>IpProtocolInfo</i> did not refer to a <b>WSAPROTOCOL_INFO</b> structure supported by the provider.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

## WSPAsyncSelect

The **WSPAsyncSelect** function requests Windows message-based event notification of network events for a socket.

```
int WSPAsyncSelect (
    SOCKET          s,
    HWND           hWnd,
    unsigned int   wMsg,
    long           lEvent,
    LPINT          lpErrno
);
```

## Parameters

*s*

[in] Descriptor identifying the socket for which event notification is required.

*hWnd*

[in] Handle identifying the window that should receive a message when a network event occurs.

*wMsg*

[in] Message to be sent when a network event occurs.

*lEvent*

[in] Bitmask that specifies a combination of network events in which the Windows Sockets SPI client is interested.

*lpErrno*

[out] Pointer to the error code.

This function is used to request that the service provider send a Windows message to the client's window *hWnd* whenever it detects any of the network events specified by the *lEvent* parameter. The service provider should use the **WPUPostMessage** function to post the message. The message to be sent is specified by the *wMsg* parameter. The socket for which notification is required is identified by *s*.

This function automatically sets socket *s* to nonblocking mode, regardless of the value of *lEvent*. See **WSPIoctl** about how to set the socket back to blocking mode.

The *lEvent* parameter is constructed by using the bitwise OR operator with any of the values specified in the following table.

Value	Meaning
FD_READ	Issues notification of readiness for reading.
FD_WRITE	Issues notification of readiness for writing.
FD_OOB	Issues notification of the arrival of OOB data.
FD_ACCEPT	Issues notification of incoming connections.
FD_CONNECT	Issues notification of completed connections.
FD_CLOSE	Issues notification of socket closure.
FD_QOS	Issues notification of socket (QOS) changes.
FD_GROUP_QOS	Reserved.
FD_ROUTING_INTERFACE_CHANGE	Issues notification of routing interface change for the specified destination.
FD_ADDRESS_LIST_CHANGE	Issues notification of local address list change for the socket's protocol family.



Invoking **WSPAsyncSelect** for a socket cancels any previous **WSPAsyncSelect** or **WSPEventSelect** for the same socket. For example, to receive notification for both reading and writing, the Windows Sockets SPI client must call **WSPAsyncSelect** with both `FD_READ` and `FD_WRITE`, as follows:

```
rc = WSPAsyncSelect(s, hWnd, wMsg, FD_READ|FD_WRITE, &error);
```

It is not possible to specify different messages for different events. The following code will *not* work; the second call will cancel the effects of the first, and only `FD_WRITE` events will be reported with message `wMsg2`:

```
rc = WSPAsyncSelect(s, hWnd, wMsg1, FD_READ, &error);
rc = WSPAsyncSelect(s, hWnd, wMsg2, FD_WRITE, &error);
```

To cancel all notification (for example, to indicate that the service provider should send no further messages related to network events on the socket) *lEvent* will be set to zero:

```
rc = WSPAsyncSelect(s, hWnd, 0, 0, &error);
```

Since a socket created by **WSPAccept** has the same properties as the listening socket used to accept it, any **WSPAsyncSelect** events set for the listening socket apply to the accepted socket. For example, if a listening socket has **WSPAsyncSelect** events `FD_ACCEPT`, `FD_READ` and `FD_WRITE`, then any socket accepted on that listening socket will also have `FD_ACCEPT`, `FD_READ`, and `FD_WRITE` events with the same *wMsg* value used for messages. If a different *wMsg* or events are needed, the Windows Sockets SPI client must call **WSPAsyncSelect**, passing the accepted socket and the new information.

When one of the nominated network events occurs on the specified socket *s*, the service provider uses **WPUPostMessage** to send message *wMsg* to the Windows Sockets SPI client's window *hWnd*. The *wParam* argument identifies the socket on which a network event has occurred. The low word of *lParam* specifies the network event that has occurred. The high word of *lParam* contains any error code. The error code can be any error as defined in `Ws2spi.h`.

The possible network event codes that may be indicated are shown in the following table.

Value	Meaning
<code>FD_READ</code>	Socket <i>s</i> ready for reading.
<code>FD_WRITE</code>	Socket <i>s</i> ready for writing.
<code>FD_OOB</code>	OOB data ready for reading on socket <i>s</i> .
<code>FD_ACCEPT</code>	Socket <i>s</i> ready for accepting a new incoming connection.
<code>FD_CONNECT</code>	Connection initiated on socket <i>s</i> completed.
<code>FD_CLOSE</code>	Connection identified by socket <i>s</i> has been closed.
<code>FD_QOS</code>	QOS associated with socket <i>s</i> has changed.

Value	Meaning
FD_GROUP_QOS	Reserved.
FD_ROUTING_INTERFACE_CHANGE	Local interface that should be used to send to the specified destination has changed.
FD_ADDRESS_LIST_CHANGE	List of addresses of the socket's protocol family to which the Windows Socket SPI client can bind has changed.

### Return Values

The return value is zero if the Windows Sockets SPI client's declaration of interest in the network event set was successful. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error code is available in *lpErrno*.

### Remarks

Although **WSPAsyncSelect** can be called with interest in multiple events, the service provider issues the same Windows message for each event.

A Windows Sockets 2 provider will not continually flood a Windows Sockets SPI client with messages for a particular network event. Having successfully posted notification of a particular event to a Windows Sockets SPI client window, no further message(s) for that network event will be posted to the window until the Windows Sockets SPI client makes the function call that implicitly reenables notification of that network event.

Event	Re-enabling function
FD_READ	<b>WSPRecv</b> or <b>WSPRecvFrom</b>
FD_WRITE	<b>WSPSend</b> or <b>WSPSendTo</b>
FD_OOB	<b>WSPRecv</b> or <b>WSPRecvFrom</b>
FD_ACCEPT	<b>WSPAccept</b> unless the error code returned is <code>WSATRY_AGAIN</code> indicating that the condition function returned <code>CF_DEFER</code>
FD_CONNECT	None
FD_CLOSE	None
FD_QOS	<b>WSPIoctl</b> with <code>SIO_GET_QOS</code>
FD_GROUP_QOS	Reserved.
FD_ROUTING_INTERFACE_CHANGE	<b>WSPIoctl</b> with command <code>SIO_ROUTING_INTERFACE_CHANGE</code>
FD_ADDRESS_LIST_CHANGE	<b>WSPIoctl</b> with command <code>SIO_ADDRESS_LIST_CHANGE</code>

Any call to the reenabling routine, even one that fails, results in reenabling of message posting for the relevant event.

For `FD_READ`, `FD_OOB`, and `FD_ACCEPT` events, message posting is level-triggered. This means if the reenabling routine is called and the relevant condition is still met after the call, a **WSPAsyncSelect** message is posted to the Windows Sockets SPI client.

The `FD_QOS` event is considered edge triggered. A message will be posted exactly once when a QOS change occurs. Further messages will *not* be forthcoming until either the provider detects a further change in quality of service or the Windows Sockets SPI client renegotiates the quality of service for the socket.

The `FD_ROUTING_INTERFACE_CHANGE` and `FD_ADDRESS_LIST_CHANGE` events are considered edge triggered as well. A message will be posted exactly once when a change occurs after the Windows Socket 2 SPI client has requested the notification by issuing **WSIoctl** with `SIO_ROUTING_INTERFACE_CHANGE` or `SIO_ADDRESS_LIST_CHANGE` correspondingly. Further messages will not be forthcoming until the SPI client reissues the `IOCTL` and another change is detected since the `IOCTL` has been issued.

If any event has already occurred when the Windows Sockets SPI client calls **WSPAsyncSelect** or when the reenabling function is called, then an appropriate message is posted. For example, consider the following sequence:

- An SPI client calls **WSPListen**.
- A connect request is received but not yet accepted.
- The Windows Sockets SPI client calls **WSPAsyncSelect** specifying that it wants to receive `FD_ACCEPT` messages for the socket.

Due to the persistence of events, the Windows Sockets service provider posts an `FD_ACCEPT` message immediately.

The `FD_WRITE` event is handled slightly differently. An `FD_WRITE` message is posted when a socket is first connected with **WSPConnect** (after `FD_CONNECT`, if also registered) or accepted with **WSPAccept**, and then after a **WSPSend** or **WSPSendTo** fails with `WSAEWOULDBLOCK` and buffer space becomes available. Therefore, a Windows Sockets SPI client can assume that sends are possible starting from the first `FD_WRITE` message and lasting until a send returns `WSAEWOULDBLOCK`. After such a failure the Windows Sockets SPI client will be notified that sends are again possible with an `FD_WRITE` message.

The `FD_OOB` event is used only when a socket is configured to receive OOB data separately. If the socket is configured to receive OOB data inline, the OOB (expedited) data is treated as normal data and the Windows Sockets SPI client must register an interest in `FD_READ` events, not `FD_OOB` events.

The error code in an `FD_CLOSE` message indicates whether the socket close was graceful or abortive. If the error code is zero, then the close was graceful; if the error code is `WSAECONNRESET`, then the socket's virtual circuit was reset. This only applies to connection-oriented sockets such as **SOCK\_STREAM**.

The `FD_CLOSE` message is posted when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means the `FD_CLOSE` is posted when the connection goes into the `TIME WAIT` or `CLOSE WAIT` states. This results from the remote end performing a **WSPShutdown** on the send side or a **WSPCloseSocket**. `FD_CLOSE` shall only be posted after all data is read from a socket.

In the case of a graceful close, the service provider should only send an `FD_CLOSE` message to indicate virtual circuit closure after all the received data has been read. It should *not* send an `FD_READ` message to indicate this condition.

The `FD_QOS` message is posted when any member in the flow specification associated with socket `s` has changed, respectively. The service provider must update the QOS information available to the client through **WSPIoctl** with `SIO_GET_QOS`.

The `FD_ROUTING_INTERFACE_CHANGE` message is posted when the local interface that should be used to reach the destination specified in **WSPIoctl** with `SIO_ROUTING_INTERFACE_CHANGE` changes *after* such `IOCTL` has been issued.

The `FD_ADDRESS_LIST_CHANGE` message is posted when the list of addresses to which the Windows Socket 2 SPI client can bind changes *after* **WSPIoctl** with `SIO_ADDRESS_LIST_CHANGE` has been issued.

Here is a summary of events and conditions for each asynchronous notification message:

- **FD\_READ:**

- When **WSPAsyncSelect** is called, if there is data currently available to receive.
- When data arrives, if `FD_READ` is not already posted.
- After **WSPRecv** or **WSPRecvfrom** is called (with or without `MSG_PEEK`), if data is still available to receive.

---

**Note** When **WSPSetSockOpt** `SO_OOBINLINE` is enabled, data includes both normal data and OOB data in the instances noted in the preceding.

---

- **FD\_WRITE:**

- When **WSPAsyncSelect** is called, if a **WSPSend** or **WSPSendTo** is possible.
- After **WSPConnect** or **WSPAccept** is called, when connection is established.
- After **WSPSend** or **WSPSendTo** fails with `WSAEWOULDBLOCK`, when **WSPSend** or **WSPSendTo** is likely to succeed.
- After **WSPBind** on a datagram socket. `FD_WRITE` may or may not occur at this time (implementation dependent). In any case, a connectionless socket is always writeable immediately after **WSPBind**.

- **FD\_OOB:** Only valid when **WSPSetSockOpt** `SO_OOBINLINE` is disabled (default).

- When **WSPAsyncSelect** is called, if there is OOB data currently available to receive with the `MSG_OOB` flag.
- When OOB data arrives, if `FD_OOB` is not already posted.

- After **WSPRecv** or **WSPRecvfrom** is called with *or without* MSG\_OOB flag, if OOB data is still available to receive.
- **FD\_ACCEPT:**
  - When **WSPAsyncSelect** is called, if there is currently a connection request available to accept.
  - When a connection request arrives, if FD\_ACCEPT is not already posted.
  - After **WSPAccept** is called, if there is another connection request available to accept.
- **FD\_CONNECT:**
  - When **WSPAsyncSelect** is called, if there is currently a connection established.
  - After **WSPConnect** is called, when connection is established (even when **WSPConnect** succeeds immediately, as is typical with a datagram socket, and even when it fails immediately).
  - After **WSAJoinLeaf** is called, when the join operation completes.
  - After **connect**, **WSAConnect**, or **WSAJoinLeaf** was called with a nonblocking, connection-oriented socket. The initial operation returned with a specific error of WSAEWOULDBLOCK, but the network operation went ahead. Whether the operation eventually succeeds or not, when the outcome has been determined, FD\_CONNECT happens. The client should check the error code to determine whether the outcome was a success or failure.
- **FD\_CLOSE:** Only valid on connection-oriented sockets (for example, **SOCK\_STREAM**):
  - When **WSPAsyncSelect** is called, if socket connection has been closed.
  - After the remote system initiated a graceful close, when no data is currently available to receive (if data has been received and is waiting to be read when the remote system initiates a graceful close, the FD\_CLOSE is not delivered until all pending data has been read).
  - After the local system initiates a graceful close with **WSPShutdown** and the remote system has responded with End of Data notification (for example, TCP FIN), when no data is currently available to receive.
  - When the remote system terminates connection (for example, sent TCP RST), and *IPParam* will contain the WSAECONNRESET error value.

---

**Note** FD\_CLOSE is not posted after WSPClosesocket is called.

---

- **FD\_QOS:**
  - When **WSPAsyncSelect** is called, if the QOS associated with the socket has been changed,
  - After **WSPIoctl** with SIO\_GET\_QOS is called, when the QOS is changed.
- **FD\_GROUP\_QOS:** Reserved.
- **FD\_ROUTING\_INTERFACE\_CHANGE:**

- After **WSPIoctl** with `SIO_ROUTING_INTERFACE_CHANGE` is called, when the local interface that should be used to reach the destination specified in the IOCTL changes.
- **FD\_ADDRESS\_LIST\_CHANGE:**
  - After **WSPIoctl** with `SIO_ADDRESS_LIST_CHANGE` is called, when the list of local addresses to which the Windows Socket 2 SPI client can bind changes.

### Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid such as the window handle not referring to an existing window, or the specified socket is in an invalid state.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	Descriptor is not a socket.

Additional error codes can be set when the service provider issues a message to a Windows Sockets SPI client's window. This error code is embedded in the **IPParam** member of the message. Possible error codes for each network event are as shown in the following tables.

### Event: FD\_CONNECT

Error code	Meaning
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	Attempt to connect was forcefully rejected.
WSAENETUNREACH	Network cannot be reached from this host at this time.
WSAEFAULT	The <i>namelen</i> argument is incorrect.
WSAEINVAL	Socket is already bound to an address.
WSAEISCONN	Socket is already connected.
WSAEMFILE	No more file descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTCONN	Socket is not connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.

**Event: FD\_CLOSE**

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAECONNRESET	Connection was reset by the remote side.
WSAECONNABORTED	Connection was terminated due to a time-out or other failure.

**Event: FD\_READ****Event: FD\_WRITE****Event: FD\_OOB****Event: FD\_ACCEPT****Event: FD\_QOS****Event: FD\_GROUP\_QOS****Event: FD\_ADDRESS\_LIST\_CHANGE**

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.

**Event: FD\_ROUTING\_INTERFACE\_CHANGE**

Error code	Meaning
WSAENETUNREACH	Specified destination can no longer be reached.
WSAENETDOWN	Network subsystem has failed.

**! Requirements**

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

**+ See Also**

**WSPSelect**

---

## WSPBind

The **WSPBind** function associates a local address (that is, name) with a socket.

```
int WSPBind (
    SOCKET
```

```

const struct sockaddr FAR    *name,
int                          namelen,
LPINT                        lpErrno
);

```

## Parameters

*s*

[in] Descriptor identifying an unbound socket.

*name*

[in] Address to assign to the socket. The **SOCKADDR** structure is defined as follows:

```

sockaddr {
    _short    sa_family;
    char      sa_data[14];
};

```

Except for the **sa\_family** member, **SOCKADDR** contents are expressed in network byte order. In Windows Sockets 2, the *name* parameter is not strictly interpreted as a pointer to a **SOCKADDR** structure. It is cast this way for Windows Sockets compatibility. The actual structure is interpreted differently in the context of different address families. The only requirements are that the first *u\_short* is the address family and the total size of the memory buffer in bytes is *namelen*.

*namelen*

[in] Length of the *name*.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WSPBind** returns zero. Otherwise, it returns **SOCKET\_ERROR**, and a specific error code is available in *lpErrno*.

## Remarks

This routine is used on an unconnected connectionless or connection-oriented socket, before subsequent calls to **WSPConnect** or **WSPListen**. When a socket is created with **WSPSocket**, it exists in a name space (address family), but it has no name or local address assigned. **WSPBind** establishes the local association of the socket by assigning a local name to an unnamed socket.

As an example, in the Internet address family, a name consists of three parts: the address family, a host address, and a port number that identifies the Windows Sockets SPI client. In Windows Sockets 2, the *name* parameter is not strictly interpreted as a pointer to a **SOCKADDR** structure. Service providers are free to regard it as a pointer to a block of memory of size *namelen*. The first two bytes in this block (corresponding to **sa\_family** in the **SOCKADDR** declaration) must contain the address family that was used to create the socket. Otherwise, the error **WSAEFAULT** will be indicated.



If a Windows Sockets 2 SPI client does not care what local address is assigned to it, it will specify the manifest constant value `ADDR_ANY` for the **sa\_data** member of the *name* parameter. This instructs the service provider to use any appropriate network address. For TCP/IP, if the port is specified as zero, the service provider will assign a unique port to the Windows Sockets SPI client with a value between 1024 and 5000. The SPI client can use **WSPGetSockName** after **WSPBind** to learn the address and the port that has been assigned to it. However, note that if the Internet address is equal to `INADDR_ANY`, **WSPGetSockOpt** will *not* necessarily be able to supply the address until the socket is connected, since several addresses can be valid if the host is multihomed.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEADDRINUSE	Some process on the machine has already bound to the same fully-qualified address (for example, IP address and port in the <i>af_inet</i> case) and the socket has not been marked to allow address reuse with <code>SO_REUSEADDR</code> . (See the <code>SO_REUSEADDR</code> socket option under <b>WSPSetSockOpt</b> .)
WSAEADDRNOTAVAIL	Specified address is not a valid address for this machine.
WSAEFAULT	<i>Name</i> or the <i>namelen</i> argument is not a valid part of the user address space, the <i>namelen</i> argument is too small, the <i>name</i> argument contains incorrect address format for the associated address family, or the first two bytes of the memory block specified by <i>name</i> do not match the address family associated with the socket descriptor <i>s</i> .
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAEINVAL	Socket is already bound to an address.
WSAENOBUFS	Not enough buffers available, too many connections.
WSAENOTSOCK	Descriptor is not a socket.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

### + See Also

**WSPConnect**, **WSPListen**, **WSPGetSockName**, **WSPSetSockOpt**, **WSPSocket**

## WSPCancelBlockingCall

The **WSPCancelBlockingCall** function cancels a blocking call that is currently in progress.

```
int WSPCancelBlockingCall (
    LPINT lpErrno
);
```

## Parameters

*lpErrno*

[out] Pointer to the error code.

## Return Values

The value returned by **WSPCancelBlockingCall** is zero if the operation was successfully canceled. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error code is available in *lpErrno*.

## Remarks

This function cancels any outstanding blocking operation for this thread. It is typically used in two situations:

- A Windows Sockets SPI client is processing a message that has been received while a service provider is implementing pseudo blocking. In this case, **WSAIsBlocking** will be true.
- A blocking call is in progress and the Windows Sockets service provider has called back to the Windows Sockets SPI client's blocking hook function (through the callback function retrieved from **WPUQueryBlockingCallback**), which in turn is invoking this function. Such a situation might arise, for instance, in implementing a Cancel option for an operation that requires an extended time to complete.

In each case, the original blocking call will terminate as soon as possible with the error `WSAEINTR`. (In the first instance the termination will *not* take place until Windows message scheduling has caused control to revert back to the pseudo blocking routine in Windows Sockets. In the second instance, the blocking call will be terminated as soon as the blocking hook function completes.)

In the case of a blocking **WSPConnect** operation, Windows Sockets will terminate the blocking call as soon as possible, but it cannot be possible for the socket resources to be released until the connection has completed (and then been reset) or timed out. This is likely to be noticeable only if the Windows Sockets SPI client immediately tries to open a new socket (if no sockets are available), or to connect to the same peer through a **WSPConnect** call.

Canceling a **WSPAccept** or a **WSPSelect** call does not adversely impact the sockets passed to these calls. Only the particular call fails; any operation that was legal before the cancel is legal after the cancel, and the state of the socket is not affected in any way.

Canceling any operation other than **WSPAccept** and **WSPSelect** can leave the socket in an indeterminate state. If a Windows Sockets SPI client cancels a blocking operation on a socket, the only operation the Windows Sockets SPI client will be able to perform on the socket is a call to **WSPCloseSocket**, although other operations can work on

some Windows Sockets service providers. If a Windows Sockets SPI client requires maximum portability, it must be careful not to depend on performing operations after a cancel operation. A Windows Sockets SPI client can reset the connection by setting the time-out on `SO_LINGER` to zero and calling **WSPCloseSocket**.

If a cancel operation compromised the integrity of a **SOCK\_STREAM**'s data stream in any way, the Windows Sockets provider will reset the connection and fail all future operations other than **WSPCloseSocket** with `WSAECONNABORTED`.

### Remarks

It is acceptable for **WSPCancelBlockingCall** to return successfully if the blocking network operation completes prior to being canceled. In this case, the blocking operation will return successfully as if **WSPCancelBlockingCall** had never been called. The only way for the Windows Sockets SPI client to know with certainty that an operation was actually canceled is to check for a return code of `WSAEINTR` from the blocking call.

### Error Codes

Error code	Meaning
<code>WSAENETDOWN</code>	Network subsystem has failed.
<code>WSAEINVAL</code>	Indicates there is no outstanding blocking call.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

## WSPCleanup

The **WSPCleanup** function terminates use of the Windows Sockets service provider.

```
int WSPCleanup (
    LPINT lpErrno
);
```

### Parameters

*lpErrno*  
[out] Pointer to the error code.

### Return Values

The return value is zero if the operation has been successfully initiated. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error number is available in *lpErrno*.

## Remarks

The Windows Sockets 2 SPI client is required to perform a successful **WSPStartup** call before it can use Windows Sockets service providers. When it has completed the use of Windows Sockets service providers, the SPI client will call **WSPCleanup** to deregister itself from a Windows Sockets service provider and allow the service provider to free any resources allocated on behalf of the Windows Sockets 2 client. It is permissible for SPI clients to make more than one **WSPStartup** call. For each **WSPStartup** call, a corresponding **WSPCleanup** call will also be issued. Only the final **WSPCleanup** for the service provider does the actual cleanup; the preceding calls simply decrement an internal reference count in the Windows Sockets service provider.

When the internal reference count reaches zero and actual cleanup operations commence, any pending blocking or asynchronous calls issued by any thread in this process are canceled without posting any notification messages or signaling any event objects. Any pending overlapped send and receive operations (**WSPSend**, **WSPSendTo**, **WSPRecv**, **WSPRecvFrom** with an overlapped socket) issued by any thread in this process are also canceled without setting the event object or invoking the completion routine, if specified. In this case, the pending overlapped operations fail with the error status `WSA_OPERATION_ABORTED`. Any sockets open when **WSPCleanup** is called are reset and automatically deallocated as if **WSPCloseSocket** was called; sockets that have been closed with **WSPCloseSocket** but still have pending data to be sent are not affected—the pending data is still sent.

This function should not return until the service provider DLL is prepared to be unloaded from memory. In particular, any data remaining to be transmitted must either already have been sent or be queued for transmission by portions of the transport stack that will *not* be unloaded from memory along with the service provider's DLL.

## Remarks

A Windows Sockets service provider must be prepared to deal with a process that terminates without invoking **WSPCleanup** (for example, as a result of an error). A Windows Sockets service provider must ensure that **WSPCleanup** leaves things in a state in which the `Ws2_32.dll` can immediately invoke **WSPStartup** to reestablish Windows Sockets usage.

## Error Codes

Error code	Meaning
<code>WSANOTINITIALISED</code>	A successful <b>WSPStartup</b> call must occur before using this function.
<code>WSAENETDOWN</code>	Network subsystem has failed.
<code>WSAEINVAL</code>	Provider identifier given to the name-space provider is not managed by the name-space provider.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

**+** See Also

**WSPStartup**, **WSPClosesocket**, **WSPShutdown**

---

## WSPCloseSocket

The **WSPCloseSocket** function closes a socket.

```
int WSPCloseSocket (  
    SOCKET    s,  
    LPINT     lpErrno  
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*lpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, **WSPCloseSocket** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code is available in *lpErrno*.

### Remarks

This function closes a socket. More precisely, it releases the socket descriptor *s*, so further references to *s* should fail with the error `WSAENOTSOCK`. If this is the last reference to an underlying socket, the associated naming information and queued data are discarded. Any blocking or asynchronous calls pending on the socket (issued by any thread in this process) are canceled without posting any notification messages. Any pending overlapped operations issued by any thread in this process are also canceled. Whatever completion action was specified for these overlapped operations is performed (for example, event, completion routine, or completion port). In this case, the pending overlapped operations fail with the error status `WSA_OPERATION_ABORTED`. `FD_CLOSE` will not be posted after **WSPCloseSocket** is called.

**WSPClosesocket** behavior is summarized as follows:

- If `SO_DONTLINGER` is enabled (the default setting), **WSPCloseSocket** returns immediately and connection is gracefully closed in the background.

- If `SO_LINGER` is enabled with a zero time-out, **WSPCloseSocket** returns immediately and the connection is reset/terminated.
- or
- If `SO_LINGER` is enabled with a nonzero time-out with a blocking socket, **WSPCloseSocket** blocks until all data is sent or the time-out expires.
- If `SO_LINGER` is enabled with a nonzero time-out with a nonblocking socket, **WSPCloseSocket** returns immediately, thus indicating failure.

The semantics of **WSPCloseSocket** are affected by the socket options `SO_LINGER` and `SO_DONTLINGER` as follows.

Option	Interval	Type of close	Wait for close?
<code>SO_DONTLINGER</code>	Do not care	Graceful	No
<code>SO_LINGER</code>	Zero	Hard	No
<code>SO_LINGER</code>	Nonzero	Graceful	Yes

If `SO_LINGER` is set (that is, the `l_onoff` member of the linger structure is nonzero) and the time-out interval, `l_linger`, is zero, **WSPClosesocket** is not blocked even if queued data has not yet been sent or acknowledged. This is called a hard or abortive close, because the socket's virtual circuit is reset immediately, and any unsent data is lost. Any **WSPRecv** call on the remote side of the circuit will fail with `WSAECONNRESET`.

If `SO_LINGER` is set with a nonzero time-out interval on a blocking socket, the **WSPClosesocket** call blocks on a blocking socket until the remaining data has been sent or until the time-out expires. This is called a graceful disconnect. If the time-out expires before all data has been sent, the service provider should terminate the connection before **WSPClosesocket** returns.

Enabling `SO_LINGER` with a nonzero time-out interval on a nonblocking socket is not recommended. In this case, the call to **WSPClosesocket** will fail with an error of `WSAEWOULDBLOCK` if the close operation cannot be completed immediately. If **WSPClosesocket** fails with `WSAEWOULDBLOCK`, the socket handle is still valid and a disconnect is not initiated.

The Windows Sockets SPI client *must* call **WSPClosesocket** again to close the socket, although **WSPClosesocket** can continue to fail unless the Windows Sockets SPI client does one of the following:

- Disables `SO_DONTLINGER`.
- Enables `SO_LINGER` with a zero time-out.
- Calls **WSPShutdown** to initiate closure.

If `SO_DONTLINGER` is set on a stream socket (that is, the `l_onoff` member of the linger structure is zero), the **WSPClosesocket** call will return immediately and does not get `WSAEWOULDBLOCK`, whether the socket is blocking or nonblocking. However, any

data queued for transmission will be sent if possible before the underlying socket is closed. This is called a graceful disconnect and is the default behavior.

Note that in this case the Windows Sockets provider is allowed to retain any resources associated with the socket until such time as the graceful disconnect has completed or the provider terminates the connection due to an inability to complete the operation in a provider-determined amount of time. This can affect Windows Sockets clients that expect to use all available sockets. This is the default behavior; `SO_DONTLINGER` is set by default.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	Descriptor is not a socket.
WSAEWOULDBLOCK	Socket is marked as nonblocking and <code>SO_LINGER</code> is set to a nonzero time-out value.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

### + See Also

**WSPAccept**, **WSPSocket**, **WSPIoctl**, **WSPSetSockOpt**

## WSPConnect

The **WSPConnect** function establishes a connection to a peer, exchanges connect data, and specifies needed quality of service based on the supplied flow specification.

```
int WSPConnect (
    SOCKET                s,
    const struct sockaddr FAR *name,
    int                  namelen,
    LPWSABUF             lpCallerData,
    LPWSABUF             lpCalleeData,
    LPQOS                lpSQOS,
    LPQOS                lpGQOS,
    LPINT                lpErrno
);
```

## Parameters

*s*

[in] Descriptor identifying an unconnected socket.

*name*

[in] Name of the peer to which the socket is to be connected.

*namelen*

[in] Length of the *name*.

*lpCallerData*

[in] Pointer to the user data that is to be transferred to the peer during connection establishment.

*lpCalleeData*

[out] Pointer to a buffer into which any user data received from the peer during connection establishment can be copied.

*lpSQOS*

[in] Pointer to the flow specifications for socket *s*, one for each direction.

*lpGQOS*

[in] Reserved.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WSPConnect** returns zero. Otherwise, it returns `SOCKET_ERROR`, and a specific error code is available in *lpErrno*.

On a blocking socket, the return value indicates success or failure of the connection attempt. If the return error code indicates the connection attempt failed (that is, `WSAECONNREFUSED`, `WSAENETUNREACH`, `WSAETIMEDOUT`) the Windows Sockets SPI client can call **WSPConnect** again for the same socket.

## Remarks

This function is used to create a connection to the specified destination and to perform a number of other ancillary operations that occur at connect time as well. If the socket, *s*, is unbound, unique values are assigned to the local association by the system and the socket is marked as bound.

For connection-oriented sockets (for example, type `SOCK_STREAM`), an active connection is initiated to the specified host using *name* (an address in the name space of the socket. (For a detailed description, see **WSPBind**.) When this call completes successfully, the socket is ready to send and receive data. If the address member of the *name* structure is all zeroes, **WSPConnect** will return the error `WSAEADDRNOTAVAIL`. Any attempt to reconnect an active connection will fail with the error code `WSAEISCONN`.



For connection-oriented, nonblocking sockets it is often not possible to complete the connection immediately. In such a case, this function returns with the error `WSAEWOULDBLOCK` but the operation proceeds. When the success or failure outcome becomes known, it may be reported in one of several ways depending on how the client registers for notification. If the client uses **WSPSelect**, success is reported in the *writesfds* set and failure is reported in the *exceptfds* set. If the client uses **WSPAsyncSelect** or **WSPEventSelect**, the notification is announced with `FD_CONNECT` and the error code associated with the `FD_CONNECT` indicates either success or a specific reason for failure.

For a connectionless socket (for example, type **SOCK\_DGRAM**), the operation performed by **WSPConnect** is to establish a default destination address so the socket can be used with subsequent connection-oriented send and receive operations (**WSPSend**, **WSPRecv**). Any datagrams received from an address other than the destination address specified will be discarded. If the address member of the *name* structure is all zeroes, the socket will be disconnected—the default remote address will be indeterminate, so **WSPSend** and **WSPRecv** calls will return the error code `WSAENOTCONN`. However, **WSPSendTo** and **WSPRecvFrom** can still be used. The default destination can be changed by simply calling **WSPConnect** again, even if the socket is already connected. Any datagrams queued for receipt are discarded if *name* is different from the previous **WSPConnect**.

For connectionless sockets, *name* can indicate any valid address, including a broadcast address. However, to connect to a broadcast address, a socket must have **WSPSetSockOpt** `SO_BROADCAST` enabled. Otherwise, **WSPConnect** will fail with the error code `WSAEACCES`.

On connectionless sockets, exchange of user-to-user data is not possible and the corresponding parameters will be silently ignored.

The Windows Sockets SPI client is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specifies.

The *IpCallerData* is a value parameter that contains any user data to be sent along with the connection request. If *IpCallerData* is `NULL`, no user data will be passed to the peer. The *IpCalleeData* is a result parameter that will reference any user data passed back from the peer as part of the connection establishment. The *IpCalleeData->len* initially contains the length of the buffer allocated by the Windows Sockets SPI client and pointed to by *IpCalleeData->buf*. The *IpCalleeData->len* will be set to zero if no user data has been passed back. The *IpCalleeData* information will be valid when the connection operation is complete. For blocking sockets, this will be when the **WSPConnect** function returns. For nonblocking sockets, this will be after the `FD_CONNECT` notification has occurred. If *IpCalleeData* is `NULL`, no user data will be passed back. The exact format of the user data is specific to the address family the socket belongs to and/or the applications involved.

At connect time, a Windows Sockets SPI client can use the *IpSQOS* parameter to override any previous QOS specification made for the socket through **WSPIoctl** with the `SIO_SET_QOS` opcode.

The *lpSQOS* specifies the flow specifications for socket *s*, one for each direction, followed by any additional provider-specific parameters. If either the associated transport provider in general or the specific type of socket in particular cannot honor the QOS request, an error will be returned as indicated below. The sending or receiving flow specification values will be ignored, respectively, for any unidirectional sockets. If no provider-specific parameters are supplied, the *buf* and *len* members of *lpSQOS->ProviderSpecific* should be set to NULL and zero, respectively. A NULL value for *lpSQOS* indicates that no application supplied quality of service.

---

**Note** When connected sockets break (that is, become closed for whatever reason), they should be discarded and recreated. It is safest to assume that when things go awry for any reason on a connected socket, the Windows Sockets SPI client must discard and recreate the needed sockets in order to return to a stable point.

---

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEADDRINUSE	Local address of the socket is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs at the time of <b>bind</b> , but could be delayed until this function if the <b>bind</b> was to a partially wildcard address (involving ADDR_ANY) and if a specific address needs to be committed at the time of this function..
WSAEINTR	(Blocking) call was canceled through <b>WSPCancelBlockingCall</b> .
WSAEINPROGRESS	Blocking Windows Sockets call is in progress or the service provider is still processing a callback function.
WSAEALREADY	Nonblocking <b>WSPConnect</b> call is in progress on the specified socket.  In order to preserve backward compatibility, this error is reported as WSAEINVAL to Windows Sockets 1.1 applications that link to either Winsock.dll or Wsock32.dll.
WSAEADDRNOTAVAIL	Remote address is not a valid address (for example, ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	Attempt to connect was rejected.
WSAEFAULT	<i>Name</i> or the <i>namelen</i> argument is not a valid part of the user address space, the <i>namelen</i> argument is too small, the buffer length for <i>lpCalleeData</i> , <i>lpSQOS</i> , and <i>lpGQOS</i> is too small, or the buffer length for <i>lpCallerData</i> is too large.
WSAEINVAL	Parameter <i>s</i> is a listening socket.
WSAEISCONN	Socket is already connected (connection-oriented sockets only).

(continued)

*(continued)*

Error code	Meaning
WSAENETUNREACH	Network cannot be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTSOCK	Descriptor is not a socket.
WSAEOPNOTSUPP	Flow specifications specified in <i>IpSQOS</i> cannot be satisfied.
WSAEPROTONOSUPPORT	The <i>IpCallerData</i> augment is not supported by the service provider.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.
WSAEWOULDBLOCK	Socket is marked as nonblocking and the connection cannot be completed immediately. It is possible to select the socket using the <b>WSPSelect</b> function while it is connecting by using the <b>WSPSelect</b> function to select it for writing.
WSAEACCES	Attempt to connect datagram socket to broadcast address failed because <b>WSPSetSockOpt</b> SO_BROADCAST is not enabled.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

#### + See Also

**WSPAccept**, **WSPBind**, **WSPGetSockName**, **WSPGetSockOpt**, **WSPSocket**, **WSPSelect**, **WSPEventSelect**, **WSPEnumNetworkEvents**

## WSPDuplicateSocket

The **WSPDuplicateSocket** function returns a **WSAPROTOCOL\_INFOW** structure that can be used to create a new socket descriptor for a shared socket.

```
int WSPDuplicateSocket (
    SOCKET          s,
    DWORD           dwProcessId,
    LPWSAPROTOCOL_INFOW lpProtocolInfo,
    LPINT           lpErrno
);
```

### Parameters

**s**

[in] Specifies the local socket descriptor.

**dwProcessId**

[in] Specifies the identifier of the target process for which the shared socket will be used.

*lpProtocolInfo*

[out] Pointer to a buffer allocated by the client that is large enough to contain a **WSAPROTOCOL\_INFOW** structure. The service provider copies the protocol information structure contents to this buffer.

*lpErrno*

[out] Pointer to the error code.

**Return Values**

If no error occurs, **WSPDuplicateSocket** returns zero. Otherwise, the value of **SOCKET\_ERROR** is returned, and a specific error number is available in *lpErrno*.

**Remarks**

A source process calls **WSPDuplicateSocket** to obtain a special **WSAPROTOCOL\_INFOW** structure. It uses some interprocess communications (IPC) mechanism to pass the contents of this structure to a target process, which in turn uses it in a call to **WSPSocket** to obtain a descriptor for the duplicated socket. Note that the special **WSAPROTOCOL\_INFOW** structure can only be used once by the target process.

It is the service provider's responsibility to perform whatever operations are needed in the source process context and to create a **WSAPROTOCOL\_INFOW** structure that will be recognized when it subsequently appears as a parameter to **WSPSocket** in the target processes' context. The provider must then return a socket descriptor that references a common underlying socket. The **dwProviderReserved** member of the **WSAPROTOCOL\_INFOW** structure is available for the service provider's use and can be used to store any useful context information, including a duplicated handle.

When new socket descriptors are allocated, IFS providers must call **WPUModifyIFSHandle** and non-IFS providers must call **WPUCreateSocketHandle**.

One possible scenario for establishing and using a shared socket in a handoff mode is illustrated in the following.

Source process	IPC	Destination process
1) <b>WSPSocket</b> , <b>WSPConnect</b> .		
2) Requests target process identifier.	⇒	
		3) Receives process identifier request and respond.
4) Receives process identifier.	⇐	
5) Calls <b>WSPDuplicateSocket</b> to get a special <b>WSAPROTOCOL_INFOW</b> structure.		
6) Sends <b>WSAPROTOCOL_INFOW</b> structure to target.		

(continued)

(continued)

Source process	IPC	Destination process
	⇒	7) Receives <b>WSAPROTOCOL_INFOW</b> structure.
		8) Calls <b>WSPSocket</b> to create shared socket descriptor.
10) <b>WSPClosesocket</b>		9) Uses shared socket for data exchange.

The descriptors that reference a shared socket can be used independently as far as I/O is concerned. However, the Windows Sockets interface does not implement any type of access control, so it is up to the processes involved to coordinate their operations on a shared socket. A typical use for shared sockets is to have one process that is responsible for creating sockets and establishing connections, hand off sockets to other processes that are responsible for information exchange.

Since what is duplicated are the socket descriptors and not the underlying socket, all the states associated with a socket are held in common across all the descriptors. For example a **WSPSetSockOpt** operation performed using one descriptor is subsequently visible using a **WSPGetSockOpt** from any or all descriptors. A process can call **WSPClosesocket** on a duplicated socket and the descriptor will become deallocated. The underlying socket, however, will remain open until **WSPClosesocket** is called by the last remaining descriptor.

Notification on shared sockets is subject to the usual constraints of **WSPAsyncSelect** and **WSPEventSelect**. Issuing either of these calls using any of the shared descriptors cancels any previous event registration for the socket, regardless of which descriptor was used to make that registration. Thus, for example, a shared socket cannot deliver FD\_READ events to process A and FD\_WRITE events to process B. For situations when such tight coordination is required, it is suggested that developers use threads instead of separate processes.

A layered service provider supplies an implementation of this function, but it is also a client of this function if and when it calls **WSPDuplicateSocket** of the next layer in the protocol chain. Some special considerations apply to this function's *IpProtocolInfo* parameter as it is propagated down through the layers of the protocol chain.

If the next layer in the protocol chain is another layer then when the next layer's **WSPDuplicateSocket** is called, this layer must pass to the next layer a *IpProtocolInfo* that references the same unmodified **WSAPROTOCOL\_INFOW** structure with the same unmodified chain information. However, if the next layer is the base protocol (that is, the last element in the chain), this layer performs a substitution when calling the base provider's **WSPDuplicateSocket**. In this case, the base provider's **WSAPROTOCOL\_INFOW** structure should be referenced by the *IpProtocolInfo* parameter.

One vital benefit of this policy is that base service providers do not have to be aware of protocol chains. This same policy applies when propagating a **WSAPROTOCOL\_INFOW** structure through a layered sequence of other functions such as **WSPAddressToString**, **WSPStartup**, **WSPSocket**, or **WSPStringToAddress**.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAENOTSOCK	Descriptor is not a socket.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

### + See Also

**WPUCreateSocketHandle**, **WPUModifyIFSHandle**

## WSPEnumNetworkEvents

The **WSPEnumNetworkEvents** function reports occurrences of network events for the indicated socket.

```
int WSPEnumNetworkEvents (
    SOCKET          s,
    WSAEVENT       hEventObject,
    LPWSANETWORKEVENTS lpNetworkEvents,
    LPINT          lpErrno
);
```

### Parameters

*s*

[in] Descriptor identifying the socket.

*hEventObject*

[in] An optional handle identifying an associated event object to be reset.

*lpNetworkEvents*

[out] Pointer to a **WSANETWORKEVENTS** structure that is filled with a record of occurred network events and any associated error codes.

The **WSANETWORKEVENTS** structure is defined in the following text.

*lpErrno*

[out] Pointer to the error code.

**Return Values**

The return value is zero if the operation was successful. Otherwise, the value **SOCKET\_ERROR** is returned, and a specific error number is available in *lpErrno*.

**Remarks**

This function is used to report which network events have occurred for the indicated socket since the last invocation of this function. It is intended for use in conjunction with **WSPEventSelect**, which associates an event object with one or more network events. Recording of network events commences when **WSPEventSelect** is called with a nonzero *INetworkEvents* parameter and remains in effect until another call is made to **WSPEventSelect** with the *INetworkEvents* parameter set to zero, or until a call is made to **WSPAsyncSelect**.

**WSPEnumNetworkEvents** only reports network activity and errors nominated through **WSPEventSelect**. See the descriptions of **WSPSelect** and **WSPAsyncSelect** to find out how those functions report network activity and errors.

The socket's internal record of network events is copied to the structure referenced by *lpNetworkEvents*, whereafter the internal network events record is cleared. If *hEventObject* is non-NULL, the indicated event object is also reset. The Windows Sockets provider guarantees that the operations of copying the network event record, clearing it, and resetting any associated event object are atomic, such that the next occurrence of a nominated network event will cause the event object to become set. In the case of this function returning **SOCKET\_ERROR**, the associated event object is not reset and the record of network events is not cleared.

The **WSANETWORKEVENTS** structure is defined as follows:

```
typedef struct _WSANETWORKEVENTS {
    long lNetworkEvents;
    int iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

The **INetworkEvent** member of the structure indicates which of the **FD\_XXX** network events have occurred. The **iErrorCode** array is used to contain any associated error codes, with array index corresponding to the position of event bits in **INetworkEvents**. The identifiers such as **FD\_READ\_BIT** and **FD\_WRITE\_BIT** can be used to index the **iErrorCode** array.

Note that only those elements of the **iErrorCode** array are set that correspond to the bits set in **INetworkEvents** member. Other members are not modified (this is important for backward compatibility with the Windows Socket 2 SPI clients that are not aware of new **FD\_ROUTING\_INTERFACE\_CHANGE** and **FD\_ADDRESS\_LIST\_CHANGE** events).

The following error codes can be returned along with the respective network event.

#### Event: **FD\_CONNECT**

<b>Error Code</b>	<b>Meaning</b>
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	Attempt to connect was forcefully rejected.
WSAENETUNREACH	Network cannot be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.

#### Event: **FD\_CLOSE**

<b>Error Code</b>	<b>Meaning</b>
WSAENETDOWN	Network subsystem has failed.
WSAECONNRESET	Connection was reset by the remote side.
WSAECONNABORTED	Connection was terminated due to a time-out or other failure.

#### Event: **FD\_READ**

#### Event: **FD\_WRITE**

#### Event: **FD\_OOB**

#### Event: **FD\_ACCEPT**

#### Event: **FD\_QOS**

#### Event: **FD\_GROUP\_QOS**

#### Event: **FD\_ADDRESS\_LIST\_CHANGE**

#### Event: **FD\_ROUTING\_INTERFACE\_CHANGE**

<b>Error Code</b>	<b>Meaning</b>
WSAENETUNREACH	Specified destination is no longer reachable.
WSAENETDOWN	Network subsystem has failed.



Error Code	Meaning
WSAENETDOWN	Network subsystem has failed.

Error Code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAEINPROGRESS	A blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	Descriptor is not a socket.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

### + See Also

**WSPEventSelect**

## WSPEventSelect

The **WSPEventSelect** function specifies an event object to be associated with the supplied set of network events.

```
int WSPEventSelect (
    SOCKET          s,
    WSAEVENT        hEventObject,
    long            lNetworkEvents,
    LPINT           lpErrno
);
```

### Parameters

**s**

[in] Descriptor identifying the socket.

**hEventObject**

[in] Handle identifying the event object to be associated with the supplied set of network events.

**lNetworkEvents**

[in] Bitmask that specifies the combination of network events in which the Windows Sockets SPI client has interest.

*lpErrno*

[out] Pointer to the error code.

## Return Values

The return value is zero if the Windows Sockets SPI client's specification of the network events and the associated event object was successful. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error number is available in *lpErrno*.

## Remarks

This function is used to specify an event object, *hEventObject*, to be associated with the selected network events, *INetworkEvents*. The socket for which an event object is specified is identified by *s*. The event object is set when any of the nominated network events occur.

**WSPEventSelect** operates very similarly to **WSPAsyncSelect**, the difference being in the actions taken when a nominated network event occurs. Whereas **WSPAsyncSelect** causes a Windows Sockets SPI client-specified Windows message to be posted, **WSPEventSelect** sets the associated event object and records the occurrence of this event in an internal network event record. A Windows Sockets SPI client can use **WSPEnumNetworkEvents** to retrieve the contents of the internal network event record and thus determine which of the nominated network events have occurred.

**WSPEventSelect** is the only function that causes network activity and errors to be recorded and retrievable through **WSPEnumNetworkEvents**. See the descriptions of **WSPSelect** and **WSPAsyncSelect** to find out how those functions report network activity and errors.

This function automatically sets socket *s* to nonblocking mode, regardless of the value of *INetworkEvents*.

The *INetworkEvents* parameter is constructed by using the bitwise OR operator with any of the values specified in the following table.

Value	Meaning
<code>FD_READ</code>	Issues notification of readiness for reading.
<code>FD_WRITE</code>	Issues notification of readiness for writing.
<code>FD_OOB</code>	Issues notification of the arrival of OOB data.
<code>FD_ACCEPT</code>	Issues notification of incoming connections.
<code>FD_CONNECT</code>	Issues notification of completed connection.
<code>FD_CLOSE</code>	Issues notification of socket closure.
<code>FD_QOS</code>	Issues notification of socket (QOS) changes.
<code>FD_GROUP_QOS</code>	Reserved.

(continued)

(continued)

Value	Meaning
FD_ROUTING_INTERFACE_CHANGE	Issues notification of routing interface changes for the specified destination(s).
FD_ADDRESS_LIST_CHANGE	Issues notification of local address list changes for the socket's address family.

Issuing a **WSPEventSelect** for a socket cancels any previous **WSPAsyncSelect** or **WSPEventSelect** for the same socket and clears the internal network event record. For example, to associate an event object with both reading and writing network events, the Windows Sockets SPI client must call **WSPEventSelect** with both **FD\_READ** and **FD\_WRITE**, as follows:

```
rc = WSPEventSelect(s, hEventObject, FD_READ|FD_WRITE);\
```

It is not possible to specify different event objects for different network events. The following code will *not* work; the second call will cancel the effects of the first, and only **FD\_WRITE** network event will be associated with **hEventObject2**:

```
rc = WSPEventSelect(s, hEventObject1, FD_READ);
rc = WSPEventSelect(s, hEventObject2, FD_WRITE); //bad
```

To cancel the association and selection of network events on a socket, *INetworkEvents* should be set to zero, in which case the *hEventObject* parameter will be ignored.

```
rc = WSPEventSelect(s, hEventObject, 0);
```

Closing a socket with **WSPCloseSocket** also cancels the association and selection of network events specified in **WSPEventSelect** for the socket. The Windows Sockets SPI client, however, still must call **WSACloseEvent** to explicitly close the event object and free any resources.

Since an accepted (**WSPAccept**) socket has the same properties as the listening socket used to accept it, any **WSPEventSelect** association and network events selection set for the listening socket apply to the accepted socket. For example, if a listening socket has **WSPEventSelect** association of *hEventObject* with **FD\_ACCEPT**, **FD\_READ**, and **FD\_WRITE**, then any socket accepted on that listening socket will also have **FD\_ACCEPT**, **FD\_READ**, and **FD\_WRITE** network events associated with the same *hEventObject*. If a different *hEventObject* or network events are needed, the Windows Sockets SPI client should call **WSPEventSelect**, passing the accepted socket and the new information.

---

**Note** Having successfully recorded the occurrence of the network event and signaled the associated event object, no further actions are taken for that network event until the Windows Sockets SPI client makes the function call that implicitly reenables the setting of that network event and the signaling of the associated event object.

---

Network event	Reenabling function
FD_READ	<b>WSPRecv</b> or <b>WSPRecvFrom</b>
FD_WRITE	<b>WSPSend</b> or <b>WSPSendTo</b>
FD_OOB	<b>WSPRecv</b> or <b>WSPRecvFrom</b>
FD_ACCEPT	<b>WSPAccept</b> unless the error code returned is WSATRY_AGAIN indicating that the condition function returned CF_DEFER
FD_CONNECT	None
FD_CLOSE	None
FD_QOS	<b>WSPIoctl</b> with SIO_QOS
FD_GROUP_QOS	Reserved
FD_ROUTING_INTERFACE_CHANGE	<b>WSPIoctl</b> with command SIO_ROUTING_INTERFACE_CHANGE
FD_ADDRESS_LIST_CHANGE	<b>WSPIoctl</b> with command SIO_ADDRESS_LIST_CHANGE

Any call to the reenabling routine, even one that fails, results in reenabling of recording and signaling for the relevant network event and event object, respectively.

For FD\_READ, FD\_OOB, and FD\_ACCEPT network events, network event recording and event object signaling are level-triggered. This means that if the reenabling routine is called and the relevant network condition is still valid after the call, the network event is recorded and the associated event object is signaled. This allows a Windows Sockets SPI client to be event-driven and not be concerned with the amount of data that arrives at any one time. This is illustrated by the following sequence.

- Service provider receives 100 bytes of data on socket *s*, records the FD\_READ network event and signals the associated event object.
- The Windows Sockets SPI client issues **WSPRecv**( *s*, *buffptr*, 50, 0) to read 50 bytes.
- The service provider records the FD\_READ network event and signals the associated event object again since there is still data to be read.

With these semantics, a Windows Sockets SPI client need not read all available data in response to an FD\_READ network event—a single **WSPRecv** in response to each FD\_READ network event is appropriate.

The FD\_QOS event is considered edge triggered. A message will be posted exactly once when a QOS change occurs. Further indications will *not* be issued until either the service provider detects a further change in quality of service or the Windows Sockets SPI client renegotiates the quality of service for the socket.

The FD\_ROUTING\_INTERFACE\_CHANGE and FD\_ADDRESS\_LIST\_CHANGE events are considered edge triggered as well. A message will be posted exactly once when a change occurs AFTER the Windows Socket SPI client has requested the notification by

issuing **WSAIoctl** with `SIO_ROUTING_INTERFACE_CHANGE` or `SIO_ADDRESS_LIST_CHANGE` correspondingly. Further messages will not be forthcoming until the SPI client reissues the IOCTL *and* another change is detected since the IOCTL has been issued.

If a network event has already occurred when the Windows Sockets SPI client calls **WSPEventSelect** or when the reenabling function is called, then a network event is recorded and the associated event object is signaled as appropriate. This is illustrated in the following sequence.

- A Windows Sockets SPI client calls **WSPListen**.
- A connect request is received but not yet accepted.
- The Windows Sockets SPI client calls **WSPEventSelect** specifying that it is interested in the `FD_ACCEPT` network event for the socket. The service provider records the `FD_ACCEPT` network event and signals the associated event object immediately.

The `FD_WRITE` network event is handled slightly differently. An `FD_WRITE` network event is recorded when a socket is first connected with **WSPConnect** or accepted with **WSPAccept**, and then after a **WSPSend** or **WSPSendTo** fails with `WSAEWOULDBLOCK` and buffer space becomes available. Therefore, a Windows Sockets SPI client can assume that sends are possible starting from the first `FD_WRITE` network event setting and lasting until a send returns `WSAEWOULDBLOCK`. After such a failure the Windows Sockets SPI client will find out that sends are again possible when an `FD_WRITE` network event is recorded and the associated event object is signaled.

The `FD_OOB` network event is used only when a socket is configured to receive OOB data separately. If the socket is configured to receive OOB data inline, the OOB (expedited) data is treated as normal data and the Windows Sockets SPI client should register an interest in, and will get, `FD_READ` network event, not `FD_OOB` network event. A Windows Sockets SPI client can set or inspect the way in which OOB data is to be handled by using **WSPSetSockOpt** or **WSPGetSockOpt** for the `SO_OOBINLINE` option.

The error code in an `FD_CLOSE` network event indicates whether the socket close was graceful or abortive. If the error code is zero, then the close was graceful; if the error code is `WSAECONNRESET`, then the socket's virtual circuit was reset. This only applies to connection-oriented sockets such as **SOCK\_STREAM**.

The `FD_CLOSE` network event is recorded when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the `FD_CLOSE` is recorded when the connection goes into the `FIN_WAIT` or `CLOSE_WAIT` states. This results from the remote end performing a **WSPShutdown** on the send side or a **WSPCloseSocket**.

Service providers shall record *only* an `FD_CLOSE` network event to indicate closure of a virtual circuit, they must *not* record an `FD_READ` network event to indicate this condition.

The `FD_QOS` network event is recorded when any member in the flow specification associated with socket `s` has changed. This change must be made available to Windows Sockets SPI clients through the **WSPIoctl** function with `SIO_GET_QOS` to retrieve the current quality of service for socket `s`.

The `FD_ROUTING_INTERFACE_CHANGE` network event is recorded when the local interface that should be used to reach the destination specified in **WSAIoctl** with `SIO_ROUTING_INTERFACE_CHANGE` changes AFTER such IOCTL has been issued.

The `FD_ADDRESS_LIST_CHANGE` network event is recorded when the list of addresses of socket's protocol family to which the Windows Socket SPI client can bind changes *after* **WSAIoctl** with `SIO_ADDRESS_LIST_CHANGE` has been issued.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid, or the specified socket is in an invalid state.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress or the service provider is still processing a callback function.
WSAENOTSOCK	Descriptor is not a socket.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

### + See Also

**WSPEnumNetworkEvents**

## WSPGetOverlappedResult

The **WSPGetOverlappedResult** function returns the results of an overlapped operation on the specified socket.

```

BOOL WSPGetOverlappedResult (
    SOCKET          s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD         lpcbTransfer,
    BOOL            fWait,
    LPDWORD         lpdwFlags,
    LPINT           lpErrno
);

```

## Parameters

*s*

[in] Identifies the socket. This is the same socket that was specified when the overlapped operation was started by a call to **WSPRecv**, **WSPRecvFrom**, **WSPSend**, **WSPSendTo**, or **WSPIoctl**.

*lpOverlapped*

[in] Points to a **WSAOVERLAPPED** structure that was specified when the overlapped operation was started.

*lpcbTransfer*

[out] Points to a 32-bit variable that receives the number of bytes that were actually transferred by a send or receive operation, or by **WSPIoctl**.

*fWait*

[in] Specifies whether the function should wait for the pending overlapped operation to complete. If **TRUE**, the function does not return until the operation has been completed. If **FALSE** and the operation is still pending, the function returns **FALSE** and *lpErrno* is **WSA\_IO\_INCOMPLETE**. The *fWait* parameter may be set to **TRUE** only if the overlapped operation selected event-based completion notification.

*lpdwFlags*

[out] Points to a 32-bit variable that will receive one or more flags that supplement the completion status. If the overlapped operation was initiated through **WSPRecv** or **WSPRecvFrom**, this parameter will contain the results value for *lpFlags* parameter.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If **WSPGetOverlappedResult** succeeds, the return value is **TRUE**. This means the overlapped operation has completed successfully and the value pointed to by *lpcbTransfer* has been updated. If **WSPGetOverlappedResult** returns **FALSE**, this means that the overlapped operation has not completed or the overlapped operation completed but with errors, or completion status could not be determined due to errors in one or more parameters to **WSPGetOverlappedResult**. On failure, the value pointed to by *lpcbTransfer* will *not* be updated. The *lpErrno* parameter indicates the cause of the failure (either of **WSPGetOverlappedResult** or of the associated overlapped operation).

## Remarks

The results reported by the **WSPGetOverlappedResult** function are those of the specified socket's last overlapped operation to which the specified **WSAOVERLAPPED** structure was provided, and for which the operation's results were pending. A pending operation is indicated when the function that started the operation returns **SOCKET\_ERROR**, and the *lpErrno* is **WSA\_IO\_PENDING**. When an I/O operation is pending, the function that started the operation resets the **hEvent** member of the **WSAOVERLAPPED** structure to the nonsignaled state. Then, when the pending operation has been completed, the system sets the event object to the signaled state.

If the *fWait* parameter is TRUE, **WSPGetOverlappedResult** determines whether the pending operation has been completed by blocking and waiting for the event object to be in the signaled state. A client may set the *fWait* parameter to TRUE only if it selected event-based completion notification when the I/O operation was requested. If another form of notification was selected, the usage of the **hEvent** member of the **WSAOVERLAPPED** structure is different, and setting *fWait* to TRUE causes unpredictable results.

### Interaction with **WPUCompleteOverlappedRequest**

The behavior of **WPUCompleteOverlappedRequest** places some constraints on how a service provider implements **WSPGetOverlappedResult** since only the **Offset** and **OffsetHigh** members of the **WSAOVERLAPPED** structure are exclusively controlled by the service provider even though three values (byte count, flags, and error) must be retrieved from the structure by **WSPGetOverlappedResult**. A service provider may accomplish this any way it chooses as long as it interacts with the behavior of **WPUCompleteOverlappedRequest** properly. The following description presents a typical implementation:

At the start of overlapped processing, the service provider sets *Internal* to **WSS\_OPERATION\_IN\_PROGRESS**.

When the I/O operation is complete, the provider sets **OffsetHigh** to the Windows Sockets 2 error code resulting from the operation, sets **Offset** to the flags resulting from the I/O operation, and calls **WPUCompleteOverlappedRequest**, passing the transfer byte count as one of the parameters. **WPUCompleteOverlappedRequest** eventually sets **InternalHigh** to the transfer byte count, then sets **Internal** to a value other than **WSS\_OPERATION\_IN\_PROGRESS**.

When **WSPGetOverlappedResult** is called, the service provider checks **Internal**. If it is **WSS\_OPERATION\_IN\_PROGRESS**, the provider waits on the event handle in the **hEvent** member or returns an error, based on the setting of the *fWait* flag of **WSPGetOverlappedResult**. If not in progress, or after completion of waiting, the provider returns the values from **InternalHigh**, **OffsetHigh**, and **Offset** as the transfer count, operation result error code, and flags respectively.

### Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAENOTSOCK	Descriptor is not a socket.
WSA_INVALID_HANDLE	The <b>hEvent</b> member of the <b>WSAOVERLAPPED</b> structure does not contain a valid event object handle.
WSA_EINVAL	One of the parameters is unacceptable.
WSA_IO_INCOMPLETE	The <i>fWait</i> parameter is FALSE and the I/O operation has not yet completed.



**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

**+** See Also

**WSPRecv**, **WSPRecvFrom**, **WSPSend**, **WSPSendTo**, **WSPConnect**, **WSPAccept**, **WSPIoctl**, **WPUCompleteOverlappedRequest**

## WSPGetPeerName

The **WSPGetPeerName** function gets the address of the peer to which a socket is connected.

```
int WSPGetPeerName (
    SOCKET          s,
    struct sockaddr FAR *name,
    LPINT          namelen,
    LPINT          lpErrno
);
```

### Parameters

*s*

[in] Descriptor identifying a connected socket.

*name*

[out] Pointer to the structure to receive the name of the peer.

*namelen*

[in/out] Pointer to an integer that, on input, indicates the size of the structure pointed to by *name*, and on output indicates the size of the returned name.

*lpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, **WSPGetPeerName** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code is available in *lpErrno*.

### Remarks

**WSPGetPeerName** supplies the name of the peer connected to the socket *s* and stores it in the structure **SOCKADDR** referenced by *name*. It can be used only on a connected socket. For datagram sockets, only the name of a peer specified in a previous **WSPConnect** call will be returned—any name specified by a previous **WSPSendTo** call will *not* be returned by **WSPGetPeerName**.

On return, the *namelen* argument contains the actual size of the name returned in bytes.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEFAULT	<i>Name</i> or the <i>namelen</i> argument is not a valid part of the user address space, or the <i>namelen</i> argument is too small.
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAENOTCONN	Socket is not connected.
WSAENOTSOCK	Descriptor is not a socket.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

### + See Also

**WSPBind**, **WSPSocket**, **WSPGetSockName**

## WSPGetQOSByName

The **WSPGetQOSByName** function initializes a **QOS** structure based on a named template, or retrieves an enumeration of the available template names.

```

BOOL WSPGetQOSByName (
    SOCKET      s,
    LPWSABUF   lpQOSName,
    LPQOS      lpQOS,
    LPINT      lpErrno
);

```

### Parameters

*s*

[in] Descriptor identifying a socket.

*lpQOSName*

[in out] Specifies the QOS template name, or supplies a buffer to retrieve an enumeration of the available template names.

*lpQOS*

[out] Pointer to the **QOS** structure to be filled.

*lpErrno*

[out] A pointer to the error code.

## Return Values

If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE, and a specific error code is available in *lpErrno*.

## Remarks

Clients can use **WSPGetQOSByName** to initialize a **QOS** structure to a set of known values appropriate for a particular service class or media type. These values are stored in a template that is referenced by a well-known name. The client may retrieve these values by setting the **buf** member of the **WSABUF** indicated by *lpQOSName* to point to a Unicode string of nonzero length specifying a template name. In this case the usage of *lpQOSName* is IN only, and results are returned through *lpQOS*.

Alternatively, the client may use **WSPGetQOSByName** to retrieve an enumeration of available template names. The client may do this by setting the **buf** member of the **WSABUF** indicated by *lpQOSName* to a zero-length null-terminated Unicode string. In this case, the buffer indicated by **buf** is overwritten with a sequence of as many null-terminated Unicode template name strings as are available up to the number of bytes available in **buf** as indicated by the **len** member of the **WSABUF** indicated by *lpQOSName*. The list of names itself is terminated by a zero-length Unicode name string. When **WSPGetQOSByName** is used to retrieve template names, the *lpQOS* parameter is ignored.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAENOTSOCK	Descriptor is not a socket.
WSAEFAULT	The <i>lpQOS</i> argument is not a valid part of the user address space, or the buffer length for <i>lpQOS</i> is too small.
WSAEINVAL	Specified QOS template name is invalid.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in *Ws2spi.h*.

### + See Also

**WSPConnect**, **WSPAccept**, **WSPGetSockOpt**

---

# WSPGetSockName

The **WSPGetSockName** function gets the local name for a socket.

```

int WSPGetSockName (
    SOCKET          s,
    struct sockaddr FAR *name,
    LPINT          nameLen,
    LPINT          lpErrno
);

```

## Parameters

*s*

[in] Descriptor identifying a bound socket.

*name*

[out] Pointer to a structure used to supply the address (name) of the socket.

*nameLen*

[in/out] Pointer to an integer that, on input, indicates the size of the structure pointed to by *name*, and on output indicates the size of the returned name.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WSPGetSockName** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code is available in *lpErrno*.

## Remarks

**WSPGetSockName** retrieves the current name for the specified socket descriptor in *name*. It is used on a bound and/or connected socket specified by the *s* parameter. The local association is returned. This call is especially useful when a **WSPConnect** call has been made without doing a **WSPBind** first; as this call provides the only means by which the local association that has been set by the service provider can be determined.

If a socket was bound to an unspecified address (for example, `ADDR_ANY`), indicating that any of the host's addresses within the specified address family should be used for the socket, **WSPGetSockName** will *not* necessarily return information about the host address, unless the socket has been connected with **WSPConnect** or **WSPAccept**. The Windows Sockets SPI client must not assume that an address will be specified unless the socket is connected. This is because for a multihomed host, the address that will be used for the socket is unknown until the socket is connected.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEFAULT	<i>Name</i> or the <i>nameLen</i> argument is not a valid part of the user address space, or the <i>nameLen</i> argument is too small.

(continued)

(continued)

Error code	Meaning
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAENOTSOCK	Descriptor is not a socket.
WSAEINVAL	Socket has not been bound to an address with <b>WSPBind</b> , or ADDR_ANY is specified in <b>WSPBind</b> but connection has not yet occurred.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

#### + See Also

**WSPBind**, **WSPSocket**, **WSPGetPeerName**

## WSPGetSockOpt

The **WSPGetSockOpt** function retrieves a socket option.

```
int WSPGetSockOpt (
    SOCKET      s,
    int         level,
    int         optname,
    char FAR    *optval,
    LPINT       optlen,
    LPINT       lpErrno
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*level*

[in] Level at which the option is defined; the supported levels include SOL\_SOCKET. (See *Annex* for more protocol-specific levels.)

*optname*

[in] Socket option for which the value is to be retrieved.

*optval*

[out] Pointer to the buffer in which the value for the requested option is to be returned.

*optlen*

[in/out] Pointer to the size of the *optval* buffer.

*lpErrno*

[out] A pointer to the error code.

## Return Values

If no error occurs, **WSPGetSockOpt** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code is available in *lpErrno*.

## Remarks

**WSPGetSockOpt** retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in *optval*. Options can exist at multiple protocol levels, but they are always present at the uppermost socket level. Options affect socket operations, such as the routing of packets and OOB data transfer.

The value associated with the selected option is returned in the buffer *optval*. The integer pointed to by *optlen* should originally contain the size of this buffer; on return, it will be set to the size of the value returned. For `SO_LINGER`, this will be the size of a structure `linger`; for most other options it will be the size of an integer.

The Windows Sockets SPI client is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specifies.

If the option was never set with **WSPSetSockOpt**, then **WSPGetSockOpt** returns the default value for the option.

*level* = `SOL_SOCKET`

Value	Type	Meaning	Default
<code>SO_ACCEPTCONN</code>	BOOL	Socket is listening through <b>WSPListen</b> .	FALSE unless a <b>WSPListen</b> has been performed.
<code>SO_BROADCAST</code>	BOOL	Socket is configured for the transmission of broadcast messages.	FALSE
<code>SO_DEBUG</code>	BOOL	Debugging is enabled.	FALSE
<code>SO_DONTLINGER</code>	BOOL	If true, the <code>SO_LINGER</code> option is disabled.	TRUE
<code>SO_DONTROUTE</code>	BOOL	Routing is disabled. Not supported on ATM sockets (results in an error).	FALSE
<code>SO_ERROR</code>	integer	Retrieves error status and clears.	0
<code>SO_GROUP_ID</code>	GROUP	Reserved.	Null
<code>SO_GROUP_PRIORITY</code>	integer	Reserved.	0

(continued)

*(continued)*

<b>Value</b>	<b>Type</b>	<b>Meaning</b>	<b>Default</b>
SO_KEEPAIVE	BOOL	Keepalives are being sent. Not supported on ATM sockets (results in an error).	FALSE
SO_LINGER	<b>LINGER</b> structure	Returns the current linger options.	1 is on (default), 0 is off
SO_MAX_MSG_SIZE	unsigned integer	Maximum size of a message for message-oriented socket types (for example, <b>SOCK_DGRAM</b> ). Has no meaning for stream oriented sockets.	Implementation dependent
SO_OOINLINE	BOOL	OOB data is being received in the normal data stream.	FALSE
SO_PROTOCOL_INFO	<b>WSAPROTOCOL_INFO</b> structure	Description of protocol information for protocol that is bound to this socket.	Protocol dependent
SO_RCVBUF	integer	Total per-socket buffer space reserved for receives. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.	Implementation dependent
SO_REUSEADDR	BOOL	Socket can be bound to an address that is already in use. Not applicable on ATM sockets.	FALSE.
SO_SNDBUF	integer	Total per-socket buffer space reserved for sends. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.	Implementation dependent
SO_TYPE	integer	Type of socket (for example, <b>SOCK_STREAM</b> ).	As created with <b>socket</b>
PVD_CONFIG	Service Provider Dependent	An opaque data structure object from the service provider associated with socket <i>s</i> . This object stores the current configuration information of the service provider. The exact format of this data structure is service provider-specific.	Implementation dependent

Calling **WSPGetSockOpt** with an unsupported option will result in an error code of **WSAENOPROTOOPT** being returned in *IpErrno*.

### SO\_DEBUG

Windows Sockets service providers are encouraged (but not required) to supply output debug information if the `SO_DEBUG` option is set by a Windows Sockets SPI client. The mechanism for generating the debug information and the form it takes are beyond the scope of this specification.

### SO\_ERROR

The `SO_ERROR` option returns and resets the per-socket-based error code (that is not necessarily the same as the per-thread-error code that is maintained by the `Ws2_32.dll`). A successful Windows Sockets call on the socket does not reset the socket-based error code returned by the `SO_ERROR` option.

### SO\_GROUP\_ID

Reserved. This value should be `NULL`.

### SO\_GROUP\_PRIORITY

Reserved.

### SO\_KEEPAIVE

A Windows Sockets SPI client can request that a TCP/IP service provider enable the use of keep-alive packets on TCP-connections by turning on the `SO_KEEPAIVE` socket option. A Windows Sockets provider need not support the use of keep-alives: if it does, the precise semantics are implementation specific but should conform to section 4.2.3.6 of RFC 1122: *Requirements for Internet Hosts—Communication Layers*. If a connection is dropped as the result of keep-alives, the error code `WSAENETRESET` is returned to any calls in progress on the socket, and any subsequent calls will fail with `WSAENOTCONN`.

### SO\_LINGER

`SO_LINGER` controls the action taken when unsent data is queued on a socket and a **WSPCloseSocket** is performed. See **WSPCloseSocket** for a description of the way in which the `SO_LINGER` settings affect the semantics of **WSPCloseSocket**. The Windows Sockets SPI client obtains the desired behavior by creating a **LINGER** structure (pointed to by the *optval* argument) with the following elements:

```
struct linger {
    u_short    l_onoff;
    u_short    l_linger;
}
```

### SO\_MAX\_MSG\_SIZE

This is a get-only socket option, which indicates the maximum size of an outbound send message for message-oriented socket types (for example, **SOCK\_DGRAM**) as implemented by the service provider. It has no meaning for byte stream-oriented sockets. There is no provision to determine the maximum inbound message size.

### SO\_PROTOCOL\_INFOW

This is a get-only option that supplies the **WSAPROTOCOL\_INFO** structure associated with this socket. See **WSCEnumProtocols** for more information about this structure.



**SO\_SNDBUF**

When a Windows Sockets service provider supports the `SO_RCVBUF` and `SO_SNDBUF` options, a Windows Sockets SPI client can use **WSPSetSockOpt** to request different buffer sizes (larger or smaller). The call can succeed even though the service provider did not make available the entire amount requested. A Windows Sockets SPI client must call this function with the same option to check the buffer size actually provided.

**SO\_REUSEADDR**

By default, a socket can not be bound (see **WSPBind**) to a local address that is already in use. On occasion, however, it may be desirable to reuse an address in this way. Since every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the Windows Sockets provider that a **WSPBind** on a socket should be allowed to bind to a local address that is already in use by another socket, the Windows Sockets SPI client should set the `SO_REUSEADDR` socket option for the socket before issuing the **WSPBind**. Note that the option is interpreted only at the time of the **WSPBind**. It is therefore unnecessary (but harmless) to set the option on a socket that is not to be bound to an existing address, and setting or resetting the option after the **WSPBind** has no effect on this or any other socket.

**PVD\_CONFIG**

This option retrieves an opaque data structure object from the service provider associated with socket *s*. This object stores the current configuration information of the service provider. The exact format of this data structure is service-provider specific.

**Error Codes**

<b>Error code</b>	<b>Meaning</b>
WSAENETDOWN	Network subsystem has failed.
WSAEFAULT	One of the <i>optval</i> or the <i>optlen</i> arguments is not a valid part of the user address space, or the <i>optlen</i> argument is too small.
WSAEINVAL	The <i>level</i> is unknown or invalid.
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAENOPROTOOPT	Option is unknown or unsupported by the indicated protocol family.
WSAENOTSOCK	Descriptor is not a socket.

**! Requirements**

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

**+** See Also

**WSPSetSockOpt, WSPSocket**

## WSPIoctl

The **WSPIoctl** function controls the mode of a socket.

```
int WSPIoctl (
    SOCKET          s,
    DWORD           dwIoControlCode,
    LPVOID          lpvInBuffer,
    DWORD           cbInBuffer,
    LPVOID          lpvOutBuffer,
    DWORD           cbOutBuffer,
    LPDWORD         lpcbBytesReturned,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    LPWSATHREADID  lpThreadId,
    LPINT           lpErrno
);
```

### Parameters

*s*

[in] Handle to a socket.

*dwIoControlCode*

[in] Control code of the operation to perform.

*lpvInBuffer*

[in] Address of input buffer.

*cbInBuffer*

[in] Size of input buffer.

*lpvOutBuffer*

[out] Address of output buffer.

*cbOutBuffer*

[in] Size of output buffer.

*lpcbBytesReturned*

[out] Pointer to the size of output buffer's contents.

*lpOverlapped*

[in] Address of **WSAOVERLAPPED** structure (ignored for nonoverlapped sockets).

*lpCompletionRoutine*

[in] Pointer to the completion routine called when the operation has been completed (ignored for nonoverlapped sockets).

*IpThreadId*

[in] Pointer to a **WSATHREADID** structure to be used by the provider in a subsequent call to **WPUQueueApc**. The provider should store the referenced **WSATHREADID** structure (not the pointer to same) until after the **WPUQueueApc** function returns.

*IpErrno*

[out] Pointer to the error code.

**Remarks**

This routine is used to set or retrieve operating parameters associated with the socket, the transport protocol, or the communications subsystem. If both *IpOverlapped* and *IpCompletionRoutine* are NULL, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, *IpOverlapped* and *IpCompletionRoutine* parameters are ignored and this function can block if socket *s* is in blocking mode. Note that if socket *s* is in nonblocking mode, this function can return WSAEWOULDBLOCK if the specified operation cannot be finished immediately. In this case, the Windows Sockets SPI client may change the socket to blocking mode and reissue the request or wait for the corresponding network event (such as FD\_ROUTING\_INTERFACE\_CHANGE or FD\_ADDRESS\_LIST\_CHANGE in case of SIO\_ROUTING\_INTERFACE\_CHANGE or SIO\_ADDRESS\_LIST\_CHANGE) using Windows message (through **WSPAsyncSelect** or event (using **WSPEventSelect**) based notification mechanism. For overlapped sockets, operations that cannot be completed immediately will be initiated and completion will be indicated at a later time. Final completion status is retrieved through **WSPGetOverlappedResult**.

Any IOCTL may block indefinitely, depending on the implementation of the service provider. If the Windows Sockets SPI client cannot tolerate blocking in a **WSPIoctl** call, overlapped I/O would be advised for ioctls that are most likely to block including:

- SIO\_FINDROUTE
- SIO\_FLUSH
- SIO\_GET\_QOS
- SIO\_GET\_GROUP\_QOS
- SIO\_SET\_QOS
- SIO\_SET\_GROUP\_QOS
- SIO\_ROUTING\_INTERFACE\_CHANGE
- SIO\_ADDRESS\_LIST\_CHANGE

Some protocol-specific ioctls may also be particularly likely to block. Check the relevant protocol-specific annex for available information.

In as much as the *dwIoControlCode* parameter is now a 32-bit entity, it is possible to adopt an encoding scheme that provides a convenient way to partition the opcode identifier space. The *dwIoControlCode* parameter is constructed to allow for protocol and

vendor independence when adding new control codes, while retaining backward compatibility with Windows Sockets 1.1 and Unix control codes. The *dwIoControlCode* parameter has the following form.

I	O	V	T	Vendor/address family	Code
3	3	2	2 2	2 2 2 2 2 2 2 1 1 1 1	1 1 1 1 1 1
1	0	9	8 7	6 5 4 3 2 1 0 9 8 7 6	5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

- I** Set if the input buffer is valid for the code, as with **IOC\_IN**.
- O** Set if the output buffer is valid for the code, as with **IOC\_OUT**. Note that for codes with both input and output parameters, both **I** and **O** will be set.
- V** Set if there are no parameters for the code, as with **IOC\_VOID**.
- T** A two-bit quantity that defines the type of IOCTL. The following values are defined:
  - 0** The IOCTL is a standard Unix IOCTL code, as with **FIONREAD** and **FIONBIO**.
  - 1** The IOCTL is a generic Windows Sockets 2 IOCTL code. New IOCTL codes defined for Windows Sockets 2 will have **T == 1**.
  - 2** The IOCTL applies only to a specific address family.
  - 3** The IOCTL applies only to a specific vendor's provider. This type allows companies to be assigned a vendor number that appears in the **Vendor/AddressFamily** member. Then, the vendor can define new ioctls specific to that vendor without having to register the IOCTL with a clearinghouse, thereby providing vendor flexibility and privacy.

The **Vendor/Address Family** is an 11-bit quantity that defines the vendor who owns the code (if **T == 3**) or that contains the address family to which the code applies (if **T == 2**). If this is a Unix IOCTL code (**T == 0**) then this member has the same value as the code on Unix. If this is a generic Windows Sockets 2 IOCTL (**T == 1**) then this member can be used as an extension of the code member to provide additional code values.

**Code** The specific IOCTL code for the operation.

The following Unix commands are supported.

## Parameters

### FIONBIO

Enables or disables nonblocking mode on socket *s*. *IpvInBuffer* points at an *unsigned long*, which is nonzero if nonblocking mode is to be enabled and zero if it is to be disabled. When a socket is created, it operates in blocking mode (that is, nonblocking mode is disabled). This is consistent with BSD sockets.

The **WSPAsyncSelect** or **WSPEventSelect** routine automatically sets a socket to nonblocking mode. If **WSPAsyncSelect** or **WSPEventSelect** has been issued on a socket, then any attempt to use **WSPIoctl** to set the socket back to blocking mode will fail with **WSAEINVAL**. To set the socket back to blocking mode, a Windows Sockets

SPI client must first disable **WSPAsyncSelect** by calling **WSPAsyncSelect** with the *lEvent* parameter equal to zero, or disable **WSPEventSelect** by calling **WSPEventSelect** with the *lNetworkEvents* parameter equal to zero.

### **FIONREAD**

Determines the amount of data that can be read atomically from socket *s*. *lpvOutBuffer* points at an unsigned long in which **WSIoctl** stores the result. If *s* is stream oriented (for example, type **SOCK\_STREAM**), **FIONREAD** returns the total amount of data that can be read in a single receive operation; this is normally the same as the total amount of data queued on the socket. If *s* is message oriented (for example, type **SOCK\_DGRAM**), **FIONREAD** returns the size of the first datagram (message) queued on the socket.

### **SIOCATMARK**

Determines whether or not all OOB data has been read. This applies only to a socket of stream style (for example, type **SOCK\_STREAM**) that has been configured for inline reception of any OOB data (**SO\_OOBINLINE**). If no OOB data is waiting to be read, the operation returns **TRUE**. Otherwise, it returns **FALSE**, and the next receive operation performed on the socket will retrieve some or all of the data preceding the mark; the Windows Sockets SPI client should use the **SIOCATMARK** operation to determine whether any remains. If there is any normal data preceding the urgent (OOB) data, it will be received in order. (Note that receive operations will *never* mix OOB and normal data in the same call.) *lpvOutBuffer* points at a **BOOL** in which **WSIoctl** stores the result.

The following Windows Sockets 2 commands are supported.

### **Parameters**

#### **SIO\_ASSOCIATE\_HANDLE** (opcode setting: I, T==1)

Associates this socket with the specified handle of a companion interface. The input buffer contains the integer value corresponding to the manifest constant for the companion interface (for example, **TH\_NETDEV** and **TH\_TAPI**), followed by a value that is a handle of the specified companion interface, along with any other required information. Refer to the appropriate section in the **Windows Sockets 2 Protocol-Specific Annex** and/or documentation for the particular companion interface for additional details. The total size is reflected in the input buffer length. No output buffer is required. The **WSAENOPROTOPT** error code is indicated for service providers that do not support this IOCTL. The handle associated by this IOCTL can be retrieved using **SIO\_TRANSLATE\_HANDLE**.

A companion interface might be used, for example, if a particular provider provides:

- A great deal of additional control over the behavior of a socket.
- Provider-specific controls that do not map to existing Windows Socket functions (or those likely for the future).

It is recommended that the Component Object Model (COM) be used instead of this IOCTL to discover and track other interfaces that might be supported by a socket. This IOCTL is present for backward compatibility with systems where COM is not available or cannot be used for some other reason.

**SIO\_ENABLE\_CIRCULAR\_QUEUEING** (opcode setting: V, T==1)

Indicates to a message-oriented service provider that a newly arrived message should never be dropped because of a buffer queue overflow. Instead, the oldest message in the queue should be eliminated in order to accommodate the newly arrived message. No input and output buffers are required. Note that this IOCTL is only valid for sockets associated with unreliable, message-oriented protocols. The WSAENOPROTOOPT error code is indicated for service providers that do not support this IOCTL.

**SIO\_FIND\_ROUTE** (opcode setting: O, T==1)

When issued, this IOCTL requests that the route to the remote address specified as a **SOCKADDR** in the input buffer be discovered. If the address already exists in the local cache, its entry is invalidated. In the case of Novell's IPX, this call initiates an IPX GetLocalTarget (GLT), that queries the network for the given remote address.

**SIO\_FLUSH** (opcode setting: V, T==1)

Discards current contents of the sending queue associated with this socket. No input and output buffers are required. The WSAENOPROTOOPT error code is indicated for service providers that do not support this IOCTL.

**SIO\_GET\_BROADCAST\_ADDRESS** (opcode setting: O, T==1)

This IOCTL fills the output buffer with a **SOCKADDR** structure containing a suitable broadcast address for use with **WSPSendTo**.

**SIO\_GET\_EXTENSION\_FUNCTION\_POINTER** (opcode setting: O, I, T==1)

Retrieves a pointer to the specified extension function supported by the associated service provider. The input buffer contains a GUID whose value identifies the extension function in question. The pointer to the desired function is returned in the output buffer. Extension function identifiers are established by service provider vendors and should be included in vendor documentation that describes extension function capabilities and semantics.

**SIO\_GET\_QOS** (opcode setting: O, T==1)

Retrieves the **QOS** structure associated with the socket. The input buffer is optional. Some protocols (for example, RSVP) allow the input buffer to be used to qualify a QOS request. The **QOS** structure will be copied into the output buffer. The output buffer must be sized large enough to be able to contain the full **QOS** structure. The WSAENOPROTOOPT error code is indicated for service providers that do not support quality of service.

**SIO\_GET\_GROUP\_QOS** (opcode setting: O, T==1)

Reserved.

**SIO\_MULTIPOINT\_LOOPBACK** (opcode setting: I, T==1)

Controls whether data sent in a multipoint session will also be received by the same socket on the local host. A value of TRUE causes loopback reception to occur while a value of FALSE prohibits this.

**SIO\_MULTICAST\_SCOPE** (opcode setting: I, T==1)

Specifies the scope over which multicast transmissions will occur. Scope is defined as the number of routed network segments to be covered. A scope of zero would indicate that the multicast transmission would not be placed on the wire, but could be disseminated across sockets within the local host. A scope value of 1 (the default) indicates that the transmission will be placed on the wire, but will *not* cross any routers. Higher scope values determine the number of routers that can be crossed. Note that this corresponds to the time-to-live (TTL) parameter in IP multicasting.

**SIO\_SET\_QOS** (opcode setting: I, T==1)

Associate the supplied **QOS** structure with the socket. No output buffer is required, the **QOS** structure will be obtained from the input buffer. The **WSAENOPROTOPT** error code is indicated for service providers that do not support quality of service.

**SIO\_SET\_GROUP\_QOS** (opcode setting: I, T==1)

Reserved.

**SIO\_TRANSLATE\_HANDLE** (opcode setting: I, O, T==1)

To obtain a corresponding handle for socket *s* that is valid in the context of a companion interface (for example, **TH\_NETDEV** and **TH\_TAPI**). A manifest constant identifying the companion interface along with any other needed parameters are specified in the input buffer. The corresponding handle will be available in the output buffer upon completion of this function. Refer to the appropriate section in the *Windows Sockets 2 Protocol-Specific Annex* and/or documentation for the particular companion interface for additional details. The **WSAENOPROTOPT** error code is indicated for service providers that do not support this **IOCTL** for the specified companion interface. This **IOCTL** retrieves the handle associated using

**SIO\_TRANSLATE\_HANDLE.**

It is recommended that **COM** be used instead of this **IOCTL** to discover and track other interfaces that might be supported by a socket. This **IOCTL** is present for backward compatibility with systems where **COM** is not available or cannot be used for some other reason.

**SIO\_ROUTING\_INTERFACE\_QUERY** (opcode setting: I, O, T==1)

To obtain the address of the local interface (represented as **SOCKADDR** structure) that should be used to send to the remote address specified in the input buffer (as **SOCKADDR**). Remote multicast addresses may be submitted in the input buffer to get the address of the preferred interface for multicast transmission. In any case, the interface address returned may be used by the application in a subsequent **bind** request.

Note that routes are subject to change. Therefore, Windows Socket SPI clients cannot rely on the information returned by **SIO\_ROUTING\_INTERFACE\_QUERY** to be persistent. SPI clients may register for routing change notifications using the **SIO\_ROUTING\_INTERFACE\_CHANGE\_IOCTL**, which provides for notification through either overlapped I/O or a **FD\_ROUTING\_INTERFACE\_CHANGE** event. The following sequence of actions can be used to guarantee that the Windows Socket SPI client always has current routing interface information for a given destination.

- Issue **SIO\_ROUTING\_INTERFACE\_CHANGE\_IOCTL**.
- Issue **SIO\_ROUTING\_INTERFACE\_QUERY\_IOCTL**.
- Whenever **SIO\_ROUTING\_INTERFACE\_CHANGE\_IOCTL** notifies the WinSock SPI client of routing change (either through overlapped I/O or by signaling **FD\_ROUTING\_INTERFACE\_CHANGE** event), the whole sequence of actions should be repeated.

If output buffer is not large enough to contain the interface address, **SOCKET\_ERROR** is returned as the result of this IOCTL and **WSPGetLastError** returns **WSAEFAULT**. The required size of the output buffer will be returned in *lpcbBytesReturned* in this case. Note the **WSAEFAULT** error code is also returned if the *lpvInBuffer*, *lpvOutBuffer*, or *lpcbBytesReturned* parameter is not totally contained in a valid part of the user address space.

If the destination address specified in the input buffer cannot be reached through any of the available interfaces, **SOCKET\_ERROR** is returned as the result of this IOCTL and **WSAGetLastError** returns **WSAENETUNREACH** or even **WSAENETDOWN** if all of the network connectivity is lost.

#### **SIO\_ROUTING\_INTERFACE\_CHANGE** (opcode setting: I, T==1)

To receive notification of the interface change that should be used to reach the remote address in the input buffer (specified as a **SOCKADDR** structure). No output information will be provided upon completion of this IOCTL; the completion merely indicates that the routing interface for a given destination has changed and should be queried again through **SIO\_ROUTING\_INTERFACE\_QUERY**.

It is assumed (although not required) that the Windows Socket SPI client uses overlapped I/O to be notified of routing interface change through completion of **SIO\_ROUTING\_INTERFACE\_CHANGE** request. Alternatively, if the **SIO\_ROUTING\_INTERFACE\_CHANGE\_IOCTL** is issued on a nonblocking socket and without overlapped parameters (*lpOverlapped / CompletionRoutine* are set to NULL), it will complete immediately with error **WSAEWOULDBLOCK** and the Windows Socket SPI client can then wait for routing change events using a call to **WSPEventSelect** or **WSPAsyncSelect** with the **FD\_ROUTING\_INTERFACE\_CHANGE** bit set in the network event bitmask.

It is recognized that routing information remains stable in most cases. So requiring the Windows Sockets SPI client to keep multiple outstanding IOCTLs—for notifications about all destinations that it is interested in as well as having the service provider keep track of all them—will unnecessarily tie up significant system resources. This situation can be avoided by extending the meaning of the input parameters and relaxing the service provider requirements as follows:

The Windows Sockets SPI client can specify a protocol family specific wildcard address (same as one used in **bind** call when requesting to bind to any available address) to request notifications of any routing changes. This allows the Windows Sockets SPI client to keep only one outstanding **SIO\_ROUTING\_INTERFACE\_CHANGE** for all the sockets/destinations it has and then use **SIO\_ROUTING\_INTERFACE\_QUERY** to get the actual routing information.



Service provider can opt to ignore the information supplied by the Windows Sockets SPI client in the input buffer of the **SIO\_ROUTING\_INTERFACE\_CHANGE** (as though the Windows Sockets SPI client specified a wildcard address) and complete the **SIO\_ROUTING\_INTERFACE\_CHANGE\_IOCTL** or signal **FD\_ROUTING\_INTERFACE\_CHANGE** event in the event of any routing information change (not just the route to the destination specified in the input buffer).

### **SIO\_ADDRESS\_LIST\_QUERY** (opcode setting: I, O, T==1)

To obtain a list of local transport addresses of the socket's protocol family to which the Windows Sockets SPI client can bind. The list returned in the output buffer using the following format:

```
typedef struct _SOCKET_ADDRESS_LIST {
...INT iAddressCount;
...SOCKET_ADDRESS Address[1];
} SOCKET_ADDRESS_LIST, FAR * LPSOCKET_ADDRESS_LIST;
Members:
...iAddressCount- number of address structures in the list;
...Address- array of protocol family specific address structures.
```

Note that in Win32 Plug and Play environments, addresses can be added and removed dynamically. Therefore, Windows Sockets SPI clients cannot rely on the information returned by **SIO\_ADDRESS\_LIST\_QUERY** to be persistent. Windows Sockets SPI clients may register for address change notifications through the **SIO\_ADDRESS\_LIST\_CHANGE\_IOCTL** that provides for notification through either overlapped I/O or **FD\_ADDRESS\_LIST\_CHANGE** event. The following sequence of actions can be used to guarantee that the Windows Sockets SPI client always has current address list information:

- Issue **SIO\_ADDRESS\_LIST\_CHANGE\_IOCTL**.
- Issue **SIO\_ADDRESS\_LIST\_QUERY\_IOCTL**.
- Whenever **SIO\_ADDRESS\_LIST\_CHANGE\_IOCTL** notifies the Windows Sockets SPI client of address list change (either through overlapped I/O or by signaling **FD\_ADDRESS\_LIST\_CHANGE** event), the whole sequence of actions should be repeated.

If the output buffer is not large enough to contain the address list, **SOCKET\_ERROR** is returned as the result of this IOCTL and **WSPGetLastError** returns **WSAEFAULT**. The required size of the output buffer will be returned in *lpcbBytesReturned* in this case. Note the **WSAEFAULT** error code is also returned if the *lpvInBuffer*, *lpvOutBuffer*, or *lpcbBytesReturned* parameter is not totally contained in a valid part of the user address space.

### **SIO\_ADDRESS\_LIST\_CHANGE** (opcode setting: T==1)

To receive notification of changes in the list of local transport addresses of the socket's protocol family to which the Windows Sockets SPI client can bind. No output information will be provided upon completion of this IOCTL; the completion merely

indicates that the list of available local addresses has changed and should be queried again through **SIO\_ADDRESS\_LIST\_QUERY**.

It is assumed (although not required) that the Windows Sockets SPI client uses overlapped I/O to be notified of change by completion of **SIO\_ADDRESS\_LIST\_CHANGE** request. Alternatively, if the **SIO\_ADDRESS\_LIST\_CHANGE** IOCTL is issued on a nonblocking socket *and* without overlapped parameters (*IpOverlapped* and *IpCompletionRoutine* are set to NULL), it will complete immediately with error **WSAEWOULDBLOCK**. The Windows Sockets SPI client can then wait for address list change events through a call to **WSPEventSelect** or **WSPAsyncSelect** with the **FD\_ADDRESS\_LIST\_CHANGE** bit set in the network event bitmask.

### **SIO\_QUERY\_PNP\_TARGET\_HANDLE** (opcode setting: O, T==1)

To obtain the socket descriptor of the next provider in the chain on which the current socket depends in PnP sense. This IOCTL is invoked by the Windows Sockets 2 DLL only on sockets of non-IFS service providers created through

**WPUCreateSocketHandle** call. The provider should return in the output buffer the socket handle of the next provider in the chain on which a given socket handle depends in PnP sense (for example, the removal of the device that supports the underlying handle will result in the invalidation of the handle above it in the chain).

If an overlapped operation completes immediately, this function returns a value of zero and the *IpCbBytesReturned* parameter is updated with the number of bytes in the output buffer. If the overlapped operation is successfully initiated and will complete later, this function returns **SOCKET\_ERROR** and indicates error code **WSA\_IO\_PENDING**. In this case, *IpCbBytesReturned* is not updated. When the overlapped operation completes, the amount of data in the output buffer is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *IpCbTransfer* parameter in **WSPGetOverlappedResult**.

When called with an overlapped socket, the *IpOverlapped* parameter must be valid for the duration of the overlapped operation. The **WSAOVERLAPPED** structure has the following form:

```
typedef struct _WSAOVERLAPPED {
    DWORD        Internal;        // reserved
    DWORD        InternalHigh;    // reserved
    DWORD        Offset;         // reserved
    DWORD        OffsetHigh;     // reserved
    WSAEVENT     hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

If the *IpCompletionRoutine* parameter is NULL, the service provider signals the **hEvent** member of *IpOverlapped* when the overlapped operation completes if it contains a valid event object handle. The Windows Sockets SPI client can use **WSPGetOverlappedResult** to poll or wait on the event object.

If *IpCompletionRoutine* is not NULL, the **hEvent** member is ignored and can be used by the Windows Sockets SPI client to pass context information to the completion routine. A client that passes a non-NULL *IpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to TRUE. In this case, the usage of the **hEvent** member is undefined, and attempting to wait on the **hEvent** member would produce unpredictable results.

It is the service provider's responsibility to arrange for invocation of the client specified—completion routine when the overlapped operation completes. Since the completion routine must be executed in the context of the same thread that initiated the overlapped operation, it cannot be invoked directly from the service provider. The *Ws2\_32.dll* offers an asynchronous procedure call (APC) mechanism to facilitate invocation of completion routines.

A service provider arranges for a function to be executed in the proper thread and process context by calling **WPUQueueApc**. This function can be called from any process and thread context, even a context different from the thread and process that was used to initiate the overlapped operation.

**WPUQueueApc** takes as input parameters a pointer to a **WSATHREADID** structure (supplied to the provider through the *IpThreadId* input parameter), a pointer to an APC function to be invoked, and a 32-bit context value that is subsequently passed to the APC function. Because only a single 32-bit context value is available, the APC function itself cannot be the client specified—completion routine. The service provider must instead supply a pointer to its own APC function that uses the supplied context value to access the needed result information for the overlapped operation, and then invokes the client specified—completion routine.

The prototype for the client-supplied completion routine is as follows:

```
void CALLBACK CompletionRoutine (
    IN    DWORD           dwError,
    IN    DWORD           cbTransferred,
    IN    LPWSAOVERLAPPED lpOverlapped,
    IN    DWORD           dwFlags
);
```

**CompletionRoutine** is a placeholder for a client supplied function. The *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* specifies the number of bytes returned. Currently, there are no flag values defined and *dwFlags* will be zero. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. The completion routines can be called in any order, though not necessarily in the same order that the overlapped operations are completed.

## Compatibility

The IOCTL codes with **T == 0** are a subset of the IOCTL codes used in Berkeley sockets. In particular, there is no command that is equivalent to **FIOASYNC**.

## Return Values

If no error occurs and the operation has completed immediately, **WSPIoctl** returns zero. Note that in this case the completion routine, if specified, will have already been queued. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code is available in *lpErrno*. The error code **WSA\_IO\_PENDING** indicates that an overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that no overlapped operation was initiated and no completion indication will occur.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEFAULT	The <i>IpvInBuffer</i> , <i>IpvOutBuffer</i> or <i>IpcbBytesReturned</i> argument is not totally contained in a valid part of the user address space, or the <i>cbInBuffer</i> or <i>cbOutBuffer</i> argument is too small.
WSAEINVAL	The <i>dwIoControlCode</i> is not a valid command, or a supplied input parameter is not acceptable, or the command is not applicable to the type of socket supplied.
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAENOTSOCK	Descriptor <i>s</i> is not a socket.
WSAEOPNOTSUPP	Specified IOCTL command cannot be realized. For example, the flow specifications specified in <b>SIO_SET_QOS</b> cannot be satisfied.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSAEWOULDBLOCK	Socket is marked as nonblocking and the requested operation would block.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

### + See Also

**WSPSocket**, **WSPSetSockOpt**, **WSPGetSockOpt**, **WPUQueueApc**

## WSPJoinLeaf

The **WSPJoinLeaf** function joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the supplied flow specifications.

```
SOCKET WSPJoinLeaf (  
    SOCKET                                s,  
    const struct sockaddr FAR            *name,  
    int                                  namelen,  
    LPWSABUF                             lpCallerData,  
    LPWSABUF                             lpCalleeData,  
    LPQOS                                 lpSQOS,  
    LPQOS                                 lpGQOS,  
    DWORD                                 dwFlags,  
    LPINT                                 lpErrno  
);
```

### Parameters

*s*

[in] Descriptor identifying a multipoint socket.

*name*

[in] Name of the peer to which the socket is to be joined.

*namelen*

[in] Length of the *name*.

*lpCallerData*

[in] Pointer to the user data that is to be transferred to the peer during multipoint session establishment.

*lpCalleeData*

[out] Pointer to the user data that is to be transferred back from the peer during multipoint session establishment.

*lpSQOS*

[in] Pointer to the flow specifications for socket *s*, one for each direction.

*lpGQOS*

[in] Reserved.

*dwFlags*

[in] Flags to indicate that the socket is acting as a sender, receiver, or both.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WSPJoinLeaf** returns a value of type **SOCKET** that is a descriptor for the newly created multipoint socket. Otherwise, a value of **INVALID\_SOCKET** is returned, and a specific error code is available in *lpErrno*.

On a blocking socket, the return value indicates success or failure of the join operation.

With a nonblocking socket, successful initiation of a join operation is indicated by a return value of a valid socket descriptor. Subsequently, an **FD\_CONNECT** indication is given when the join operation completes, either successfully or otherwise. The error code associated with the **FD\_CONNECT** indicates the success or failure of the **WSPJoinLeaf**.

Also, until the multipoint session join attempt completes all subsequent calls to **WSPJoinLeaf** on the same socket will fail with the error code **WSAEALREADY**. After the **WSAJoinLeaf** completes successfully a subsequent attempt will usually fail with the error code **WSAEISCONN**. An exception to the **WSAEISCONN** rule occurs for a **c\_root** socket that allows root-initiated joins. In such a case another join may be initiated after a prior **WSAJoinLeaf** completes.

If the return error code indicates the multipoint session join attempt failed (that is, **WSAECONNREFUSED**, **WSAENETUNREACH**, **WSAETIMEDOUT**) the Windows Sockets SPI client can call **WSPJoinLeaf** again for the same socket.

## Remarks

This function is used to join a leaf node to a multipoint session, and to perform a number of other ancillary operations that occur at session join time as well. If the socket, *s*, is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound.

**WSPJoinLeaf** has the same parameters and semantics as **WSPConnect** except that it returns a socket descriptor (as in **WSPAccept**), and it has an additional *dwFlags* parameter. Only multipoint sockets created using **WSPSocket** with appropriate multipoint flags set can be used for input parameter *s* in this function. If the socket is in the nonblocking mode, the returned socket descriptor will *not* be useable until after a corresponding **FD\_CONNECT** indication on the original socket *s* has been received, except that **closesocket** can be invoked on this new socket descriptor to cancel a pending join operation. A root node in a multipoint session can call **WSPJoinLeaf** one or more times in order to add a number of leaf nodes, however at most one multipoint connection request can be outstanding at a time. Refer to *Protocol-Independent Multicast and Multipoint in the SPI* for additional information.

For nonblocking sockets it is often not possible to complete the connection immediately. In such a case, this function returns an as-yet unusable socket descriptor and the operation proceeds. There is no error code such as **WSAEWOULDBLOCK** in this case, since the function has effectively returned a “successful start” indication. When the final outcome success or failure becomes known, it may be reported through **WSPAsyncSelect** or **WSPEventSelect** depending on how the client registers for

notification on the original socket *s*. In either case, the notification is announced with `FD_CONNECT` and the error code associated with the `FD_CONNECT` indicates either success or a specific reason for failure. Note that **WSPSelect** cannot be used to detect completion notification for **WSAJoinLeaf**.

The socket descriptor returned by **WSPJoinLeaf** is different depending on whether the input socket descriptor, *s*, is a `c_root` or a `c_leaf`. When used with a `c_root` socket, the *name* parameter designates a particular leaf node to be added and the returned socket descriptor is a `c_leaf` socket corresponding to the newly added leaf node. (As is described in section *Descriptor Allocation*, when new socket descriptors are allocated IFS providers must call **WPUModifyIFSHandle** and non-IFS providers must call **WPUCreateSocketHandle**). The newly created socket has the same properties as *s* including asynchronous events registered with **WSPAsyncSelect** or with **WSPEventSelect**. It is not intended to be used for exchange of multipoint data, but rather is used to receive network event indications (for example, `FD_CLOSE`) for the connection that exists to the particular `c_leaf`. Some multipoint implementations can also allow this socket to be used for “side chats” between the root and an individual leaf node. An `FD_CLOSE` indication will be received for this socket if the corresponding leaf node calls **WSPCloseSocket** to drop out of the multipoint session. Symmetrically, invoking **WSPCloseSocket** on the `c_leaf` socket returned from **WSPJoinLeaf** will cause the socket in the corresponding leaf node to get `FD_CLOSE` notification.

When **WSPJoinLeaf** is invoked with a `c_leaf` socket, the *name* parameter contains the address of the root node (for a rooted control scheme) or an existing multipoint session (nonrooted control scheme), and the returned socket descriptor is the same as the input socket descriptor. In other words, a new socket descriptor is *not* allocated. In a rooted control scheme, the root application would put its `c_root` socket in the listening mode by calling **WSPListen**. The standard `FD_ACCEPT` notification will be delivered when the leaf node requests to join itself to the multipoint session. The root application uses the usual **WSPAccept** functions to admit the new leaf node. The value returned from **WSPAccept** is also a `c_leaf` socket descriptor just like those returned from **WSPJoinLeaf**. To accommodate multipoint schemes that allow both root-initiated and leaf-initiated joins, it is acceptable for a `c_root` socket that is already in listening mode to be used as an input to **WSPJoinLeaf**.

The Windows Sockets SPI client is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specifies.

The *IpCallerData* is a value parameter that contains any user data that is to be sent along with the multipoint session join request. If *IpCallerData* is `NULL`, no user data will be passed to the peer. The *IpCalleeData* is a result parameter that will contain any user data passed back from the peer as part of the multipoint session establishment. *IpCalleeData->len* initially contains the length of the buffer allocated by the Windows Sockets SPI client and pointed to by *IpCalleeData->buf*. *IpCalleeData->len* will be set to zero if no user data has been passed back. The *IpCalleeData* information will be valid when the multipoint join operation is complete. For blocking sockets, this will be when the **WSPJoinLeaf** function returns. For nonblocking sockets, this will be after the `FD_CONNECT` notification has occurred on the original socket *s*. If *IpCalleeData* is

NULL, no user data will be passed back. The exact format of the user data is specific to the address family to which the socket belongs and/or the applications involved.

At multipoint session establishment time, a Windows Sockets SPI client can use the *lpSQOS* parameters to override any previous QOS specification made for the socket through **WSPIoctl** with the SIO\_SET\_QOS opcode.

*lpSQOS* specifies the flow specifications for socket *s*, one for each direction, followed by any additional provider-specific parameters. If either the associated transport provider in general or the specific type of socket in particular cannot honor the QOS request, an error will be returned as indicated below. The sending or receiving flow specification values will be ignored, respectively, for any unidirectional sockets. If no provider-specific parameters are supplied, the *buf* and *len* members of *lpSQOS->ProviderSpecific* should be set to NULL and zero, respectively. A NULL value for *lpSQOS* indicates no application supplied quality of service.

The *dwFlags* parameter is used to indicate whether the socket will be acting only as a sender (JL\_SENDER\_ONLY), only as a receiver (JL\_RECEIVER\_ONLY), or both (JL\_BOTH).

---

**Note** When connected sockets break (that is, become closed for whatever reason), they should be discarded and recreated. It is safest to assume that when things go awry for any reason on a connected socket, the Windows Sockets SPI client must discard and recreate the needed sockets in order to return to a stable point.

---

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEADDRINUSE	Socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs at the time of <b>bind</b> , but could be delayed until this function if the <b>bind</b> was to a partially wild-card address (involving ADDR_ANY) and if a specific address needs to be "committed" at the time of this function.
WSAEINTR	(Blocking) call was canceled through <b>WSPCancelBlockingCall</b> .
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	Nonblocking <b>WSPJoinLeaf</b> call is in progress on the specified socket.
WSAEADDRNOTAVAIL	Remote address is not a valid address (for example, ADDR_ANY).

(continued)



*(continued)*

<b>Error code</b>	<b>Meaning</b>
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to join was forcefully rejected.
WSAEFAULT	<i>Name</i> or the <i>namelen</i> argument is not a valid part of the user address space, the <i>namelen</i> argument is too small, the buffer length for <i>lpCalleeData</i> , <i>lpSQOS</i> , and <i>lpGQOS</i> are too small, or the buffer length for <i>lpCallerData</i> is too large.
WSAEISCONN	Socket is already member of the multipoint session.
WSAENETUNREACH	Network cannot be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be joined.
WSAENOTSOCK	Descriptor is not a socket.
WSAEOPNOTSUPP	Flow specifications specified in <i>lpSQOS</i> cannot be satisfied.
WSAEPROTONOSUPPORT	The <i>lpCallerData</i> argument is not supported by the service provider.
WSAETIMEDOUT	Attempt to join timed out without establishing a multipoint session.

**!** Requirements**Version:** Requires Windows Sockets 2.0.**Header:** Declared in `Ws2spi.h`.**+** See Also**WSPBind**, **WSPSelect**, **WSPAccept**, **WSPAsyncSelect**, **WSPEventSelect**, **WSPSocket**

---

## WSPListen

The **WSPListen** function establishes a socket to listen for incoming connections.

```
int WSPListen (
    SOCKET    s,
    int       backlog,
    LPINT     lpErrno
);
```

## Parameters

*s*

[in] Descriptor identifying a bound, unconnected socket.

*backlog*

[in] Maximum length to which the queue of pending connections can grow. If this value is SOMAXCONN, then the service provider should set the backlog to a maximum “reasonable” value. There is no standard provision to find out the actual backlog value.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WSPListen** returns zero. Otherwise, a value of SOCKET\_ERROR is returned, and a specific error code is available in *lpErrno*.

## Remarks

To accept connections, a socket is first created with **WSPSocket** bound to a local address with **WSPBind**, a backlog for incoming connections is specified with **WSPListen**, and then the connections are accepted with **WSPAccept**. **WSPListen** applies only to sockets that are connection oriented (for example, **SOCK\_STREAM**). The socket *s* is put into passive mode where incoming connection requests are acknowledged and queued pending acceptance by the Windows Sockets SPI client.

This function is typically used by servers that could have more than one connection request at a time: if a connection request arrives with the queue full, the client will receive an error with an indication of WSAECONNREFUSED.

**WSPListen** should continue to function rationally when there are no available descriptors. It should accept connections until the queue is emptied. If descriptors become available, a later call to **WSPListen** or **WSPAccept** will refill the queue to the current or most recent “backlog”, if possible, and resume listening for incoming connections.

A Windows Sockets SPI client can call **WSPListen** more than once on the same socket. This has the effect of updating the current backlog for the listening socket. Should there be more pending connections than the new *backlog* value, the excess pending connections will be reset and dropped.

## Compatibility

*backlog* is limited (silently) to a reasonable value as determined by the service provider. Illegal values are replaced by the nearest legal value. There is no standard provision to find out the actual backlog value.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEADDRINUSE	Socket's local address is already in use and the socket was not marked to allow address reuse with <code>SO_REUSEADDR</code> . This error usually occurs at the time of <b>bind</b> , but could be delayed until this function if the <b>bind</b> was to a partially wildcard address (involving <code>ADDR_ANY</code> ) and if a specific address needs to be committed at the time of this function.
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAEINVAL	Socket has not been bound with <b>WSPBind</b> .
WSAEISCONN	Socket is already connected.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	Descriptor is not a socket.
WSAEOPNOTSUPP	Referenced socket is not of a type that supports the <b>WSPListen</b> operation.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

### + See Also

**WSPAccept**, **WSPConnect**, **WSPSocket**

## WSPRecv

The **WSPRecv** function receives data on a socket.

```
int WSPRecv (
    SOCKET          s,
    LPWSABUF       lpBuffers,
    DWORD          dwBufferCount,
    LPDWORD        lpNumberOfBytesRecvd,
    LPDWORD        lpFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    LPWSATHREADID lpThreadId,
    LPINT          lpErrno
);
```

## Parameters

*s*

[in] Descriptor identifying a connected socket.

*lpBuffers*

[in/out] Pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer.

*dwBufferCount*

[in] Number of **WSABUF** structures in the *lpBuffers* array.

*lpNumberOfBytesRecv*

[out] Pointer to the number of bytes received by this call.

*lpFlags*

[in/out] Pointer to flags.

*lpOverlapped*

[in] Pointer to a **WSAOVERLAPPED** structure (ignored for nonoverlapped structures).

*lpCompletionRoutine*

[in] Pointer to the completion routine called when the receive operation has been completed (ignored for nonoverlapped structures).

*lpThreadId*

[in] Pointer to a **WSATHREADID** structure to be used by the provider in a subsequent call to **WPUQueueApc**. The provider should store the referenced **WSATHREADID** structure (not the pointer to same) until after the **WPUQueueApc** function returns.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs and the receive operation has completed immediately, **WSPRecv** returns zero. Note that in this case the completion routine, if specified, will have already been queued. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code is available in *lpErrno*. The error code **WSA\_IO\_PENDING** indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that no overlapped operations was initiated and no completion indication will occur.

## Remarks

**WSPRecv** is used on connected sockets or bound connectionless sockets specified by the *s* parameter and is used to read incoming data. The socket's local address must be known. This may be done explicitly through **WSPBind** or implicitly through **WSPAccept**, **WSPConnect**, **WSPSendTo**, or **WSPJoinLeaf**.

For connected, connectionless sockets, this function restricts the addresses from which received messages are accepted. The function only returns messages from the remote address specified in the connection. Messages from other addresses are (silently) discarded.

For overlapped sockets **WSPRecv** is used to post one or more buffers into which incoming data will be placed as it becomes available, after which the Windows Sockets SPI client-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or **WSPGetOverlappedResult**.

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the *lpOverlapped*, *lpCompletionRoutine*, and *lpThreadId* parameters are ignored. Any data that has already been received and buffered by the transport will be copied into the supplied user buffers. For the case of a blocking socket with no data currently having been received and buffered by the transport, the call will block until data is received. Windows Sockets 2 does not define any standard blocking timeout mechanism for this function. For protocols acting as byte-stream protocols the stack tries to return as much data as possible subject to the supplied buffer space and amount of received data available. However, receipt of a single byte is sufficient to unblock the caller. There is no guarantee that more than a single byte will be returned. For protocols acting as message-oriented, a full message is required to unblock the caller.

Whether or not a protocol is acting as byte-stream is determined by the setting of **XP1\_MESSAGE\_ORIENTED** and **XP1\_PSEUDO\_STREAM** in its **WSAPROTOCOL\_INFO** structure and the setting of the **MSG\_PARTIAL** flag passed in to this function (for protocols that support it). The relevant combinations are summarized in the following table (an asterisk (\*) indicates that the setting of this bit does not matter in this case).

<b>XP1_MESSAGE_ORIENTED</b>	<b>XP1_PSEUDO_STREAM</b>	<b>MSG_PARTIAL</b>	<b>Acts as</b>
not set	*	*	byte-stream
*	set	*	byte-stream
set	not set	set	byte-stream
set	not set	not set	message-oriented

The supplied buffers are filled in the order in which they appear in the array pointed to by *lpBuffers*, and the buffers are packed so that no holes are created.

The array of **WSABUF** structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture this array of pointers to **WSABUF** structures before returning from this call. This enables Windows Sockets SPI clients to build stack-based **WSABUF** arrays.

For byte stream-style sockets (for example, type **SOCK\_STREAM**), incoming data is placed into the buffers until the buffers are filled, the connection is closed, or internally buffered data is exhausted. Regardless of whether or not the incoming data fills all the buffers, the completion indication occurs for overlapped sockets. For message-oriented sockets (for example, type **SOCK\_DGRAM**), an incoming message is placed into the supplied buffers, up to the total size of the buffers supplied, and the completion indication occurs for overlapped sockets. If the message is larger than the buffers supplied, the buffers are filled with the first part of the message. If the **MSG\_PARTIAL** feature is supported by the service provider, the **MSG\_PARTIAL** flag is set in *lpFlags* and subsequent receive operation(s) can be used to retrieve the rest of the message. If **MSG\_PARTIAL** is not supported but the protocol is reliable, **WSARecv** generates the error **WSAEMSGSIZE** and a subsequent receive operation with a larger buffer can be used to retrieve the entire message. Otherwise, (that is, the protocol is unreliable and does not support **MSG\_PARTIAL**), the excess data is lost, and **WSARecv** generates the error **WSAEMSGSIZE**.

For connection-oriented sockets, **WSARecv** can indicate the graceful termination of the virtual circuit in one of two ways, depending on whether the socket is a byte stream or message oriented. For byte streams, zero bytes having been read indicates graceful closure and that no more bytes will ever be read. For message-oriented sockets, where a zero byte message is often allowable, a return error code of **WSAEDISCON** is used to indicate graceful closure. In any case a return error code of **WSAECONNRESET** indicates an abortive close has occurred.

*lpFlags* can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *lpFlags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

Value	Meaning
<b>MSG_PEEK</b>	Peeks at the incoming data. The data is copied into the buffer but is not removed from the input queue. This flag is valid only for nonoverlapped sockets.
<b>MSG_OOB</b>	Processes OOB data (See section. <i>DECnet Out-Of-band data</i> for a discussion of this topic.)
<b>MSG_PARTIAL</b>	This flag is for message-oriented sockets only. On output, indicates that the data supplied is a portion of the message transmitted by the sender. Remaining portions of the message will be supplied in subsequent receive operations. A subsequent receive operation with <b>MSG_PARTIAL</b> flag cleared indicates end of sender's message.  As an input parameter, <b>MSG_PARTIAL</b> indicates that the receive operation should complete even if only part of a message has been received by the service provider.

## Overlapped Socket I/O

If an overlapped operation completes immediately, **WSPRecv** returns a value of zero and the *lpNumberOfBytesRecv* parameter is updated with the number of bytes received and the flag bits pointed by the *lpFlags* parameter are also updated. If the overlapped operation is successfully initiated and will complete later, **WSPRecv** returns **SOCKET\_ERROR** and indicates error code **WSA\_IO\_PENDING**. In this case, *lpNumberOfBytesRecv* and *lpFlags* are not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **WSPGetOverlappedResult**. Flag values are obtained either through the *dwFlags* parameter of the completion routine, or by examining the *lpdwFlags* parameter of **WSPGetOverlappedResult**.

Providers must allow this function to be called from within the completion routine of a previous **WSPRecv**, **WSPRecvFrom**, **WSPSend** or **WSPSendTo** function. However, for a given socket, I/O completion routines can not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The **WSAOVERLAPPED** structure has the following form:

```
typedef struct _WSAOVERLAPPED {
    DWORD      Internal;           // reserved
    DWORD      InternalHigh;      // reserved
    DWORD      Offset;            // reserved
    DWORD      OffsetHigh;        // reserved
    WSAEVENT   hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

If the *lpCompletionRoutine* parameter is **NULL**, the service provider signals the *hEvent* member of *lpOverlapped* when the overlapped operation completes if it contains a valid event object handle. The Windows Sockets SPI client can use **WSPGetOverlappedResult** to wait or poll on the event object.

If *lpCompletionRoutine* is not **NULL**, the *hEvent* member is ignored and can be used by the Windows Sockets SPI client to pass context information to the completion routine. A client that passes a non-**NULL** *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case the usage of the *hEvent* member is undefined, and attempting to wait on the *hEvent* member would produce unpredictable results.

It is the service provider's responsibility to arrange for invocation of the client specified—completion routine when the overlapped operation completes. Since the completion routine must be executed in the context of the same thread that initiated the overlapped

operation, it cannot be invoked directly from the service provider. The `Ws2_32.dll` offers an asynchronous procedure call (APC) mechanism to facilitate invocation of completion routines.

A service provider arranges for a function to be executed in the proper thread and process context by calling **WPUQueueApc**, which was used to initiate the overlapped operation. This function can be called from any process and thread context, even a context different from the thread and process that was used to initiate the overlapped operation.

**WPUQueueApc** takes as input parameters a pointer to a **WSATHREADID** structure (supplied to the provider through the *lpThreadId* input parameter), a pointer to an APC function to be invoked, and a 32-bit context value that is subsequently passed to the APC function. Because only a single 32-bit context value is available, the APC function itself cannot be the client-specified completion routine. The service provider must instead supply a pointer to its own APC function that uses the supplied context value to access the needed result information for the overlapped operation, and then invokes the client-specified completion routine.

The prototype for the client-supplied completion routine is as follows:

```
void CALLBACK CompletionRoutine (
    IN    DWORD           dwError,
    IN    DWORD           cbTransferred,
    IN    LPWSAOVERLAPPED lpOverlapped,
    IN    DWORD           dwFlags
);
```

**CompletionRoutine** is a placeholder for a client-supplied function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes received. *dwFlags* contains information that would have appeared in *lpFlags* if the receive operation had completed immediately. This function does not return a value.

The completion routines can be called in any order, but not necessarily the same order in which the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order in which they are supplied.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAENOTCONN	Socket is not connected.
WSAEINTR	(Blocking) call was canceled through <b>WSPCancelBlockingCall</b> .
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.

(continued)



*(continued)*

Error code	Meaning
WSAENETRESET	Connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAEFAULT	The <i>lpBuffers</i> argument is not totally contained in a valid part of the user address space.
WSAENOTSOCK	Descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	Socket has been shut down; it is not possible to receive through <b>WSPRecv</b> on a socket after <b>WSPShutdown</b> has been invoked with <i>how</i> set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: there are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the receive operation cannot be completed immediately.
WSAEMSGSIZE	Message was too large to fit into the specified buffer and (for unreliable protocols only) any trailing portion of the message that did not fit into the buffer has been discarded.
WSAEINVAL	Socket has not been bound (for example, with <b>WSPBind</b> ) or the socket is not created with the overlapped flag.
WSAECONNABORTED	Virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	Virtual circuit was reset by the remote side.
WSAEDISCON	Socket <i>s</i> is message oriented and the virtual circuit was gracefully closed by the remote side.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	Overlapped operation has been canceled due to the closure of the socket.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

#### + See Also

**WPUCloseEvent**, **WPUCreateEvent**, **WSPGetOverlappedResult**, **WSPSocket**, **WPUQueueApc**

# WSPRecvDisconnect

The **WSPRecvDisconnect** function terminates reception on a socket and retrieves the disconnect data, if the socket is connection oriented.

```
int WSPRecvDisconnect (
    SOCKET      s,
    LPWSABUF   lpInboundDisconnectData,
    LPINT       lpErrno
);
```

## Parameters

*s*

[in] Descriptor identifying a socket.

*lpInboundDisconnectData*

[out] Pointer to a buffer into which disconnect data is to be copied.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WSPRecvDisconnect** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code is available in *lpErrno*.

## Remarks

**WSPRecvDisconnect** is used on connection-oriented sockets to disable reception, and retrieve any incoming disconnect data from the remote party.

After this function has been successfully issued, subsequent receives on the socket will be disallowed. This has no effect on the lower protocol layers. For TCP, the TCP window is not changed and incoming data will be accepted (but not acknowledged) until the window is exhausted. For UDP, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated.

To successfully receive incoming disconnect data, a Windows Sockets SPI client must use other mechanisms to determine that the circuit has been closed. For example, a client needs to receive an `FD_CLOSE` notification, or get a zero return value, or a `WSAEDISCON` error code from **WSPRecv**.

Note that **WSPRecvDisconnect** does not close the socket, and resources attached to the socket will *not* be freed until **WSPCloseSocket** is invoked.

---

**Note** **WSPRecvDisconnect** does not block regardless of the `SO_LINGER` setting on the socket.

A Windows Sockets SPI client should not rely on being able to reuse a socket after it has been **WSPRecvDisconnected**. In particular, a Windows Sockets provider is not required to support the use of **WSPConnect** on such a socket.

---

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEFAULT	Buffer referenced by the parameter <i>lpinboundDisconnectData</i> is too small.
WSAENOPROTOOPT	Disconnect data is not supported by the indicated protocol family.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	Socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	Descriptor is not a socket.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

### + See Also

**WSPConnect**, **WSPSocket**

---

## WSPRecvFrom

The **WSPRecvFrom** function receives a datagram and stores the source address.

```
int WSPRecvFrom (
    SOCKET                s,
    LPWSABUF             lpBuffers,
    DWORD                dwBufferCount,
    LPDWORD              lpNumberOfBytesRecvd,
    LPDWORD              lpFlags,
    struct sockaddr FAR *lpFrom,
    LPINT               lpFromlen,
    LPWSAOVERLAPPED     lpOverlapped,
```

```

LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
LPWSATHREADID lpThreadId,
LPINT lpErrno
);

```

## Parameters

*s*

[in] Descriptor identifying a socket.

*lpBuffers*

[in/out] Pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer.

*dwBufferCount*

[in] Number of **WSABUF** structures in the *lpBuffers* array.

*lpNumberOfBytesRecv*

[out] Pointer to the number of bytes received by this call.

*lpFlags*

[in/out] Pointer to flags.

*lpFrom*

[out] An optional pointer to a buffer that will hold the source address upon the completion of the overlapped operation.

*lpFromlen*

[in/out] Pointer to the size of the *from* buffer, required only if *lpFrom* is specified.

*lpOverlapped*

[in] Pointer to a **WSAOVERLAPPED** structure (ignored for nonoverlapped sockets).

*lpCompletionRoutine*

[in] Pointer to the completion routine called when the receive operation has been completed (ignored for nonoverlapped sockets).

*lpThreadId*

[in] Pointer to a **WSATHREADID** structure to be used by the provider in a subsequent call to **WPUQueueApc**. The provider should store the referenced **WSATHREADID** structure (not the pointer to same) until after the **WPUQueueApc** function returns.

*lpErrno*

[in/out] Pointer to the error code.

## Return Values

If no error occurs and the receive operation has completed immediately, **WSPRecvFrom** returns zero. Note that in this case the completion routine, if specified will have already been queued. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code is available in *lpErrno*. The error code **WSA\_IO\_PENDING** indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that no overlapped operations was initiated and no completion indication will occur.

## Remarks

**WSPRecvFrom** is used primarily on a connectionless socket specified by *s*. The socket must not be connected. The local address of the socket must be known. This may be done explicitly through **WSPBind** or implicitly through **WSPSendTo** or **WSPJoinLeaf**.

For overlapped sockets, this function is used to post one or more buffers into which incoming data will be placed as it becomes available on a (possibly connected) socket, after which the client-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or **WSPGetOverlappedResult**. Also note that the values pointed to by *lpFrom* and *lpFromlen* are not updated until completion is indicated. Applications must not use or disturb these values until they have been updated, therefore the client must not use automatic (that is, stack-based) variables for these parameters.

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the *lpOverlapped*, *lpCompletionRoutine*, and *lpThreadId* parameters are ignored. Any data that has already been received and buffered by the transport will be copied into the supplied user buffers. For the case of a blocking socket with no data currently having been received and buffered by the transport, the call will block until data is received according to the assigned blocking semantics for **WSPRecv**.

The supplied buffers are filled in the order in which they appear in the array pointed to by *lpBuffers*, and the buffers are packed so that no holes are created.

The array of **WSABUF** structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture this array of pointers to **WSABUF** structures before returning from this call. This enables Windows Sockets SPI clients to build stack-based **WSABUF** arrays.

For connectionless socket types, the address from which the data originated is copied to the buffer pointed by *lpFrom*. On input, the value pointed to by *lpFromlen* is initialized to the size of this buffer, and is modified on completion to indicate the actual size of the address stored there.

As noted previously for overlapped sockets, the *lpFrom* and *lpFromlen* parameters are not updated until after the overlapped I/O has completed. The memory pointed to by these parameters must, therefore, remain available to the service provider and cannot be allocated on the Windows Sockets SPI client's stack frame. The *lpFrom* and *lpFromlen* parameters are ignored for connection-oriented sockets.

For byte stream style sockets (for example, type **SOCK\_STREAM**), incoming data is placed into the buffers until the buffers are filled, the connection is closed, or internally buffered data is exhausted. Regardless of whether or not the incoming data fills all the buffers, the completion indication occurs for overlapped sockets.

For message-oriented sockets, a single incoming message is placed into the supplied buffers, up to the total size of the buffers supplied, and the completion indication occurs for overlapped sockets. If the message is larger than the buffers supplied, the buffers are filled with the first part of the message. If the `MSG_PARTIAL` feature is supported by the service provider, the `MSG_PARTIAL` flag is set in *lpFlags* for the socket and subsequent receive operation(s) will retrieve the rest of the message. If `MSG_PARTIAL` is not supported but the protocol is reliable, **WSPRecvFrom** generates the error `WSAEMSGSIZE` and a subsequent receive operation with a larger buffer can be used to retrieve the entire message. Otherwise, (that is, the protocol is unreliable and does not support `MSG_PARTIAL`), the excess data is lost, and **WSPRecvFrom** generates the error `WSAEMSGSIZE`.

*lpFlags* can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *lpFlags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

Value	Meaning
<code>MSG_PEEK</code>	Peeks at the incoming data. The data is copied into the buffer but is not removed from the input queue. This flag is valid only for nonoverlapped sockets.
<code>MSG_OOB</code>	Processes OOB data (See <i>DECnet Out-Of-band data</i> for a discussion of this topic.)
<code>MSG_PARTIAL</code>	This flag is for message-oriented sockets only. On output, indicates that the data supplied is a portion of the message transmitted by the sender. Remaining portions of the message will be supplied in subsequent receive operations. A subsequent receive operation with <code>MSG_PARTIAL</code> flag cleared indicates end of sender's message.  As an input parameter, <code>MSG_PARTIAL</code> indicates that the receive operation should complete even if only part of a message has been received by the service provider.

For message-oriented sockets, the `MSG_PARTIAL` bit is set in the *lpFlags* parameter if a partial message is received. If a complete message is received, `MSG_PARTIAL` is cleared in *lpFlags*. In the case of delayed completion, the value pointed to by *lpFlags* is not updated. When completion has been indicated the Windows Sockets SPI client should call **WSPGetOverlappedResult** and examine the flags pointed to by the *lpdwFlags* parameter.

### Overlapped Socket I/O

If an overlapped operation completes immediately, **WSPRecv** returns a value of zero and the *lpNumberOfBytesRecv* parameter is updated with the number of bytes received and the flag bits pointed by the *lpFlags* parameter are also updated. If the overlapped operation is successfully initiated and will complete later, **WSPRecv** returns `SOCKET_ERROR` and indicates error code `WSA_IO_PENDING`. In this case,

*lpNumberOfBytesRecv* and *lpFlags* is not updated. When the overlapped operation completes, the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **WSPGetOverlappedResult**. Flag values are obtained by examining the *lpdwFlags* parameter of **WSPGetOverlappedResult**.

Providers must allow this function to be called from within the completion routine of a previous **WSPRecv**, **WSPRecvFrom**, **WSPSend**, or **WSPSendTo** function. However, for a given socket, I/O completion routines cannot be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The **WSAOVERLAPPED** structure has the following form:

```
typedef struct _WSAOVERLAPPED {
    DWORD      Internal;      // reserved
    DWORD      InternalHigh;  // reserved
    DWORD      Offset;       // reserved
    DWORD      OffsetHigh;   // reserved
    WSAEVENT   hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

If the *lpCompletionRoutine* parameter is null, the service provider signals the *hEvent* member of *lpOverlapped* when the overlapped operation completes if it contains a valid event object handle. A Windows Sockets SPI client can use **WSPGetOverlappedResult** to wait or poll on the event object.

If *lpCompletionRoutine* is not null, the *hEvent* member is ignored and can be used by the Windows Sockets SPI client to pass context information to the completion routine. It is the service provider's responsibility to arrange for invocation of the client-specified completion routine when the overlapped operation completes. Since the completion routine must be executed in the context of the same thread that initiated the overlapped operation, it cannot be invoked directly from the service provider. The *Ws2\_32.dll* offers an asynchronous procedure call (APC) mechanism to facilitate invocation of completion routines.

A service provider arranges for a function to be executed in the proper thread and process context by calling **WPUQueueApc**. This function can be called from any process and thread context, even a context different from the thread and process that was used to initiate the overlapped operation.

**WPUQueueApc** takes as input parameters a pointer to a **WSATHREADID** structure (supplied to the provider through the *lpThreadId* input parameter), a pointer to an APC function to be invoked, and a 32-bit context value that is subsequently passed to the APC function. Because only a single 32-bit context value is available, the APC function itself cannot be the client specified—completion routine. The service provider must

instead supply a pointer to its own APC function that uses the supplied context value to access the needed result information for the overlapped operation, and then invokes the client specified—completion routine.

The prototype for the client-supplied completion routine is as follows:

```
void CALLBACK CompletionRoutine (
    IN    DWORD           dwError,
    IN    DWORD           cbTransferred,
    IN    LPWSAOVERLAPPED lpOverlapped,
    IN    DWORD           dwFlags
);
```

**CompletionRoutine** is a placeholder for a client-supplied function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes received. *dwFlags* contains information that would have appeared in *lpFlags* if the receive operation had completed immediately. This function does not return a value.

The completion routines can be called in any order, though not necessarily in the same order that the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order they are supplied.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEFAULT	The <i>lpFromlen</i> argument was invalid: the <i>lpFrom</i> buffer was too small to accommodate the peer address or <i>lpbuffers</i> is not totally contained within a valid part of the user address space.
WSAEINTR	(Blocking) call was canceled through <b>WSPCancelBlockingCall</b> .
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	Socket has not been bound (for example, with <b>WSPBind</b> ) or the socket is not created with the overlapped flag.
WSAEISCONN	Socket is connected. This function is not permitted with a connected socket, whether the socket is connection-oriented or connectionless.
WSAENETRESET	Connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOTSOCK	Descriptor is not a socket.

(continued)



*(continued)*

Error code	Meaning
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	Socket has been shut down; it is not possible to run <b>WSPRecvFrom</b> on a socket after <b>WSPShutdown</b> has been invoked with <i>how</i> set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: There are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the receive operation cannot be completed immediately.
WSAEMSGSIZE	Message was too large to fit into the specified buffer and (for unreliable protocols only) any trailing portion of the message that did not fit into the buffer has been discarded.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer useable. On a UDP datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEDISCON	Socket <i>s</i> is message oriented and the virtual circuit was gracefully closed by the remote side.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	Overlapped operation has been canceled due to the closure of the socket.

#### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

#### + See Also

**WSPSocket**, **WSPGetOverlappedResult**, **WPUQueueApc**

## WSPSelect

The **WSPSelect** function determines the status of one or more sockets.

```
int WSPSelect (
    int
```

```
    nfd,
```

```

fd_set FAR          *readfds,
fd_set FAR          *writefds,
fd_set FAR          *exceptfds,
const struct timeval FAR *timeout,
LPINT               lpErrno
);

```

## Parameters

*nfds*

[in] Ignored and included only for the sake of compatibility.

*readfds*

[in/out] An optional pointer to a set of sockets to be checked for readability.

*writefds*

[in/out] An optional pointer to a set of sockets to be checked for writability

*exceptfds*

[in/out] An optional pointer to a set of sockets to be checked for errors.

*timeout*

[in] Maximum time for **WSPSelect** to wait, or NULL for a blocking operation.

*lpErrno*

[out] Pointer to the error code.

## Return Values

**WSPSelect** returns the total number of descriptors that are ready and contained in the **FD\_SET** structures, or **SOCKET\_ERROR** if an error occurred. If the return value is **SOCKET\_ERROR**, a specific error code is available in *lpErrno*.

## Remarks

This function is used to determine the status of one or more sockets. For each socket, the caller can request information on read, write, or error status. The set of sockets for which a given status is requested is indicated by an **FD\_SET** structure. All entries in an **FD\_SET** correspond to sockets created by the service provider (that is, the **WSAPROTOCOL\_INFOW** structures describing their protocols have the same *providerId* value). Upon return, the structures are updated to reflect the subset of these sockets that meet the specified condition, and **WSPSelect** returns the total number of sockets meeting the conditions. A set of macros is provided for manipulating an **FD\_SET**. These macros are compatible with those used in the Berkeley software, but the underlying representation is completely different.

The parameter *readfds* identifies those sockets that are to be checked for readability. If the socket is currently listening through **WSPListen**, it will be marked as readable if an incoming connection request has been received, so that a **WSPAccept** is guaranteed to complete without blocking. For other sockets, readability means that queued data is available for reading so that a **WSPRecv** or **WSPRecvfrom** is guaranteed not to block.

For connection-oriented sockets, readability can also indicate that a close request has been received from the peer. If the virtual circuit was closed gracefully, then a **WSPRecv** will return immediately with zero bytes read. If the virtual circuit was reset, then a **WSPRecv** will complete immediately with an error code, such as **WSAECONNRESET**. The presence of OOB data will be checked if the socket option **SO\_OOBINLINE** has been enabled (see **WSPSetSockOpt**).

The parameter *writefds* identifies those sockets that are to be checked for writability:

- If a socket is connecting through **WSPConnect**, writability means that the connection establishment successfully completed.
- If the socket is not in the process of listening through **WSPConnect**, writability means that a **WSPSend** or **WSPSendTo** are guaranteed to succeed.

However, they can block on a blocking socket if the *len* exceeds the amount of outgoing system buffer space available. It is not specified how long these guarantees can be assumed to be valid, particularly in a multithreaded environment.

The parameter *exceptfds* identifies those sockets that are to be checked for the presence of OOB data or any exceptional error conditions. Note that OOB data will only be reported in this way if the option **SO\_OOBINLINE** is **FALSE**. If a socket is making a **WSPConnect** (nonblocking) connection, failure of the connect attempt is indicated in *exceptfds*. This specification does not define which other errors will be included.

Any two of *readfds*, *writefds*, or *exceptfds* can be given as **NULL** if no descriptors are to be checked for the condition of interest. At least one must be non-**NULL**, and any non-**NULL** descriptor set must contain at least one socket descriptor.

Summary: A socket will be identified in a particular set when **WSPSelect** returns according to the following:

- readfds*: If **WSPListen** is called, a connection is pending, **WSPAccept** will succeed. Data is available for reading (includes OOB data if **SO\_OOBINLINE** is enabled). Connection has been closed/reset/terminated.
- writefds*: If **WSPConnect** (nonblocking), connection has succeeded. Data can be sent.
- Exceptfds*: If **WSPConnect** (nonblocking), connection attempt failed. OOB data is available for reading (only if **SO\_OOBINLINE** is disabled).

Three macros and one upcall function are defined in the header file **Ws2spi.h** for manipulating and checking the descriptor sets. The variable **FD\_SETSIZE** determines the maximum number of descriptors in a set. (The default value of **FD\_SETSIZE** is 64, which can be modified by #defining **FD\_SETSIZE** to another value before #including **Ws2spi.h**.) Internally, socket handles in a **FD\_SET** are not represented as bit flags as in Berkeley Unix. Their data representation is opaque. Use of these macros will maintain software portability between different socket environments.

The macros to manipulate and check **FD\_SET** contents are.

**FD\_CLR(*s*, \**set*)**

Removes the descriptor *s* from *set*.

**FD\_SET(*s*, \**set*)**

Adds descriptor *s* to *set*.

**FD\_ZERO(\**set*)**

Initializes the *set* to the NULL set.

The upcall function used to check the membership is:

```
int WPUFDIsSet ( SOCKET s, FD_SET FAR * set );
```

which will return nonzero if *s* is a member of the *set* or otherwise zero.

The parameter *timeout* controls how long the **WSPSelect** can take to complete. If *timeout* is a NULL pointer, **WSPSelect** will block indefinitely until at least one descriptor meets the specified criteria. Otherwise, *timeout* points to a **TIMEVAL** structure that specifies the maximum time that **WSPSelect** should wait before returning. When **WSPSelect** returns, the contents of the **TIMEVAL** structure are not altered. If **TIMEVAL** is initialized to {0, 0}, **WSPSelect** will return immediately; this is used to poll the state of the selected sockets. If this is the case, then the **WSPSelect** call is considered nonblocking and the standard assumptions for nonblocking calls apply. For example, the blocking hook will *not* be called, and the Windows Sockets provider will not yield.

---

**Note** **WSPSelect** has no effect on the persistence of socket events registered with **WSPAsyncSelect** or **WSPEventSelect**.

---

## Error Codes

Error code	Meaning
WSAEFAULT	Windows Sockets service provider was unable to allocated needed resources for its internal operations, or the <i>readfds</i> , <i>writfds</i> , <i>exceptfds</i> or <i>timeval</i> parameters are not part of the user address space.
WSAENETDOWN	Network subsystem has failed.
WSAEINVAL	The <i>timeout</i> value is not valid, or all three descriptor parameters were NULL.
WSAEINTR	(Blocking) call was canceled through <b>WSPCancelBlockingCall</b> .
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	One of the descriptor sets contains an entry that is not a socket.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

## + See Also

**WSPAccept, WSPConnect, WSPRecv, WSPRecvFrom, WSPSend, WSPSendTo, WSPEventSelect**

# WSPSend

The **WSPSend** function sends data on a connected socket.

```
int WSPSend (
    SOCKET                s,
    LPWSABUF             lpBuffers,
    DWORD                dwBufferCount,
    LPDWORD              lpNumberOfBytesSent,
    DWORD                dwFlags,
    LPWSAOVERLAPPED     lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    LPWSATHREADID       lpThreadId,
    LPINT                lpErrno
);
```

## Parameters

**s**

[in] Descriptor identifying a connected socket.

**lpBuffers**

[in] Pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer. This array must remain valid for the duration of the send operation.

**dwBufferCount**

[in] Number of **WSABUF** structures in the *lpBuffers* array.

**lpNumberOfBytesSent**

[out] Pointer to the number of bytes sent by this call.

**dwFlags**

[in] Specifies the way in which the call is made.

**lpOverlapped**

[in] Pointer to a **WSAOVERLAPPED** structure (ignored for nonoverlapped sockets.)

**lpCompletionRoutine**

[in] Pointer to the completion routine called when the send operation has been completed (ignored for nonoverlapped sockets.)

**lpThreadId**

[in] Pointer to a **WSATHREADID** structure to be used by the provider in a subsequent call to **WPUQueueApc**. The provider should store the referenced **WSATHREADID** structure (not the pointer to same) until after the **WPUQueueApc** function returns.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs and the send operation has completed immediately, **WSPSend** returns zero. Note that in this case the completion routine, if specified, will have already been queued. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code is available in *lpErrno*. The error code `WSA_IO_PENDING` indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that no overlapped operation was initiated and no completion indication will occur.

## Remarks

**WSPSend** is used to write outgoing data from one or more buffers on a connection-oriented socket specified by *s*. It can also be used, however, on connectionless sockets that have a stipulated default peer address established through the **WSPConnect** function.

For overlapped sockets (created using **WSPSocket** with flag `WSA_FLAG_OVERLAPPED`) this will occur using overlapped I/O, unless both *lpOverlapped* and *lpCompletionRoutine* are `NULL` in which case the socket is treated as a nonoverlapped socket. A completion indication will occur (invocation of the completion routine or setting of an event object) when the supplied buffer(s) have been consumed by the transport. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or **WSPGetOverlappedResult**.

For nonoverlapped sockets, the parameters *lpOverlapped*, *lpCompletionRoutine*, and *lpThreadId* are ignored and **WSPSend** adopts the regular synchronous semantics. Data is copied from the supplied buffer(s) into the transport's buffer. If the socket is nonblocking and stream oriented, and there is not sufficient space in the transport's buffer, **WSPSend** will return with only part of the supplied buffers having been consumed. Given the same buffer situation and a blocking socket, **WSPSend** will block until all of the supplied buffer contents have been consumed.

The array of **WSABUF** structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For message-oriented sockets, care must be taken not to exceed the maximum message size of the underlying provider, which can be obtained by getting the value of socket option `SO_MAX_MSG_SIZE`. If the data is too long to pass atomically through the underlying protocol the error `WSAEMSGSIZE` is returned, and no data is transmitted.

Note that the successful completion of a **WSPSend** does not indicate that the data was successfully delivered.

*dwFlags* can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *dwFlags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag;
MSG_OOB	Sends OOB data (stream-style socket such as <b>SOCK_STREAM</b> only).
MSG_PARTIAL	Specifies that <i>lpBuffers</i> only contains a partial message. Note that the error code WSAEOPNOTSUPP will be returned for messages that do not support partial message transmissions.

### Overlapped Socket I/O

If an overlapped operation completes immediately, **WSPSend** returns a value of zero and the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent. If the overlapped operation is successfully initiated and will complete later, **WSPSend** returns SOCKET\_ERROR and indicates error code WSA\_IO\_PENDING. In this case, *lpNumberOfBytesSent* is not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **WSPGetOverlappedResult**.

Providers must allow this function to be called from within the completion routine of a previous **WSPRecv**, **WSPRecvFrom**, **WSPSend** or **WSPSendTo** function. However, for a given socket, I/O completion routines cannot be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The **WSAOVERLAPPED** structure has the following form:

```
typedef struct _WSAOVERLAPPED {
    DWORD      Internal;           // reserved
    DWORD      InternalHigh;      // reserved
    DWORD      Offset;            // reserved
    DWORD      OffsetHigh;        // reserved
    WSAEVENT   hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

If the *lpCompletionRoutine* parameter is NULL, the service provider signals the *hEvent* member of *lpOverlapped* when the overlapped operation completes if it contains a valid event object handle. The Windows Sockets SPI client can use **WSPGetOverlappedResult** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* member is ignored and can be used by the Windows Sockets SPI client to pass context information to the completion routine. A client that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to TRUE. In this case the usage of the *hEvent* member is undefined, and attempting to wait on the *hEvent* member would produce unpredictable results.

A service provider arranges for a function to be executed in the proper thread and process context by calling **WPUQueueApc**. This function can be called from any process and thread context, even a context different from the thread and process that was used to initiate the overlapped operation.

A service provider arranges for a function to be executed in the proper thread by calling **WPUQueueApc**. Note that this function must be invoked while in the context of the same process (but not necessarily the same thread) that was used to initiate the overlapped operation. It is the service provider's responsibility to arrange for this process context to be active prior to calling **WPUQueueApc**.

**WPUQueueApc** takes as input parameters a pointer to a **WSATHREADID** structure (supplied to the provider through the *lpThreadId* input parameter), a pointer to an APC function to be invoked, and a 32-bit context value that is subsequently passed to the APC function. Because only a single 32-bit context value is available, the APC function itself cannot be the client specified—completion routine. The service provider must instead supply a pointer to its own APC function that uses the supplied context value to access the needed result information for the overlapped operation, and then invokes the client specified—completion routine.

The prototype for the client-supplied completion routine is as follows:

```
void CALLBACK CompletionRoutine (
    IN    DWORD           dwError,
    IN    DWORD           cbTransferred,
    IN    LPWSAOVERLAPPED lpOverlapped,
    IN    DWORD           dwFlags
);
```

**CompletionRoutine** is a placeholder for a client supplied function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes sent. No flag values are currently defined and the *dwFlags* value will be zero. This function does not return a value.



The completion routines can be called in any order, though not necessarily in the same order that the overlapped operations are completed. However, the service provider guarantees to the client that posted buffers are sent in the same order they are supplied.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEACCES	Requested address is a broadcast address, but the appropriate flag was not set.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>IpBuffers</i> argument is not totally contained in a valid part of the user address space.
WSAENETRESET	Connection has been broken due to the remote host resetting.
WSAENOBUFS	Connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOTCONN	Socket is not connected.
WSAENOTSOCK	Descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, MSG_PARTIAL is not supported, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	Socket has been shut down; it is not possible to <b>WSPSend</b> on a socket after <b>WSPShutdown</b> has been invoked with how set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: There are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the send operation cannot be completed immediately.
WSAEMSGSIZE	Socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	Socket has not been bound with <b>WSPBind</b> , or the socket is not created with the overlapped flag.
WSAECONNABORTED	Virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	Virtual circuit was reset by the remote side.
WSA_OPERATION_ABORTED	Overlapped operation has been canceled due to the closure of the socket, or the execution of the SIO_FLUSH command in <b>WSIoctl</b> .

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

**+** See Also

**WSPSocket**, **WSPGetOverlappedResult**, **WPUQueueApc**

## WSPSendDisconnect

The **WSPSendDisconnect** function initiates termination of the connection for the socket and sends disconnect data.

```
int WSPSendDisconnect (
    SOCKET      s,
    LPWSABUF   lpOutboundDisconnectData,
    LPINT       lpErrno
);
```

### Parameters

*s*

[in] A descriptor identifying a socket.

*lpOutboundDisconnectData*

[in] A pointer to the outgoing disconnect data.

*lpErrno*

[out] A pointer to the error code.

### Return Values

If no error occurs, **WSPSendDisconnect** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code is available in *lpErrno*.

### Remarks

**WSPSendDisconnect** is used on connection-oriented sockets to disable transmission, and to initiate termination of the connection along with the transmission of disconnect data, if any.

After this function has been successfully issued, subsequent sends are disallowed.

*lpOutboundDisconnectData*, if not `NULL`, points to a buffer containing the outgoing disconnect data to be sent to the remote party.

Note that **WSPSendDisconnect** does not close the socket, and that resources attached to the socket will not be freed until **WSPCloseSocket** is invoked.

**Note** **WSPSendDisconnect** does not block regardless of the **SO\_LINGER** setting on the socket.

A Windows Sockets SPI client should not rely on being able to reuse a socket after it has been disconnected. In particular, a Windows Sockets provider is not required to support the use of **WSPConnect** on such a socket.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAENOPROTOPT	Parameter <i>IpOutboundDisconnectData</i> is not NULL, and the disconnect data is not supported by the service provider.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	Socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	Descriptor is not a socket.
WSAEFAULT	The <i>IpOutboundDisconnectData</i> argument is not totally contained in a valid part of the user address space.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

### + See Also

**WSPConnect**, **WSPSocket**

## WSPSendTo

The **WSPSendTo** function sends data to a specific destination using overlapped I/O.

```
int WSPSendTo (
    SOCKET                s,
    LPWSABUF             lpBuffers,
    DWORD                dwBufferCount,
    LPDWORD              lpNumberOfBytesSent,
    DWORD                dwFlags,
    const struct sockaddr FAR *lpTo,
    int                  iToLen,
    LPWSAOVERLAPPED     lpOverlapped,
```

```

LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
LPWSATHREADID                      lpThreadId,
LPINT                               lpErrno
);

```

## Parameters

*s*

[in] A descriptor identifying a socket.

*lpBuffers*

[in] A pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer. This array must remain valid for the duration of the send operation.

*dwBufferCount*

[in] The number of **WSABUF** structures in the *lpBuffers* array.

*lpNumberOfBytesSent*

[out] A pointer to the number of bytes sent by this call.

*dwFlags*

[in] Specifies the way in which the call is made.

*lpTo*

[in] An optional pointer to the address of the target socket.

*iToLen*

[in] The size of the address in *lpTo*.

*lpOverlapped*

[in] A pointer to a **WSAOVERLAPPED** structure (ignored for nonoverlapped sockets).

*lpCompletionRoutine*

[in] A pointer to the completion routine called when the send operation has been completed (ignored for nonoverlapped sockets).

*lpThreadId*

[in] A pointer to a **WSATHREADID** structure to be used by the provider in a subsequent call to **WPUQueueApc**. The provider should store the referenced **WSATHREADID** structure (not the pointer to same) until after the **WPUQueueApc** function returns.

*lpErrno*

[out] A pointer to the error code.

## Return Values

If no error occurs and the receive operation has completed immediately, **WSPSendTo** returns zero. Note that in this case the completion routine, if specified, will have already been queued. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code is available in *lpErrno*. The error code **WSA\_IO\_PENDING** indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that no overlapped operation was initiated and no completion indication will occur.

## Remarks

**WSPSendTo** is normally used on a connectionless socket specified by *s* to send a datagram contained in one or more buffers to a specific peer socket identified by the *IpTo* parameter. Even if the connectionless socket has been previously connected to a specific address with the **connect** function, *IpTo* overrides the destination address for that particular datagram only. On a connection-oriented socket, the *IpTo* and *iToLen* parameters are ignored; in this case the **WSPSendTo** function is equivalent to **WSPSend**.

For overlapped sockets (created using **WSPSocket** with flag `WSA_FLAG_OVERLAPPED`) this will occur using overlapped I/O, unless both *IpOverlapped* and *IpCompletionRoutine* are NULL in which case the socket is treated as a nonoverlapped socket. A completion indication will occur (invocation of the completion routine or setting of an event object) when the supplied buffer(s) have been consumed by the transport. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or **WSPGetOverlappedResult**.

For nonoverlapped sockets, the parameters *IpOverlapped*, *IpCompletionRoutine*, and *IpThreadId* are ignored and **WSPSendTo** adopts the regular synchronous semantics. Data is copied from the supplied buffer(s) into the transport's buffer. If the socket is nonblocking and stream oriented, and there is not sufficient space in the transport's buffer, **WSPSendTo** will return with only part of the Windows Sockets SPI client's buffers having been consumed. Given the same buffer situation and a blocking socket, **WSPSendTo** will block until all of the Windows Sockets SPI client's buffer contents have been consumed.

The array of **WSABUF** structures pointed to by the *IpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For message-oriented sockets, care must be taken not to exceed the maximum message size of the underlying transport, which can be obtained by getting the value of socket option `SO_MAX_MSG_SIZE`. If the data is too long to pass atomically through the underlying protocol the error `WSAEMSGSIZE` is returned, and no data is transmitted.

Note that the successful completion of a **WSPSendTo** does not indicate that the data was successfully delivered.

*iFlags* can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *dwFlags* parameter. The latter is constructed by using the bitwise OR operator with any of the values on the following page.

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Sends OOB data (stream-style socket such as <b>SOCK_STREAM</b> only).
MSG_PARTIAL	Specifies that <i>lpBuffers</i> only contains a partial message. Note that the error code WSAEOPNOTSUPP will be returned by transports that do not support partial message transmissions.

### Overlapped Socket I/O

If an overlapped operation completes immediately, **WSPSendTo** returns a value of zero and the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent. If the overlapped operation is successfully initiated and will complete later, **WSPSendTo** returns SOCKET\_ERROR and indicates error code WSA\_IO\_PENDING. In this case, *lpNumberOfBytesSent* is not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **WSPGetOverlappedResult**.

Providers must allow this function to be called from within the completion routine of a previous **WSPRecv**, **WSPRecvFrom**, **WSPSend** or **WSPSendTo** function. However, for a given socket, I/O completion routines cannot be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The **WSAOVERLAPPED** structure has the following form:

```
typedef struct _WSAOVERLAPPED {
    DWORD      Internal;           // reserved
    DWORD      InternalHigh;      // reserved
    DWORD      Offset;            // reserved
    DWORD      OffsetHigh;        // reserved
    WSAEVENT   hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

If the *lpCompletionRoutine* parameter is NULL, the service provider signals the *hEvent* member of *lpOverlapped* when the overlapped operation completes if it contains a valid event object handle. Windows Sockets SPI clients can use **WSPGetOverlappedResult** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* member is ignored and can be used by the Windows Sockets SPI client to pass context information to the completion routine. A client that passes a non-NULL *lpCompletionRoutine* and later calls

**WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to TRUE. In this case the usage of the *hEvent* member is undefined, and attempting to wait on the *hEvent* member would produce unpredictable results.

It is the service provider's responsibility to arrange for invocation of the client specified—completion routine when the overlapped operation completes. Since the completion routine must be executed in the context of the same thread that initiated the overlapped operation, it cannot be invoked directly from the service provider. The *Ws2\_32.dll* offers an asynchronous procedure call (APC) mechanism to facilitate invocation of completion routines.

A service provider arranges for a function to be executed in the proper thread by calling **WPUQueueApc**. This function can be called from any process and thread context, even a context different from the thread and process that was used to initiate the overlapped operation.

**WPUQueueApc** takes as input parameters a pointer to a **WSATHREADID** structure (supplied to the provider through the *lpThreadId* input parameter), a pointer to an APC function to be invoked, and a 32-bit context value that is subsequently passed to the APC function. Because only a single 32-bit context value is available, the APC function itself cannot be the client specified—completion routine. The service provider must instead supply a pointer to its own APC function, which uses the supplied context value to access the needed result information for the overlapped operation, and then invokes the client specified—completion routine.

The prototype for the client-supplied completion routine is as follows:

```
void CALLBACK CompletionRoutine (
    IN    DWORD           dwError,
    IN    DWORD           cbTransferred,
    IN    LPWSAOVERLAPPED lpOverlapped,
    IN    DWORD           dwFlags
);
```

**CompletionRoutine** is a placeholder for a client-supplied function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes sent. No flag values are currently defined and the *dwFlags* value will be zero. This function does not return a value.

The completion routines can be called in any order, though not necessarily in the same order that the overlapped operations are completed. However, the service provider guarantees to the client that posted buffers are sent in the same order they are supplied.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEACCES	Requested address is a broadcast address, but the appropriate flag was not set.
WSAEINTR	(Blocking) call was canceled through <b>WSPCancelBlockingCall</b> .
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>IpBuffers</i> or <i>IpTo</i> parameters are not part of the user address space, or the <i>IpTo</i> argument is too small.
WSAENETRESET	Connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOBUFS	Windows Sockets provider reports a buffer deadlock.
WSAENOTCONN	Socket is not connected (connection-oriented sockets only.)
WSAENOTSOCK	Descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type <b>SOCK_STREAM</b> , OOB data is not supported in the communication domain associated with this socket, MSG_PARTIAL is not supported, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	Socket has been shut down; it is not possible to use <b>WSPSendTo</b> on a socket after <b>WSPShutdown</b> has been invoked with <i>how</i> set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: there are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the send operation cannot be completed immediately.
WSAEMSGSIZE	Socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	Socket has not been bound with <b>WSPBind</b> , or the socket is not created with the overlapped flag.
WSAECONNABORTED	Virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	Virtual circuit was reset by the remote side.
WSAEADDRNOTAVAIL	Remote address is not a valid address (for example, ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDRREQ	Destination address is required.
WSAENETUNREACH	Network cannot be reached from this host at this time.
WSA_OPERATION_ ABORTED	Overlapped operation has been canceled due to the closure of the socket, or the execution of the SIO_FLUSH command in <b>WSPIoctl</b> .



**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

**+** See Also

**WSPSocket, WSPGetOverlappedResult, WPUQueueApc**

---

## WSPSetSockOpt

The **WSPSetSockOpt** function sets a socket option.

```
int WSPSetSockOpt (  
    SOCKET          s,  
    int             level,  
    int             optname,  
    const char FAR *optval,  
    int             optlen,  
    LPINT           lpErrno  
);
```

### Parameters

*s*

[in] Descriptor identifying a socket.

*level*

[in] Level at which the option is defined; the supported *levels* include SOL\_SOCKET. (See the *Windows Sockets Protocol-Specific Annex* for more protocol-specific *levels*.)

*optname*

[in] Socket option for which the value is to be set.

*optval*

[in] Pointer to the buffer in which the value for the requested option is supplied.

*optlen*

[in] Size of the *optval* buffer.

*lpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, **WSPSetSockOpt** returns zero. Otherwise, a value of SOCKET\_ERROR is returned, and a specific error code is available in *lpErrno*.

## Remarks

**WSPSetSockOpt** sets the current value for a socket option associated with a socket of any type, in any state. Although options can exist at multiple protocol levels, they are always present at the uppermost socket level. Options affect socket operations, such as whether broadcast messages can be sent on the socket.

There are two types of socket options: Boolean options that enable or disable a feature or behavior, and options that require an integer value or structure. To enable a Boolean option, *optval* points to a nonzero integer. To disable the option, *optval* points to an integer equal to zero. The *optlen* parameter should be equal to `sizeof (int)` for Boolean options. For other options, *optval* points to the an integer or structure that contains the desired value for the option, and *optlen* is the length of the integer or structure.

*level* = SOL\_SOCKET

Value	Type	Meaning
SO_BROADCAST	BOOL	Allows transmission of broadcast messages on the socket.
SO_DEBUG	BOOL	Records debugging information.
SO_DONTLINGER	BOOL	Reserved.
SO_DONTROUTE	BOOL	Does not route: sends directly to interface. Not supported on ATM sockets (results in an error).
SO_GROUP_PRIORITY	int	Reserved.
SO_KEEPAIVE	BOOL	Sends keep-alives. Not supported on ATM sockets (results in an error).
SO_LINGER	struct linger	Lingers on close if unsend data is present.
SO_OOINLINE	BOOL	Receives OOB data in the normal data stream.
SO_RCVBUF	int	Specifies the total per-socket buffer space reserved for receives. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.
SO_REUSEADDR	BOOL	Allows the socket to be bound to an address that is already in use. (See <i>bind</i> .) Not applicable on ATM sockets.
SO_SNDBUF	int	Specifies the total per-socket buffer space reserved for sends. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.
PVD_CONFIG	Service Provider Dependent	This object stores the configuration information for the service provider associated with socket <i>s</i> . The exact format of this data structure is service provider specific.

Calling **WSPGetSockOpt** with an unsupported option will result in an error code of WSAENOPROTOOPT being returned in *lpErrno*.

**SO\_DEBUG**

Windows Sockets service providers are encouraged (but not required) to supply output debug information if the **SO\_DEBUG** option is set by a Windows Sockets SPI client. The mechanism for generating the debug information and the form it takes are beyond the scope of this specification.

**SO\_GROUP\_PRIORITY**

Reserved.

**SO\_KEEPAIVE**

A Windows Sockets SPI client can request that a TCP/IP provider enable the use of keep-alive packets on TCP-connections by turning on the **SO\_KEEPAIVE** socket option. A Windows Sockets provider need not support the use of keep-alives: if it does, the precise semantics are implementation specific but should conform to section 4.2.3.6 of RFC 1122: *Requirements for Internet Hosts—Communication Layers*. If a connection is dropped as the result of keep-alive the error code WSAENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with WSAENOTCONN.

**SO\_LINGER**

**SO\_LINGER** controls the action taken when unsend data is queued on a socket and a **WSPCloseSocket** is performed. See **WSPCloseSocket** for a description of the way in which the **SO\_LINGER** settings affect the semantics of **WSPCloseSocket**. The Windows Sockets SPI client sets the desired behavior by creating a *struct linger* (pointed to by the *optval* argument) with the following elements:

```
struct linger {
    u_short  l_onoff;
    u_short  l_linger;
}
```

To enable **SO\_LINGER**, a Windows Sockets SPI client should set *l\_onoff* to a nonzero value, set *l\_linger* to zero or the desired time-out (in seconds), and call **WSPSetSockOpt**. To enable **SO\_DONTLINGER** (that is, disable **SO\_LINGER**) *l\_onoff* should be set to zero and **WSPSetSockOpt** should be called. Note that enabling **SO\_LINGER** with a nonzero time-out on a nonblocking socket is not recommended (see section 4.1.7. *WSPCloseSocket* for details.)

Enabling **SO\_LINGER** also disables **SO\_DONTLINGER**, and vice versa. Note that if **SO\_DONTLINGER** is *disabled* (that is, **SO\_LINGER** is *enabled*) then no time-out value is specified. In this case, the time-out used is implementation dependent. If a previous time-out has been established for a socket (by enabling **SO\_LINGER**), then this time-out value should be reinstated by the service provider.

**SO\_REUSEADDR**

By default, a socket cannot be bound (see **WSPBind**) to a local address that is already in use. On occasion, however, it may be desirable to reuse an address in this way. Since every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the Windows Sockets provider that a **WSPBind** on a socket should be allowed to bind to a local

address that is already in use by another socket, the Windows Sockets SPI client should set the **SO\_REUSEADDR** socket option for the socket before issuing the **WSPBind**. Note that the option is interpreted only at the time of the **WSPBind**: it is therefore unnecessary (but harmless) to set the option on a socket that is not to be bound to an existing address, and setting or resetting the option after the **WSPBind** has no effect on this or any other socket.

### SO\_SNDBUF

When a Windows Sockets implementation supports the **SO\_RCVBUF** and **SO\_SNDBUF** options, a Windows Sockets SPI client can request different buffer sizes (larger or smaller). The call can succeed even though the service provider did not make available the entire amount requested. A Windows Sockets SPI client must call **WSPGetSockOpt** with the same option to check the buffer size actually provided.

### PVD\_CONFIG

This object stores the configuration information for the service provider associated with socket *s*. The exact format of this data structure is service provider specific.

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEFAULT	The <i>optval</i> is not in a valid part of the process address space or <i>optlen</i> argument is too small.
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The <i>level</i> is not valid, or the information in <i>optval</i> is not valid.
WSAENETRESET	Connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOPROTOOPT	Option is unknown or unsupported for the specified provider.
WSAENOTCONN	Connection has been reset when <b>SO_KEEPALIVE</b> is set.
WSAENOTSOCK	Descriptor is not a socket.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

### + See Also

**WSPBind**, **WSPGetSockOpt**, **WSIoctl**, **WSPSocket**, **WSPEventSelect**

# WSPShutdown

The **WSPShutdown** function disables sends and/or receives on a socket.

```
int WSPShutdown (  
    SOCKET s,  
    int how,  
    LPINT lpErrno  
);
```

## Parameters

*s*

[in] Descriptor identifying a socket.

*how*

[in] Flag that describes what types of operation will no longer be allowed.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WSPShutdown** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code is available in *lpErrno*.

## Remarks

**WSPShutdown** is used on all types of sockets to disable reception, transmission, or both.

If *how* is `SD_RECEIVE`, subsequent receives on the socket will be disallowed. This has no effect on the lower protocol layers. For TCP sockets, if there is still data queued on the socket waiting to be received, or data arrives subsequently, the connection is reset, since the data cannot be delivered to the user. For UDP sockets, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated.

If *how* is `SD_SEND`, subsequent sends on the socket are disallowed. For TCP sockets, a FIN will be sent. Setting *how* to `SD_BOTH` disables both sends and receives as described above.

Note that **WSPShutdown** does not close the socket, and resources attached to the socket will *not* be freed until **WSPCloseSocket** is invoked.

---

**Note** **WSPShutdown** does not block regardless of the **SO\_LINGER** setting on the socket. A Windows Sockets SPI client should not rely on being able to reuse a socket after it has been shut down. In particular, a Windows Sockets service provider is not required to support the use of **WSPConnect** on such a socket.

---

## Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEINVAL	The <i>how</i> is not valid, or is not consistent with the socket type. For example, SD_SEND is used with a UNI_RECV socket type.
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAENOTCONN	Socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	Descriptor is not a socket.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Ws2spi.h.

### + See Also

**WSPConnect**, **WSPSocket**

## WSPSocket

The **WSPSocket** function creates a socket.

```
SOCKET WSPSocket (
    int                af,
    int                type,
    int                protocol,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    GROUP              g,
    DWORD              dwFlags,
    LPINT              lpErrno
);
```

### Parameters

*af*

[in] Address family specification.

*type*

[in] Type specification for the new socket.

*protocol*

[in] Protocol to be used with the socket that is specific to the indicated address family.

*IpProtocolInfo*

[in] Pointer to a **WSAPROTOCOL\_INFOW** structure that defines the characteristics of the socket to be created.

*g*

[in] Reserved.

*dwFlags*

Socket attribute specification.

*IpErrno*

[out] Pointer to the error code.

## Return Values

If no error occurs, **WSPSocket** returns a descriptor referencing the new socket. Otherwise, a value of **INVALID\_SOCKET** is returned, and a specific error code is available in *IpErrno*.

## Remarks

**WSPSocket** causes a socket descriptor and any related resources to be allocated. By default, the created socket will *not* have the overlapped attribute. Windows Sockets providers are encouraged to be realized as Windows installable file systems, and supply system file handles as socket descriptors. These providers must call **WPUModifyIFSHandle** prior to returning from this function. For nonfile-system Windows Sockets providers, **WPUCreateSocketHandle** must be used to acquire a unique socket descriptor from the *Ws2\_32.dll* prior to returning from this function. See section *Descriptor Allocation* for more information.

The values for *af*, *type*, and *protocol* are those supplied by the application in the corresponding API functions **socket** or **WSASocket**. A service provider is free to ignore or pay attention to any or all of these values as is appropriate for the particular protocol. However, the provider must be willing to accept the value of zero for *af* and *type*, since the *Ws2\_32.dll* considers these to be wildcard values. Also the value of manifest constant **FROM\_PROTOCOL\_INFO** must be accepted for any of *af*, *type* and *protocol*. This value indicates that the Windows Sockets 2 application needs to use the corresponding values from the indicated **WSAPROTOCOL\_INFOW** structure: (*iAddressFamily*, *iSocketType*, *iProtocol*).

The *dwFlags* parameter can be used to specify the attributes of the socket by using the bitwise OR operator with any of the flags on the following page.

Flag	Meaning
WSA_FLAG_OVERLAPPED	This flag causes an overlapped socket to be created. Overlapped sockets can utilize <b>WSPSend</b> , <b>WSPSendTo</b> , <b>WSPRecv</b> , <b>WSPRecvFrom</b> and <b>WSPIoctl</b> for overlapped I/O operations, which allow multiple operations to be initiated and in process simultaneously. All functions that allow overlapped operations also support nonoverlapped usage on an overlapped socket if the values for parameters related to overlapped operation are NULL.
WSA_FLAG_MULTIPOINT_C_ROOT	Indicates that the socket created will be a c_root in a multipoint session. Only allowed if a rooted control plane is indicated in the protocol's <b>WSAPROTOCOL_INFOW</b> structure.
WSA_FLAG_MULTIPOINT_C_LEAF	Indicates that the socket created will be a c_leaf in a multicast session. Only allowed if XP1_SUPPORT_MULTIPOINT is indicated in the protocol's <b>WSAPROTOCOL_INFOW</b> structure.
WSA_FLAG_MULTIPOINT_D_ROOT	Indicates that the socket created will be a d_root in a multipoint session. Only allowed if a rooted data plane is indicated in the protocol's <b>WSAPROTOCOL_INFOW</b> structure.
WSA_FLAG_MULTIPOINT_D_LEAF	Indicates that the socket created will be a d_leaf in a multipoint session. Only allowed if XP1_SUPPORT_MULTIPOINT is indicated in the protocol's <b>WSAPROTOCOL_INFOW</b> structure.

---

**Important** For multipoint sockets, exactly one of **WSA\_FLAG\_MULTIPOINT\_C\_ROOT** or **WSA\_FLAG\_MULTIPOINT\_C\_LEAF** must be specified, and exactly one of **WSA\_FLAG\_MULTIPOINT\_D\_ROOT** or **WSA\_FLAG\_MULTIPOINT\_D\_LEAF** must be specified. Refer to *Protocol-Independent Multicast and Multipoint in the SPI* for additional information.

---

Connection-oriented sockets such as **SOCK\_STREAM** provide full-duplex connections, and must be in a connected state before any data can be sent or received on them. A connection to another socket is created with a **WSPConnect** call. Once connected, data can be transferred using **WSPSend** and **WSPRecv** calls. When a session has been completed, a **WSPCloseSocket** must be performed.

The communications protocols used to implement a reliable, connection-oriented socket ensure that data is not lost or duplicated. If data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and subsequent calls will fail with the error code set to **WSAETIMEDOUT**.

Connectionless, message-oriented sockets allow sending and receiving of datagrams to and from arbitrary peers using **WSPSendTo** and **WSPRecvFrom**. If such a socket is connected by using **WSPConnect** to a specific peer, datagrams can be sent to that peer using **WSPSend** and can be received from (only) this peer using **WSPRecv**.



Support for sockets with type **SOCK RAW** is not required but service providers are encouraged to support raw sockets whenever it makes sense to do so.

### Shared Sockets

When a special **WSAPROTOCOL\_INFOW** structure (obtained through the **WSPDuplicateSocket** function and used to create additional descriptors for a shared socket) is passed as an input parameter to **WSPSocket**, the *g* and *dwFlags* parameters are **ignored**.

### Layered Service Provider Considerations

A layered service provider supplies an implementation of this function, but it is also a client of this function if and when it calls **WSPSocket** of the next layer in the protocol chain. Some special considerations apply to this function's *lpProtocolInfo* parameter as it is propagated down through the layers of the protocol chain.

If the next layer in the protocol chain is another layer then when the next layer's **WSPSocket** is called, this layer must pass to the next layer a *lpProtocolInfo* that references the same unmodified **WSAPROTOCOL\_INFOW** structure with the same unmodified chain information. However, if the next layer is the base protocol (that is, the last element in the chain), this layer performs a substitution when calling the base provider's **WSPSocket**. In this case, the base provider's **WSAPROTOCOL\_INFOW** structure should be referenced by the *lpProtocolInfo* parameter.

One vital benefit of this policy is that base service providers do not have to be aware of protocol chains.

This same propagation policy applies when propagating a **WSAPROTOCOL\_INFOW** structure through a layered sequence of other functions such as **WSPAddressToString**, **WSPDuplicateSocket**, **WSPStartup**, or **WSPStringToAddress**.

### Error Codes

Error code	Meaning
WSAENETDOWN	Network subsystem has failed.
WSAEAFNOSUPPORT	Specified address family is not supported.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	Specified protocol is not supported.
WSAEPROTOTYPE	Specified protocol is the wrong type for this socket.
WSAESOCKTNOSUPPORT	Specified socket type is not supported in this address family.
WSAEINVAL	Parameter <i>g</i> specified is not valid.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in `Ws2spi.h`.

**+** See Also

**WSPAccept**, **WSPBind**, **WSPConnect**, **WSPGetSockName**, **WSPGetSockOpt**, **WSPSetSockOpt**, **WSPListen**, **WSPRecv**, **WSPRecvFrom**, **WSPSend**, **WSPSendTo**, **WSPShutdown**, **WSPIoctl**, **WPUCreateSocketHandle**

## WSPStartup

The **WSPStartup** function initiates use of a Windows Sockets service provider by a client.

```
int WSPStartup (
    WORD                wVersionRequested,
    LPWSPDATA           lpWSPData,
    LPWSAPROTOCOL_INFOW lpProtocolInfo,
    WSPUPCALLTABLE     UpcallTable,
    LPWSPPROC_TABLE     lpProcTable
);
```

### Parameters

#### *wVersionRequested*

[in] Highest version of Windows Sockets SPI support that the caller can use. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

#### *lpWSPData*

[out] Pointer to the **WSPDATA** data structure that is to receive details of the Windows Sockets service provider.

#### *lpProtocolInfo*

[in] Pointer to a **WSAPROTOCOL\_INFOW** structure that defines the characteristics of the desired protocol. This is especially useful when a single provider DLL is capable of instantiating multiple different service providers.

#### *UpcallTable*

[in] `Ws2_32.dll`'s upcall dispatch table.

#### *lpProcTable*

[out] Pointer to the table of SPI function pointers.

### Return Values

**WSPStartup** returns zero if successful. Otherwise, it returns one of the error codes listed below:

## Remarks

This function *must* be the first Windows Sockets SPI function called by a Windows Sockets SPI client on a per-process basis. It allows the client to specify the version of Windows Sockets SPI required and to provide its upcall dispatch table. All upcalls (that is, functions prefixed with **WPU**) made by the Windows Sockets service provider are invoked through the client's upcall dispatch table. This function also allows the client to retrieve details of the specific Windows Sockets service provider implementation. The Windows Sockets SPI client can only issue further Windows Sockets SPI functions after a successful **WSPStartup** invocation. A table of pointers to the rest of the SPI functions is retrieved through the *lpProcTable* parameter.

In order to support future versions of the Windows Sockets SPI and the *Ws2\_32.dll*, which may have functionality differences from the current Windows Sockets SPI, a negotiation takes place in **WSPStartup**. The caller of **WSPStartup** (either the *Ws2\_32.dll* or a layered protocol) and the Windows Sockets service provider indicate to each other the highest version that they can support, and each confirms that the other's highest version is acceptable. Upon entry to **WSPStartup**, the Windows Sockets service provider examines the version requested by the client. If this version is equal to or higher than the lowest version supported by the service provider, the call succeeds and the service provider returns in *wHighVersion* the highest version it supports and in *wVersion* the minimum of its high version and *wVersionRequested*. The Windows Sockets service provider then assumes that the Windows Sockets SPI client will use *wVersion*. If the *wVersion* member of the **WSPDATA** structure is unacceptable to the caller, it should call **WSPCleanup** and either search for another Windows Sockets service provider or fail to initialize.

This negotiation allows both a Windows Sockets service provider and a Windows Sockets SPI client to support a range of Windows Sockets versions. A client can successfully utilize a Windows Sockets service provider if there is any overlap in the version ranges.

The following chart gives examples of how **WSPStartup** works in conjunction with different *Ws2\_32.dll* and Windows Sockets service provider (SP) versions.

DLL versions	SP versions	<i>wVersion requested</i>	<i>wVersion</i>	<i>wHigh version</i>	End result
1.1	1.1	1.1	1.1	1.1	use 1.1
1.0 1.1	1.0	1.1	1.0	1.0	use 1.0
1.0	1.0 1.1	1.0	1.0	1.1	use 1.0
1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1	1.0	1.1	1.0	1.0	DLL fails
1.0	1.1	1.0	---	---	WSAVERNOTSUPPORTED
1.0 1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1 2.0	1.1	2.0	1.1	1.1	use 1.1
2.0	2.0	2.0	2.0	2.0	use 2.0

The following code fragment demonstrates how a Windows Sockets SPI client, which supports only version 2 of Windows Sockets SPI, makes a **WSPStartup** call:

```
WORD wVersionRequested;
WSPDATA WSPData;

int err;

WSPUPCALLTABLE upcallTable =
{
    /* initialize upcallTable with function pointers */
};

LPWSPPROC_TABLE lpProcTable =
{
    /* allocate memory for the ProcTable */
};

wVersionRequested = MAKEWORD( 2, 2 );

err = WSPStartup( wVersionRequested, &WSPData, lpProtocolBuffer, upcallTable,
lpProcTable );
if ( err != 0 ) {
    /* Tell the user that we could not find a useable */
    /* Windows Sockets service provider. */
    return;
}

/* Confirm that the Windows Sockets service provider supports 2.2.*/
/* Note that if the service provider supports versions */
/* greater than 2.2 in addition to 2.2, it will still */
/* return 2.2 in wVersion since that is the version we */
/* requested. */

if ( LOBYTE( WSPData.wVersion ) != 2 ||
    HIBYTE( WSPData.wVersion ) != 2 ) {
    /* Tell the user that we could not find a useable */
    /* Windows Sockets service provider. */
    WSPCleanup( );
    return;
}

/* The Windows Sockets service provider is acceptable. Proceed. */
```

And this code fragment demonstrates how a Windows Sockets service provider that supports only version 2 performs the **WSPStartup** negotiation.

```

/* Make sure that the version requested is >= 2.2. */
/* The low byte is the major version and the high */
/* byte is the minor version. */

if ( LOBYTE( wVersionRequested ) < 2 ) ||
    ((LOBYTE( wVersionRequested ) == 2) &&
    (HIBYTE( wVersionRequested ) < 2))) {
    return WSAVERNOTSUPPORTED;
}

/* Since we only support 2.2, set both wVersion and */
/* wHighVersion to 2.2. */

lpWSPData->wVersion = MAKEWORD( 2, 2 );
lpWSPData->wHighVersion = MAKEWORD( 2, 2 );

```

Once the Windows Sockets SPI client has made a successful **WSPStartup** call, it can proceed to make other Windows Sockets SPI calls as needed. When it has finished using the services of the Windows Sockets service provider, the client must call **WSPCleanup** in order to allow the Windows Sockets service provider to free any resources allocated for the client.

Details of how Windows Sockets service provider information is encoded in the **WSPDATA** structure is as follows:

```

typedef struct WSPData {
    WORD        wVersion;
    WORD        wHighVersion;
    WCHAR       szDescription[WSPDESCRIPTION_LEN+1];
} WSPDATA, FAR * LPWSPDATA;

```

The members of this structure are shown in the following table.

Member	Usage
<b>wVersion</b>	Version of the Windows Sockets SPI specification that the Windows Sockets service provider expects the caller to use.
<b>wHighVersion</b>	Highest version of the Windows Sockets SPI specification that this service provider can support (also encoded as above). Normally this will be the same as <i>wVersion</i> .
<b>szDescription</b>	Null-terminated Unicode string into which the Windows Sockets provider copies a description of itself. The text (up to 256 characters in length) can contain any characters except control and formatting characters: the most likely use to which an SPI client will put this is to display it (possibly truncated) in a status message.

A Windows Sockets SPI client can call **WSPStartup** more than once if it needs to obtain the **WSPData** structure information more than once. On each such call the client can specify any version number supported by the provider.

There must be one **WSPCleanup** call corresponding to every successful **WSPStartup** call to allow third-party DLLs to make use of a Windows Sockets provider. This means, for example, that if **WSPStartup** is called three times, the corresponding call to **WSPCleanup** must occur three times. The first two calls to **WSPCleanup** do nothing except decrement an internal counter; the final **WSPCleanup** call does all necessary resource deallocation.

This function (and most other service provider functions) can be invoked in a thread that started out as a 16-bit process if the client is a 16-bit Windows Sockets 1.1 client. One important limitation of 16-bit processes is that a 16-bit process cannot create threads. This is significant to service provider implementers that plan to use an internal service thread as part of the implementation.

Fortunately, there are usually only two areas where the conditions for a service thread are strong:

- In the implementation of overlapped I/O completion.
- In the implementation of **WSPEventSelect**.

Both of these areas are only accessible through new Windows Sockets 2 functions, which can only be invoked by 32-bit processes.

A service thread can be safely used if these two design rules are carefully followed:

- Use a service thread only for functionality that is unavailable to 16-bit Windows Sockets 1.1 clients.
- Create the service thread only on demand.

Several other cautions apply to the use of internal service threads. First, threads generally carry some performance penalty. Use as few as possible, and avoid thread transitions wherever possible. Second, your code should always check for errors in creating threads and fail gracefully and informatively (for example, with **WSAEOPNOTSUPP**) in case some execution event you did not expect results in a 16-bit process executing a code path that needs threads.

### Layered Service Provider Considerations

A layered service provider supplies an implementation of this function, but it is also a client of this function when it calls **WSPStartup** to initialize the next layer in the protocol chain. The call to the next layer's **WSPStartup** may happen during the execution of this layer's **WSPStartup** or it may be delayed and called on demand, such as when **WSPSocket** is called. In any case, some special considerations apply to this function's *IpProtocolInfo* parameter as it is propagated down through the layers of the protocol chain.

The layered provider searches the *ProtocolChain* of the structure referenced by *lpProtocolInfo* to determine its own location in the chain (by searching for the layer's own catalog entry *Id*) and the identity of the next element in the chain. If the next element is another layer, then, when the next layer's **WSPStartup** is called, this layer must pass to the next layer a *lpProtocolInfo* that references the same unmodified **WSAPROTOCOL\_INFOW** structure with the same unmodified chain information. However, if the next layer is the base protocol (that is, the last element in the chain), this layer performs a substitution when calling the base provider's **WSPStartup**. In this case, the base provider's **WSAPROTOCOL\_INFOW** structure should be referenced by the *lpProtocolInfo* parameter.

One vital benefit of this policy is that base service providers do not have to be aware of protocol chains.

This same propagation policy applies when propagating a **WSAPROTOCOL\_INFOW** structure through a layered sequence of other functions such as **WSPAddressToString**, **WSPDuplicateSocket**, **WSPSocket**, or **WSPStringToAddress**.

## Error Codes

Error code	Meaning
WSASYSNOTREADY	Indicates that the underlying network subsystem is not ready for network communication.
WSAVERNOTSUPPORTED	Version of Windows Sockets SPI support requested is not provided by this particular Windows Sockets service provider.
WSAEINPROGRESS	Blocking Windows Sockets operation is in progress.
WSAEPROCLIM	Limit on the number of clients supported by the Windows Sockets implementation has been reached.
WSAEFAULT	The <i>lpWSPData</i> or <i>lpProcTable</i> parameter is invalid.

### ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

### + See Also

**WSPSend**, **WSPSendTo**, **WSPCleanup**

## WSPStringToAddress

The **WSPStringToAddress** function converts a human-readable numeric string to a socket address structure (**SOCKADDR**) suitable to passing to Windows Sockets routines that take such a structure.



Any missing components of the address will be defaulted to a reasonable value if possible. For example, a missing port number will default to zero.

```
int WSPStringToAddress (
    LPWSTR           AddressString,
    INT              AddressFamily,
    LPWSAPROTOCOL_INFOW lpProtocolInfo,
    LPSOCKADDR       lpAddress,
    LPINT            lpAddressLength,
    LPINT            lpErrno
);
```

### Parameters

#### *AddressString*

[in] Points to the zero-terminated human-readable string to convert.

#### *AddressFamily*

[in] Address family to which the string belongs, or AF\_UNSPEC if it is unknown.

#### *lpProtocolInfo*

[in] (required) Provider's **WSAPROTOCOL\_INFOW** structure.

#### *lpAddress*

[out] Buffer that is filled with a single **SOCKADDR** structure.

#### *lpAddressLength*

[in/out] Length of the Address buffer. Returns the size of the resultant **SOCKADDR** structure. If the supplied buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size in bytes.

#### *lpErrno*

[out] Pointer to the error code.

### Return Values

If no error occurs, **WSPStringToAddress** returns zero. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code is available in *lpErrno*.

### Layered Service Provider Considerations

A layered service provider supplies an implementation of this function, but it is also a client of this function if and when it calls **WSPStringToAddress** of the next layer in the protocol chain. Some special considerations apply to this function's *lpProtocolInfo* parameter as it is propagated down through the layers of the protocol chain.



If the next layer in the protocol chain is another layer then when the next layer's **WSPStringToAddress** is called, this layer must pass to the next layer a *IpProtocolInfo* that references the same unmodified **WSAPROTOCOL\_INFOW** structure with the same unmodified chain information. However, if the next layer is the base protocol (that is, the last element in the chain), this layer performs a substitution when calling the base provider's **WSPStringToAddress**. In this case, the base provider's **WSAPROTOCOL\_INFOW** structure should be referenced by the *IpProtocolInfo* parameter.

One vital benefit of this policy is that base service providers do not have to be aware of protocol chains.

This same propagation policy applies when propagating a **WSAPROTOCOL\_INFOW** structure through a layered sequence of other functions such as **WSPAddressToString**, **WSPDuplicateSocket**, **WSPStartup**, or **WSPSocket**.

## Error Codes

Error code	Meaning
WSAEFAULT	Specified address buffer is too small. Pass in a larger buffer.
WSAEINVAL	Unable to translate the string into a <b>SOCKADDR</b> , or the provider was unable to support the indicated address family, or the specified <i>IpProtocolInfo</i> did not refer to a <b>WSAPROTOCOL_INFOW</b> structure supported by the provider.

### Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in *Ws2spi.h*.

---

## CHAPTER 12

# Winsock 2 Protocol-Specific Annex

This chapter describes the Microsoft® Windows® Sockets 2 Protocol-Specific Annex. It presents information drawn from the Windows Sockets 2 Protocol-Specific Annex specification.

## Using the Annex

This chapter and its sections provide the information needed to create a Windows Sockets (Winsock) application for Microsoft® Windows NT®/Windows® 2000, Windows 98®, and Windows 95® operating systems, using the Microsoft implementation of Windows Sockets 2. It is intended as a reference tool and outlines the functions in the Windows Sockets Protocol-Specific Annex.

To make the best use of this chapter, you should have a working knowledge of Microsoft Win32® programming concepts. The Win32® Developer's Reference Library, another library in the Windows Programming Reference Series (WPRS) is available from Microsoft Press. If you do not have a working knowledge of Win32®, you may want to refer to the Win32 Developer's Reference Library or, other references that provide a more systematic guide to writing Winsock applications.

---

**Note** This documentation is intended for application developers. If you are developing a transport or service provider, see the Service Provider documentation installed with the Platform SDK.

---

## Overview of Windows Sockets 2

Windows Sockets 2 is a superset of the widely deployed Windows Sockets 1.1. Windows Sockets 2 extends the 1.1 interface in a number of areas while maintaining full backward compatibility. Among other enhancements, it provides access to protocols other than TCP/IP. Winsock enables an application to use the familiar socket interface to achieve simultaneous access to any number of installed transport protocols.

While most of these protocols may be used with standard Windows Sockets 2–interface mechanisms, each supported protocol contains conventions and behaviors that developers need to be aware of. Some individual protocols support special features that do *not* lend themselves to being treated generically. This document provides the details that developers need to effectively use all of the supported protocols.

## Microsoft Extensions and Windows Sockets 2

The Windows Sockets 2 specification defines an extension mechanism that exposes advanced transport functionality to application programs. For more information, see *Function Extension Mechanism*.

The following Microsoft-specific extensions were added to Windows Sockets 1.1. They are also available in Windows Sockets 2:

- **AcceptEx**
- **GetAcceptExSockaddr**
- **TransmitFile**
- **WSARecvEx**

These functions are not exported from the `Ws2_32.dll`; they are exported from `Mswsock.dll`.

An application written to use the Microsoft-specific extensions to Winsock will not run correctly over a Windows Sockets service provider that does not support those extensions.

### Socket Handles for Windows Sockets 2

A socket handle can optionally be a file handle in Windows Sockets 2. It is possible to use socket handles with **ReadFile**, **WriteFile**, **ReadFileEx**, **WriteFileEx**, **DuplicateHandle**, and other Win32 functions. Not all transport service providers will support this option. However, for an application to run over the widest possible number of service providers, it should not assume that socket handles are file handles.

Windows Sockets 2 has expanded certain functions that transfer data between sockets using handles. The functions offer advantages specific to sockets for transferring data and include **WSARecv**, **WSASend**, and **WSADuplicateSocket**.

## TCP/IP

This section describes TCP/IP functions, data structures, and TCP/IP controls.

### TCP/IP Introduction

This section covers extensions to Windows Sockets 2 that are specific to TCP/IP protocols. It also describes aspects of base Windows Sockets 2 functions that require special consideration or that may exhibit unique behavior when using TCP/IP.

**Fast Facts**

<b>Protocol name(s)</b>	TCP, UDP
<b>Description</b>	Provides two transport services over the IP networking layer: UDP for unreliable datagrams, TCP for reliable, connection-oriented byte streams.
<b>Address family</b>	AF_INET
<b>Header file</b>	Ws2tcpip.h

## TCP/IP Overview

The TCP/IP protocol suite forms the backbone of the global Internet. The original Windows Sockets specification (version 1.1) addressed only TCP/IP protocols, and still contains a few IP-specific attributes.

Two basic types of transport services are offered: unreliable datagrams (UDP), and reliable connection-oriented byte streams (TCP). In addition, a raw socket is optionally supported. Raw sockets allow an application to communicate through protocols other than TCP and UDP such as ICMP.

## TCP/IP Data Structures

The **INTERFACE\_INFO** structure is used in conjunction with the **SIO\_GET\_INTERFACE\_LIST** ioctl command. It is defined in Ws2tcpip.h file and is reproduced here.

```
typedef struct _INTERFACE_INFO
{
    u_long          iiFlags;        // Type and status of
                                   // the interface
    struct sockaddr iiAddress;     // Interface address
    struct sockaddr iiBroadcastAddress; // Broadcast
                                   // address
    struct sockaddr iiNetmask;    // Network mask
} INTERFACE_INFO;
```

### Members

#### iiFlags

A bitmask describing the status of the interface. The following flags are possible.

<b>Flag</b>	<b>Meaning</b>
IFF_UP	The interface is running.
IFF_BROADCAST	The broadcast feature is supported.
IFF_LOOPBACK	The loopback interface is running.
IFF_POINTTOPOINT	The interface is using point-to-point link.
IFF_MULTICAST	The multicast feature is supported.

**iiAddress**

The address of the interface.

**iiBroadcastAddress**

The broadcast address of the interface or the address of the other side for point-to-point links.

**iiNetmask**

The netmask used by the interface.

## TCP/IP Controls

The following controls are available in all TCP/IP implementations:

- UNIX `ioctl`s
- TCP/IP Socket Options

### UNIX `ioctl`s

The **SIODCONF** command provided by most UNIX implementations is supported in the form of **WSAIoctl** and **WSPIOctl** functions with the command **SIO\_GET\_INTERFACE\_LIST**. This command returns the list of configured interfaces and their parameters. Support of this command is *mandatory* for Windows Sockets 2 compliant-TCP/IP service providers.

The parameter *IpvOutBuffer* points to the buffer in which **WSAIoctl** and **WSPIOctl** store the information about interfaces. The number of interfaces (number of structures returned in *IpvOutBuffer*) can be determined based on the actual length of the output buffer returned in *IpcbBytesReturned*.

### TCP/IP Socket Options

The following TCP/IP-specific options are defined.

Value	Meaning
SIO_GET_INTERFACE_LIST	Returns a list of all IP interfaces on the system.

#### *level=IPPROTO\_IP*

Value	Type	Meaning
IP_OPTIONS	char FAR *	Lists IP options to be inserted into the outgoing packets.
IP_TOS	int	Specifies the type of service to be used. See following for more information about setting TOS.
IP_TTL	int	Specifies the TTL to be used.

Value	Type	Meaning
IP_HDRINCL	BOOL	If TRUE, the application provides the IP header in the packets sent over <b>SOCK_RAW</b> interface. Otherwise, the header is provided by the service provider.
IP_MULTICAST_IF	<b>IN_ADDR FAR</b> *structure	Selects the interface for the outgoing multicast packets. The <i>optval</i> should point to the address of the interface to be used. If NULL, the default interface is used.
IP_MULTICAST_TTL	int	TTL used for the multicast packets.
IP_MULTICAST_LOOP	BOOL	If TRUE, multicast loopback is enabled. Otherwise, it is disabled.
IP_ADD_MEMBERSHIP	<b>IP_MREQ FAR *</b> structure	Specifies the multicast group to join.
IP_DROP_MEMBERSHIP	<b>IP_MREQ FAR *</b> structure	Specifies the multicast group to leave.

**level= IPPROTO\_UDP**

Value	Type	Meaning
UDP_NOCHECKSUM	BOOL	If this option is set, UDP datagrams are sent with the checksum of zero. This option is required. If a service provider does not have a mechanism to disable UDP checksum calculation, it may simply store this option without taking any action.

**level= IPPROTO\_TCP**

Value	Type	Meaning
TCP_EXPEDITED_1122	BOOL	If set, the SP implements the expedited data as specified in RFC-1222. Otherwise, the BSD style (default) is used. This option can be set on the connection only once, that is, once on, this option can not be turned off. This option is not required.

### IP\_OPTIONS

Specifies IP options to be inserted into outgoing datagrams. Setting the new options overwrites all the previously specified options. Setting *optval* to zero means removing all previously specified options. The support of IP\_OPTIONS is not required. In order to check whether IP\_OPTIONS is supported, an application should use **getsockopt** to attempt to get the current options. If **getsockopt** fails, the IP\_OPTIONS is not supported.

### IP\_TOS

Changes the default value set by the TCP/IP service provider in the TOS field of the IP header in outgoing datagrams. The support of IP\_TOS is not required. To check whether IP\_TOS is supported, an application should use **getsockopt** to attempt to get the current options. If **getsockopt** fails, the IP\_TOS is not supported.

There are two important points to keep in mind regarding TOS:

- TOS and IP precedence bits have been deprecated, and the corresponding TOS RFC obviated, by Diffserv code point (DSCP). Many DSCP values have been recommended for standardization by the IETF. For more information about these standardization recommendations, consult the IETF Web site at [www.ietf.org](http://www.ietf.org) and look into the Diffserv working group.
- Programmers should never directly set TOS bits. Use the QOS API to set the bits. Diffserv code point is explained in the Quality of Service SDK reference section.

It is important to note that when QOS is enabled on a Windows 2000 computer, all TOS settings are overridden by settings implemented or set using the traffic control API (TC API) or the QOS API.

### IP\_TTL

Changes the default value set by the TCP/IP service provider in the TTL field of the IP header in outgoing datagrams. The support of IP\_TTL is not required. In order to check if IP\_TTL is supported, an application should use **getsockopt** to attempt to get the current options. If **getsockopt** fails, the IP\_TTL is not supported.

### IP\_HDRINCL

By default TCP/IP service provider forms the IP header for the outgoing datagrams. Some applications, however, may wish to provide their own IP header. Such applications should set **IP\_HDRINCL** option to TRUE and then supply a completed IP header at the front of each outgoing datagram. The only modification TCP/IP service provider may make to the supplied IP header is setting the ID field, if the value supplied by the application is 0. The **IP\_HDRINCL** option is applied only to the SOCK\_RAW type of protocol. If a TCP/IP service provider supports SOCK\_RAW protocol, it should also support **IP\_HDRINCL** option.

### IP\_MULTICAST\_IF

Information supplied at a later release.

### IP\_MULTICAST\_TTL

Information supplied at a later release.

### IP\_MULTICAST\_LOOP

Information supplied at a later release.

### IP\_ADD\_MEMBERSHIP

Information supplied at a later release.

### IP\_DROP\_MEMBERSHIP

Support of these options is required if a protocol supports multicast. This will be indicated in the **WSAPROTOCOL\_INFO** structure returned by **WSAEnumProtocols** as follows:

XPI\_SUPPORTS\_MULTIPOINT = 1

XP1\_MULTIPOINT\_CONTROL\_PLANE = 0

XP1\_MULTIPOINT\_DATA\_PLANE = 0

## TCP/IP Function Details

This section presents general information about TCP/IP function details for multicast, raw sockets, and Ipv6 support.

### Multicast

Generic Winsock multipoint functions support IP multicast. However, the TCP/IP transport providers who support multicast must also provide BSD style–multicast support by supporting the corresponding multicast options. This will simplify the porting of existing multicast applications to Windows Sockets 2.

### TCP/IP Raw Sockets

The TCP/IP service providers may support the **SOCK\_RAW** socket type. There are two types of such sockets:

- The first type assumes a known protocol type as written in the IP header. An example of the first type of socket is ICMP.
- The second type allows any protocol number. An example of the second type would be an experimental protocol that is not supported by the service provider.

If a TCP/IP service provider supports **SOCK\_RAW** sockets for the **AF\_INET** family, the corresponding protocol(s) should be included in the list returned by **WSAEnumProtocols**. The *ipProtocol* field of the **WSAPROTOCOL\_INFO** structure may be set to zero if the service provider allows an application to specify any value for the *protocol* parameter for the **Socket**, **WSASocket**, and **WSPSocket** functions.

---

**Note** An application may not specify zero (0) as the *protocol* parameter for the **Socket**, **WSASocket**, and **WSPSocket** functions if **SOCK\_RAW** sockets are used.

---

The following rules are applied to the operations over **SOCK\_RAW** sockets:

- When an application sends a datagram it may or may not include the IP header at the front of the outgoing datagrams depending on the **IP\_HDRINCL** option set for the socket.



- An application always gets the IP header at the front of each received datagram regardless of the **IP\_HDRINCL** option.
- Received datagrams are copied into all **SOCK\_RAW** sockets that satisfy the following conditions:
  - The protocol number specified for the socket should match the protocol number in the IP header of the received datagram.
  - If a local IP address is defined for the socket, it should correspond to the destination address as specified in the IP header of the received datagram. An application may specify the local IP address by calling **bind** functions. If no local IP address is specified for the socket, the datagrams are copied into the socket regardless of the destination IP address in the IP header of the received datagram.
  - If a foreign address is defined for the socket, it should correspond to the source address as specified in the IP header of the received datagram. An application may specify the foreign IP address by calling **connect** functions. If no foreign IP address is specified for the socket, the datagrams are copied into the socket regardless of the source IP address in the IP header of the received datagram.

It is important to understand that **SOCK\_RAW** sockets may get many unexpected datagrams. For example, a PING program may use **SOCK\_RAW** sockets to send ICMP echo requests. While the application is expecting ICMP echo responses, all other ICMP messages (such as ICMP **HOST\_UNREACHABLE**) may be delivered to this application also. Moreover, if several **SOCK\_RAW** sockets are open on a machine at the same time, the same datagrams may be delivered to all the open sockets. An application must have a mechanism to recognize its datagram and to ignore all others. Such mechanism may include inspecting the received IP header—using unique identifiers in the ICMP header (**ProcessID**, for example), and so forth.

---

**Note** On Windows NT/Windows 2000, raw socket support requires administrative privileges. Users running Winsock applications that make use of raw sockets must have administrative privileges on the computer, otherwise raw socket calls fail with an error code of **WSAEACCESS**.

---

## IPv6 Support

If TCP/IP service provider supports IPv6 addressing, it must install itself twice:

- Once for IPv4.
- Once for IPv6 address family.

So, **WSAEnumProtocols** returns two **WSAPROTOCOL\_INFO** structures for each of the supported socket types (**SOCK\_STREAM**, **SOCK\_DGRAM**, **SOCK\_RAW**). The *iAddressFamily* must be set to **AF\_INET** for IPv4 addressing, and to **AF\_INET6** for IPv6 addressing.

The IPv6 addresses are described in the following structures.

The IPv6 addresses are described in the following structures:

```

struct sockaddr_in6 {
    short          sin6_family;    /* AF_INET6 */
    u_short       sin6_port;      /* Transport level port number */
    u_long        sin6_flowinfo;  /* IPv6 flow information */
    struct in_addr sin6_addr;     /* IPv6 address */
};

struct    in_addr6 {
    u_char   s6_addr[16];        /* IPv6 address */
};

```

If an application uses Windows Sockets 1.1 functions and wants to use IPv6 addresses, it may continue to use all the old functions that take the **SOCKADDR** structure as one of the parameters (**bind**, **connect**, **sendto**, and **recvfrom**, **accept**, and so forth). The only change that is required is to use **SOCKADDR\_IN6** instead of **SOCKADDR**.

However, the name resolution functions (**gethostbyname**, **gethostbyaddr**, and so forth) and address conversion functions (**inet\_addr**, **inet\_ntoa**) can not be reused because they assume an IP address 4 bytes in length. An application that wants to perform name resolution and address conversion for IPv6 addresses must use the Windows Sockets 2-specific functions (**WSAStringToAddress**, **WSAAddressToString**, and so forth).

## Text Representation of IPv6 Addresses

This section is copied from the IP Version 6 Addressing Architecture by R.Hinden and S. Deering. There are three conventional forms for representing IPv6 addresses as text strings:

- The preferred form is x:x:x:x:x:x:x, where the “x”s are the hexadecimal values of the eight 16-bit pieces of the address.

Examples:

```

FEDC:BA98:7654:3210:FEDC:BA98:7654:3210
1080:0:0:0:8:800:200C:417A

```

---

**Note** It is not necessary to write the leading zeros in an individual field, but there must be at least one numeral in every field (except for the case described in the second form).

---

- Due to the method of allocating certain styles of IPv6 addresses, it is common for addresses to contain long strings of zero bits. In order to make writing addresses containing zero bits easier, a special syntax is available to compress the zeros. The use of double quotation marks (“::”) indicates multiple groups of 16-bits of zeros.

For example, the multicast address

```
FF01:0:0:0:0:0:0:43
```

may be represented as:

```
FF01::43
```

The double quotation marks (“:”) can only appear once in an address. They can be used to compress leading or trailing zeros in an address.

- An alternative form that may be more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is x:x:x:x:d.d.d.d, where the “x”s are the hexadecimal values of the six high-order 16-bit pieces of the address, and the “d”s are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation).

Examples:

0:0:0:0:0:0:13.1.68.3

0:0:0:0:0:FFFF:129.144.52.38

or in compressed form:

::13.1.68.3

::FFFF:129.144.52.38

## TCP/IP Header File

Use the `Ws2tcpip.h` header file for TCP/IP transactions.

## IPX/SPX

This section introduces and provides important specifics for IPX/SPX:

### IPX/SPX Introduction

This section covers extensions to Windows Sockets 2 that are specific to the IPX family of transport protocols. It also describes aspects of base Windows Sockets 2 functions that require special consideration or that may exhibit unique behavior.

#### Fast Facts

<b>Protocol name(s)</b>	IPX, SPX
<b>Description</b>	Provides transport services over the IPX networking layer: IPX for unreliable datagrams, SPX for reliable, connection-oriented message streams.
<b>Address family</b>	AF_IPX
<b>Header file</b>	Wsipx.h

### IPX/SPX Overview

This section discusses how to use the Windows Sockets 2 API with the IPX family of protocols. Traditionally, the Windows Sockets 1.x specification has been used with the IP family of protocols such as TCP and UDP. With the advent of Windows Sockets 2, the API has been updated to access a wide range of transport and network types more easily.

The Windows Sockets 2 API is sufficiently generic. That is, programmers need to know very little about the specifics of the IPX/SPX implementation. However, if you are moving traditional IPX applications to Winsock or want more knowledge of the IPX/SPX implementation, this appendix is for you. Be aware that IPX networks operate differently than IP networks; consideration of this fact will most likely be manifest in your code.

IPX/SPX features are defined in the header file `Wsipx.h`.

## AF\_IPX Address Family

The IPX *address family* defines the addressing structure for protocols that use standard IPX socket addressing. For these transports, an endpoint address consists of a *network number*, *node address*, and *socket number*.

The network number is an administrative domain and typically names a single Ethernet or token ring segment. The node number is a station's physical address. The combination of net and node form a unique station address that is presumed to be unique in the world. Net and node numbers are represented in ASCII text in either block or dashed notation as: "0101a040,00001b498765" or "01-01-a0-40,00-00-1b-49-87-65". Leading zeros need not be present.

The IPX socket number is a network/transport service number much like a TCP port number and is not to be confused with the Winsock socket descriptor. IPX socket numbers are global to the end station and cannot be bound to specific net/node addresses. For instance, if the end station has two network interface cards, a bound socket can send and receive on both cards. In particular, datagram sockets would receive broadcast datagrams on both cards.

---

**Caution** `SOCKADDR_IPX` is 14 bytes long and is shorter than the 16-byte `SOCKADDR` reference structure. IPX/SPX implementations may accept the 16-byte length as well as the true length. If you use `SOCKADDR_IPX` and a hard-coded length of 16 bytes, the implementation may assume that it has access to the 2 bytes following your structure.

---

Field	Value
<code>sa_family</code>	Address family <code>AF_IPX</code> in host order.
<code>sa_netnum</code>	IPX network identifier in network order.
<code>sa_nodenum</code>	Station node address, flushed right.
<code>sa_socket</code>	IPX socket number in network order.

## IPX Family of Protocol Identifiers

The *protocol* parameter in `socket` and `WSASocket` is an identifier that establishes a network type and a method for identifying the various transport protocols that run on the network. Unlike IP, IPX does not use a single protocol value for selecting a transport stack. Since there is no network requirement to use a specific value for each transport

protocol, we are free to assign them in a manner convenient for Winsock applications. We avoid values 0–255 in order to avoid collisions with the corresponding PF\_INET protocol values.

Name	Value	Socket types	Description
Reserved	0–255		Reserved for PF_INET protocol values.
NSPROTO_IPX	1000–1255	<b>SOCK_DGRAM</b> <b>SOCK_RAW</b>	Datagram service for IPX.
NSPROTO_SPX	1256	<b>SOCK_STREAM</b> <b>SOCK_SEQPKT</b>	Reliable packet exchange using fixed-sized packets.

---

**Note** When NSPROTO\_SPX is specified, the SPX II protocol is automatically utilized if both endpoints are capable of doing so.

---

## Broadcast to Local Network

A broadcast can be made to the locally attached network by setting *sa\_netnum* to binary zeros and *sa\_nodenum* to binary ones. This broadcast may be sent to the primary network for the device, or to all locally attached networks at the option of the service provider. Broadcasts to the local network are not propagated by routers.

## All Routes Broadcast

A general broadcast through the Internet is achieved by setting the *sa\_netnum* and *sa\_nodenum* fields to binary ones (–1). The service provider translates this request to a type 20 packet, which IPX routers recognize and forward. The packet visits all subnets and may be duplicated several times. Receivers must handle several duplicate copies of the datagram.

Use of this broadcast type is unpopular among network administrators, so its use should be extremely limited. Many routers disable this type of broadcast, leaving parts of the subnet invisible to the packet.

## Directed Broadcast

Generally considered more network friendly than an all-routes broadcast, a directed broadcast limits itself to the local network that you specify. Fill *sa\_netnum* with the target network number and *sa\_nodenum* with binary ones. Routers forward this request to the destination network where it becomes a local broadcast. Intermediate networks do not see this packet as a broadcast.

## About Media Packet Size

Media packet size affects the ability of IPX protocols to transfer data across networks and can prove challenging to deal with in a transport-independent manner. IPX does not segment packets, nor does it report when packets are dropped due to size violations. This means that some entity on the end station must maintain knowledge of the maximum packet size to be used on any given internetwork path. Traditionally, IPX datagram and SPX connection-oriented services have left this burden to the application, while SPXII has used Large Internet Packet negotiation to handle it transparently.

Winsock attempts to set rational packet size limits for its various IPX protocols as described in the next section. These limits can be viewed and modified by applications through get/set socket options. When determining maximum packet size, the three areas of concern are:

- # Media packet size
- # Routable packet size
- # End-station packet size

*Media packet size* reflects the maximum packet size acceptable on any media the packet traverses to its destination. Packet size varies among differing media such as Ethernet and token ring. The amount of data space within a packet can also vary within a given media, depending on the packet header arrangement. For instance, the effective data size of an Ethernet packet varies depending on whether it is of the type Ethernet II, Ethernet 802.2, or Ethernet SNAP.

*Routable packet size* reflects the maximum packet size an intermediate router is willing to forward. Modern IPX routers are built to route any size datagram as long as it remains within the media size of the sending and receiving network. However, older routers may limit maximum packet size to 576 bytes, including protocol headers.

*End-station packet size* reflects the size of the listen buffers that end stations have posted to receive incoming packets. Even when the media and router limits allow a packet through, it may be discarded by the end station if the receiving application has posted a smaller buffer. Many traditional IPX/SPX applications limit receive buffer size such that the data portion must be no larger than 512 or 1024 bytes.

## How Packet Size Affects Protocols

Media packet size issues discussed in the topic, *About Media Packet Size*, affect the various PF\_IPX protocols differently. A common strategy for handling various router and media size constraints is to use the full media size when the remote station's network number matches the local station, and fall back to the minimum packet size otherwise.

### NSPROTO\_IPX

Provides a datagram service; each datagram must reside within the maximum packet size. Winsock sets the maximum datagram packet size to the local media packet size minus the IPX header. Keep in mind, however, that if the packet is routed, it may hit router restrictions en route. Make sure your application can fall back to 546-byte datagrams.

**NSPROTO\_SPX**

Provides stream and sequenced-packet services. Winsock IPX/SPX lets data streams and messages span multiple packets, so packet size does not limit the amount of data handled by **send** and **recv**. However, the underlying media size must be set correctly or the first large packet will be undeliverable and the connection will reset. If the target station is on the local network, Winsock sets its packet size to the media packet size. Otherwise, it defaults to 512 bytes. This size can be changed immediately after **connect** or **accept** through **setsockopt**.

**NSPROTO\_SPXII**

SPXII features *large Internet packet* negotiation to maintain a best-fit size for packets and does not require programmer intervention. However, if the remote station does not support SPXII and negotiates down to standard SPX, the NSPROTO\_SPX rules are in effect.

Protocol	Media	Type	Data packet size
NSPROTO_IPX	Ethernet	Ethernet II	
		802.3	
		802.2	1466
		SNAP	
NSPROTO_SPX	Token Ring	four megabit	
		16 megabit	
NSPROTO_SPX	Ethernet	Ethernet II	
		802.3	
		802.2	1454
		SNAP	
NSPROTO_SPX	Token Ring	four megabit	
		16 megabit	

## IPX/SPX Data Structures

The IPX/SPX data structures in this section are Microsoft-specific implementations, and can be found in Wsnwlink.h.

---

## IPX\_ADDRESS\_DATA

The **IPX\_ADDRESS\_DATA** structure provides information about a specific adapter to which IPX is bound. Used in conjunction with **getsockopt** function calls that specify **IPX\_ADDRESS** in the *optname* parameter.

```
typedef struct _IPX_ADDRESS_DATA {
    INT    adapternum; // input: 0-based adapter number
    UCHAR netnum[4];  // output: IPX network number
    UCHAR nodenum[6]; // output: IPX node address
    BOOLEAN wan;      // output: TRUE = adapter is on
                    // a wan link
    BOOLEAN status;  // output: TRUE = wan link is up
                    // (or adapter is not wan)
    INT    maxpkt;   // output: max packet size,
                    // not including IPX header
    ULONG linkspeed; // output: link speed in 100
                    // bytes/sec (i.e. 96 == 9600 bps)
} IPX_ADDRESS_DATA, *PIPX_ADDRESS_DATA;
```

## Members

### **adapternum**

[in] Adapter number.

### **netnum**

[out] IPX network number for the associated adapter.

### **nodenum**

[out] IPX node address for the associated adapter.

### **wan**

[out] Specifies whether the adapter is on a wide area network (WAN) link. When TRUE, the adapter is on a WAN link.

### **status**

[out] Specifies whether the WAN link is up. FALSE indicates that the WAN link is up or the adapter is not on a WAN. Compare with **wan** to determine the meaning.

### **maxpkt**

[out] Maximum allowable packet size, excluding the IPX header.

### **linkspeed**

[out] Link speed, returned in 100 bytes per second increments. For example, a 9600 byte per second link would be return a value of 96.

## Remarks

Adapter numbers are base zero, so if there are eight adapters on a given computer, they are numbered 0–7. To determine the number of adapters present on the computer, call the **getsockopt** function with **IPX\_MAX\_ADAPTER\_NUM**.

## ! Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Wsnwlink.h.



**+** See Also

**getsockopt**

## IPX\_NETNUM\_DATA

The **IPX\_NETNUM\_DATA** structure provides information about a specified IPX network number. Used in conjunction with **getsockopt** function calls that specify **IPX\_GETNETINFO** in the *optname* parameter.

```
typedef struct _IPX_NETNUM_DATA {
    UCHAR  netnum[4]; // input: IPX network number
    USHORT hopcount; // output: hop count to this network,
                    // in machine order
    USHORT netdelay; // output: tick count to this
                    // network, in machine order
    INT    cardnum;  // output: 0-based adapter number
                    // used to route to this net;
                    // can be used as adapternum input
                    // to IPX_ADDRESS
    UCHAR  router[6]; // output: MAC address of the next
                    // hop router, zeroed if
                    // the network is directly attached
} IPX_NETNUM_DATA, *PIPX_NETNUM_DATA;
```

### Members

#### netnum

[in] IPX network number for which information is being requested.

#### hopcount

[out] Number of hops to the IPX network being queried, in machine order.

#### netdelay

[out] Network delay tick count required to reach the IPX network, in machine order.

#### cardnum

[out] Adapter number used to reach the IPX network. The adapter number is zero based, such that if eight adapters are on a given computer, they are numbered 0–7.

#### router

[out] Media Access Control (MAC) address of the next-hop router in the path between the computer and the IPX network. This value is zero if the computer is directly attached to the IPX network.

### Remarks

If information about the IPX network is in the computer's IPX cache, the call will return immediately. If not, RIP requests are used to resolve the information.

**!** Requirements

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Wsnwlink.h.

**+** See Also

**getsockopt**

## IPX\_SPXCONNSTATUS\_DATA

The **IPX\_SPXCONNSTATUS\_DATA** structure provides information about a connected SPX socket. Used in conjunction with **getsockopt** function calls that specify **IPX\_SPXGETCONNECTIONSTATUS** in the *optname* parameter. All numbers in **IPX\_SPXCONNSTATUS\_DATA** are in Novell (high-low) order.

```
typedef struct _IPX_SPXCONNSTATUS_DATA {
    UCHAR   ConnectionState;
    UCHAR   WatchDogActive;
    USHORT  LocalConnectionId;
    USHORT  RemoteConnectionId;
    USHORT  LocalSequenceNumber;
    USHORT  LocalAckNumber;
    USHORT  LocalAllocNumber;
    USHORT  RemoteAckNumber;
    USHORT  RemoteAllocNumber;
    USHORT  LocalSocket;
    UCHAR   ImmediateAddress[6];
    UCHAR   RemoteNetwork[4];
    UCHAR   RemoteNode[6];
    USHORT  RemoteSocket;
    USHORT  RetransmissionCount;
    USHORT  EstimatedRoundTripDelay; /* In milliseconds */
    USHORT  RetransmittedPackets;
    USHORT  SuppressedPacket;
} IPX_SPXCONNSTATUS_DATA, *PIPX_SPXCONNSTATUS_DATA;
```

### Members

**ConnectionState**

Specifies the connection state.

**WatchDogActive**

Specifies whether watch dog capabilities are active.

**LocalConnectionId**

Specifies the local connection ID.

**RemoteConnectionId**

Specifies the remote connection ID.

**LocalSequenceNumber**

Specifies the local sequence number.

**LocalAckNumber**

Specifies the local acknowledgment (ACK) number.

**LocalAllocNumber**

Specifies the local allocation number.

**RemoteAckNumber**

Specifies the remote acknowledgment (ACK) number.

**RemoteAllocNumber**

Specifies the remote allocation number.

**LocalSocket**

Specifies the local socket.

**ImmediateAddress**

Specifies the IPX address to which the local computer is attached.

**RemoteNetwork**

Specifies the network to which the remote host is attached.

**RemoteNode**

Specifies the remote node.

**RemoteSocket**

Specifies the remote socket.

**RetransmissionCount**

Specifies the number of retransmissions.

**EstimatedRoundTripDelay**

Specifies the estimated round trip–time delay for a given packet.

**RetransmittedPackets**

Specifies the number of retransmitted packets on the socket.

**SuppressedPacket**

Specifies the number of suppressed packets on the socket.

**! Requirements**

**Version:** Requires Windows Sockets 2.0.

**Header:** Declared in Wsnwlink.h.

**+ See Also**

**getsockopt**

## IPX/SPX Controls

This section describes IPX/SPX socket options.

## NSPROTO\_IPX Socket Options

NSPROTO\_IPX options are handled through **getsockopt** and **setsockopt**. Each option is accessed by setting *level = NSPROTO\_IPX* and *optname* to one of the following values.

### *level = NSPROTO\_IPX*

Option	Type	Default	Meaning
IPX_CHECKSUM	Bool	off	When set, IPX performs a checksum on outgoing packets and verifies the checksum of incoming packets.
IPX_TXPKTSIZE	int	Media size to a maximum of 1466	Sets maximum send datagram size. This size does not include the IPX header or any media headers that may also be used. May be increased to media size.
IPX_RXPKTSIZE	int	Media size to a maximum of 1466	Sets maximum receive datagram size. This size does not include the IPX header or any media headers that may also be used. May be increased to media size.
IPX_TXMEDIASIZE	int	Primary board	Returns send media size that sets an upper bound for datagram size.
IPX_RXMEDIASIZE	int	Primary board	Returns receive media size that sets an upper bound for datagram size.
IPX_PRIMARY	Bool	Primary	Restricts traffic to the primary network board.

### *level = NSPROTO\_SPX*

Option	Type	Default	Meaning
SPX_CHECKSUM	Bool	off	When set, IPX performs a checksum on outgoing packets and verifies the checksum of incoming packets. Not supported on all platforms.
SPX_TXPKTSIZE	int	Media size to a maximum of 1466	Sets maximum send datagram size. This size does not include the SPX header or any media headers that may also be used. May be increased to media size.

(continued)

*(continued)*

Option	Type	Default	Meaning
SPX_RXPKTSIZE	int	Media size to a maximum of 1466	Sets maximum receive datagram size. This size does not include the SPX header or any media headers that may also be used. May be increased to media size.
SPX_TXMEDIASIZE	int	Primary board	Returns send media size minus SPX and media headers. This sets an upper bound for message segmentation packet size.
SPX_RXMEDIASIZE	int	Primary board	Returns receive media size minus SPX and media headers. This sets an upper bound for receive packet size.
SPX_RAWSPX	Bool	off	When set, the IPX/SPX protocol header is passed with the data.

**SPX\_RAWSPX**

When set, `SPX_RAWSPX` lets the application manage the IPX/SPX header. In this mode, Winsock will not segment messages, restricting maximum **send** and **recv** message size to the underlying packet size. Packet size options are automatically adjusted to include the IPX/SPX headers. Fields that can be set by the application are detailed in `Wsipx.h`.

## DECnet

This section covers extensions to Windows Sockets 2 that are specific to DECnet. It also describes aspects of base Windows Sockets 2 functions that require special consideration or that may exhibit unique behavior.

**Fast Facts**

<b>Protocol name(s)</b>	DNPROTO_NSP
<b>Socket type</b>	SOCK_SEQPACKET
<b>Description</b>	Provides transport service over the DECnet network layer.
<b>Address family</b>	AF_DECnet
<b>Header file</b>	Ws2dnet.h

## DECnet Overview

The Digital Network Architecture (DNA) consists of an architectural overview and a set of specifications defining various network protocol layers. DECnet refers to a set of

products that implement the Digital Network Architecture. DECnet Phase IV, as introduced in 1982, supports peer-to-peer connectivity in both local and wide area networks.

## **DNPROTO\_NSP Protocol Family**

DECnet Phase IV uses Network Services Protocol (NSP) as its transport layer.

## **AF\_DECnet Address Families**

The AF\_DECnet Address families include:

### **DECnet Phase IV Node Addresses**

DECnet Phase IV node addresses are hierarchical, indicating the routing area and the node number within that area. The binary format of the address is a 16-bit unsigned integer. The high-order 6 bits indicate the area, the low-order 10 bits are the node number within the area.

The ASCII format of the address is area.number with area in the range 1–63, and number in the range 1–1023.

For example: A DECnet node address in area 5, number 7 is represented as shown in the following table.

ASCII format	“5.7”
Binary value	000101 0000000111.
Hexadecimal value	0x1407

### **DECnet Extended Addressing**

DECnet extended addressing allows the DECnet NSP transport to be run over the OSI-routing layer. Sockets opened through AF\_DECnet assumes that addresses of 3 to 20 bytes in length are OSI-style addresses.

### **DECnet Objects**

DECnet client tasks specify the server task that they want to communicate with by using network object number and task names. The DECnet object number is an 8-bit unsigned value. Object numbers in the range 1–127 are reserved as generic objects for digital use. Numbers 1–128 are available for user-written generic objects.

If the object number is zero, then the network connect is done to a specific server task name. Task names are 1–16-byte ASCII strings.

#17	FAL	Generic DECnet file access listener.
#19	NML	Generic DECnet network management listener.
#0	DEBUG_TASK	User-specified debug server task.

## SOCK\_SEQPACKET Socket Type

DECnet sockets use sequenced packets that maintain message boundaries across the network.

## DECnet Data Structures

The DECnet structures and constants include Winsock2.h and Ws2dnet.h. This section presents information about their constants and controls.

### Manifest Constants (Winsock2.h)

```
#define AF_DECnet      12    // DECnet address family
#define SOCK_SEQPACKET 5    // Socket type - Message-oriented
```

### Manifest Constants (Ws2dnet.h)

```
#define DNPROTO_NSP    1    // DECnet NSP transport protocol
#define SO_LINKINFO    7    // Get DECnet logical link information

#define DN_MAXADDL     20   // Maximum DECnet address length
#define DN_ADDL        2    // DECnet NSP address length
#define DN_MAXOPTL     16   // Maximum DECnet optional data
#define DN_MAXOBJL     16   // Maximum DECnet object name
#define DN_MAXACCL     40   // Maximum DECnet access string
                             // length (39 bytes + trailing zero)
#define DN_MAXALIASL   128  // Maximum DECnet alias string length
#define DN_MAXNODEL   256  // Maximum node string length

// DECnet logical link states
#define LL_INACTIVE    0    // logical link inactive
#define LL_CONNECTING 1    // logical link connecting
#define LL_RUNNING     2    // logical link running
#define LL_DISCONNECTING 3  // logical link disconnecting

// DECnet incoming access types
#define DN_NONE        0x00
#define DN_RO          0x01
#define DN_WO          0x02
#define DN_RW          0x03
```

### Data Structures (Ws2dnet.h)

This section lists address type information for the header files.

#### DECnet Node Address

```
// if a_len = 2, DECnet NSP transport & DECnet routing
// if a_len > 2, DECnet NSP transport & OSI routing
```



```

struct dn_naddr {
    unsigned short  a_len;    // address length - set to DN_ADDL
    unsigned char   a_addr[DN_MAXADDL]; // DECnet address
};

```

### DECnet Socket

```

// DECnet uses object names & numbers instead of ports
struct sockaddr_dn {
    unsigned short  sdn_family;           // AF_DECnet
    unsigned char   sdn_flags;           // Reserved - set to zero
    unsigned char   sdn_objnum;         // DECnet object number
    unsigned short  sdn_objname1;       // DECnet object name length
    unsigned char   sdn_objname[DN_MAXOBJL]; // DECnet object name
    struct dn_naddr sdn_add;            // DECnet address
};

```

### DECnet Node Entity

```

struct nodeent_f {
    char FAR *n_name;           // pointer to name of DECnet node
    unsigned short n_addrtype;  // address type
    unsigned short n_length;    // address length
    unsigned char FAR *n_addr;  // pointer to DECnet address
    unsigned char FAR *n_params;
    unsigned char  n_reserved[16]; // Reserved
};

```

### DECnet Optional Data

```

struct optdata_dn {
    unsigned short  opt_status;        // status return
    unsigned short  opt_optl;         // length of data
    unsigned char   opt_data[DN_MAXOPTL]; // Optional data
};

```

### DECnet Outgoing Access Control

```

struct accessdata_dn {
    unsigned short  acc_acc1;         // account length
    unsigned char   acc_acc[DN_MAXACCL]; // account string
    unsigned short  acc_pass1;       // password length
    unsigned char   acc_pass[DN_MAXACCL]; // password string
    unsigned short  acc_user1;       // user length
    unsigned char   acc_user[DN_MAXACCL]; // user string
};

```

### DECnet Incoming Access Control

```

// DECnet incoming access control information
struct dnet_accent
{

```

(continued)



(continued)

```

unsigned char dac_status;
unsigned char dac_type; // DN_NONE, etc.
char dac_username[DN_MAXACCL];
char dac_password[DN_MAXACCL];
};

```

### DECnet Call Data

```

struct calldata_dn {
struct optdata_dn {
    unsigned short opt_status;
    unsigned short opt_opt1;
    unsigned char opt_data[DN_MAXOPTL];
};
struct accessdata_dn {
    unsigned short acc_acc1;
    unsigned char acc_acc[DN_MAXACCL];
    unsigned short acc_pass1;
    unsigned char acc_pass[DN_MAXACCL];
    unsigned short acc_user1;
    unsigned char acc_user[DN_MAXACCL];
}
};

```

### DECnet Logical Link

```

struct linkinfo_dn {
    unsigned short idn_segsize;
    unsigned char idn_linkstate;
};

```

## DECnet Function Details

This section presents connection information for DECnet functions.

### Connections Using Accept/WSAAccept/WSPAccept

This section describes various data connections for **accept** functions.

#### Immediate Accept with No Optional or Access Data

In order to accept a connection on a DECnet socket, the parameter *addr* should point to a **SOCKADDR\_DN** structure.

```

#include <ws2dnet.h>

SOCKET WINAPI accept (
    SOCKET s,
    struct sockaddr_dn FAR * addr,
    int FAR * addrlen
);

```

```

SOCKET WINAPI WSAAccept (
    SOCKET s,
    struct sockaddr_dn FAR * addr,
    int FAR * addrlen,
    LPCONDITIONPROC lpfnCondition,
    DWORD dwCallbackData
);

SOCKET WINAPI WSPAccept(
    SOCKET s,
    struct sockaddr_dn FAR * addr,
    int FAR * addrlen,
    LPCONDITIONPROC lpfnCondition,
    DWORD dwCallbackData,    int FAR * lpErrno
);

```

**addr**

Pointer to a **SOCKADDR\_DN** structure that receives the address of the connecting entity, as known to the communications layer.

**Deferred Accept with Optional and Access Data**

DECnet sockets support both immediate and deferred accepts. It also supports the exchange of up to 16 bytes of optional data on **accept** and **connect**. It also supports the receipt of DECnet access control information by the server from the client on a connect request.

```

#include <ws2dnet.h>

SOCKET WINAPI WSAAccept (
    SOCKET s,
    struct sockaddr_dn FAR * addr,
    int FAR * addrlen,
    LPCONDITIONPROC lpfnCondition,
    DWORD dwCallbackData
);

SOCKET WINAPI WSPAccept(
    SOCKET s,
    struct sockaddr_dn FAR * addr,
    int FAR * addrlen,
    LPCONDITIONPROC lpfnCondition,
    DWORD dwCallbackData,
    int FAR * lpErrno
);

```

DECnet optional data is passed in an **OPTDATA\_DN** structure. Access control data is passed in an **ACCESSDATA\_DN** structure.

In the **CALLBACK** function, the *lpCallerData* should point to a **CALLDATA\_DN** structure that contains concatenated **OPTDATA\_DN** and **ACCESSDATA\_DN** structures. If *lpCallerData* is set to **NULL**, no additional data will be read from the caller. If either the **OPTDATA\_DN.OPT\_OPTL** or the **ACCESSDATA\_DN.ACC\_USERL** are set to zero, that portion of the structure will be ignored.

The **accept** only reads **ACCESSDATA\_DN**, it does not write it, so only **OPTDATA\_DN** can be returned by the server. The *lpCalleeData* pointer should point to a buffer containing the **OPTDATA\_DN** structure. If *lpCalleeData* is set to **NULL**, no optional data can be read from the server.

## Structure Information for Bind/WSPBind

In order to bind a local DECnet address to a socket, the parameter *name* should point to a **SOCKADDR\_DN** structure.

```
#include <ws2dnet.h>

int WINAPI bind (SOCKET s, const struct sockaddr_dn FAR * name, int namelen);

int WINAPI WSPBind ( SOCKET s, const struct sockaddr_dn FAR * name, int namelen,
int FAR * lpErrno );
```

*name*

Pointer to a **SOCKADDR\_DN** structure that contains the DECnet address to be bound to this socket.

## Connections Using Connect/WSAConnect/WSPConnect

This section provides connection information using **Connect** functions.

### Connect with no Optional or Access Data

In order to establish a connection to a DECnet peer, the parameter *name* should point to a **SOCKADDR\_DN** structure.

```
#include <ws2dnet.h>

int WINAPI connect (
    SOCKET s,
    const struct sockaddr_dn FAR * name,
    int namelen
);

int WINAPI WSAConnect (
    SOCKET s,
```



```

const struct sockaddr_dn FAR * name,
int namelen,
LPWSABUF lpCallerData,
LPWSABUF lpCalleeData,
GROUP g,
LPQOS lpSQOS,
LPQOS lpGQOS
);

int WSPAPI WSPConnect(
SOCKET s,
const struct sockaddr_dn FAR * name,
int namelen,
LPWSABUF lpCallerData,
LPWSABUF lpCalleeData,
LPQOS lpSQOS,
LPQOS lpGQOS,
int FAR * lpErrno
);

```

***name***

Pointer to a **SOCKADDR\_DN** structure that contains the DECnet address and object to which the socket is to be connected.

**Optional and Access Data on Connect**

DECnet sockets support the exchange of up to 16 bytes of optional data on a **WSAConnect**. It also supports the sending of DECnet access control information from the client to the server on a connect request.

```

#include <ws2dnet.h>

int WSAAPI WSAConnect ( SOCKET s, const struct sockaddr_dn FAR * name,
int namelen, LPWSABUF lpCallerData, LPWSABUF lpCalleeData,
GROUP g, LPQOS lpSQOS, LPQOS lpGQOS );

int WSPAPI WSPConnect( SOCKET s, const struct sockaddr_dn FAR * name, int
namelen, LPWSABUF lpCallerData, LPWSABUF lpCalleeData, LPQOS lpSQOS, LPQOS lpGQOS,
int FAR * lpErrno );

```

Parameter	Description
<i>lpCallerData</i>	Pointer to the user data that is to be transferred from the client to the server during connection establishment.
<i>lpCalleeData</i>	Pointer to the optional data that is to be transferred back from the server during connection establishment.

DECnet optional data is passed in an **OPTDATA\_DN** structure. Access control data is passed in an **ACCESSDATA\_DN** structure.

The *IpCallerData* should point to a **CALLDATA\_DN** structure that contains concatenated **OPTDATA\_DN** and **ACCESSDATA\_DN** structures. If *IpCallerData* is set to NULL, no additional data will be sent to the server. If either the *opdata\_dn.opt\_optl* or the *accessdata\_dn.acc\_userl* are set to zero, that portion of the structure should be ignored.

If *IpCalleeData* is set to NULL, no optional data will be read from the server. If either the *opdata\_dn.opt\_optl* or the *accessdata\_dn.acc\_userl* are set to zero, that structure should be ignored. The **accept** function only returns optional data, not access data. So the *IpCalleeData* pointer should point to a buffer containing the **OPTDATA\_DN** structure. If *IpCalleeData* is set to NULL no optional data will be read from the server.

## Addressing with GetPeerName/WSPGetPeerName

To get the address of the DECnet peer to which a socket is connected, the parameter *name* should point to a **SOCKADDR\_DN** structure.

```
#include <ws2dnet.h>

int WINAPI getpeername (
    SOCKET s,
    struct sockaddr_dn FAR * name,
    int FAR * namelen
);

int WSPAPI WSPGetPeerName(
    SOCKET s,
    struct sockaddr_dn FAR * name,
    int FAR * namelen,
    int FAR * lpErrno
);
```

*name*

Pointer to a **SOCKADDR\_DN** structure that will return the DECnet address of the DECnet peer to which a socket is connected.

## Receiving Local Name with getsockname/WSPGetSockName

To get the local name for a DECnet socket, the parameter *name* should point to a **SOCKADDR\_DN** structure.

```
#include <ws2dnet.h>

int WINAPI getsockname (
    SOCKET s,
    struct sockaddr_dn FAR * name,
```

```

    int FAR * nameLen
);

int WSPAPI WSPGetSockName(
    SOCKET s,
    struct sockaddr_dn FAR * name,
    int FAR * nameLen,
    int FAR * lpErrno
);

```

*name*

Pointer to a **SOCKADDR\_DN** structure that will return the local name to which a DECnet socket is connected.

## Using Getsockopt/WSPGetSockOpt

The **SO\_LINKINFO** socket option returns a **LINKINFO\_DN** structure containing the current state of the specified DECnet logical link.

```

int WSPAPI WSPGetSockOpt(
    SOCKET s,
    int level,
    int optname,
    char FAR * optval,
    int far * optlen,
    int FAR * lpErrno
);

```

*s*

Descriptor identifying socket

*level*

DNPROTO\_NSP

*optname*

**SO\_LINKINFO**

*optval*

FAR \* **LINKINFO\_DN** structure

*optlen*

Sizeof(**LINKINFO\_DN**)

*lpErrno*

Pointer to error code

## Return Values

If no error occurs, **WSPGetSockOpt** returns zero and the **LINKINFO\_DN** structure pointed to by *optval* contains the current transport segment size and logical link state. See the **LL\_\*** manifest constants for valid link states. Otherwise, a value of **SOCKET\_ERROR** is returned, and a specific error code is available in *lpErrno*.

## Using Socket/WSASocket/WSPSocket

To create a socket that is bound to a DECnet service provider, the following values should be passed to the socket call.

```
#include <ws2dnet.h>

SOCKET WINAPI socket ( int af, int type, int protocol );

SOCKET WINAPI WSASocket(int af, int type, int protocol, ...)

SOCKET WSPAPI WSPSocket(int af, int type, int protocol, ...)
```

*af* = **AF\_DECnet**

DECnet address family.

*type* = **SOCK\_SEQPACKET**

Sequenced packets (message-oriented).

*protocol* = **DNPROTO\_NSP**

DECnet NSP transport protocol.

## DECnet Out-of-Band Data

DECnet supports the sending and receiving of OOB data through the **MSG\_OOB** flag for **recv** and **send**. If a **send** with the **MSG\_OOB** flag is sent, the **recv** must be posted with **MSG\_OOB** in order to read the data. To check for the presence of OOB data, use **select** with **exceptfds** set, or a **WSAASYNCSELECT** with **FD\_OOB** set.

DECnet does not support inline OOB data through the **setsockopt** **SO\_OOBINLINE** flag.

## DECnet-Specific Extended Functions Identifiers

This section presents the DECnet-specific extensions to the Windows Sockets 2 specification.

The Windows Sockets 2 function extension mechanism works as follows: a call is made to **WSAIoctl** with the control code **SIO\_GET\_EXTENSION\_FUNCTION\_POINTER** that returns a function pointer to a specified extension function.

The following DECnet-specific extension identifiers have been allocated by the WS2 identifier clearinghouse.

---

**Note** We reserve 32 in each block.

---

```
//
// DECnet
//
```

```
#define WS2API_DECNET_dnet_addr          1
#define WS2API_DECNET_dnet_eof          2
#define WS2API_DECNET_dnet_getacc       3
#define WS2API_DECNET_dnet_getalias     4
#define WS2API_DECNET_dnet_htoa         5
#define WS2API_DECNET_dnet_ntoa         6
#define WS2API_DECNET_getnodeadd        7
#define WS2API_DECNET_getnodebyaddr     8
#define WS2API_DECNET_getnodebyname     9
#define WS2API_DECNET_getnodename      10
//
// Next ones start at 33
//
```

## dnet\_addr

The **dnet\_addr** identifier converts an ASCIZ DECnet Phase IV node address string to a binary address.

```
#include <ws2dnet.h>

struct dn_naddr FAR * WSAAPI dnet_addr(const char FAR *cp);
```

*cp*

Specifies the address of a character string that contains a DECnet node address in the form *a.n* (area.node). For example, 9.440 is a DECnet Phase IV node number.

## Return Values

**dnet\_addr** returns a pointer to the **DN\_NADDR** structure

If successful, it returns a pointer to a **DN\_NADDR** structure that contains a binary DECnet address. Applications should copy this data before issuing another **dnet\_addr** call. Otherwise, it returns a NULL pointer.

## dnet\_eof

The **dnet\_eof** identifier tests a DECnet socket for an end-of-file condition.

```
int WSAAPI dnet_eof(SOCKET s);
```

*s*

Specifies DECnet socket to test.

## Return Values

If the connection is still active (either in a running or connected state), it returns zero. If the connection is inactive, a one is returned.



## dnet\_getacc

The **dnet\_getacc** identifier retrieves local access control for incoming DECnet connections based on the specified user name.

```
#include <ws2dnet.h>
```

```
struct dnet_accnt FAR * WINAPI dnet_getacc(struct dnet_accnt FAR *nacc )
```

*nacc*

Pointer to the incoming access control record. *nacc.dac\_username* contains the user name. This ASCIZ character string has a maximum length of DN\_MAXACCL.

### Return Values

If successful, it returns a pointer to a **DNET\_ACCENT** structure. Applications should copy this data before issuing another **dnet\_getacc** call. Otherwise, it returns a NULL pointer.

## dnet\_getalias

The **dnet\_getalias** identifier returns any default access control information associated with a specific node.

```
char FAR * WINAPI dnet_getalias(const char FAR *node)
```

The following ASCIZ strings are all valid returns:

“node”

“node/username”

“node/username/password”

“node/username/password/account”

“node/username//account”

*node*

Pointer to the node name for which **dnet\_getalias** should search.

### Return Values

The call returns the address of a static buffer that contains the information from the local DECnet node database. Applications should copy this data before issuing another **dnet\_getalias** call.

## dnet\_htoa

The **dnet\_htoa** identifier searches the local node database. If it finds the node name for the specified node address, it returns a pointer to an ASCIZ DECnet name string. Otherwise, it returns an ASCIZ node address string.

```
#include <ws2dnet.h>
```

```
char FAR *WSAAPI dnet_htoa(const struct dn_naddr FAR *add);
```

*add*

Specifies the address of a **DN\_NADDR** structure that contains the node number for which to search.

### Return Values

The function returns the address of a static buffer that contains the node string. This string must be copied before **dnnet\_htoa** is called again.

### dnnet\_ntoa

The **dnnet\_ntoa** identifier converts a DECnet node address from binary to ASCII format. It converts the pointer to a **DN\_NADDR** to a string in the form 9.123, for example.

```
#include <ws2dnet.h>
```

```
char FAR *WSAAPI dnet_ntoa(const struct dn_naddr FAR *add);
```

*add*

Specifies the address of a **DN\_NADDR** structure that contains the node number to convert.

### Return Values

The function returns a pointer to a static string that contains the node address. This string must be copied before **dnnet\_ntoa** is called again.

### getnodeadd

The **getnodeadd** identifier returns the local node's DECnet address.

```
#include <ws2dnet.h>
```

```
struct dn_naddr FAR *WSAAPI getnodeadd(void);
```

### Return Values

The function returns the address of a static **DN\_NADDR** structure containing the local node's DECnet address. Applications should copy this data before issuing another **getnodeadd** call. If DECnet is not installed, it returns a NULL pointer

### getnodebyaddr

The **getnodebyaddr** identifier searches for a DECnet node name that matches a specified DECnet address. The address is a 16-bit binary DECnet address, where the

high 6 bits are the DECnet area number and the low 10 bits are the DECnet node number. DECnet Phase IV node names consist of one to six alphanumeric characters with at least one alphabetic character.

```
#include <ws2dnet.h>

struct nodeent_f FAR *WSAAPI getnodebyaddr(const unsigned char FAR *addr, int
len, int type);
```

*addr*

Pointer to the address for which **getnodebyaddr** should search.

*len*

Length in bytes of the requested address (range DN\_ADDL to DN\_MAXADDL).

*type*

Address family of the address (AF\_DECnet).

### Return Values

The function returns a pointer to a static **NODEENT\_F** structure. The application must copy this data before issuing another **getnodebyaddr** call. If the end of file is reached, a NULL pointer is returned.

### getnodebyname

The **getnodebyname** identifier searches for a DECnet node address that matches the specified DECnet node name. DECnet Phase IV node names consist of one to six alphanumeric characters with at least one alphabetic character.

```
#include <ws2dnet.h>

struct nodeent_f FAR *WSAAPI getnodebyname(const char FAR *);
```

*name*

Pointer to an ASCII node name for which to search.

### Return Values

The function returns a pointer to a static **NODEENT\_F** structure. The application must copy this data before issuing another **getnodebyaddr** call. If the end of file is reached, a NULL pointer is returned.

### getnodename

The **getnodename** identifier returns the name of the local DECnet node. DECnet Phase IV node names consist of one to six alphanumeric characters with at least one alphabetic character.

```
char FAR *WSAAPI getnodename(void);
```

## Return Values

The function returns the address of a static string that contains the local node's ASCII DECnet name, or a NULL pointer if DECnet is not installed. The maximum size for a returned string is DN\_MAXNODEL. The application must copy this data before issuing another **getnodename** call.

## DECnet Header File

Use Ws2dnet.h.

## Open Systems Interconnection (OSI)

This section presents OSI-transport information in the following topics.

### OSI Introduction

This section covers extensions to Windows Sockets 2 that are specific to OSI transports. It also describes aspects of base Windows Sockets 2 functions that require special consideration or that may exhibit unique behavior when used with OSI transports.

#### Fast Facts

<b>Protocol name(s)</b>	TP4/CLNS, TP4/NULL, and so forth.
<b>Description</b>	Provides OSI transport services over OSI networking layer: CLTS for unreliable datagrams and COTS for reliable connection-oriented message streams.
<b>Address family</b>	AF_OSI
<b>Header file</b>	Ws2osi.h

## International Organization for Standardization (IOS)

The International Organization for Standardization (IOS) produces a set of standards to facilitate interconnection of computer systems. The Open Systems Interconnection (OSI) Reference Model subdivides the set into a series of layers. Winsock allows an application to use OSI Transport protocols.

Examples of OSI profiles that may be implemented are shown in the following table.

<b>OSI profile</b>	<b>Description</b>
TP4/CLNS	Transport class four over Connectionless-mode Network Service
TP4/NULL	Transport class four over Null Network Service
TP0/CONS	Transport class zero over Connection-mode Network Service
NULL/CONS	Null Transport over Connection-mode Network Service
TP0/TCP	Transport class zero over TCP/IP (RFC1006)

The OSI profile is designated either by the *protocol* parameter or the *IpProtocolInfo* parameter to the **WSASocket** API. The TP4/NULL profile is a subset of the TP4/CLNS profile and is selected by the addressing information. The default OSI profile is TP4/CLNS.

An application must use the **WSAPROTOCOL\_INFO** structure returned by **WSAEnumProtocols** to discover the services provided by a particular OSI profile. For example, not all OSI profiles support connect and disconnect data.

## OSI Expedited Data

OSI protocols may support expedited data. Expedited data is not subject to the flow control procedures for normal data and may overtake normal data.

## ISO Qualified Data

The ISO 8208/X.25 protocol supports qualified data. Qualified data is sent by using the **WSAIoctl** function to mark the following data as qualified, and then using the normal Winsock **send** functions. Qualified data is received by using the **WSAIoctl** function to wait for notification of qualified data, and then using the normal Winsock **recv** functions.

## ISO Reset

The ISO 8208/X.25 protocol supports resets. A reset may be generated by using the **WSAIoctl** function. Two bytes of reset data are permitted. These are interpreted as reset cause and reset diagnostic. For further details of this data see the X.25 specification.

## OSI Quality of Service

The provider-specific parameters in the flow specification are encoded in a Type/Length/Data format to enable multiple parameters to be provided.

For example ISO 8208/X.25 call facilities may be specified by using the type `OSI_PARAM_ID_X25_CALL_FACILITY`.

## Option Profiles

OSI protocols typically have a large number of options. These options may include:

- Preferred class
- Alternative classes
- Timers
- Checksums
- Lifetime
- Others

An option profile name may be used to select a set of options available in a particular vendor's OSI-protocol implementation. The Windows Sockets specification does not define the options that may be selected.

Option profile names are specified in the OSI address structure. If an option profile name is not specified, the OSI protocol uses its default option profile.

## Address Format

Two address formats for the OSI protocol are defined in this section. The SOCKADDR\_TP format is retained for compatibility with Windows Sockets 1.1-OSI protocols. Windows Sockets 2 OSI protocols support the SOCKADDR\_OSITP format. It contains:

- A Transport Selector (TSEL).
- Network Service Access Point (NSAP).
- Sub Network address (SNPA).
- Extended addressing information.
- The option profile name.

The SNPA follows the rules of the Sub Network in use. For example, an SNPA would contain a 6-byte MAC address followed by a 1-byte Link Service Access Point (LSAP) giving a 7-byte SNPA.

Some X.25 protocols use connect user data to select the listening socket. The extended addressing information contains connect user data. This field must not be specified for other OSI protocols.

The address structure also contains the option profile name. If the name length is zero, the default option profile is used.

The TP4/NULL OSI-protocol profile is selected by using the TP4/CLNS protocol, selecting a zero-length NSAP, and specifying the MAC address and LSAP in the Sub Network address field.

## OSI Data Structures

The following structure is used to encode the provider-specific parameters in the flow specification:

```
typedef struct _osispecflowparam
{
    u_short paramID;           // Parameter ID.
    u_short paramLength;      // Length of following data.
    u_char param;             // Parameter data.
} OSISPECFLOWPARAM;
```

## OSI Controls

This section describes OSI control features using `ioctl`s and socket options.

## Ioctl's

The following commands are available.

Command	Semantics
SIO_OSI_X25_GET_RESET_DATA	Waits for an ISO 8208/X.25 reset to occur, and then returns the reset data associated with the reset. No input buffer is required. The 2 bytes of reset data are copied to the output buffer. The WSAENOPROTOOPT error code is indicated for service providers that do not support X.25 resets.
SIO_OSI_X25_GENERATE_RESET	Generates an ISO 8208/X.25 reset. No output buffer is required, the 2 bytes of reset data are obtained from the input buffer. The WSAENOPROTOOPT error code is indicated for service providers that do not support X.25 resets.
SIO_OSI_X25_SEND_QUALIFIED	Indicates that the next message sent using one of the send functions, or <b>WSASend</b> , will be sent as qualified data. No buffers are required. It is not necessary to call this function for subsequent sends of the same message where the MSG_PARTIAL flag has been used. The WSAENOPROTOOPT error code is indicated for service providers that do not support X.25-qualified data.
SIO_OSI_X25_GET_QUALIFIED	Waits until the next message that is received by the <b>recv</b> or <b>WSARecv</b> functions is ISO 8208/X.25-qualified data. This command only completes once for each message received, even if the MSG_PARTIAL flag is returned. The WSAENOPROTOOPT error code is indicated for service providers that do not support X.25-qualified data.

## Socket Options

The following socket options are supported for **setsockopt** and **getsockopt**. The Type identifies the type of data addressed by *optval*.

level = SOL\_SOCKET

Value	Type	Meaning
SO_EXPEDITED	BOOL	Negotiates expedited data.
SO_X25_CONFIRM_DELIVERY	BOOL	ISO 8208/X.25 delivery confirmation.

### SO\_EXPEDITED

This option is negotiated during connection establishment. The **setsockopt** function must be used before the connection is established to specify the proposed option. The **getsockopt** function may be used after the circuit has been established to retrieve the final negotiated option. See ISO 8073 for further details.

**SO\_X25\_CONFIRM\_DELIVERY**

This option controls the state of the Delivery Confirmation bit (D-bit) for X.25 protocols. If the D-bit is set, end-to-end confirmation of data occurs. The `SO_X25_CONFIRM_DELIVERY` option may be used to change the state of the D-bit many times during the life of a connection.

## OSI Function Specifics

This section describes provider-specific parameters for Quality of Service and lists the OSI header file.

### Quality of Service

The provider-specific parameters in the flow specification are encoded in a Type/Length/Data format to enable multiple parameters to be provided. The `OSISPECFLOWPARAM` structure is used to encode the parameter.

The following parameter type is defined.

Parameter	Type	Meaning
<code>OSI_PARAM_ID_X25_CALL_FACILITY</code>	<code>u_char[]</code>	ISO 8208/X.25 call facilities.

**OSI\_PARAM\_ID\_X25\_CALL\_FACILITY**

This parameter specifies the ISO 8208/X.25 call facilities to be used. The format of the data is the same as is coded in an X.25 call request/call confirm packet. For further details see the X.25 specification.

## OSI Header File

Use `ws2osi.h`.

## ATM-Specific Extensions

This section presents information describing the Asynchronous Transfer Mode (ATM) specific extensions.

## ATM Introduction

This section describes the Asynchronous Transfer Mode (ATM)-specific extensions needed to support the native ATM services as exposed and specified in the ATM Forum User Network Interface (UNI) specification version 3.x (3.0 and 3.1). This document supports AAL type 5 (message mode) and user-defined AAL. Future versions of this document will support other types of AAL as well as UNI 4.0 after it's finalized.



## Fast Facts

<b>Protocol name(s)</b>	ATMPROTO_AAL5, ATMPROTO_AALUSER
<b>Description</b>	ATM AAL5 provides a transport service that is connection-oriented, message-boundary preserved, and QOS guaranteed. ATMPROTO_AALUSER is a user-defined AAL.
<b>Address family</b>	AF_ATM
<b>Header file</b>	Ws2atm.h

## ATM Overview

ATM is an emerging high-speed networking technology that is applicable to both LAN and WAN arenas. An ATM network simultaneously transports a wide variety of network traffic—voice, data, image, and video. It provides users with a guaranteed quality of service on a per-virtual channel (VC) basis.

## ATM Data Structures

A new address family, AF\_ATM, is introduced for native ATM services, and the corresponding **SOCKADDR** structure, **sockaddr\_atm**, is defined in the following. To open a socket for native ATM services, parameters in **socket** should contain AF\_ATM, **SOCK\_RAW**, and ATMPROTO\_AAL5 or ATMPROTO\_AALUSER, respectively.

```
struct sockaddr_atm {
    u_short satm_family:      // address family should be
                             // AF_ATM
    ATM_ADDRESS satm_number; // ATM address
    ATM_BLLI satm_blli;      // B-LLI
    ATM_BHLI satm_bhli;      // B-HLI
};
```

### Members

#### satm\_family

Identifies the address family, which is AF\_ATM in this case.

#### satm\_number

Identifies the ATM address that could be either in E.164 or NSAP-style ATM End Systems Address format. See *Using the ATM\_ADDRESS Structure* for more details about the **ATM\_ADDRESS** structure. This field will be mapped to the Called Party Number IE (Information Element) if it is specified in **bind** and **WSPBind** for a listening socket, or in **connect**, **WSAConnect**, **WSPConnect**, **WSAJoinLeaf**, or **WSPJoinLeaf** for a connecting socket. It will be mapped to the Calling Party Number IE if specified in **bind** and **WSPBind** for a connecting socket.

#### satm\_blli

Identifies the fields in the B-LLI Information Element that are used along with **satm\_bhli** to identify an application. See the *Using the ATM\_ADDRESS Structure* section for more details about the **ATM\_BLLI** structure. Note that the B-LLI layer two

information is treated as not present if its *Layer2Protocol* field contains `SAP_FIELD_ABSENT`, or as a wildcard if it contains `SAP_FIELD_ANY`. Similarly, the B-LLI layer three information is treated as not present if its *Layer3Protocol* field contains `SAP_FIELD_ABSENT`, or as a wildcard if it contains `SAP_FIELD_ANY`.

### **satm\_bhli**

Identifies the fields in the B-HLI Information Element that are used along with **satm\_blli** to identify an application. See *Using the ATM\_ADDRESS Structure* for more details about the **ATM\_BHLI** structure.

---

**Note** **Satm\_bhli** is treated as not present if its *HighLayerInfoType* field contains `SAP_FIELD_ABSENT`, or as a wildcard if it contains `SAP_FIELD_ANY`.

---

For listening sockets, the **SOCKADDR\_ATM** structure is used in **bind/WSPBind** to register a Service Access Point (SAP) to receive incoming connection requests destined to this SAP. SAP registration is used to match against the SAP specified in an incoming connection request in order to determine which listening socket is to receive this request. In the current version of this specification, overlapping registration is not allowed. Overlapping registration is defined as having more than one registered SAP to potentially match the SAP specified in any incoming connection request. **Listen** and **WSPListen** will return the error code `WSAEADDRINUSE` if the SAP associated with the listening socket overlaps with any currently registered SAPs in the system.

The fields in a SAP to be registered must contain either a valid value, or one of two special manifest constants: `SAP_FIELD_ABSENT` or `SAP_FIELD_ANY`.

`SAP_FIELD_ABSENT` simply means that this field is not presented as part of a SAP. `SAP_FIELD_ANY` means using wildcards.

Note that the requirement of nonoverlapping registration does not preclude using wildcards. For example, it is possible to have two registered SAPs that both contain `SAP_FIELD_ANY` in some fields and different values in other fields.

---

**Note** The called party ATM number is mandatory, thus the **satm\_number** field cannot contain `SAP_FIELD_ABSENT`.

---

For connecting sockets, the **SOCKADDR\_ATM** structure is used to specify the destination SAP in **connect/WSAConnect/WSPConnect** for point-to-point connections, and **WSAJoinLeaf/WSPJoinLeaf** for point-to-multipoint connections. The fields in the destination SAP of a connecting socket must contain either a valid value or `SAP_FIELD_ABSENT`, that is, `SAP_FIELD_ANY` is not allowed.

Furthermore, `SAP_FIELD_ABSENT` is not allowed for the **satm\_number** field. The destination SAP is used to match against all the registered SAPs in the destination machine to determine the forwarding destination for this connection request. If each and every field of the destination SAP of an incoming request either equals the

corresponding field of a registered SAP, or the corresponding field contains the `SAP_FIELD_ANY`, the listening socket associated with this registered SAP will receive the incoming connection request.

If `bind` and/or `WSPBind` are used on a connecting socket to specify the calling party ATM address, the `satm_blli` and `satm_bhli` fields should be ignored and the ones specified in `connect`, `WSAConnect`, or `WSPConnect` will be used.

## Using the ATM\_ADDRESS Structure

```

/*
 * values used for AddressType in struct ATM_ADDRESS
 */
#define ATM_E164                0x01    // E.164 addressing scheme
#define ATM_NSAP                0x02    // NSAP-style ATM Endsystem
                                        // Address scheme
#define ATM_AESA                0x02    // NSAP-style ATM Endsystem
                                        // Address scheme

typedef struct {
    DWORD AddressType;           // E.164 or NSAP-style ATM Endsystem
                                // Address
    DWORD NumofDigits;          // number of digits;
    UCHAR Addr[20];             // IA5 digits for E164, BCD encoding
                                // for NSAP
                                // format as defined in the ATM Forum
                                // UNI 3.1
} ATM_ADDRESS;

```

For `ATM_E164`, enter the numbered digits in the same order in which they would be entered on a numeric keypad; that is, the number digit that would be entered first is located in `addr`. Digits are coded in IA5 characters. Bit 8 is set to zero.

For `ATM_NSAP`, code the address using Binary Coded Decimal (BCD) as defined in the ATM Forum UNI 3.1. The `NumofDigits` field are ignored in this case, and the NSAP-style address always contains 20 bytes.

A value of `SAP_FIELD_ANY` in `AddressType` indicates that the `satm_number` field is a wildcard. There are two more specialized wildcard values:

`SAP_FIELD_ANY_AESA_SEL` and `SAP_FIELD_ANY_AESA_REST`.

`SAP_FIELD_ANY_AESA_SEL` means that this is an NSAP-style ATM Endsystem Address and the selector octet is set as a wildcard. `SAP_FIELD_ANY_AESA_REST` means that this is an NSAP-style ATM Endsystem Address and all the octets except for the selector octet are set as wildcards.

## ATM\_BLLI Structure and Associated Manifest Constants

```

/*
 * values used for Layer2Protocol in struct B-LLI
 */
#define BLLI_L2_ISO_1745      0x01 // Basic mode ISO 1745
#define BLLI_L2_Q921         0x02 // CCITT Rec. Q.921
#define BLLI_L2_X25L         0x06 // CCITT Rec. X.25, link layer
#define BLLI_L2_X25M         0x07 // CCITT Rec. X.25, multilink
#define BLLI_L2_ELAPB        0x08 // Extended LAPB; for half
                                // duplex operation
#define BLLI_L2_HDLC_NRM     0x09 // HDLC NRM (ISO 4335)
#define BLLI_L2_HDLC_ABM     0x0A // HDLC ABM (ISO 4335)
#define BLLI_L2_HDLC_ARM     0x0B // HDLC ARM (ISO 4335)
#define BLLI_L2_LLC          0x0C // LAN logical link control
                                // (ISO 8802/2)
#define BLLI_L2_X75          0x0D // CCITT Rec. X.75, single
                                // link procedure
#define BLLI_L2_Q922         0x0E // CCITT Rec. Q.922
#define BLLI_L2_USER_SPECIFIED 0x10 // User Specified
#define BLLI_L2_ISO_7776     0x11 // ISO 7776 DTE-DTE operation

/*
 * values used for Layer3Protocol in struct B-LLI
 */
#define BLLI_L3_X25          0x06 // CCITT Rec. X.25, packet
                                // layer
#define BLLI_L3_ISO_8208     0x07 // ISO/IEC 8208 (X.25 packet
                                // layer for DTE
#define BLLI_L3_X223         0x08 // X.223/ISO 8878
#define BLLI_L3_SIO_8473     0x09 // ISO/IEC 8473 (OSI
                                // connectionless)
#define BLLI_L3_T70          0x0A // CCITT Rec. T.70 min.
                                // network layer
#define BLLI_L3_ISO_TR9577   0x0B // ISO/IEC TR 9577 Network
                                // Layer Protocol ID
#define BLLI_L3_USER_SPECIFIED 0x10 // User Specified

/*
 * values used for Layer3IPI in struct B-LLI
 */
#define BLLI_L3_IPI_SNAP     0x80 // IEEE 802.1 SNAP identifier
#define BLLI_L3_IPI_IP       0xCC // Internet Protocol (IP)
                                // identifier

```

(continued)

(continued)

```
typedef struct {
    DWORD Layer2Protocol;           // User information layer 2
                                    // protocol
    DWORD Layer2UserSpecifiedProtocol; // User specified layer 2
                                    // protocol information
    DWORD Layer3Protocol;           // User information layer 3
                                    // protocol
    DWORD Layer3UserSpecifiedProtocol; // User specified layer 3
                                    // protocol information
    DWORD Layer3IPI;                // ISO/IEC TR 9577 Initial
                                    // Protocol Identifier
    UCHAR SnapID[5];                // SNAP ID consisting of OUI
                                    // and PID
} ATM_BLLI;
```

## Parameters

### *Layer2Protocol*

Identifies the layer-two protocol. Corresponds to the *User information layer 2 protocol* field in the B-LLI information element. A value of `SAP_FIELD_ABSENT` indicates that this field is not used, and a value of `SAP_FIELD_ANY` means wildcard.

### *Layer2UserSpecifiedProtocol*

Identifies the user-specified layer-two protocol. Only used if the *Layer2Protocol* parameter is set to `BLLI_L2_USER_SPECIFIED`. The valid values range from zero–127. Corresponds to the *User specified layer 2 protocol information* field in the B-LLI information element.

### *Layer3Protocol*

Identifies the layer-three protocol. Corresponds to the *User information layer 3 protocol* field in the B-LLI information element. A value of `SAP_FIELD_ABSENT` indicates that this field is not used, and a value of `SAP_FIELD_ANY` means wildcard.

### *Layer3UserSpecifiedProtocol*

Identifies the user-specified layer-three protocol. Only used if the *Layer3Protocol* parameter is set to `BLLI_L3_USER_SPECIFIED`. The valid values range from zero–127. Corresponds to the *User specified layer 3 protocol information* field in the B-LLI information element.

### *Layer3IPI*

Identifies the layer-three Initial Protocol Identifier. Only used if the *Layer3Protocol* parameter is set to `BLLI_L3_ISO_TR9577`. Corresponds to the *ISO/IEC TR 9577 Initial Protocol Identifier* field in the B-LLI information element.

### *SnapID*

Identifies the 802.1 SNAP identifier. Only used if the *Layer3Protocol* parameter is set to `BLLI_L3_ISO_TR9577` and *Layer3IPI* is set to `BLLI_L3_IPI_SNAP`, indicating an IEEE 802.1 SNAP identifier. Corresponds to the *OUI* and *PID* fields in the B-LLI information element.



## ATM\_BHLI Structure and Associated Manifest Constants

```

/*
 * values used for the HighLayerInfoType field in struct ATM_BHLI
 */
#define BHLI_ISO                0x00    // ISO
#define BHLI_UserSpecific       0x01    // User Specific
#define BHLI_HighLayerProfile   0x02    // High layer profile (only in
                                        // UNI3.0)
#define BHLI_VendorSpecificAppId 0x03    // Vendor-Specific
                                        // Application ID

typedef struct {
    DWORD HighLayerInfoType;           // High Layer Information Type
    DWORD HighLayerInfoLength;        // number of bytes in HighLayerInfo
    UCHAR HighLayerInfo[8];           // the value dependent on the
                                        // HighLayerInfoType field
} ATM_BHLI;

```

### Parameters

#### *HighLayerInfoType*

Identifies the *high layer information type* field in the B-LLI information element. Note that the type `BHLI_HighLayerProfile` has been eliminated in UNI 3.1. A value of `SAP_FIELD_ABSENT` indicates that B-HLI is not present, and a value of `SAP_FIELD_ANY` means wildcard.

#### *HighLayerInfoLength*

Identifies the number of bytes from one to eight in the *HighLayerInfo* array. Valid values include eight for the cases of `BHLI_ISO` and `BHLI_UserSpecific`, four for `BHLI_HighLayerProfile`, and seven for `BHLI_VendorSpecificAppId`.

#### *HighLayerInfo*

Identifies the *high layer information* field in the B-LLI information element. In the case of *HighLayerInfoType* being `BHLI_VendorSpecificAppId`, the first 3 bytes consist of a globally-administered Organizationally Unique Identifier (OUI) (as per IEEE standard 802-1990), followed by a 4-byte application identifier, which is administered by the vendor identified by the OUI. Value for the case of `BHLI_UserSpecific` is user defined and requires bilateral agreement between two end users.

## ATM Controls

ATM point-to-point and point-to-multipoint connection setup and teardown are natively supported by the Windows Sockets 2 specification. In fact, Windows Sockets 2 QOS specification and protocol-independent multipoint/multicast mechanisms were designed with ATM in mind, along with other protocols. See section 2.7 and appendix D of the Windows Sockets 2 API specification for Windows Sockets 2 QOS and multipoint support, respectively. Therefore, no ATM-specific ioctls need to be introduced in this document.

## ATM Function Specifics

Based on the taxonomy defined for Windows Sockets 2 protocol-independent multipoint/multicast schemes, ATM falls into the category of rooted data and rooted control planes. (See the Windows Sockets 2 API specification, Appendix D for more information.) An application acting as the root would create `c_root` sockets, and counterparts running on leaf nodes would utilize `c_leaf` sockets. The root application will use **WSAJoinLeaf** to add new leaf nodes. The corresponding applications on the leaf nodes will have set their `c_leaf` sockets into the listening mode. **WSAJoinLeaf** with a `c_root` socket specified will be mapped to the Q.2931 SETUP message (for the first leaf) or ADD PARTY message (for any subsequent leaves).

---

**Note** The QOS parameters specified in **WSAJoinLeaf** for any subsequent leaves should be ignored per the ATM Forum UNI specification.

---

The leaf-initiated join is not part of the ATM UNI 3.1, but is supported in the ATM UNI 4.0. Thus **WSAJoinLeaf** with a `c_leaf` socket specified can be mapped to the appropriate ATM UNI 4.0 message.

The AAL Parameter and B-LLI negotiation is supported through the modification of the corresponding IEs in the `lpSQOS` parameter upon the invocation of the condition function specified in **WSAAccept**.

---

**Note** Only certain fields in those two IEs can be modified. See the ATM Forum UNI specification Appendix C and Appendix F for details.

---

## ATM-Specific Quality of Service Extension

This section describes the protocol-specific Quality of Service (**QOS**) structure for ATM, which is contained in the **ProviderSpecific**.buf field of the **QOS** structure. Note that the use of this ATM-specific **QOS** structure is optional by Windows Sockets 2 clients, and the ATM service provider is required to map the generic **FLOWSPEC** structure to appropriate Q.2931 Information Elements. However, if both the generic **FLOWSPEC** structure and ATM-specific **QOS** structure are specified, the value specified in the ATM-specific **QOS** structure should take precedence should there be any conflicts. See Windows Sockets 2 API specification section 2.7 for more information about the QOS provisions and **FLOWSPEC** structure.

The protocol-specific **QOS** structure for ATM is a concatenation of Q.2931 Information Element (IE) structures, which are defined in the following text. If an application omits an IE that UNI 3.x mandates, the service provider should insert a reasonable default value, taking the information in the **FLOWSPEC** structure into account if applicable.

The handling of repeated IEs is dependent on the IE itself. If an IE is repeated and it is one that is allowed to be repeated per the ATM Forum UNI specification then the provider must handle it properly. In this case, order in the list determines preference

order, with elements appearing earlier in the list being more preferred. If an IE is repeated and this is not allowed per ATM Forum UNI specification, the provider may either fail the request from the client (the preferred option) or use the last IE of that type found.

Each individual **IE** structure is formatted in the following manner and identified by the **IEType** field:

```
typedef struct {
    Q2931_IE_TYPE IEType;
    ULONG         IELength;
    UCHAR         IE[1];
} Q2931_IE;
```

Legal values for the **IEType** field are defined as:

```
typedef enum {
    IE_AALParameters,
    IE_TrafficDescriptor,
    IE_BroadbandBearerCapability,
    IE_BHLI,
    IE_BLLI,
    IE_CalledPartyNumber,
    IE_CalledPartySubaddress,
    IE_CallingPartyNumber,
    IE_CallingPartySubaddress,
    IE_Cause,
    IE_QOSClass,
    IE_TransitNetworkSelection,
} Q2931_IE_TYPE;
```

The **IE** field is overlaid by a specific **IE** structure and the **IELength** field is the total length in bytes of the **IE** structure including the **IEType** and **IELength** fields. The semantics and legal values for each element of these **IE** structures are per ATM UNI 3.x specification. **SAP\_FIELD\_ABSENT** can be used for those elements that are optional for a given **IE** structure, per ATM UNI 3.x specification.

## AAL Parameters

```
/*
 * manifest constants for the AALType field in struct AAL_PARAMETERS_IE
 */
typedef enum {
    AALTYPE_5      = 5,    /* AAL 5 */
    AALTYPE_USER  = 16    /* user-defined AAL */
} AAL_TYPE;
```

(continued)



*(continued)*

```

/*
 * values used for the Mode field in struct AAL5_PARAMETERS
 */
#define AAL5_MODE_MESSAGE          0x01
#define AAL5_MODE_STREAMING       0x02

/*
 * values used for the SSCSType field in struct AAL5_PARAMETERS
 */
#define AAL5_SSCS_NULL            0x00
#define AAL5_SSCS_SSCOP_ASSURED  0x01
#define AAL5_SSCS_SSCOP_NON_ASSURED 0x02
#define AAL5_SSCS_FRAME_RELAY    0x04

typedef struct {
    ULONG ForwardMaxCPCSSDUSize;
    ULONG BackwardMaxCPCSSDUSize;
    UCHAR Mode;                               /* only available in UNI 3.0 */
    UCHAR SSCSType;
} AAL5_PARAMETERS;

typedef struct {
    ULONG UserDefined;
} AALUSER_PARAMETERS;

typedef struct {
    AAL_TYPE AALType;
    union {
        AAL5_PARAMETERS    AAL5Parameters;
        AALUSER_PARAMETERS AALUserParameters;
    } AALSpecificParameters;
} AAL_PARAMETERS_IE;

```

## ATM Traffic Descriptor

This sections lists the ATM Traffic Descriptor.

```

typedef struct {
    ULONG PeakCellRate_CLP0;
    ULONG PeakCellRate_CLP01;
    ULONG SustainableCellRate_CLP0;
    ULONG SustainableCellRate_CLP01;
    ULONG MaxBurstSize_CLP0;
    ULONG MaxBurstSize_CLP01;
    BOOL Tagging;
} ATM_TD;

```

```
typedef struct {
    ATM_TD Forward;
    ATM_TD Backward;
    BOOL BestEffort;
} ATM_TRAFFIC_DESCRIPTOR_IE;
```

## Broadband Bearer Capability

This section lists the values used for the Broadband Bearer Capability.

```
/*
 * values used for the BearerClass field in struct
ATM_BROADBAND_BEARER_CAPABILITY_IE
 */
#define BCOB_A          0x00 /* Bearer class A          */
#define BCOB_C          0x03 /* Bearer class C          */
#define BCOB_X          0x10 /* Bearer class X          */

/*
 * values used for the TrafficType field in struct
ATM_BROADBAND_BEARER_CAPABILITY_IE
 */
#define TT_NOIND        0x00 /* No indication of traffic type */
#define TT_CBR          0x04 /* Constant bit rate          */
#define TT_VBR          0x06 /* Variable bit rate          */

/*
 * values used for the TimingRequirements field in struct
ATM_BROADBAND_BEARER_CAPABILITY_IE
 */
#define TR_NOIND        0x00 /* No timing requirement indication */
#define TR_END_TO_END   0x01 /* End-to-end timing required */
#define TR_NO_END_TO_END 0x02 /* End-to-end timing not required */

/*
 * values used for the ClippingSusceptability field in struct
ATM_BROADBAND_BEARER_CAPABILITY_IE
 */
#define CLIP_NOT        0x00 /* Not susceptible to clipping */
#define CLIP_SUS        0x20 /* Susceptible to clipping */

/*
 * values used for the UserPlaneConnectionConfig field in
 * struct ATM_BROADBAND_BEARER_CAPABILITY_IE
 */
#define UP_P2P          0x00 /* Point-to-point connection */
```

(continued)

*(continued)*

```
#define UP_P2MP                0x01    /* Point-to-multipoint connection */

typedef struct {
    UCHAR BearerClass;
    UCHAR TrafficType;
    UCHAR TimingRequirements;
    UCHAR ClippingSusceptability;
    UCHAR UserPlaneConnectionConfig;
} ATM_BROADBAND_BEARER_CAPABILITY_IE;
```

## Broadband High Layer Information

This section lists the type definition for the broadband high-layer information.

```
typedef ATM_BHLI ATM_BHLI_IE;
```

## Broadband Lower Layer Information

This section lists the type definition for the broadband lower-layer information.

```
/*
 * values used for the Layer2Mode field in struct ATM_BLLI_IE
 */
#define BLLI_L2_MODE_NORMAL    0x40
#define BLLI_L2_MODE_EXT      0x80

/*
 * values used for the Layer3Mode field in struct ATM_BLLI_IE
 */
#define BLLI_L3_MODE_NORMAL    0x40
#define BLLI_L3_MODE_EXT      0x80

/*
 * values used for the Layer3DefaultPacketSize field in struct ATM_BLLI_IE
 */
#define BLLI_L3_PACKET_16      0x04
#define BLLI_L3_PACKET_32      0x05
#define BLLI_L3_PACKET_64      0x06
#define BLLI_L3_PACKET_128     0x07
#define BLLI_L3_PACKET_256     0x08
#define BLLI_L3_PACKET_512     0x09
#define BLLI_L3_PACKET_1024    0x0A
#define BLLI_L3_PACKET_2048    0x0B
#define BLLI_L3_PACKET_4096    0x0C

typedef struct {
    DWORD Layer2Protocol;    /* User information layer 2 protocol */
    UCHAR Layer2Mode;
```



```

    UCHAR Layer2WindowSize;
    DWORD Layer2UserSpecifiedProtocol;    /* User specified layer 2
                                           protocol information */
    DWORD Layer3Protocol;                 /* User information layer 3
                                           protocol */
    UCHAR Layer3Mode;
    UCHAR Layer3DefaultPacketSize;
    UCHAR Layer3PacketWindowSize;
    DWORD Layer3UserSpecifiedProtocol;    /* User specified layer 3
                                           protocol information */
    DWORD Layer3IPI;                      /* ISO/IEC TR 9577 Initial
                                           Protocol Identifier */
    UCHAR SnapID[5];                      /* SNAP ID consisting of OUI
                                           and PID */
} ATM_BLLI_IE;

```

## Called Party Number

This section lists the type definition for the called party number.

```
typedef ATM_ADDRESS ATM_CALLED_PARTY_NUMBER_IE;
```

## Called Party Subaddress

This section lists the type definition for the called party subaddress.

```
typedef ATM_ADDRESS ATM_CALLED_PARTY_SUBADDRESS_IE;
```

## Calling Party Number

This section lists the type definition for the calling party number.

```

/*
 * values used for the Presentation_Indication field in
 * struct ATM_CALLING_PARTY_NUMBER_IE
 */
#define PI_ALLOWED                0x00
#define PI_RESTRICTED            0x40
#define PI_NUMBER_NOT_AVAILABLE  0x80

/*
 * values used for the Screening_Indicator field in
 * struct ATM_CALLING_PARTY_NUMBER_IE
 */
#define SI_USER_NOT_SCREENED     0x00
#define SI_USER_PASSED           0x01
#define SI_USER_FAILED           0x02
#define SI_NETWORK               0x03

```

(continued)

(continued)

```
typedef struct {
    ATM_ADDRESS ATM_Number;
    UCHAR      Presentation_Indication;
    UCHAR      Screening_Indicator;
} ATM_CALLING_PARTY_NUMBER_IE;
```

## Calling Party Subaddress

This section lists the type definition for the calling party subaddress.

```
typedef ATM_ADDRESS ATM_CALLING_PARTY_SUBADDRESS_IE;
```

## Quality of Service Parameter

This section lists the parameters used for the quality of service (QOS).

```
/*
 * values used for the QOSClassForward and QOSClassBackward
 * field in struct ATM_QOS_CLASS_IE
 */
#define QOS_CLASS0          0x00
#define QOS_CLASS1          0x01
#define QOS_CLASS2          0x02
#define QOS_CLASS3          0x03
#define QOS_CLASS4          0x04

typedef struct {
    UCHAR QOSClassForward;
    UCHAR QOSClassBackward;
} ATM_QOS_CLASS_IE;
```

## Transit Network Selection

This section lists values used for the transit network selection.

```
/*
 * values used for the TypeOfNetworkId field in struct
 ATM_TRANSIT_NETWORK_SELECTION_IE
 */
#define TNS_TYPE_NATIONAL    0x40

/*
 * values used for the NetworkIdPlan field in struct
 ATM_TRANSIT_NETWORK_SELECTION_IE
 */
#define TNS_PLAN_CARRIER_ID_CODE    0x01

typedef struct {
    UCHAR TypeOfNetworkId;
```

```

    UCHAR NetworkIdPlan;
    UCHAR NetworkIdLength;
    UCHAR NetworkId[1];
} ATM_TRANSIT_NETWORK_SELECTION_IE;

```

## Cause

In addition to all the IEs previously described, which could be specified in the ATM-specific **QOS** structure while calling **WSAConnect**, there is a Cause IE that can only be used during the call release. Upon disconnecting, Windows Sockets 2 applications can optionally specify this IE as the disconnect data in **WSASendDisconnect**. The remote party can retrieve this IE through **WSARecvDisconnect** after receiving the **FD\_CLOSE** notification.

```

/*
 * values used for the Location field in struct ATM_CAUSE_IE
 */
#define CAUSE_LOC_USER                0x00
#define CAUSE_LOC_PRIVATE_LOCAL       0x01
#define CAUSE_LOC_PUBLIC_LOCAL        0x02
#define CAUSE_LOC_TRANSIT_NETWORK     0x03
#define CAUSE_LOC_PUBLIC_REMOTE       0x04
#define CAUSE_LOC_PRIVATE_REMOTE      0x05
#define CAUSE_LOC_INTERNATIONAL_NETWORK 0x06
#define CAUSE_LOC_BEYOND_INTERWORKING 0x0A

/*
 * values used for the Cause field in struct ATM_CAUSE_IE
 */
#define CAUSE_UNALLOCATED_NUMBER      0x01
#define CAUSE_NO_ROUTE_TO_TRANSIT_NETWORK 0x02
#define CAUSE_NO_ROUTE_TO_DESTINATION 0x03
#define CAUSE_VPI_VCI_UNACCEPTABLE    0x0A
#define CAUSE_NORMAL_CALL_CLEARING    0x10
#define CAUSE_USER_BUSY                0x11
#define CAUSE_NO_USER_RESPONDING      0x12
#define CAUSE_CALL_REJECTED           0x15
#define CAUSE_NUMBER_CHANGED          0x16
#define CAUSE_USER_REJECTS_CLIR       0x17
#define CAUSE_DESTINATION_OUT_OF_ORDER 0x1B
#define CAUSE_INVALID_NUMBER_FORMAT   0x1C
#define CAUSE_STATUS_ENQUIRY_RESPONSE 0x1E
#define CAUSE_NORMAL_UNSPECIFIED      0x1F
#define CAUSE_VPI_VCI_UNAVAILABLE     0x23
#define CAUSE_NETWORK_OUT_OF_ORDER    0x26
#define CAUSE_TEMPORARY_FAILURE       0x29

```

(continued)



(continued)

```
#define CAUSE_ACCESS_INFORMATION_DISCARDED      0x2B
#define CAUSE_NO_VPI_VCI_AVAILABLE             0x2D
#define CAUSE_RESOURCE_UNAVAILABLE             0x2F
#define CAUSE_QOS_UNAVAILABLE                  0x31
#define CAUSE_USER_CELL_RATE_UNAVAILABLE       0x33
#define CAUSE_BEARER_CAPABILITY_UNAUTHORIZED  0x39
#define CAUSE_BEARER_CAPABILITY_UNAVAILABLE   0x3A
#define CAUSE_OPTION_UNAVAILABLE              0x3F
#define CAUSE_BEARER_CAPABILITY_UNIMPLEMENTED 0x41
#define CAUSE_UNSUPPORTED_TRAFFIC_PARAMETERS   0x49
#define CAUSE_INVALID_CALL_REFERENCE           0x51
#define CAUSE_CHANNEL_NONEXISTENT             0x52
#define CAUSE_INCOMPATIBLE_DESTINATION        0x58
#define CAUSE_INVALID_ENDPOINT_REFERENCE      0x59
#define CAUSE_INVALID_TRANSIT_NETWORK_SELECTION 0x5B
#define CAUSE_TOO_MANY_PENDING_ADD_PARTY     0x5C
#define CAUSE_AAL_PARAMETERS_UNSUPPORTED      0x5D
#define CAUSE_MANDATORY_IE_MISSING            0x60
#define CAUSE_UNIMPLEMENTED_MESSAGE_TYPE     0x61
#define CAUSE_UNIMPLEMENTED_IE               0x63
#define CAUSE_INVALID_IE_CONTENTS            0x64
#define CAUSE_INVALID_STATE_FOR_MESSAGE      0x65
#define CAUSE_RECOVERY_ON_TIMEOUT             0x66
#define CAUSE_INCORRECT_MESSAGE_LENGTH       0x68
#define CAUSE_PROTOCOL_ERROR                  0x6F

/*
 * values used for the Condition portion of the Diagnostics field
 * in struct ATM_CAUSE_IE, for certain Cause values
 */
#define CAUSE_COND_UNKNOWN                     0x00
#define CAUSE_COND_PERMANENT                   0x01
#define CAUSE_COND_TRANSIENT                   0x02

/*
 * values used for the Rejection Reason portion of the Diagnostics field
 * in struct ATM_CAUSE_IE, for certain Cause values
 */
#define CAUSE_REASON_USER                      0x00
#define CAUSE_REASON_IE_MISSING               0x04
#define CAUSE_REASON_IE_INSUFFICIENT          0x08

/*
 * values used for the P-U flag of the Diagnostics field
 * in struct ATM_CAUSE_IE, for certain Cause values
```

```
*/
#define CAUSE_PU_PROVIDER          0x00
#define CAUSE_PU_USER             0x08

/*
 * values used for the N-A flag of the Diagnostics field
 * in struct ATM_CAUSE_IE, for certain Cause values
 */
#define CAUSE_NA_NORMAL           0x00
#define CAUSE_NA_ABNORMAL        0x04

typedef struct {
    UCHAR Location;
    UCHAR Cause;
    UCHAR DiagnosticsLength;
    UCHAR Diagnostics[4];
} ATM_CAUSE_IE;
```

## ATM Header File

Use ws2atm.h.

## Other Windows Sockets 2 Considerations

### Secure Sockets Layer (SSL)

Secure Sockets Layer (SSL) is not natively supported in Windows Sockets 2. Microsoft makes available the Security Support Provider Interface (SSPI) to enable application programmers to provide security-enabled communications. See the section titled Security Support Provider Interface (SSPI), found under Security in the Platform SDK, for more information.

### RSVP

The Resource Reservation Protocol (RSVP) implementation of the Windows Sockets 2 Protocol-Specific Annex, found at <ftp://ftp.microsoft.com/bussys/winsock/winsock2/wsax203.doc>, is supported by Microsoft through Windows 2000 QOS functionality. For more information on RSVP in Windows 2000, check following chapters in this volume of the *Networking Services Developer's Reference Library*.





---

## CHAPTER 13

# QOS Overview

## QOS Documentation Structure

To facilitate explanation, Windows 2000 Quality of Service (QOS) documentation divides its QOS components into three categories:

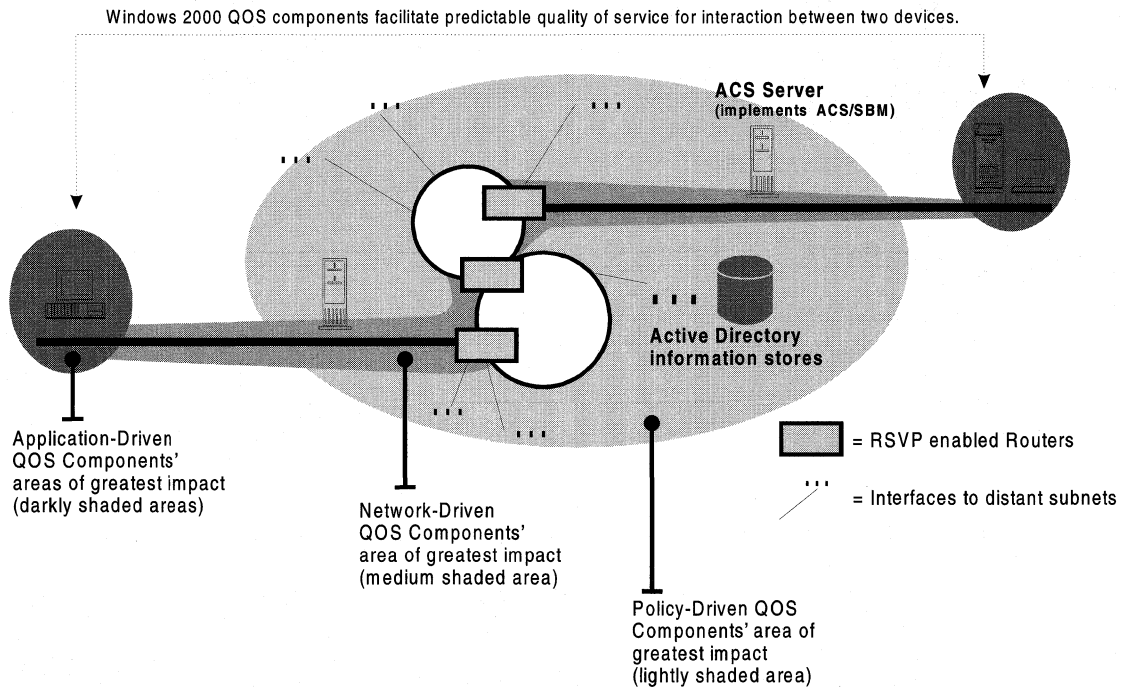
- Application-Driven QOS Components
- Network-Driven QOS Components
- Policy-Driven QOS Components

This division is purely for the sake of convenience and clarity. Though each of these QOS components is in some way initiated at the client or end-node, and in pure semantics, is initiated by the application, the impact of the component may actually be greatest elsewhere.

For an example (which may be clearer once the 802.1p QOS component is further explained), the 802.1p precedence bits are actually set in the end-node's network stack. This is done because it was an application-initiated sequence of QOS events that eventually triggered the setting of the bits. The cause of this bit-setting, then, could be argued to be the application's initiation of QOS service (thus application-driven). However, because the effect of setting the priority on 802.1p bits has the greatest impact when the packets associated with this session cross their local segment, 802.1p is included in the Network-Driven QOS Components category, but *not* in the Application-Driven QOS Components category. Despite the fact that no QOS requests would have been instigated to set the 802.1p bit *without* the application's invocation of a QOS component, its explanation is best placed in a discussion of network-driven components. The ripple-effect of 802.1p bit setting, then, is felt greatest in the network.

Although individual components from among the three categories can function independently at times to provide subsets of QOS functionality, Windows 2000 QOS overall is an integrated technology.

Figure 13-1 provides a visual representation of the structure of QOS documentation, and the reasoning behind its structure; certain QOS components have greatest impact in one of the three categories, and are thus discussed there.



**Figure 13-1: Visual Representation of QOS Documentation.**

## Determining Which Discussion Is for You

Most developers of QOS-enabled applications will find their requirements met within the QOS API, and the requisite knowledge of components will fall largely under the Application-Driven QOS Components section. Thus, reading through that overview material, combined with the API reference information in QOS Reference, should provide nearly all of the information necessary to develop QOS-aware applications.

There is more information in other sections of QOS documentation that facilitates the development of granular Traffic Control. For example, that found in the *Network-Driven QOS Components* section, can provide explanations of the underlying technologies and programmatic reference to interfaces available for such efforts. In such circumstances, a reading of the Application-Driven and Network-Driven sections is suggested.

For development of policy module components, such as policy-based decision services intended to provide network resource-admission services, the Policy-Driven sections are recommended.

For a complete understanding of QOS and its components, as they are implemented and interact within the Microsoft® Windows 2000® (and Microsoft® Windows® 98) framework, reading all of the overview sections is recommended.

## Additional Information on QOS

For additional information on Windows 2000 Quality of Service, including implementation recommendations and approaches, the following book is available:

Iseminger, David, *Windows 2000 Quality of Service*, (Macmillan Technical Publishing, 1999).

## About Quality of Service

Quality of Service in Microsoft® Windows® 2000 is a collection of components that enable differentiation and preferential treatment for subsets of data transmitted over the network. QOS components constitute the Microsoft implementation of Quality of Service.

Quality of Service is a defined term that loosely means subsets of data get preferential treatment when traversing a network. QOS technology has implications for the application programmer and the network administrator.

By writing QOS-enabled applications, programmers can:

- Specify or request bandwidth requirements particular to their application, such as latency requirements for streaming audio.
- Write mission-critical applications for the corporate environment that are guaranteed to get their required bandwidth—provided permissions and bandwidth availability exist.

Just as importantly, network administrators can leverage knowledge of Quality of Service to:

- Control network device resources based on user policy and/or application usage.
- Provision portions of a given bandwidth, whether an Ethernet subnet or a WAN interface, for applications or users that require such availability for core business activities.
- Shape and smooth the traffic that clients submit to the network, thereby avoiding the overburdening of switches and routers suffered with traditional burst transmissions.

Windows 2000 Quality of Service achieves this through programmatic interfaces, the cooperation of multiple components, and communication with network devices throughout the end-to-end network solution.

## Introduction to QOS

If you've ever endured demonstrations of new software touting real-time audio and video to the desktop (gobbling megabits per second on the already-taxed network resources), only to find your important file transfer inching along, you may be interested in Quality of Service. If you're the network administrator, charged with ensuring that mission-critical

applications have sufficient bandwidth to keep the company functioning, while users flood the network with multimedia application data, you, too may be interested in Quality of Service.

## Quality of Service Defined

Quality of Service is an industry-wide movement that aims to get more efficient use out of network resources by differentiating between subsets of data. The IETF (Internet Engineering Task Force) has played a central role in ensuring that QOS standards enable all affected network devices to participate in the end-to-end QOS-enabled connection.

Quality of Service provides applications (or network administrators) with a means by which network resources—such as available bandwidth and latency—can be predicted and managed on both local computers and devices throughout the network.

With such an all-network encompassing definition, QOS functionality requires cooperation among end nodes, switches, routers, and wide area network (WAN) links through which data must pass. Without some level of cooperation among those network devices, the quality of data transmission services can break down. In other words, if each such network device is left to make its own decisions about transmitting data, it would likely treat all data equally, and thus provide service on a first come—first served basis. Although such service may be satisfactory in network devices or transmission media that are not heavily loaded, when congestion occurs, such service can delay all data. With this information, we can extend the definition of quality of service—it allows preferential treatment for certain subsets of data as they traverse any QOS-enabled part of (or devices in) the network.

Microsoft® Windows 2000® implements quality of service by including a number of components that can cooperate with (or invoke) one another. These components are found in the DLLs, protocols, services, and individual network device functionality.

## Windows 2000 Quality of Service Defined

The Microsoft implementation of Quality of Service enables developers to use generic Windows Sockets 2 calls to create QOS-enabled applications. With the Windows 2000 QOS capabilities, developers do not need to consider how the various operating system components interact to achieve quality of service. The components that constitute QOS implementation are instead abstracted from the QOS application development effort, allowing a single or generic QOS interface—instead of individual interfaces—for each QOS component. This provides a generic interface for the developer, and also provides a mechanism by which new QOS components (perhaps with increased functionality) can be added, without the need to completely rewrite existing QOS applications.

## What QOS Solves

As computing and applications become more mission-critical, not to mention more content-rich and multimedia-oriented, the bandwidth necessary to service desktop functionality increases. However, bandwidth availability doesn't necessarily keep up with

the bandwidth appetite of today's desktop applications, creating an environment where there is often more data to be transmitted than there are resources to transmit. The *bursty* nature of network transmissions only aggravates the problem.

Traditional data transfers are also increasing as a result of the continual addition of new network nodes to existing networks. At the crux of this problem is the fact that there is no inherent means of differentiating between important data—such as data transmitted by mission-critical applications, and excessive data, or data transmitted by interesting (but not necessarily critical) multimedia applications.

Such traditional business applications are continuing to increase in size and in network use, but they aren't alone in their hunger for network resources. New applications such as multimedia applications, make extensive use of the network, pushing network utilization to its limit—and sometimes beyond. For example, video transmission applications require significant bandwidth to transmit with acceptable levels of quality. Due to the send-as-much-data-as-you-can nature of the most common networking protocol, IP, even a few active instances of these data-intensive programs can create bandwidth strain for networks that were never designed to carry the burden. With data-intensive multimedia applications putting such hefty data loads onto the network, the network often becomes less available for other applications. If the load is significant enough, overall network performance will wane.

Poor network performance is especially threatening to real-time audio and interactive conferencing transmissions; they are time-sensitive and especially susceptible to delays or individual frame drops delivery. Such delays in the delivery of individual packets, known as *latency*, can render real-time audio and real-time conferencing applications substandard at best and at worst, burdensome.

In the midst of larger traditional applications' higher network utilization, as well as increased network use by emerging desktop applications, core business applications are left vying, sometimes unsuccessfully, for adequate access to the network. These bandwidth-poor and latency-laden characteristics of an overburdened network have an even larger and more dramatic effect on mission-critical applications' use of new multimedia desktop technology: not only do they need access to the burdened network, they need more of the increasingly precious network resources, putting core business applications that use multimedia features in double jeopardy.

If such over-subscription of available network resources isn't a bleak enough picture, consider the prevalence of WAN, which introduces an even more critical and more precious bandwidth restriction at the WAN link. With this, the situation is exacerbated.

Mechanisms that manage network activity from an end-to-end perspective are needed to manage over-subscription of network resources, to regulate the allocation of their availability, and to present network data in a means more friendly to a shared network environment (that is, in a less bursty manner). These mechanisms are found in Windows 2000 Quality of Service.

## How Windows 2000 QOS Works

To achieve manageable and predictable quality of service from one end of the network to the other, the collection of components that must communicate and interact results in a fairly complex process. Microsoft® Windows 2000 QOS has the ability to facilitate priority along every step of a packet's journey: in the sender's network stack, at the switch, and even at each QOS-enabled router hop. Quality of Service also has the ability to facilitate how much data can and should be sent in a given unit of time, maximum burst rates, and overall bandwidth utilization rights. These can be configured based on administratively configurable policies. These functional capabilities only scratch the surface of quality of service.

Windows 2000 QOS is comprised of a number of components. Figure 13-2 shows where many of the QOS components reside in relation to the network stack, where communication occurs between and among them, and where certain interfaces, such as APIs, facilitate developing QOS services.

Note that this is an individual node's network stack view, not a functional schematic of how Quality of Service operates over a given network. Some of these components have further defined subcomponents, each of which is explained under *Components*.

## Windows 98 QOS Notes

The following list describes Windows 2000 QOS features *not* available in Microsoft® Windows® 98:

### Error Code Granularity

Windows 98 does not support the fine-grained error codes that can be provided in ExtendedStatus1 and ExtendedStatus2.

### Kernel Traffic Control

Kernel traffic control is not available in Windows 98. Therefore, observe the following restrictions when using Windows 98:

- Applications should not pass down or retrieve traffic objects, including shaping range or shape/discard objects.

- The SERVICE\_NO\_TRAFFIC\_CONTROL control flag is not available.

- ADSPEC parameters specific to traffic control are not set.

### Registry Parameters

Windows 98 does not support the following registry parameters:

- EnablePriorityBoost

- EnableRSVP

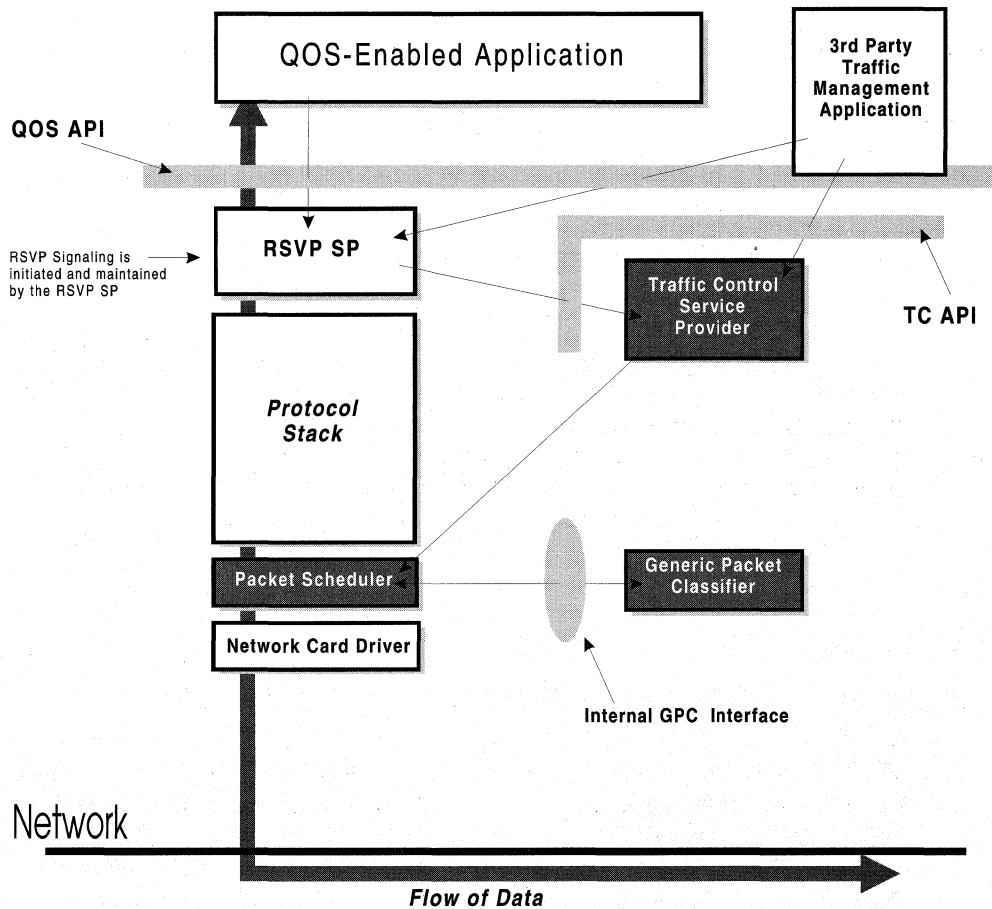
- EnableSPSetIPTOS

### RSVP\_RESERVE\_INFO

Windows 98 does not support the passing of **RSVP\_POLICY\_INFO** objects in the **RSVP\_RESERVE\_INFO** structure.

### Policy Objects

Windows 98 does not support the submission or retrieval of application specific-policy objects.



**Figure 13-2: QOS Components.**

### Service Types

The `SERVICE_NO_QOS_SIGNALING` control flag is not available.

### SIO\_CHK\_QOS

Windows 98 does not support `SIO_CHK_QOS`, and as a result, Windows 98 does not support queries for `ALLOWED_TO_SEND_DATA`, `ABLE_TO_RECV_RSVP`, or `LINE_RATE`.



### **SIO\_SET\_QOS**

Asynchronous calls to **WSAIoctl(SIO\_SET\_QOS)** are not supported in Windows 98.

### **TOS**

The RSVP service provider does not set Type of Service (TOS) bits in the IP header. TOS has been obviated by DSCP.

### **Known QOS Issues**

The following are known issues regarding the use of Quality of Service on Windows 98.

#### **Socket Handles**

With Windows 2000 or Windows 98, if a QOS-enabled application passes socket handles between different processes and requests for QOS notifications on that socket, the application may receive multiple notification events.

#### **Overlapped I/O**

The use of overlapped I/O on a blocking socket for QOS event notification is problematic.

## **QOS Header Files**

Certain QOS-specific data structures should not be used in Windows 98. The following list outlines those structures.

**ABLE\_TO\_RECV\_RSVP**  
**ALLOW\_TO\_SEND\_DATA**  
**INFO\_NOT\_AVAILABLE**  
**LINE\_RATE**  
**QOS\_OBJECT\_SD\_MODE**  
**RSVP\_OBJECT\_POLICY\_INFO**

**SERVICETYPE\_NONCONFORMING**  
**TC\_NONCONF\_BORROW**  
**TC\_NONCONF\_SHAPE**  
**TC\_NONCONF\_DISCARD**  
**TC\_NONCONF\_BORROW\_PLUS**

## **QOS Components**

To facilitate explanation, the Microsoft® Windows 2000 QOS documentation divides the QOS components into the following three topics:

- Application-Driven QOS Components
- Network-Driven QOS Components
- Policy-Driven QOS Components

---

**Note** This segmentation is purely for convenience and clarity. Though individual components among the three categories can function independently to provide certain subsets of QOS functionality, overall, Windows 2000 QOS is an integrated technology. For more information on how the documentation is organized, see *QOS Documentation Structure*.

---

This is the approach taken with all QOS components, with one exception: RSVP. Though the bulk of RSVP will be discussed within the *Application-Driven QOS Components* topic, elements that have effects elsewhere (namely in the network, and for facilitation of policy checking) will be included in their respective sections.

## Application-Driven QOS Components

### RSVP Service Provider

The RSVP service provider facilitates Windows 2000 QOS and invokes other QOS components.

### Traffic Control Modules

Traffic control modules facilitate traffic control. Traffic control modules include the packet classifier, the packet scheduler, and the packet shaper.

### Resource Reservation Protocol (RSVP) Service

Resource ReSerVation Protocol is invoked by the RSVP SP, and the carrier of RSVP messages across the entire network, with a functional interface for each QOS-enabled network device hop across the network. Though important to overall Windows 2000 QOS technology, and to QOS in general, certain QOS faculties do not require RSVP to operate, which means that quality of service can be achieved without RSVP signaling.

### QOS API

The QOS API is the programmatic interface to the RSVP SP.

### Traffic Control API (TC API)

The traffic control API is a programmatic interface to the traffic control components that regulate network traffic on local hosts.

### QOS API

The QOS API is the programmatic interface to the RSVP service provider (RSVP SP). Under most circumstances, the QOS API is the only interface that programmers will require to create QOS-aware or QOS-enabled applications. Most operations that happen on behalf of an application in the QOS sequence are a result of QOS API calls communicating requests down (and sometimes back up) through QOS components, creating a QOS-enabled flow of data that keeps important data moving through the network with preferential transmission consideration.

For more information on the programming elements included in the QOS API, see *QOS Reference*.

### RSVP Service

The RSVP service is a single instance Windows 2000 service that runs on a Windows 2000 computer. The RSVP service instigates traffic control functionality (if appropriate), and implements, maintains, and handles RSVP signaling for all Windows 2000 QOS functionality.

The RSVP service, by virtue of the fact that it implements and maintains RSVP and is the initiator of traffic control, is at the heart of Windows 2000 Quality of Service.

## RSVP Service Provider

The RSVP service provider (RSVP SP) is the name for the QOS component that invokes nearly all resulting QOS facilities. The RSVP SP communicates Windows Sockets 2 QOS semantics to the RSVP SP. The Rsvpsp DLL is loaded by Windows Sockets when a QOS-enabled socket is opened.

## Traffic Control API

The traffic control application programming interface (TC API), is a programmatic interface to the components that regulate network traffic on local hosts; both from an internal perspective (within the kernel itself), and from a network perspective (prioritization and queuing of packets based on transmission priority).

Traffic control is implicitly invoked through calls made to the QOS API and subsequently serviced by the RSVP service provider (RSVP SP). However, applications that require further or specific control over traffic control can use the TC API. The RSVP Service also uses TC API calls.

Note that the use of functions in the TC API requires administrative privilege.

## Traffic Control and Differentiated Services

Windows 2000 QOS is capable of providing differentiated services. The traffic control functionality built into Microsoft Windows 2000 QOS is also capable of operating in differentiated services mode. This capability is primarily used on routers, in which incoming IP packets are classified into generalized (differentiated) flows using the Diffserv code point (DSCP), rather than setting up and monitoring individual RSVP flows.

The packet scheduler component of TC can be put into differentiated services mode for a given interface by:

- Using **TcSetInterface**, Set GUID\_QOS\_ADAPTER\_MODE to PSADAPTER\_FLOW\_MODE\_DIFFSERV

The interface can be returned to its default mode of operation by setting the GUID to the default value of PSADAPTER\_FLOW\_MODE\_STANDARD.

While operating in Diffserv mode, standard TC function calls, such as those found in the Traffic Control API (TC API) section are still used, but a special QOS object called **QOS\_OBJECT\_DIFFSERV** can be included with the **TcAddFlow** function call. This QOS object is used to specify the list of DSCPs that are used to classify incoming IP packets on a given flow. Hence the **TcAddfilter** function and **TcDeleteFilter** function will not be used for classifying packets.

For more information about **QOS\_OBJECT\_DIFFSERV**, see *QOS Objects*.

## Traffic Control Modules

Traffic control (TC) plays a central role in the provision of quality of service. With traffic control, packets are prioritized both inside and outside the node on which TC is used. The implications for such granular control (or preferential treatment) of packets as they flow through the system and through the network, reach across the entire network realm or enterprise. Traffic control is realized through two modules, the Generic Packet Classifier and the Packet Scheduler.

### Generic Packet Classifier

Packet classification provides a means by which packets internal to a specific network node can be classified, and consequently prioritized, within and by both user and kernel-mode network components. These classification and prioritization uses include activities such as CPU processing attention or transmission onto the network. The Generic Packet Classifier (GPC) is utilized through the Generic Packet Classifier Interface, or GPC Interface, which facilitates an information store that can be used or associated with specified (defined) subsets of packets.

The importance of GPC hinges on its ability to provide lookup tables and classification services within the network stack, and is thus the first step in an overall and ubiquitous prioritization scheme for network traffic.

### Packet Scheduler

Packet scheduling is the means by which data (packet) transmission-governing—a key function of quality of service—is achieved. The packet scheduler is the traffic control module that regulates how much data an application (or flow) is allowed, essentially enforcing QOS parameters that are set for a particular flow. The packet scheduler incorporates three mechanisms in its scheduling of packets:

- A conformer
- The packet shaper
- A sequencer

The conformer and sequencer are discussed in more detail in the traffic control documentation. Since the packet scheduler's role is essential to overall traffic control understanding, it is defined here.

The packet scheduler considers the classification provided by the Generic Packet Classifier (GPC), and provides preferential treatment to higher-priority traffic. Consequently, the packet scheduler is the first step (in a sequential view) to ensuring that the prioritized network transmission of packets begins with data that has been deemed most important.

Part of the packet scheduler's responsibility is shaping the way packets are transmitted from a network device, a capability often referred to as packet shaping. Though often referenced by its own name, the packet shaper is simply a part of overall packet scheduler functionality.

The packet shaper mitigates the burst nature of computer network transmissions by smoothing transmission peaks over a given period of time, thereby smoothing out network usage to affect a more steady use of the network. The significance of the packet shaper becomes apparent: one factor that contributes to network congestion is the burst nature of computer data transmissions, a side-effect of the inherent “send it all out right now” nature of IP transmission. Packet shaping can help alleviate at least some of the effects of such activity by spacing out QOS-enabled packet transmissions.

## **Network-Driven QOS Components**

### **802.1p**

The use of flags in the Media Access Control (MAC) header to establish packet priority in shared-media 802 networks.

### **Differentiated Services**

Enables the marking of relative priority for IP packets. Differentiated Services enable the marking of packets with a code point value, called the DiffServe code point, which is used by network devices such as routers to determine the Per-Hop Behavior (PHB) treatment to which the packet is subjected. Essentially, differentiated services specifies a packet’s transmission priority as it passes through each network device on its journey through the network.

### **L2 Signaling**

The mapping of RSVP objects to Layer 2 (per the ISO OSI Model) signaling, such as Frame Relay Network Devices (FRNDs) or ATM interfaces.

### **Subnet Bandwidth Manager (SBM)**

Manages shared-media network bandwidth. In Windows 2000 Quality of Service, SBM functionality is incorporated into Admission Control Service (ACS).

### **Resource Reservation Protocol (RSVP)**

Carries and disseminates QOS information to QOS-aware network devices along the path between a sender and one or more receivers for a given flow, and also to senders and receivers.

### **802.1p**

Responsibility for QOS provisions on the local segment, and avoidance of the “all packets are treated equally” issue, falls onto the hub or switch servicing the segment. At such a level, the issue of differentiating between network packets, and perhaps treating them differently, must fall into the realm of the media access control (MAC) header. The MAC header (the lower half of Layer 2 in the ISO OSI Model) is the only part of a packet that hubs or switches investigate in their scope of work.

802.1p provides prioritization of packets traversing a subnet by the setting of a 3-bit value in the MAC header. Thus, when the local segment becomes congested and the hub/switch workload results in the delay (or dropping) of packets, those packets with flags that correspond to higher priorities will receive preferential treatment, and will be serviced before packets with lower priorities.

Note that implementing 802.1p for QOS requires an 802.1p-aware network interface card, an 802.1p-aware device driver, and an 802.1p-aware switch.

### **Differentiated Services**

Differentiated services enables packets that pass through network devices operating on Layer 3 information, such as routers, to have their relative priority differentiated from one another. Differentiated services uses 6 bits in the IP header to specify its value, called the DSCP (DiffServ code point); the first 6 bits of the TOS field, the first three of which were formerly used for IP precedence. Differentiated services has subsumed IP precedence, but maintains backward compatibility.

With differentiated services marking, Layer 3 devices can establish aggregated precedence-based queues and provide better service (when packet service is subject to queuing, as is the case under significant traffic loads) to packets that have higher relative priority. For differentiated services to be effective, Layer 3 devices must be DSCP-enabled.

### **L2 Signaling**

WAN technology manipulates Layer 1, Layer 2, and to a certain extent Layer 3 information, as it transmits data over the telecommunications network. Since quality of service is an end-to-end solution that provides quality of service for data transferred across the network, there must be a means by which data passing through WAN interfaces can be associated with some sort of preferential or nonpreferential treatment. Such a requirement necessitates the mapping of RSVP or other QOS parameters to WAN technology QOS interfaces.

Layer 2, however, is where QOS technology interacts most with the WAN's underlying signaling, since it is in Layer 2 where existing WAN technologies implement their own native QOS components. L2 signaling, in Windows 2000 QOS terms, takes QOS information such as parameters that are carried in RSVP messages to or through each network node between end devices, and maps that QOS information to native WAN technology QOS interfaces. For example, the classical IP over ATM (CLIP) module in Windows 2000 specifically maps Windows 2000 QOS settings to an appropriate ATM class of service.

### **Subnet Bandwidth Manager**

The Subnet Bandwidth Manager (SBM) is the QOS component that provides resource management and policy based-admission control for QOS-aware applications using shared media subnets (Ethernet, for example). The SBM is based on an IETF draft that defines SBM functionality and its general implementation.

SBM is implemented in Windows 2000® through the Admission Control Service (ACS). ACS is a Windows 2000 service that resides on a Windows 2000 Server.

## Policy-Driven QOS Components

### Admission Control Service (ACS)

A Windows 2000 service, residing on a Windows 2000 Server, that inserts itself into the RSVP message path to enforce Windows 2000 Directory Service based-network admission control policies for QOS-enabled clients.

### Local Policy Module (LPM)

A Microsoft-provided module that provides network resource-access decisions, based on policies configured in Active Directory services, for the Subnet Bandwidth Manager (SBM) component. The LPM makes policy decisions based on policy information contained in RSVP-based signaling messages sent by clients.

### Policy Information

Identity-based policy information contained in RSVP messages is submitted to the LPM, on which LPMs base policy decisions. Policy information is generated by a Microsoft-provided DLL that resides on Windows clients providing turnkey implementation of Windows 2000 QOS. Policy information is securely transmitted across the network in a session that is secured by a Kerberos ticket. Microsoft-provided policy information is data generated by the Microsoft-provided DLL, and not QOS service components (rather, they are data that is *generated* by a QOS component).

### RSVP

Carries policy data between end nodes and the ACS/SBM. RSVP also carries rejections to admission requests back to the requesting node.

### Local Policy Module API (LPM API)

The programmatic interface for LPMs to interface with the SBM in ACS.

For a figure that shows where these components fit into the end-to-end QOS enabled-network picture, see *QOS Documentation Structure*.

## Admission Control Service

Admission control service (ACS), is a Windows 2000 QOS component that regulates subnet usage for QOS-enabled applications. The ACS exerts its authority over QOS aware applications or clients by placing itself within the RSVP message path. With this placement, ACS effectively intercepts RSVP PATH, RESV, PATH\_ERR, RESV\_ERR, PATH\_TEAR, and RESV\_TEAR messages and passes the messages' policy information to Local Policy Modules (LPMs) for authentication. This exertion of ACS authority occurs on each interface (or shared medium) over which a given QOS flow must traverse. For a simplified example, if ACS is functioning on a source subnet and a (different) destination subnet for a given flow, policy restrictions are enforced by the ACS on *each* subnet.

ACS regulation is based on available network resources and on administratively-configurable information on users, or group policy. ACS is implemented as a Windows 2000 service on a Windows 2000 Server.

Local policy modules (LPMs) fall within the fold of ACS functionality, and can be considered an integral part of the ACS. With the default LPM, Microsoft Identity LPM (MSIDLPM) user information in the intercepted RSVP message is used to look up user policy in Windows 2000 Active Directory services. MSIDLPM then makes policy decisions based on information found in Active Directory services.

Another ACS component, the Policy Control Module (PCM), actually mediates the interaction between the ACS and LPMs. If there are multiple residential LPMs, the PCM will send all policy data objects contained in the received RSVP messages to each LPM, gather all responses, perform logical checks on the information, aggregate it, and return the combined response to the ACS.

If network resources are available and if the policy check succeeds, the RSVP message and its policy information is sent to the next hop (or the previous hop, if it is a PATH or RESV message). In this way, ACS acts as the logical gatekeeper for RSVP message propagation across the network by rejecting requests under the following conditions:

- If local segment resources aren't available to provide the requested level of QOS (the SBM functionality of the ACS).
- If the sender or receiver doesn't have appropriate policy permission to transmit data with the requested parameters.

When such conditions occur, no network nodes beyond the ACS (in the appropriate direction) receive any of the RSVP messages rejected by the ACS. However, the error messages due to the rejection will traverse the network to get to the network node that made the request.

This provides twofold service. It keeps unnecessary RSVP signaling traffic from traversing the network by keeping lame-duck RSVP messages from running across the network, and it preserves processing resources for routers and WAN Interface Cards (WANICs) since they will not have to handle such RSVP messages. Note that any node that declines requests based on policy failure, however, will return an RSVP error message to the sender, indicating failure. Clients will not transmit anything if their request is rejected by ACS.

Though ACS is a Windows 2000 QOS component, its services include other QOS components, such as the Subnet Bandwidth Manager (SBM) and its LPM interface.

### **Policy Information**

Microsoft provides identity policy information through a DLL installed on QOS-enabled Windows clients such as Windows 2000 Server and Windows 2000 Professional.

The policy information is incorporated in RSVP resource reservation requests, which in turn get sent out onto the network in the form of RSVP messages. This policy information is intercepted by the Admission Control Service (ACS), which includes SBM functionality as part of its fundamental service suite. The ACS services such requests by checking whether the requesting client has authorization to use network resources on the local subnet. If successful, the policy information is forwarded to the next network



node for subsequent policy checking, and such activity continues toward the intended receiver, along the data path of network nodes, until the request is rejected or the intended receiver (the node with which the sending client wishes to communicate) is reached.

### **Local Policy Module**

The Local Policy Module (LPM) is a QOS component responsible for retrieving and returning policy-based decisions used by the Admission Control Service (ACS). A default LPM provides an interface to policy information that is configured and stored in Windows 2000 Active Directory services. LPM is a generic term used to supply policy-based admission control decisions for ACS. An LPM makes these decisions using policies that are generally configured by network administrators, and stored in policy databases. An LPM is usually implemented as a DLL.

LPMs are used on the ACS server. Client QOS components that generate the policy element reside on the client, and are capable of creating policy information that is carried in RSVP messages to the ACS (which then gets forwarded down to the LPM). It is possible to install an LPM on the ACS server for which there is no corresponding component on the client; for example, an ACS-based LPM could enforce "time-of-day" policies, or could be capable of communicating with a COPS server.

It is important to note the difference between IETF-draft technologies and the Microsoft implementation of them. Subnet Bandwidth Manager (SBM), for example, is a technology derived from an IETF-draft proposal for regulating access to 802 subnets (draft-ietf-issll-is802-sbm-08.txt). ACS is a Windows 2000 service (and a QOS component) that incorporates SBM technology within its fold to incorporate regulation of access to 802 subnets in accordance with policy control.

### **LPM API**

The local policy module application programming interface (LPM API) is the programmatic interface by which LPMs communicate and interact with the Admission Control Service (ACS). The LPM API also specifies how LPMs are registered and initialized within the constructs of the ACS.

Such interaction is actually regulated by an abstraction module called the Policy Control Module (PCM). Because it is possible to have multiple LPMs, the PCM manages the policy-based decision information that LPM modules return. Note that LPMs may selectively accept or reject flows. For example, an LPM can receive an RSVP-based request from the PCM that has multiple flow requests; the LPM can then selectively accept or reject individual flows within that request, and return the results to the PCM. Note, too, that the PCM can manage information returned from multiple LPMs (if multiple LPMs are installed on the system), perform logical aggregation of their results, and then return aggregated information to the ACS.

The LPM API consists of a handful of functions used to allow its interaction with the ACS. The interaction of an LPM with its corresponding policy store or server is excluded from this interface; such interfacing would be proprietary to the LPM and the policy store or server.

### LPM Byte Reordering

The RSVP protocol submits policy information in network-byte order, as illustrated in the following example:

```
0 1 2 3 4 5 6 7
```

As you can see from the example, in network-byte ordering, the bits in any given byte are ordered such that the first bit (base zero) is bit zero, and the following bits are ordered sequentially through the last bit (bit position seven, base zero).

The Microsoft LPM, `Msidlpm.dll`, *reorders* these bits in order to put them into host-byte order. This is an important distinction because bit-based flags can be used to designate certain values in the PCM-LPM interaction. The following example illustrates host-byte ordering:

```
7 6 5 4 3 2 1 0
```

As you can see from the example, host byte ordering begins with the last bit in the byte (bit position seven, base zero) and progressing sequentially through the byte to bit position zero.

### Installing an LPM

For additional LPMs to function on a system, you must install the LPM on the Windows 2000 Server computer on which it will function. Note that the Microsoft-provided LPM, `Msidlpm.dll`, is installed by default with Windows 2000 Quality of Service.

#### ► To install an LPM on a system

1. Create a registry entry. The key in which the entry must be created is **HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\RSVP\PCM Config\ActiveLPMs**
  - a. The entry type is `REG_MULTI_SZ`
  - b. The value of the entry is the name of the LPM DLL
2. Then, place the DLL into the Windows 2000 `System32` directory.

### RSVP and QOS

Resource Reservation Protocol (RSVP) plays a multifaceted role in the provisioning and signaling of QOS reservation requirements in Windows 2000. The following pages outline how RSVP carries out that role.

## RSVP

Resource Reservation Protocol (RSVP) is an IETF-draft networking protocol dedicated to being the facilitator and carrier of standardized QOS information and parameters. RSVP carries generic (industry-defined) QOS parameters from end nodes (inclusive) to each QOS-aware network device included in the path between RSVP session members. That is, RSVP is a means by which end nodes and network devices can communicate and negotiate QOS parameters and network usage admission.

RSVP is a multifaceted protocol—it is central to application-driven, network-driven, and policy-driven QOS activities—it is the carrier on which Windows 2000 systems send QOS parameters and information to the network. RSVP has its own objects that can be filled or interrogated to specify or determine the requested QOS parameters that are either requested by the application, required of the network device, or applied for by the user.

Since RSVP has its own clearly-defined objects, the implementation of RSVP within Windows 2000 requires QOS parameters to be mapped into (or on the receiving end of RSVP messages, derived from) RSVP objects.

From an application-driven perspective, RSVP is the means by which an application's requests are transported through the network (a container, though one with specifically defined compartments). Calls to the QOS API provide information to the RSVP SP, which then maps such information into predefined RSVP objects for transmission across the network.

RSVP provides a mechanism for end systems to convey information to network devices about application data flow. Network devices can then take QOS- or policy-related action based on this information.

From a policy-driven perspective, RSVP provides the necessary user information within industry standard—RSVP objects (much like a set of containers, each of which is defined based on its standard formats), that facilitate the exchange of user identity and admission requests.

Although RSVP can be considered responsible for the bulk of Windows 2000 QOS interaction among end nodes, it is possible to achieve QOS without RSVP signaling, such as by using traffic control facilities available with Windows 2000 QOS.

---

## CHAPTER 14

# QOS Programming

## Basic QOS Operations

The RSVP Service Provider (RSVP SP) is where developers can enable their applications to take advantage of the QOS capabilities built into Windows 2000. The RSVP SP provides a service layer; by sitting between applications that want to take advantage of Windows QOS capabilities and QOS components. The RSVP SP shields developers from complexities involved in directly interfacing with QOS components such as RSVP, traffic control, and local policy modules.

The process of QOS-enabling an application includes enabling the application to perform the following:

- Receiving QOS-enabled Data
- Sending QOS-enabled Data
- Closing the QOS connection

Throughout the course of a QOS session, applications also need to be able to manage the QOS connection.

Note that there are other components that may affect the success of QOS-enabled connection requests—such as policies in routers or on switches. For the purpose of explaining the process of QOS-enabling applications, the effect of these other QOS components will be addressed apart from the process of QOS enabling an application with the RSVP SP.

## QOS-Enabling Your Application

QOS is enabled through the use of specific Windows Sockets APIs, as well as APIs and structures created specifically for use with QOS. Applications taking advantage of Windows QOS use Windows Sockets 2 APIs, in conjunction with QOS APIs and structures, to create a connection and provide QOS parameters that articulate the application's QOS requirements. QOS APIs and structures are also used to maintain (or change) settings associated with a particular QOS session.

The call sequence involved in establishing and maintaining a QOS-enabled connection generally includes preparation calls, such as enumerating protocols then querying available protocols for QOS capability. The process of enumerating and querying protocols for QOS capability is outlined in *Opening a QOS-Enabled Socket*.

## Opening a QOS-Enabled Socket

The services and service quality guarantees provided by the RSVP SP require a QOS-enabled socket. The process of finding a QOS-enabled protocol involves the following steps:

1. Call the **WSAEnumProtocols** function to enumerate the existing protocols.
2. Loop through the enumerated protocols to find a protocol that supports QOS. QOS support is indicated by the presence of the `XP1_QOS_SUPPORTED` flag in `dwServiceFlags1` in the **WSAPROTOCOL\_INFO** structure.
3. Call the **WSASocket** function, and pass a pointer to the **WSAPROTOCOL\_INFO** structure corresponding to the QOS-enabled protocol. Note that the RSVP SP requires an overlapped socket. Create the socket in overlapped mode by setting the `WSA_FLAG_OVERLAPPED` flag in the `dwFlags` parameter of the **WSASocket** function.

---

**Note** Do not attempt to use the **WSADuplicateSocket** function to create QOS-enabled sockets; the **WSADuplicateSocket** function cannot be used on a QOS-enabled socket. Attempting to do so will result in the return of a `WSAEINVAL` error.

---

## Invoking the RSVP SP

QOS-enabled connections are unidirectional. To enable a connection with service guarantees for both sending and receiving from a host, two individual QOS-enabled flows are required. Whether the QOS-enabled flow is for sending or receiving, the initial process of invoking the RSVP SP usually includes the use of one of the following Windows Sockets 2 APIs:

- **WSAConnect**
- **WSAJoinLeaf**
- **WSAAccept**
- **WSAIoctl(SIO\_SET\_QOS)**

Each of these functions invokes the RSVP SP on the application's behalf, and begins the process of establishing Quality of Service between the receiver and sender. Each of these functions includes a parameter that provides the application with the capability of providing QOS-specific parameters. These QOS-specific parameters are provided through the **QOS** structure, which is included as a parameter of each of the three preceding functions (**WSAAccept** generally implements the **QOS** structure through the callback function provided in its `lpfnCondition` parameter).

Whenever an application calls the **WSAConnect** function, the **WSAJoinLeaf** function, or the **WSAAccept** callback function with a non-NULL pointer to the **QOS** structure, QOS functionality is implicitly invoked. This invocation includes enlistment of any other QOS components' services (such as traffic control or RSVP signaling) by the RSVP SP on behalf of the application.

QOS technology and parameters are unidirectional, enabling different transmission parameters for sender and receiver. Additionally, RSVP semantics are receiver-centric, in that resources aren't reserved until the receiver sends out a RESV message. Due to this unidirectional approach, differences exist in the behavior of the receiver and the sender. These differences are outlined in the *Receiving QOS-Enabled Data* and *Sending QOS-Enabled Data* sections.

## Providing the RSVP SP with QOS-specific Parameters

In order for the RSVP SP to act on behalf of an application, the RSVP SP must be provided with QOS-specific parameters. These parameters come in the form of the following structures:

- **QOS**
- **FLOWSPEC**

The **QOS** structure is the all-encompassing structure for QOS-specific parameters, and is comprised of three parameters, two of which are **FLOWSPEC** structures (one for sending, one for receiving).

The **FLOWSPEC** structure includes flow members, which are **TokenRate**, **TokenBucketSize**, **PeakBandwidth**, **Latency**, **DelayVariation**, **ServiceType**, **MaxSduSize** (maximum packet size permitted in the traffic flow), and **MinimumPolicedSize**. When the application provides values for these parameters, the RSVP SP can characterize the service quality being requested; this enables the RSVP SP to make appropriate requests to corresponding components of QOS.

The third parameter of the **QOS** structure is the **ProviderSpecific** buffer, which is used for additional provider-specific QOS parameters that can not be specified in the **FLOWSPEC** structures. For more information about the **ProviderSpecific** buffer, see the section titled *Using the ProviderSpecific Buffer*.

QOS parameters for the **FLOWSPEC** structure may be common among certain application types. Such commonality lends itself to the establishment of standardized QOS parameters for a given type of transmission. The RSVP SP provides such standardization, and the ease of implementation that comes with prescribed service types, through the use of QOS templates, which are described in detail in the *QOS Templates* section.

Further details of the **QOS** structure and the **FLOWSPEC** structure can be found in their respective API entries.

## Receiving QOS-Enabled Data

An application can begin receiving data from its QOS-enabled sender before the QOS-enabled reservation is established. The result of receiving data before the QOS reservation is established is that data is transmitted over the network without QOS guarantees (as it would be in a non-QOS network). Once the QOS-enabled reservation is established, the receiver can receive data that conforms to the established QOS-parameters for the reservation.

Inherent in the occurrence of events (such as the establishment of the QOS-enabled connection) is the need for an application to query for events. The RSVP SP provides mechanisms that enable event notification. These mechanisms differ depending on whether the application is receiving or sending data.

As a *receiver* on a QOS-enabled connection, a host may initiate some or all of the following actions:

- As implied in the section titled *Providing the RSVP SP with QOS-specific parameters*, a receiver must provide QOS-specific parameters to the RSVP SP. These parameters are provided in the *ReceivingFlowSpec* parameter of the **QOS** structure, which is provided through the use of the **WSAConnect** function, the **WSAJoinLeaf** function, the **WSAAccept** callback function, or through the Windows Sockets 2 **SIO\_SET\_QOS** IOCTL opcode.
- To streamline the establishment of existing, common settings for the **QOS** structure, an application can use the **WSAGetQosByName** function to enumerate, and then retrieve an existing QOS template, and apply the values in that template (in the form of a **QOS** structure) to the QOS structure in the Windows Sockets function call. See the section titled *QOS Templates* for more information on QOS templates.
- The application may then send data. Data sent per the constraints of the established QOS-specific parameters is considered *conforming* data. Data that falls outside those parameters, or nonconforming data, is handled differently based on the **QOS\_OBJECT\_SD\_MODE** setting, which is particular to traffic control. Options for nonconforming data include relegating it to a best-effort transmission to discarding nonconforming packets.
- The application may want to monitor RSVP SP *events* in order to monitor the QOS-enabled connection status. For example, the application uses the status information and error codes provided with **FD\_QOS** events. Other RSVP SP events that an application may want to monitor include using an **RSVP\_RESERVE\_INFO** object to request arrival confirmation of a RESV message.
- When the application receives notification of certain events, it may want to take appropriate action to modify its QOS settings or parameters. For example, if an application is notified of the arrival of a PATH message, it must use the Windows Sockets 2 **SIO\_GET\_QOS** IOCTL opcode in order to retrieve pertinent QOS information, such as the sender's Tspec and the path's Adspec. Information provided in the PATH message may be used by the application to modify its initial QOS-specific parameters to match the sender's Tspec. For more information on Tspec and Adspec, see the section titled *RSVP SP and RSVP*.

## Sending QOS-Enabled Data

As a *sender* on a QOS-enabled connection, the events differ somewhat from the receiver's collection of available actions. Where senders initiate connections and request certain QOS-specific parameters, *receivers* respond to such requests (note that network devices between the receiver and sender *also* react to receiver requests, and may reject requests before they ever reach the sender).

As a *sender* on a QOS-enabled connection, a host may initiate some or all of the following actions:

- Sending hosts must inform the RSVP SP of QOS-specific parameters in order to enable the RSVP SP to interact with other QOS components on the sending host's behalf. QOS-specific parameters are provided to the RSVP SP through the **SendingFlowspec** member of the **QOS** structure, which itself is a parameter of the **WSAConnect** function, the **WSAAccept** function's **ConditionFunc** placeholder, and the **WSAJoinLeaf** function. The **SendingFlowspec** member can also be provided through the use of the SIO\_SET\_QOS IOCTL opcode.
- The QOS-specific members (**SendingFlowspec**) are based on the application's knowledge of the data transmission characteristics appropriate for the application. For example, bandwidth requirements for database queries of a mission-critical application may be different than bandwidth requirements for low-quality audio broadcasts. Latency boundaries may also be different for those types of applications (note that latency boundaries are only available with Guaranteed Service). Therefore, the application is best equipped to provide appropriate QOS-specific transmission parameters, perhaps through the use of a QOS template.
- Senders may use the **WSAGetQosByName** function to enumerate, and then retrieve preinstalled QOS templates that include QOS-specific parameters with appropriate transmission characteristics based on the type of application. For example, there may be a template called RADIOBCAST associated with a **QOS** structure that automatically applies the most appropriate QOS settings for radio broadcast senders. It is the application's responsibility to ensure its traffic remains within the bounds of **FLOWSPEC** members obtained using the **WSAGetQosByName** function; traffic that exceeds those parameters will be treated as nonconforming, and may result in degradation of quality.
- Senders may configure additional traffic control parameters by including objects in the **ProviderSpecific** buffer, such as handling nonconforming packets through setting of the SD\_MODE.
- The sender may monitor RSVP SP *events* to maintain the QOS-enabled connection, such as the arrival of RESV messages. Often, the sender will want to monitor status information and error codes provided with FD\_QOS events. For example, the sender may choose to stop transmission when the FD\_QOS event WSA\_QOS\_NO\_RECEIVER occurs, indicating that there are no QOS-enabled receivers for the transmission.



- The sending application may want to use the Windows Sockets 2 `SIO_GET_QOS` IOCTL opcode to look up QOS-specific parameters associated with the arrival of a RESV message.
- The sending application may want to use the Windows Sockets 2 `SIO_CHK_QOS` IOCTL opcode to query parameters that provide information about the QOS connection such as allowed sending rate, line rate, and others.

## Closing the QOS Connection

There are a number of ways to close a QOS-enabled connection. Generally, any event that closes a socket also ends RSVP SP servicing on the socket. Examples of events that cause the RSVP SP to stop providing QOS functionality on a socket include:

- Using the **closesocket** function to close the socket.
- Using the **shutdown** function to disable sends or receives on a socket. Note, however, that shutting down only the receive capabilities leaves the QOS-enabled send capabilities for the socket intact, and that shutting down only the send capabilities of a socket leaves the QOS-enabled receive capabilities for the socket intact as well.
- Using the **WSAConnect** function with the *name* parameter set to NULL.
- Using the **WSAIoctl** function with the `SIO_SET_QOS` IOCTL opcode, in which the **ServiceType** member corresponding to **SendingFlowspec** or **ReceivingFlowspec** (both of which are of **FLOWSPEC**) is set to `SERVICETYPE_NOTRAFFIC` or `SERVICETYPE_BESTEFFORT`.

Note that setting `SERVICETYPE_NOTRAFFIC` or `SERVICETYPE_BESTEFFORT` selectively can disable RSVP SP service for sending and receiving individually. For example, setting the **ServiceType** member of **SendingFlowspec** to `SERVICETYPE_NOTRAFFIC` or `SERVICETYPE_BESTEFFORT` only terminates QOS servicing for sending, and setting the *ServiceType* member of **ReceivingFlowspec** to `SERVICETYPE_NOTRAFFIC` or `SERVICETYPE_BESTEFFORT` only terminates QOS servicing for receiving.

## QOS Templates

The use of QOS templates enables applications to leverage well-known QOS settings for common transmission types. Templates can reduce the complexity associated with setting QOS parameters, because they enable application programmers to choose from established QOS settings (by using included and existing templates), or enable application programmers to create additional templates based for special QOS needs of applications, then simply apply that template to subsequent connections.

The RSVP SP provides three functions that facilitate the enumeration, use, creation, and deletion of QOS templates. These functions are:

- **WSAGetQOSByName**
- **WSCInstallQOSTemplate**
- **WSCRemoveQOSTemplate**

There are four activities associated with using QOS templates:

- Enumerating Available QOS Templates
- Applying a QOS Template
- Installing a QOS Template
- Removing a QOS Template

## Enumerating Available QOS Templates

In order to discover which templates are available on a given system, an application can enumerate the available QOS templates, using the **WSAGetQOSByName** function.

The process of using the **WSAGetQOSByName** function to enumerate available QOS templates follows a set of steps, as outlined in the following list:

1. The application calls the **WSAGetQOSByName** function with the *lpQOS* parameter set to NULL, and a pointer to a structure of type **WSABUF** provided for the *lpQOSName* parameter.
2. A list of available QOS template names is returned in the **WSABUF** structure pointed to by *lpQOSName*.
3. The application peruses the available templates, and selects an appropriate template, based on the template's name (codec), to service the application's required QOS parameters.

Once the list of available QOS templates is obtained, the application may apply a QOS template to a requested connection. The process of applying a QOS template is explained in the section called Applying a QOS Template.

## Applying a QOS Template

When an application knows the name of the QOS template it wants to use, the process of applying the QOS template is implemented through the **WSAGetQOSByName** function. The following steps outline the process necessary to apply a QOS template:

1. The application then implements a QOS template by making a call to the **WSAGetQOSByName** function. In the function call, the application passes the template name in the *lpQOSName* parameter, and provides a pointer to a **QOS** structure to be filled with the template's settings in the *lpQOS* parameter with a NULL string.
2. The RSVP SP fills the **QOS** structure pointed to in *lpQOS* with the parameters from the selected QOS template.

3. The application then sets members of its **SendingFlowspec** and **ReceivingFlowspec** members of the **QOS** structure with values that correspond to the values of the associated template. (**SendingFlowspec** and **ReceivingFlowspec** are members of the **QOS** structure, and both are of type **FLOWSPEC**.) When using a QOS template, you should ensure that your application conforms to the transmission characteristics specified by the template being used.
4. The application may then make a QOS-enabled connection request, and by using the **QOS** structure filled in Step 2, implement the QOS template's QOS parameters as part of the request.

For applications that do not know the name of the QOS template they want to use, or want to enumerate the available QOS templates, the process outlined in *Enumerating Available QOS Templates* should be followed.

## Installing a QOS Template

Under certain circumstances, such as when available QOS templates do not fit required or desired QOS parameters, an application may want to install a QOS template of its own. By installing a QOS template that is tailored specifically to its needs, the application can benefit from easier and more consistent implementation of QOS parameter requests.

Note that the installation of a QOS template requires administrative privilege. The process of installing a QOS template is contained within the **WSCInstallQOSTemplate** function. Essentially, a successful call to the **WSCInstallQOSTemplate** function installs a QOS template and associates the template with the name provided in the *IpQOSName* parameter. Once the QOS template is installed with the **WSCInstallQOSTemplate** function, it can be applied to a connection through the use of the **WSAGetQOSByName** function.

## Removing a QOS Template

The process of removing a QOS template is contained within the **WSCRemoveQOSTemplate** function. Note that the removal of a QOS template requires administrative privilege. For more details about the removal of a QOS template, see the **WSCRemoveQOSTemplate** reference page.

## Built-in QOS Templates

The RSVP SP provides a set of built-in QOS templates that enable applications to apply well-known codecs to QOS connection requests. The QOS templates provided are as follows:

- G711
- G723.1
- G729
- H263QCIF
- H263CIF

- H261QCIF
- H261CIF
- GSM6.10

## RSVP SP Error Codes

The RSVP SP provides a rich set of error codes and error values that enable applications and service providers to get detailed feedback when errors in processing occur.

RSVP SP errors (and all QOS errors) are retrieved using the **SIO\_GET\_QOS** IOCTL call. **SIO\_GET\_QOS** returns a **QOS** structure, and the error codes are returned in a **RSVP\_STATUS\_INFO** object provided in the **ProviderSpecific** buffer in the **QOS** structure.

The reference pages in this section enable application and service developers to become familiar with the RSVP SP error code/error value family, and provide suggested actions when such errors arise.

### Error Codes

The following table provides error codes, a description of the error code, and if applicable, the action that the application should take in the event the error is encountered.

Error code	Description	Application response
QQOS_NO_ERRORCODE	No error occurred, or the error code is unavailable.	
QQOS_RSVP	The error occurred in the local RSVP engine.	Check the QOS call sequence.
QQOS_KERNEL_TC	The error occurred in local traffic control.	Depending on the specific error value, the application may retry the operation with reduced QOS requirements.
QQOS_NET_ADMISSION	Admission failure, due to the SBM (part of the ACS).	Stop the operation, retry with reduced QOS requirements, or try later.
QQOS_NET_POLICY	A policy-related error occurred.	Stop or retry with reduced QOS requirements (depending on the specific error value).

*(continued)*

*(continued)*

Error code	Description	Application response
------------	-------------	----------------------

GQOS_ERRORCODE_UNKNOWN		
------------------------	--	--

GQOS_API		
----------	--	--

GQOS_RSVP_SYS		
---------------	--	--

GQOS_KERNEL_TC		
----------------	--	--

## Error Values

The following table provides error values, a description of the error value, and if applicable, the action that the application should take in the event the error is encountered.

Error value	Description	Application response
GQOS_NO_ERRORVALUE	No error occurred, or the error value is unavailable.	
GQOS_ERRORVALUE_UNKNOWN		

### Admission (resource) Error Values

GQOS_OTHER		
------------	--	--

GQOS_DELAYBND	An upstream QOS-enabled device cannot meet the specified delay-bound requirement.	Retry the operation with a more relaxed delay-bound requirement.
---------------	---	--

GQOS_BANDWIDTH	An upstream QOS-enabled device cannot meet bandwidth requirement.	Retry the operation with a more relaxed bandwidth requirement.
----------------	---	--

GQOS_MTU	The Maximum Transmission Unit (MTU) in <b>FLWSPEC</b> is too large.	Adjust the packet size and retry the operation.
----------	---	---

GQOS_FLOW_RATE	An upstream QOS-enabled device cannot meet bandwidth requirement.	Retry the operation with a more relaxed bandwidth requirement.
----------------	---	--

GQOS_PEAK_RATE	An upstream QOS-enabled device cannot meet bandwidth requirement.	Retry the operation with a more relaxed bandwidth requirement.
----------------	---	--

<b>Error value</b>	<b>Description</b>	<b>Application response</b>
<b>Policy Errors</b>		
GQOS_POLICY_ERROR_UNKNOWN	A policy error occurred for an unknown reason.	
GQOS_POLICY_GLOBAL_DEF_FLOW_COUNT	Policy error: the operation would exceed the global default-policy flow count.	Abort or retry at a later time.
GQOS_POLICY_GLOBAL_GRP_FLOW_COUNT	Policy error: the operation would exceed the global group-policy flow count.	Abort or retry at a later time.
GQOS_POLICY_GLOBAL_USER_FLOW_COUNT	Policy error: the operation would exceed the global user-policy flow count.	Abort or retry at a later time.
GQOS_POLICY_GLOBAL_UNK_USER_FLOW_COUNT	Policy error: the operation would exceed the unknown user-policy flow count.	Abort or retry at a later time.
GQOS_POLICY_SUBNET_DEF_FLOW_COUNT	Policy error: the operation would exceed the subnet default-policy flow count.	Abort or retry at a later time.
GQOS_POLICY_SUBNET_GRP_FLOW_COUNT	Policy error: the operation would exceed the subnet default-policy flow count.	Abort or retry at a later time.
GQOS_POLICY_SUBNET_USER_FLOW_COUNT	Policy error: the operation would exceed the subnet default-policy flow count.	Abort or retry at a later time.
GQOS_POLICY_SUBNET_UNK_USER_FLOW_COUNT	Policy error: the operation would exceed the subnet default-policy flow count.	Abort or retry at a later time.
GQOS_POLICY_GLOBAL_DEF_FLOW_DURATION	Policy error: the operation would exceed the global default-policy flow duration.	Abort.
GQOS_POLICY_GLOBAL_GRP_FLOW_DURATION	Policy error: the operation would exceed the global group-policy flow duration.	Abort.
GQOS_POLICY_GLOBAL_USER_FLOW_DURATION	Policy error: the operation would exceed the global user-policy flow duration.	Abort.

*(continued)*

*(continued)*

<b>Error value</b>	<b>Description</b>	<b>Application response</b>
<b>Policy Errors</b>		
GQOS_POLICY_GLOBAL_UNK_USER_FLOW_DURATION	Policy error: the operation would exceed the unknown user-policy flow duration.	Abort.
GQOS_POLICY_SUBNET_DEF_FLOW_DURATION	Policy error: the operation would exceed the subnet default-policy flow duration.	Abort.
GQOS_POLICY_SUBNET_GRP_FLOW_DURATION	Policy error: the operation would exceed the subnet group-policy flow duration.	Abort.
GQOS_POLICY_SUBNET_USER_FLOW_DURATION	Policy error: the operation would exceed the subnet user-policy flow duration.	Abort.
GQOS_POLICY_SUBNET_UNK_USER_FLOW_DURATION	Policy error: the operation would exceed the subnet unknown user-policy flow duration.	Abort.
GQOS_POLICY_GLOBAL_DEF_FLOW_RATE	Policy error: the operation would exceed the global default-policy flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_GRP_FLOW_RATE	Policy error: the operation would exceed the global group-policy flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_USER_FLOW_RATE	Policy error: the operation would exceed the global user-policy flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_UNK_USER_FLOW_RATE	Policy error: the operation would exceed the unknown user-policy flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_DEF_FLOW_RATE	Policy error: the operation would exceed the subnet default-policy flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_GRP_FLOW_RATE	Policy error: the operation would exceed the subnet group-policy flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_USER_FLOW_RATE	Policy error: the operation would exceed the subnet user-policy flow rate.	Abort or retry with reduced QOS requirements.

Error value	Description	Application response
<b>Policy Errors</b>		
GQOS_POLICY_SUBNET_UNK_USER_FLOW_RATE	Policy error: the operation would exceed the subnet unknown user-policy flow rate.	Abort or retry with reduced QOS requirements
GQOS_POLICY_GLOBAL_DEF_PEAK_RATE	Policy error: the operation would exceed the global default-policy peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_GRP_PEAK_RATE	Policy error: the operation would exceed the global group-policy peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_USER_PEAK_RATE	Policy error: the operation would exceed the global user-policy peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_UNK_USER_PEAK_RATE	Policy error: the operation would exceed the unknown user-policy peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_DEF_PEAK_RATE	Policy error: the operation would exceed the subnet default-policy peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_GRP_PEAK_RATE	Policy error: the operation would exceed the subnet default-policy peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_USER_PEAK_RATE	Policy error: the operation would exceed the subnet default-policy peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_UNK_USER_PEAK_RATE	Policy error: the operation would exceed the subnet default-policy peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_DEF_SUM_FLOW_RATE	Policy error: the operation would exceed the global default-policy total flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_GRP_SUM_FLOW_RATE	Policy error: the operation would exceed the global group-policy total flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_USER_SUM_FLOW_RATE	Policy error: the operation would exceed the global user-policy total flow rate.	Abort or retry with reduced QOS requirements.

*(continued)*



*(continued)*

<b>Error value</b>	<b>Description</b>	<b>Application response</b>
<b>Policy Errors</b>		
GQOS_POLICY_GLOBAL_UNK_USER_SUM_FLOW_RATE	Policy error: the operation would exceed the unknown user-policy total flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_DEF_SUM_FLOW_RATE	Policy error: the operation would exceed the subnet default-policy total flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_GRP_SUM_FLOW_RATE	Policy error: the operation would exceed the subnet group-policy total flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_UNK_USER_SUM_FLOW_RATE	Policy error: the operation would exceed the subnet unknown user-policy total flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_USER_SUM_FLOW_RATE	Policy error: the operation would exceed the subnet user-policy total flow rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_DEF_SUM_PEAK_RATE	Policy error: the operation would exceed the global default-policy total peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_GRP_SUM_PEAK_RATE	Policy error: the operation would exceed the global default-policy total peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_USER_SUM_PEAK_RATE	Policy error: the operation would exceed the global default-policy total peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_GLOBAL_UNK_USER_SUM_PEAK_RATE	Policy error: the operation would exceed the global default-policy total peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_DEF_SUM_PEAK_RATE	Policy error: the operation would exceed the subnet default-policy total peak rate.	Abort or retry with reduced QOS requirements.

Error value	Description	Application response
<b>Policy Errors</b>		
GQOS_POLICY_SUBNET_GRP_SUM_PEAK_RATE	Policy error: the operation would exceed the subnet default-policy total peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_USER_SUM_PEAK_RATE	Policy error: the operation would exceed the subnet default-policy total peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_SUBNET_UNK_USER_SUM_PEAK_RATE	Policy error: the operation would exceed the subnet default-policy total peak rate.	Abort or retry with reduced QOS requirements.
GQOS_POLICY_UNKNOWN_USER	Policy error: the user is unknown.	Check the user's identification and security attributes.
GQOS_POLICY_NO_PRIVILEGES	Policy error: the user has no privilege.	Abort. Possible shut down on sender or receiver.
GQOS_POLICY_EXPIRED_USER_TOKEN	Policy error: the user identification token has expired.	Abort or retry.
GQOS_POLICY_NO_RESOURCES	Policy error: LPM out of resources (memory).	Abort or retry at a later time.
GQOS_POLICY_PRE_EMPTED	Policy error: the operation was pre-empted by a higher priority request.	Abort or retry at a later time.
GQOS_POLICY_USER_CHANGED	Policy error: user identification has changed after the reservation was approved.	Abort.
GQOS_POLICY_NO_ACCEPTS	Policy error: the operation was rejected by all policy modules.	Abort.
GQOS_POLICY_NO_MEMORY	Policy error: LPM out of memory.	Abort or retry at a later time.
GQOS_POLICY_CRAZY_FLOWSPEC	Policy error: invalid <b>FLOWSPEC</b> .	Check the <b>FLOWSPEC</b> structure.
GQOS_POLICY_ERROR_USERID	Unable to understand the user ID	Abort.

(continued)

*(continued)*

<b>Error value</b>	<b>Description</b>	<b>Application response</b>
<b>RSVP Errors</b>		
GQOS_NO_PATH	No matching path state for the reservation request.	Check the QOS call sequence.
GQOS_NO_SENDER	No sender information for the reservation request.	Check the QOS call sequence.
GQOS_BAD_STYLE	Mismatch in Resv style.	Check the RESV filter specifications.
GQOS_UNKNOWN_STYLE	The Resv style is unknown.	Check the RESV filter specifications.
GQOS_BAD_DSTPORT	Conflicting or invalid destination port.	Check the QOS call sequence.
GQOS_BAD_SNDPORT	Conflicting or invalid source port.	Check the QOS call sequence.
GQOS_AMBIG_FILTER	Ambiguous filter specification in RESV.	Check the RESV filter specifications.
GQOS_PREEMPTED	Service preempted due to a higher priority reservation.	Try to invoke QOS again at a later time.
GQOS_UNKN_OBJ_CLASS	Invalid RSVP syntax in objects	Abort.
GQOS_UNKNOWN_CTYPE	Invalid RSVP syntax in objects	Abort.
GQOS_INVALID	Invalid operation or parameters.	
<b>API Errors</b>		
GQOS_API_BADSEND		
GQOS_API_BADRECV		
GQOS_API_BADSPORT		
<b>Traffic Control System Errors</b>		
GQOS_TC_GENERIC	Generic error.	
GQOS_TC_INVALID		
GQOS_NO_MEMORY	Not enough memory available to execute the requested RSVP/traffic control operation.	Abort the operation or retry at a later time.

Error value	Description	Application response
<b>Traffic Control System Errors</b>		
GQOS_BAD_ADDRESSTYPE	Traffic control error: invalid address type.	Check the address type of the socket.
GQOS_BAD_DUPLICATE		
GQOS_CONFLICT		
GQOS_NOTREADY		
GQOS_WOULDBLOCK	RSVP/traffic control operation would block.	Retry at a later time.
GQOS_INCOMPATIBLE		
GQOS_BAD_SDMODE		
GQOS_BAD_QOSPRIORITY	Traffic control error: invalid internal priority.	Check the traffic control priority object.
GQOS_BAD_TRAFFICCLASS		
GQOS_NO_SYS_RESOURCES	Traffic control error: out of system resources.	Abort or retry at a later time.
<b>RSVP System Errors</b>		
GQOS_OTHER_SYS		
GQOS_MEMORY_SYS		
GQOS_API_SYS		
GQOS_SETQOS_NO_LOCAL_APPS		
<b>Traffic Control Errors</b>		
GQOS_CONFLICT_SERV	Conflicting traffic control filters.	Check the QOS specifications.
GQOS_NO_SERV	The service is unknown to local traffic control.	Check the <i>Service Type</i> parameter.
GQOS_BAD_FLOWSPEC		
GQOS_BAD_TSPEC		
GQOS_BAD_ADSPEC		
<b>WSAIoctl Errors</b>		
GQOS_IOCTL_SYSTEMFAILURE		
GQOS_IOCTL_NOBYTESRETURNED		
GQOS_IOCTL_INVALIDSOCKET		

(continued)

*(continued)*

Error value	Description	Application response
<b>SIO_SET_QOS Errors</b>		
GQOS_SETQOS_BADINBUFFER		
GQOS_SETQOS_BADFLOWSPEC		
GQOS_SETQOS_COLLISION		
GQOS_SETQOS_ BADPROVSPECBUF		
GQOS_SETQOS_ILLEGALOP		
GQOS_SETQOS_INVALIDADDRESS		
GQOS_SETQOS_OUTOFMEMORY		
GQOS_SETQOS_EXCEPTION		
GQOS_SETQOS_BADADDRLEN		
GQOS_SETQOS_NOSOCKNAME		
GQOS_SETQOS_IPTOSFAIL		
GQOS_SETQOS_ OPENSESSIONFAIL		
GQOS_SETQOS_SENDFAIL		
GQOS_SETQOS_RECVFAIL		
GQOS_SETQOS_ BADPOLICYOBJECT		
GQOS_SETQOS_ UNKNOWNFILTEROBJ		
GQOS_SETQOS_BADFILTERTYPE		
GQOS_SETQOS_BADFILTERCOUNT		
GQOS_SETQOS_BADOBJLENGTH		
GQOS_SETQOS_BADFLOWCOUNT		
GQOS_SETQOS_UNKNOWNPSOBJ		
GQOS_SETQOS_BADPOLICYOBJ		
GQOS_SETQOS_BADFLOWDESC		
GQOS_SETQOS_ BADPROVSPECOBJ		
GQOS_SETQOS_NOLOOPBACK		
GQOS_SETQOS_ MODENOTSUPPORTED		
GQOS_SETQOS_ MISSINGFLOWDESC		

Error value	Description	Application response
<b>SIO_GET_QOS Errors</b>		
GQOS_GETQOS_BADOUTBUFFER		
GQOS_GETQOS_SYSTEMFAILURE		
GQOS_GETQOS_EXCE;TION		
GQOS_GETQOS_INTERNALFAILURE		
<b>SIO_CHK_QOS Errors</b>		
GQOS_CHKQOS_BADINBUFFER		
GQOS_CHKQOS_BADOUTBUFFER		
GQOS_CHKQOS_SYSTEMFAILURE		
GQOS_CHKQOS_INTERNALFAILURE		
GQOS_CHKQOS_BADPARAMETER		
GQOS_CHKQOS_EXCEPTION		

## Service Types

Service types enable an application to specify service quality requirements for a QOS-enabled connection. The availability of different service types enables the RSVP SP to categorize the QOS required by the application, and thereby decide the type of requests it (the RSVP SP) should make to complementary components on behalf of the requesting application.

There are four primary service types included in the RSVP SP.

Applications must choose which service type is most appropriate for their transmission requirements, based on traffic characteristics, performance requirements, user preferences, or any other criterion that influences how data should be transmitted. Also note that when an application invokes the RSVP SP to initiate QOS for a given connection, the **ServiceType** member of the **FLOWSPEC** structure (where the service type is specified) for both the **SendingFlowspec** and **ReceivingFlowspec** members of the **QOS** structure *must* be specified.

Specifying either CONTROLLED LOAD or GUARANTEED for the *ServiceType* parameter implicitly invokes QOS service for the corresponding direction of the QOS enabled connection.

In addition to the primary service types, the RSVP SP provides secondary service types that enable application programmers and the RSVP SP to monitor or modify service provisions under certain circumstances, and on an ongoing basis.

There are three secondary service types.

## Primary Service Types

The following four service types are considered primary service types:

- BEST EFFORT
- CONTROLLED LOAD
- GUARANTEED
- QUALITATIVE

### BEST EFFORT

The BEST EFFORT service type instructs the RSVP SP to use the application's QOS parameters as guidelines for service quality requests, and make reasonable effort to maintain the requested level of service. The BEST EFFORT service type does not make any guarantees that requested QOS parameters will be implemented or enforced, but can be used by senders to specify traffic control objects.

### CONTROLLED LOAD

When the CONTROLLED LOAD service type is specified, the RSVP SP is instructed to provide end-to-end service quality that approximates the network behavior achieved from BEST EFFORT service *under unloaded conditions*.

The result of specifying the CONTROLLED LOAD service type is that network devices and elements in the path between the QOS-enabled connection (the end-to-end path) can be expected to:

- Deliver a very high percentage of packets (packet loss approximates basic packet error rates for the transmission medium).
- Transit delay by a very high percentage of transmitted packets will not greatly exceed the minimum transit delay experienced by any successfully delivered packet at the speed of light.

These defining characteristics of the CONTROLLED LOAD service type are based on definitions provided by RFC 2210.

### GUARANTEED

When the GUARANTEED service type is specified, the RSVP SP and other QOS components included in the transmission of packets attempt to guarantee the level of service quality defined by the application's provided QOS parameters. With the GUARANTEED service type, queuing algorithms isolate an application's data *flow* (a flow is the unidirectional transmission of packets on a QOS-enabled connection) to provide the following service characteristics.

- The application's flow is isolated from other flows as much as possible.
- The application's flow is guaranteed the ability to transmit data at the **TokenRate** for the duration of the connection.
- If the application does not exceed **TokenRate** over time, latency is also guaranteed.

The GUARANTEED service type is designed for applications that require precisely known service quality (QOS parameters), but would not benefit from better service. An example of such an application is a real-time control system application. The GUARANTEED service type is also designed to transmit within a specific delay bound.

## QUALITATIVE

The QUALITATIVE service type is suited for applications that require better than BEST EFFORT service, but are unable to quantify their requirements. Applications that request QUALITATIVE service can supply an application ID policy object. Policy servers on the network use information in the application ID object to identify the application's flow and assign it an appropriate quality of service. For more information on how to insert an application ID, see the Microsoft White Paper titled *Inserting Application and Sub-application IDs* or the IETF Internet Draft on APP ID, draft-ietf-rap-rsvp-appid-00.txt.

## Secondary Service Types

The use of secondary service types enable an application programmer to modify QOS service guarantees in different ways, such as enabling QOS for only one direction of a bidirectional connection, disabling an existing QOS service guarantee, or disabling traffic control. Each of the following secondary service types are explained in this section:

- `SERVICETYPE_NOTRAFFIC`
- `SERVICETYPE_GENERAL_INFORMATION`
- `SERVICETYPE_NOCHANGE`

There are also two secondary service types that an application programmer can use in conjunction with the all primary service types, and the previously mentioned secondary service types. Application programmers can use the bitwise OR operator with these following two service type values to further refine the service type behavior they require:

- `SERVICE_NO_TRAFFIC_CONTROL`
- `SERVICE_NO_QOS_SIGNALING`

## SERVICETYPE\_NOTRAFFIC

Use of the `SERVICETYPE_NO_TRAFFIC` service type indicates (or specifies) that no QOS services are required in the associated direction. For example, if the **SendingFlowspec** specifies `SERVICETYPE_NOTRAFFIC`, then traffic sent on the specified connection would not receive QOS service provisions. In other words, QOS signaling for sent data would be disabled.



This service type is useful when QOS service guarantees are only required in one direction (and therefore *not* required in the opposite direction). The `SERVICETYPE_NOTRAFFIC` service type is also useful when QOS service provisions are *no longer* required in a specific direction, in which case the `SERVICETYPE_NOTRAFFIC` service type can be used in conjunction with the `SERVICETYPE_NOCHANGE` service type to disable QOS signaling in one direction, and leave the other direction's QOS signaling unchanged.

## **SERVICETYPE\_GENERAL\_INFORMATION**

The `SERVICETYPE_GENERAL_INFORMATION` service type is used in the `SendingFlowspec` parameter of a **QOS** structure to indicate that the sender can operate properly within any of the services. Note that `SERVICETYPE_GENERAL_INFORMATION` does not include any indication regarding the availability of the `QUALITATIVE` service type.

`SERVICETYPE_GENERAL_INFORMATION` is only available to the sender; the RSVP SP returns an error if the receiver attempts to use the `SERVICETYPE_GENERAL_INFORMATION` service type.

It is worth noting that `BEST EFFORT` is almost always available, since it makes no particular guarantee for service quality. The RSVP SP will do its best to meet service level requests for `BEST EFFORT` flows. Therefore, the `SERVICETYPE_GENERAL_INFORMATION` can more effectively be interpreted as advertising that either `CONTROLLED LOAD` or `GUARANTEED` service types can be selected by the receiver for the flow.

Note, however, that routers in the path between end nodes may indicate that they do not support one or both of the (interesting) service types—`CONTROLLED LOAD` and `GUARANTEED`.

## **SERVICETYPE\_NOCHANGE**

The `SERVICETYPE_NOCHANGE` service type is useful when making changes to QOS parameters for one direction of a given flow. Specifically, the `SERVICETYPE_NOCHANGE` enables application developers to indicate or implement changes in one direction of a flow, without having to respecify unchanged QOS parameters for the `SERVICETYPE_NOCHANGE`-specified direction of the flow.

## **SERVICE\_NO\_TRAFFIC\_CONTROL**

The `SERVICE_NO_TRAFFIC_CONTROL` service type instructs the RSVP SP not to invoke kernel traffic control. This secondary service type can be invoked using the bitwise OR operator with the following primary and secondary service types to further specify an application's quality of service requirements:

- `BEST EFFORT`
- `CONTROLLED LOAD`
- `GUARANTEED`

- QUALITATIVE
- SERVICETYPE\_NOTRAFFIC
- SERVICETYPE\_GENERAL\_INFORMATION
- SERVICETYPE\_NOCHANGE

## SERVICE\_NO\_QOS\_SIGNALING

The SERVICE\_NO\_QOS\_SIGNALING service type may be set by the sending or receiving application. When this service type is specified, the RSVP SP does not invoke RSVP signaling, enabling the application to suppress the RSVP SP's invocation of RESV messages on the application's behalf.

This secondary service type can be invoked using the bitwise OR operator with the following primary and secondary service types to further specify an application's quality of service requirements:

- BEST EFFORT
- CONTROLLED LOAD
- GUARANTEED
- QUALITATIVE
- SERVICETYPE\_NOTRAFFIC
- SERVICETYPE\_GENERAL\_INFORMATION
- SERVICETYPE\_NOCHANGE

## Using Service Types

The following sections provide information on how to use service types in your application, including the directional implications of each of the primary and secondary service types, as well as a usage example.

### Directional Implications of Service Types

The impact of specifying each of these service types differs depending on whether it is sent in the *SendingFlowspec* or *ReceivingFlowspec*. To best illustrate the impact of specifying any of these service types, the following table explains the implications of each, depending on whether the service type is specified in the *SendingFlowspec* or the *ReceivingFlowspec*.

<b>Service Types</b>	<b>In <i>SendingFlowspec</i></b>	<b>In <i>ReceivingFlowspec</i></b>
BESTEFFORT	Indicates that only best-effort service is supported for this traffic flow.	Requests best-effort service (no RSVP signaling treatment).
NOTRAFFIC	Indicates that there will be no traffic in this direction.	Indicates that there will be no traffic in this direction.
CONTROLLEDLOAD	Indicates that only controlled load service is supported for this traffic flow.	Ask the network for a controlled load reservation.
GUARANTEED	Indicates that only guaranteed service is supported for this traffic flow.	Ask the network for a guaranteed reservation.
QUALITATIVE	Indicates that only qualitative service is requested for this traffic flow.	Indicates that qualitative service is being requested for the traffic flow.
GENERAL_INFORMATION	Indicates that all service types are supported or this traffic flow.	N/A
NO_CHANGE	Allows an application to modify QOS in the receiving direction, while leaving the sending unchanged, or to provide <b>ProviderSpecific</b> parameters without altering values previously specified in the <i>flowspec</i> .	Allows an application to modify QOS in the sending direction, while leaving the receiving direction unchanged, or to provide <b>ProviderSpecific</b> parameters without altering values previously specified in the <i>flowspec</i> .

Note that a sending application can specify a *ServiceType* parameter in the *SendingFlowspec* set to CONTROLLED\_LOAD or GUARANTEED if it wants to limit the service type options presented to the receiver to just one service type. If the *SendingFlowspec* contains a service type of GENERAL\_INFORMATION then both service types will be advertised as available from the sender (although intervening routers may indicate that they do not support one or both service types).

## Examples of Setting the Service Type

The first example is a sending application that requires GUARANTEED service type, but does not want kernel traffic control. In this case, the application should set the *ServiceType* in its *SendingFlowspec* to the following:

```
SERVICETYPE_GUARANTEED | SERVICE_NO_TRAFFIC_CONTROL
```

## Using the ProviderSpecific Buffer

Most application developers will find that the RSVP SP and its functions provide all the necessary programmatic access to QOS parameters and settings. However, some applications may find it necessary to provide information or parameters that are not available through the **FLowsPEC** structure. The **ProviderSpecific** buffer enables application developers to provide special QOS information, such as RSVP style, Resv confirmation request, or traffic control parameters and settings.

The **ProviderSpecific** buffer is a member of the **QOS** structure, and is of type **WSABUF**.

### Structure of the ProviderSpecific Buffer

The **ProviderSpecific** buffer includes a length field, and a pointer to a buffer. The buffer may include multiple objects, and each object must contain the following, in the order shown:

- A type field that identifies the object.
- A length field that contains the length of the object, including the header.
- The object data itself.

Note that all objects referenced in the **ProviderSpecific** buffer must be contained within the same piece of contiguous buffer memory (the entire **QOS** structure is contained within a contiguous block of memory).

### Use of the ProviderSpecific Buffer as a Receiver

Receivers can use the **ProviderSpecific** buffer to do the following:

While making QOS requests, receivers can:

- Specify nondefault RSVP reservation style, flow descriptors, and filter specifications.
- Request RESERVE CONFIRMATION.
- Specify one or more policy elements (policy information) to RSVP.

While receiving notification, such as getting additional information from the sender, receivers can:

- Retrieve policy information.
- Retrieve QOS and RESV\_CONFIRM events.

### Use of the ProviderSpecific Buffer as a Sender

Senders can use the **ProviderSpecific** buffer to do the following.

While making QOS requests, senders can:

- Specify one or more policy elements (policy information) to RSVP.
- Specify traffic control parameters such as shape/discard mode.
- Specify policy information.

While receiving notification, such as getting additional information from the sender, senders can:

- Retrieve QOS events.
- Retrieve policy information.

## Understanding Traffic Control

Traffic control, commonly referred to as TC, regulates traffic on local hosts. Traffic control is responsible for controlling the flow of traffic based on a given flow's priority, both from an internal perspective (within the kernel itself), and from a network perspective (prioritization and queuing of packets based on transmission priority).

The following sections detail how traffic control is invoked, how traffic control settings can be checked, and how to programmatically modify how traffic control treats a given flow.

---

**Note** Ownership of TC objects, such as flows and settings, are specific to a given application. As such, one application cannot inherit flow or filter objects from another object, nor can an application change any such flows or settings it does not own. Accordingly, all TC objects are deleted when an application (process) dies.

---

### How the RSVP SP Invokes TC

The RSVP service and TC communicate in order to work together to provide overall QOS for a given sending flow. When an application requests QOS using the RSVP SP, the RSVP SP responds by initiating RSVP signaling and invoking kernel traffic control from local TC components (using the traffic control Interface). As such, traffic control and RSVP signaling are initiated concurrently upon flow setup.

In this transitional period (presuming **QOS\_OBJECT\_SD\_MODE** has not been specifically set) the reservation has not been established, and therefore traffic control is configured to transmit traffic associated with the flow's specification, as follows:

- **BEST\_EFFORT** traffic is transmitted with its **QOS\_SD\_MODE** set to borrow mode (**TC\_NONCONF\_BORROW**)
- **CONTROLLED\_LOAD** traffic is transmitted with its **QOS\_SD\_MODE** set to **TC\_NONCONF\_BORROW**
- **GUARANTEED** traffic is shaped with its **QOS\_SD\_MODE** set to **TC\_NONCONF\_SHAPE**

---

**Note** The above default configuration can be overridden by supplying the **QOS\_OBJECT\_SD\_MODE** object in the **ProviderSpecific** buffer.

---

In this transitional period, all packets conforming to the *flowspec* for **BEST\_EFFORT** and **CONTROLLED\_LOAD** are sent immediately with the appropriate traffic control marking, and nonconforming packets are sent immediately with their host and network priority demoted. Transmission settings for any given flow are aligned with the allowed sending rate specified by the system (which is in turn determined by settings in the appropriate **FLOWSPEC**).

Sending applications can determine what the allowed sending rate is by querying the **Allowed\_Rate** using **SIO\_CHK\_QOS**. More information about using **SIO\_CHK\_QOS** is provided in the section titled *Using SIO\_CHK\_QOS*.

Once **PATH** messages arrive at the receiving host (or hosts; for simplicity, ongoing references will be singular), the host may request QOS provisioning for the flow in the form of **RESV** messages sent back toward the sender. When the **RSVP SP** on the receiver receives the **RESV** message, it communicates with traffic control to enable traffic control to update its transmission information, and if appropriate, to modify the **BEST\_EFFORT** flow according to the reservation indicated in the **RESV** message.

Note that it is possible to invoke traffic control without **RSVP** signaling, and to programmatically modify traffic control's treatment of a flow. **RSVP** signaling can be suppressed on a given flow if the **SERVICE\_NO\_QOS\_SIGNALING** service type is specified, and traffic control can be suppressed on a given flow if the **SERVICE\_NO\_TRAFFIC\_CONTROL** service type is specified. Additional control over the suppressing of **RSVP** signaling and traffic control can be achieved through the use of registry entries. For more information on controlling QOS through the use of registry entries, consult the *Windows 2000 Resource Kit*, available from Microsoft Press.

## Using SIO\_CHK\_QOS

**SIO\_CHK\_QOS** can be used in conjunction with the **WSAIoctl** function to obtain information on QOS traffic characteristics. During the transitional phase on the sending system between flow setup and the receipt of a **RESV** message (see *How the RSVP SP Invokes TC* for more information on the transitional phase), traffic associated with the flow is shaped based on service type (**BEST\_EFFORT**, **CONTROLLED\_LOAD**, or **GUARANTEED**). When the **RSVP** service on the sending host receives the **RESV** message, it communicates with traffic control to modify the **BEST\_EFFORT** flow according to the change in **ServiceType** in the **FLOWSPEC**, and traffic control objects received in the **RESV** message.

Certain applications may find the results of shape/discard mode settings unacceptable, such as a **CONTROLLED\_LOAD** flow's nonconforming packets (in borrow mode) potentially being discarded. To avoid unnecessary discarding of packets, applications can use **SIO\_CHK\_QOS** immediately following QOS invocation. The results of the call may require that the application defer data transmission until receipt of the **RESV** message.

Note that the default setting for SIO\_CHK\_QOS allows the application to send data as **BEST\_EFFORT** before the reservation is in place. Network administrators in a given QOS-enabled network environment must explicitly override this setting on the ACS in order to exercise this control.

## Disabling Traffic Control

Traffic control is invoked by default, and is done so immediately upon the receipt of a sending **FLowsPEC**. Application programmers, however, can disable traffic control by using the **SERVICE\_NO\_TRAFFIC\_CONTROL** service type. When the **SERVICE\_NO\_TRAFFIC\_CONTROL** service type is used, traffic control is not invoked on the specified flow regardless of RSVP signaling. If QOS is reset (modified) without the **SERVICE\_NO\_TRAFFIC\_CONTROL** flag, traffic control is invoked.

## QOS Events

Windows 2000 QOS makes QOS event information available to applications that register interest in obtaining event-related QOS information. Windows 2000 QOS makes the following event categories available to applications:

- Policy-based or administratively based information that impacts QOS provisioning for a given flow.
- Errors that occur upon setup, or during the life of a QOS-enabled flow.
- Information regarding acceptance or rejection of QOS requests by the RSVP module or by the network. Keep in mind that rejection of a QOS request may indicate a transient failure, which may be subsequently corrected.
- Significant changes in the service quality provided by the network (when in contrast with previously negotiated QOS parameters), such as from flow preemption in the network.
- Status regarding the presence of a QOS peer to send or receive a particular flow.

Windows 2000 QOS provides this status information through the **FD\_QOS** event suite. Applications that take advantage of QOS capabilities, whether sending or receiving data, should use **FD\_QOS** events to maintain and monitor their QOS-enabled application. Registering interest in QOS events is done by listening for **FD\_QOS** event notifications.

It is important to realize that all **FD\_QOS** events are *edge-triggered*. With edge-triggered events, a message is posted exactly *once* when a quality of service change occurs. Further messages are *not* forthcoming until the provider detects a further change in quality of service, or the application re-negotiates the flow's QOS.

For more information about event notification, consult the following knowledge base article:

[support.microsoft.com/support/kb/articles/Q196/3/60.ASP](http://support.microsoft.com/support/kb/articles/Q196/3/60.ASP)

## Listening for FD\_QOS Events

QOS status is provided through FD\_QOS events. Applications that either send or receive QOS-enabled data can listen for FD\_QOS events by either of the following mechanisms:

- Register for FD\_QOS events using the **WSAAsyncSelect** or **WSAEventSelect** function.
- Perform an overlapped **WSAIoctl(SIO\_GET\_QOS)** function call.

Each of these approaches is explained in the following sections.

## Using WSAEventSelect or WSAAsyncSelect

Applications that register their interest in receiving FD\_QOS events can do so by enabling asynchronous event notification with either the **WSAAsyncSelect** or **WSAEventSelect** function. Information about how to do so can be found in the Windows Sockets 2 documentation.

When registered for event notification using this mechanism, and an event notification occurs, an application can look up the status code (by using the **WSAEnumNetworkEvents** function, for example) and subsequently issue a **WSAIoctl(SIO\_GET\_QOS)** function call to retrieve the **QOS** structure associated with the event.

The associated **QOS** structure contains the current QOS parameters. Applications should inspect the QOS parameters to determine the extent of the changes associated with the event notification. There are a couple of issues to consider when working with FD\_QOS events in this manner:

- You must issue the **WSAIoctl(SIO\_GET\_QOS)** to reenable the FD\_QOS event.
- There may be multiple QOS status indications waiting for retrieval. Use the **WSAIoctl(SIO\_GET\_QOS)** function call in a loop until **SOCKET\_ERROR** is returned.

Applications can also register their interest in FD\_QOS events using overlapped **WSAIoctl(SIO\_GET\_QOS)**, as explained in *Using Overlapped WSAIoctl(SIO\_GET\_QOS)*.

## Using Overlapped WSAIoctl(SIO\_GET\_QOS)

Applications can register their interest in FD\_QOS events by issuing an overlapped **WSAIoctl(SIO\_GET\_QOS)** function call. When an application takes this approach, an FD\_QOS event invokes the completion function specified in the *CompletionRoutine* parameter of the **WSAIoctl** function call, and the updated **QOS** structure is made



available in its output buffer. With this approach, the application could be developed to use the *CompletionRoutine* to act on the contents of the output buffer. The **WSAIoctl(SIO\_GET\_QOS)** function request completes with QOS information that corresponds only to one direction (either the **SendingFlowspec** or the **ReceivingFlowspec** is valid, but not both). The **FLOWSPEC** that is invalid has its **ServiceType** value set to **SERVICETYPE\_NOCHANGE**.

---

**Note** Sending applications cannot call **SIO\_GET\_QOS** until a connection has completed. However, a receiving application can call **SIO\_GET\_QOS** as soon as it is bound. When using overlapped **WSAIoctl(SIO\_GET\_QOS)** to monitor **FD\_QOS** events, the **RSVP SP** also passes an **RSVP\_STATUS\_INFO** object in the **ProviderSpecific** buffer of the updated **QOS** structure. This enables applications to obtain extended status information about the **FD\_QOS** event.

---

**Note** If the output buffer associated with the **WSAIoctl** function call is not sufficiently sized to contain the full **QOS** structure and the **RSVP\_STATUS\_INFO** object that is included in its **ProviderSpecific** buffer, the application will get **WSA\_ENOBUFS** and can reissue the query. The required size is returned as an unsigned integer at the beginning of the output buffer.

---

## QOS Event Codes

Applications that use the **FD\_QOS** event suite to monitor QOS events have access to status and error codes associated with the event, as well as updated QOS parameters (in the **QOS** structure associated with the event). The following is a list of common QOS-related Windows Sockets 2 status and error codes.

Event or error code	Definition
<b>WSA_QOS_RECEIVERS</b>	One or more <b>RESV</b> message has arrived.
<b>WSA_QOS_SENDERS</b>	One or more <b>PATH</b> message has arrived.
<b>WSA_QOS_NO_SENDERS</b>	There are no senders.
<b>WSA_QOS_NO_RECEIVERS</b>	There are no receivers.
<b>WSA_QOS_REQUEST_CONFIRMED</b>	Reservation has been confirmed.
<b>WSA_QOS_ADMISSION_FAILURE</b>	Error due to lack of resources.
<b>WSA_QOS_POLICY_FAILURE</b>	Rejected for administrative reasons.
<b>WSA_QOS_BAD_STYLE</b>	Unknown or conflicting style.
<b>WSA_QOS_BAD_OBJECT</b>	Problem with some part of the <b>FLOWSPEC</b> .
<b>WSA_QOS_TRAFFIC_CTRL_ERROR</b>	Problem with some part of the filter specification.
<b>WSA_QOS_GENERIC_ERROR</b>	General error.

For additional status and error codes, consult the **Winsock2.h** header file.

## RSVP SP and RSVP

Below the RSVP SP sits the Resource Reservation Protocol, or RSVP. RSVP is an IETF-standardized protocol that ferries quality of service provision requests between end nodes, and interacts with all RSVP-enabled network devices in the path between end nodes. The RSVP SP—which invokes and facilitates all aspects of Windows 2000 QOS, not just RSVP signaling—also enables application developers to fine-tune RSVP messages through the use of the **ProviderSpecific** buffer. Such fine-grained control of RSVP by an application enables the fine-tuning or special service requests to be made without depending on the RSVP SP to interpret conventional requests (through members in the **QOS** structure) and pass such requests down to RSVP.

RSVP signaling is the primary mechanism employed by the RSVP SP to create an end-to-end QOS connection. When **ServiceType** in either the **SendingFlowspec** or **ReceivingFlowspec** member of the **QOS** structure is set to initiate Quality of Service over the connection in either direction (that is, when either parameter is set to a service type other than **SERVICETYPE\_BESTEFFORT** or **SERVICETYPE\_NOTRAFFIC**), the RSVP SP initiates RSVP signaling.

Most application programmers will find that enabling an application to use the RSVP SP, which automatically initiates and maintains RSVP signaling on behalf of applications, is sufficient to enable their application to take advantage of QOS capabilities.

For those interested in the specifics of RSVP signaling, and how to get more granular control over its parameters and notifications, the following sections outline general RSVP concepts as they interact with the RSVP SP.

## Basic RSVP Operations

This section introduces basic RSVP operations, including:

- How RSVP signaling is invoked by the RSVP SP.
- How RSVP filters are defined and applied to RSVP SP Service Type specifications.

### Invoking RSVP

There are two ways to invoke RSVP:

- Using defaults, through the RSVP SP.
- Overriding defaults, by using the **RSVP\_RESERVE\_INFO** object.

RSVP signaling is invoked automatically when an application invokes the RSVP SP to provide QOS reservations on its behalf. The RSVP SP provides QOS reservations on behalf of an application when the service type in either the **SendingFlowspec** or **ReceivingFlowspec** members of the **QOS** structure is set to a service type other than **SERVICETYPE\_BESTEFFORT** or **SERVICETYPE\_NOTRAFFIC**. Note that there are a number of individual APIs that can invoke the RSVP SP. For more information, see *Invoking the RSVP SP*.

Another mechanism that an application developer can use to invoke RSVP, and to gain access to advanced RSVP features available in the **QOS** structure, is through the **ProviderSpecific** buffer in the **QOS** structure.

The **ProviderSpecific** buffer can be used in conjunction with the **RSVP\_RESERVE\_INFO** object. When specific RSVP-based reservation information is specified in the **RSVP\_RESERVE\_INFO** object, RSVP SP-supplied default information is superseded by the information, enabling application developers to fine-tune the way RSVP handles reservation requests (and responses). For more information on using the **RSVP\_RESERVE\_INFO** object, see *Using the RSVP\_RESERVE\_INFO Object*. For more information about the **FLOWSPEC** structure, including which of its members can be defaulted, see **FLOWSPEC**.

## Using the RSVP\_RESERVE\_INFO Object

The **RSVP\_RESERVE\_INFO** object enables application developers to specify granular QOS parameters directly to RSVP, providing a mechanism for fine-tuning RSVP reservations. To implement the object, you must pass a filled **RSVP\_RESERVE\_INFO** object to the RSVP SP using the **ProviderSpecific** buffer. The following members can be fine-tuned with the **RSVP\_RESERVE\_INFO** object:

- RSVP Filter Style
- RESV Confirmation Request
- Policy Objects
- List of flow descriptors

## Confirming RSVP Reservations

The **RSVP\_RESERVE\_INFO** object enables a receiving application to be notified of the outcome of an RSVP reservation request. RSVP reservation confirmation is achieved by setting the **ConfirmRequest** member of the **RSVP\_RESERVE\_INFO** object to a nonzero value. This setting is necessary because RSVP network nodes are not required to automatically generate RSVP CONFIRMATION messages, per the RSVP specification.

Until the presence of a reservation is discerned (senders learn about the presence of a reservation by listening for a **WSA\_QOS\_RECEIVERS** event), data for the connection will be treated as **BEST\_EFFORT** traffic, which provides a compelling reason for network programmers to enable RSVP confirmation immediately after a **WSAConnect**, **WSAJoinLeaf** function call, or use of the **SIO\_SET\_QOS** IOCTL.

Senders can query whether their application is allowed to send by using **SIO\_CHK\_QOS**. If the binary result of **ALLOWED\_TO\_SEND** is false (indicating that the application is not allowed to send) and the application chooses to send anyway, the traffic is treated as **BEST\_EFFORT**. Once the RESV message arrives at the sender (triggering the RSVP confirmation), the RSVP SP on the sending host modifies traffic control service to match the service type in the RESV message, and the traffic is treated as per the service type.

RSVP confirmation requests are only useful for receiving applications.

## Disabling RSVP Signaling

Both receiving applications and senders may disable RSVP signaling on a per-flow basis (a flow is a unidirectional QOS-enabled connection). Disabling RSVP signaling for a given flow is done by using the bitwise OR operator in the `SERVICE_NO_QOS_SIGNALING` flag with the value in the `ServiceType` field of the `ReceivingFlowspec` member of the `QOS` structure.

## RSVP Reservation Styles

RSVP recognizes three reservation styles that define how RSVP-enabled network devices set up reservations along the path between an end-to-end QOS-enabled connection. RSVP reservation styles are also sometimes called reservation styles. The three RSVP reservation styles are:

- Fixed Filter (FF)
- Wildcard Filter (WF)
- Shared Explicit (SE)

RSVP reservation styles either establish a distinct reservation for each upstream sender, or make a single reservation that is shared among all senders' packets. The RSVP SP automatically selects an appropriate RSVP-reservation style based on the type of connection (unicast or multicast) as part of its RSVP invocation. Application programmers can override the RSVP SP's selection of reservation style, overriding the setting that the RSVP SP provides, through use of the `Style` member of the `RSVP_RESERVE_INFO` object. For more information on overriding this setting, see *Overriding Default RSVP Filter Style Settings*.

## Base RSVP Reservation Styles

RSVP recognizes three base reservation styles. Note that not all RSVP reservation styles are available without direct interaction (through the `ProviderSpecific` buffer) with RSVP.

### RSVP Fixed Filter

The fixed filter (FF) RSVP reservation style implies a distinct reservation and an explicit sender. This contrasts with other reservation styles in that a reservation is made for individual senders, and the reservation is not shared with any other sender.

The fixed filter style is appropriate (and its definition implied) for use with unicast sessions. The RSVP SP uses the fixed filter for all TCP receivers, and for unicast UPD receivers attempting to establish a QOS-enabled connection.

### RSVP Wildcard Filter

The wildcard filter (WF) RSVP reservation style implies that a single reservation is shared among all senders in the session. The wildcard filter style is appropriate for use with applications such as audio conferencing, for example, in which only one or two speakers at a time can speak (in order for the audio conference to be meaningful); in which case only one reservation is necessary to service all participants.

### RSVP Shared Explicit

The shared explicit (SE) RSVP reservation style is a hybrid of the other two RSVP reservation styles; with the shared explicit style, the receiving application is making a reservation which can be shared (explicitly) by selected senders. The shared explicit style is appropriate for applications such as a video conference in which, unlike the WF reservation, only participants who are explicitly listed benefit from the RSVP reservation.

The RSVP SP does not use the shared explicit filter by default. Applications that want to use the shared explicit filter must set the **Style** member of the **RSVP\_RESERVE\_INFO** object to **RSVP\_SHARED\_EXPLICIT\_STYLE**, and must specify the list of senders in *flowdescriptors*.

It is important to note that the number of senders included in any individual Shared Explicit reservation should be limited to a reasonable number of senders (such as 10). If more than approximately 10 senders will be included in a given reservation, senders should use the WF reservation style.

### Default RSVP Filter Style Settings

The RSVP SP sets RSVP reservation style based on the type of socket on which the reservation request is based. Default settings are applied to their corresponding connection types when the RSVP SP makes RSVP signaling requests on behalf of an application when there is no reservation, and when **RSVP\_DEFAULT\_STYLE** is specified in the **Style** member of the **RSVP\_RESERVE\_INFO** object.

Note, however, that these default settings can be overridden by setting the **Style** member of the **RSVP\_RESERVE\_INFO** object to one of the other available RSVP filter styles.

### Unicast Receivers

Applications that request QOS service provisions for unicast sessions are assumed to require the fixed filter (FF) RSVP filter style. While TCP receivers are always assumed to be unicast receivers, only UDP unicast receivers that use the **WSAConnect** function are provided with the Fixed Filter RSVP filter style; otherwise, UDP unicast receivers are provided with the wildcard filter (WF) RSVP reservation style.

### Multicast Receivers

Applications that request QOS service provisions for multicast sessions are assumed to require the wildcard filter (WF) RSVP reservation style.

### Use of the **RSVP\_DEFAULT\_STYLE** Filter Style

The **RSVP\_DEFAULT\_STYLE** filter style enables an application developer to specify parameters in other members of the **RSVP\_RESERVE\_INFO** object without having to explicitly indicate (or override) the RSVP filter style applied by the RSVP SP.

When the connection is TCP or connected UDP, specifying **RSVP\_DEFAULT\_STYLE** implements the fixed filter (FF) style.

When the connection is multicast or unconnected UDP, specifying **RSVP\_DEFAULT\_STYLE** implements the wildcard filter (WF) style.

### Overriding Default RSVP Filter Style Settings

Default RSVP filter style settings can be overridden through the use of the **RSVP\_RESERVE\_INFO** object by specifying one of the following values in the **Style** member of the **RSVP\_RESERVE\_INFO** object, along with required *flowdescriptors*.

#### The **RSVP\_FIXED\_FILTER\_STYLE** Object

The **RSVP\_FIXED\_FILTER\_STYLE** object may be specified in the **Style** member of the **RSVP\_RESERVE\_INFO** object to override default settings of the wildcard filter (WF) RSVP filter style setting for multicast or unconnected UDP unicast receivers. This capability is useful when a multicast session has multiple senders; rather than accepting the default WF reservation that provides a shared reservation among all senders, you can use **RSVP\_FIXED\_FILTER\_STYLE** to set up individual reservations for each sender. Another similar situation is when the receiver is using unconnected UDP to receive from multiple senders, in which a **RSVP\_FIXED\_FILTER\_STYLE** can again be used to set up individual reservations for each sender.

Note that *flowdescriptors* must be specified when using the **RSVP\_FIXED\_FILTER\_STYLE** object, and the **NumFlowDesc** member of **RSVP\_RESERVE\_INFO** is set to the number of senders, and **FlowDescList** of **RSVP\_RESERVE\_INFO** is set to each sender's address/port. This is only appropriate when multiple senders are involved.

#### The **RSVP\_WILDCARD\_STYLE** Object

The **RSVP\_WILDCARD\_STYLE** object may be specified in the **Style** member of the **RSVP\_RESERVE\_INFO** object to override default settings of the fixed filter (FF) RSVP filter style setting for unicast receivers.

Note that *flowdescriptors* must not be specified when using the **RSVP\_WILDCARD\_STYLE** object, such that the **NumFlowDesc** member of **RSVP\_RESERVE\_INFO** is set to zero, and **FlowDescList** of **RSVP\_RESERVE\_INFO** is set to NULL.

#### The **RSVP\_SHARED\_EXPLICIT\_STYLE** Object

The **RSVP\_SHARED\_EXPLICIT\_STYLE** object may be specified in the **Style** member of the **RSVP\_RESERVE\_INFO** object to override default RSVP filter style settings. The **RSVP\_SHARED\_EXPLICIT\_STYLE** object is used to create an RSVP Shared Explicit

(SE) reservation (see *Base RSVP Reservation Styles*). Note that the only way to create connections that use the shared explicit (SE) RSVP filter style is through this mechanism.

The **RSVP\_SHARED\_EXPLICIT\_STYLE** object cannot be applied to TCP receivers or to connected UDP receivers.

When the **RSVP\_SHARED\_EXPLICIT\_STYLE** is used, the flow's resources are shared between all senders listed in the *FilterSpec*. Also, when using the **RSVP\_SHARED\_EXPLICIT\_STYLE** object, *flowdescriptors* must be specified, such that the **NumFlowDesc** member of **RSVP\_RESERVE\_INFO** is set to 1, and **FlowDescList** of **RSVP\_RESERVE\_INFO** is not set to NULL.

It is important to note that the number of senders included in any individual Shared Explicit reservation should be limited to a reasonable number of senders (such as 10). If more than 10 senders will be included in a given reservation, senders should use the WF reservation style.

### Generating Multiple Fixed Filter Reservations in a Single Reservation

It may be useful in certain situations to enable a receiver to reserve mutually exclusive flows for multiple, explicitly identified sources. This is achieved by generating multiple fixed filter (FF) RSVP reservations in a single reservation.

To do this, specify **RSVP\_FIXED\_FILTER\_STYLE** in the **Style** member of the **RSVP\_RESERVE\_INFO** object, followed by a list of multiple *flowdescriptors*. So for  $n$  explicitly identified sources, the **NumFlowDesc** member of the **RSVP\_RESERVE\_INFO** object is set to  $n$ , and the **FlowDescList** member of the **RSVP\_RESERVE\_INFO** object is set to  $n$  sender addresses/ports.

Note that this cannot be done with TCP receivers, as TCP receivers are assumed connected to a single-peer sender. This should not be applied to UDP receivers that have been connected using the **WSAConnect** function. In these cases, the transport discards data from all senders other than the sender specified in the **WSAConnect** function call.

## Mapping RSVP SP Parameters to RSVP

In order to provide quality of service parameters from the RSVP SP to RSVP, there must be some mechanism to translate parameters that the application provides in the **QOS** structure to RSVP. Passing RSVP-requisite service quality parameters, based on information provided by the application to the RSVP SP, is done through mapping.

The information provided to RSVP is derived from the **SendingFlowspec** and **ReceivingFlowspec** members of the **FLOWSPEC** structure, itself a member of the **QOS** structure.

## RSVP PATH and RESV Messages

RSVP establishes QOS-enabled connections through the use of PATH and RESV messages. A short explanation of each, and how they pertain to Windows 2000 QOS and the RSVP SP is merited. For a thorough explanation and discussion of RSVP PATH and RESV messages, consult IETF RFC 2205.

When a QOS-enabled connection is established and RSVP signaling is triggered (see *Invoking RSVP*), the sender and receiver(s) play specific roles in the establishment of an RSVP session:

- The sender emits PATH messages toward the receiver (or receivers).
- The receiver waits until the PATH message corresponding to the flow arrives, then issues a RESV message.

The information contained in the PATH and RESV messages is derived from the **FLOWSPEC** structures associated with the **SendingFlowspec** and **ReceivingFlowspec** members of the **QOS** structure.

### Transmission of RSVP PATH and RESV Messages

When a PATH message is received, the RSVP SP creates an RSVP session and associates a PATH state (based on the PATH message) with the RSVP session. The RSVP SP sends RESV messages once it determines that the PATH state exists for a session that matches a socket for which receiving QOS is indicated. In the absence of a received PATH message, the RSVP SP creates an RSVP session using QOS parameters (and thereby, QOS is invoked) on a receive socket. In this latter case, PATH state is not associated with the session until a matching PATH message is received.

The transmission of RESV messages, therefore, may be triggered by the following circumstances:

- Upon receipt of a PATH message that matches the session associated with a preexisting socket.
- Upon creation of a socket that matches the session associated with a preexisting PATH state.

### RSVP PATH Message Parameters

RSVP PATH messages derive their RSVP sender *Tspec* from the **SendingFlowspec** member of the **QOS** structure (the **SendingFlowspec** member of **QOS** is itself a **FLOWSPEC** structure).

The following table outlines the information required to begin transmission of RSVP PATH messages, and how the information is derived from the **QOS** structure or other information provided by the sender. The RSVP PATH parameters discussed serve the following purposes.



- *SenderTspec* contains QOS parameters for sent traffic.
- *SenderTemplate* contains the sender's address.
- *Session* contains the destination of the sent traffic.

RSVP PATH parameter	Equivalent receiver-based parameters
<i>SenderTspec</i>	<b>SendingFlowspec</b> member of the <b>QOS</b> structure.
<i>SenderTemplate</i>	Source IP address/port to which sending socket is bound.
<i>Session</i>	Destination IP address/port and protocol identifier to which the socket is sending ( <b>sockaddr_in</b> ).

Note that the RSVP session parameter includes specification of the protocol identifier can be UDP or TCP. The type of socket on which the QOS-enabled connection is being invoked determines the protocol identifier.

### RSVP RESV Message Parameters

RSVP RESV messages derive their RSVP **FLOWSPEC** parameters from the **ReceivingFlowspec** member of the **QOS** structure (the **ReceivingFlowspec** member of **QOS** is itself a **FLOWSPEC** structure).

The following table outlines the information required to begin transmission of RSVP RESV messages, and how the information is derived from the **QOS** structure or other information provided by the receiver. The RSVP RESV parameters discussed serve the following purposes:

- *Flowspec* contains desired QOS parameters for traffic to be received.
- *Filterspec* contains the source or sources from which QOS-enabled traffic will be received.
- *Session* contains the destination of the sent traffic.

RSVP RESV parameter	Derived from the following Winsock parameter
<i>flowspec</i>	<b>ReceivingFlowspec</b> member of the <b>QOS</b> structure or the <b>ProviderSpecific</b> buffer.
<i>Filterspec</i> (source(s) from which QOS traffic will be received)*.	Address(es) of peer(s) from which the socket is receiving.
<i>Session</i> (destination of sent traffic).	Local IP address and port to which the receiving socket is bound (unicast), or multicast session address to which the socket has joined (multicast).

\* Strictly speaking, *RESV* messages can be generated without knowledge of the sender's address. These type of *RESV* messages are said to be WF style, meaning that they apply to all senders in the session.

The WF reservation style RESV messages do not have any *filterspecs*, so receivers need not supply the sender an IP address.

## Tspec, FlowSpec, and Adspec

RSVP transmits request information for a QOS-enabled connection with RSVP PATH and RESV messages. Within such PATH and RESV messages, certain values are used to represent traffic and requested QOS parameters that enable a sender and receiver to establish service quality parameters for a given flow:

- The sender *Tspec* (T representing traffic) specifies parameters available for the flow. Both senders and receivers use *Tspec* (*SenderTspec* and *ReceiverTspec*, respectively).
- The **FLOWSPEC** specifies requested QOS parameters, and is used by the receiver in RESV messages.
- The *Adspec* (ad for advertisement) enables QOS-enabled network devices in the path between sender and receiver to advertise their service capabilities, resource availability, and transmission characteristics.

### RSVP Tspec

Both senders and receivers use *Tspec*, as part of *SenderTspec* and *Receiverflowspec*, respectively.

Sender provides the *Tspec* to describe the traffic it will originate, and the receiver provides the *flowspec* to describe the reservation it needs.

The RSVP *Tspec* derives its parameters from the **SendingFlowspec** member of a **QOS** structure (**SendingFlowspec** is of type **FLOWSPEC**).

The following table explains how members in **SendingFlowspec** map to **Tspec** parameters.

<b>SendingFlowspec Parameter</b>	<b>Tspec</b>
<i>TokenRate</i>	<i>TokenBucketRate</i>
<i>TokenBucketSize</i>	<i>TokenBucketSize</i>
<i>PeakBandwidth</i>	<i>PeakRate</i>
<i>MinimumPolicedSize</i>	<i>MinimumPolicedUnit</i>
<i>MaxSduSize</i>	<i>MaximumPacketSize</i>
<i>DelayVariation</i>	
<i>Latency</i>	

### SenderTspec Specifics

The following items are specific to a sender's use of the RSVP *Tspec* in PATH messages:

- If *MaxSduSize* is not specified in the *Tspec* included in the sender's PATH message, the RSVP SP will default to a value of 1,500 bytes.
- If *MinimumPolicedSize* is not specified in the *Tspec* included in the sender's PATH message, the RSVP SP will default to a value of 128 bytes.

## Receiver Flowspec Specifics

For receivers, the contents of the RSVP *flowspec* varies depending on requested service type in the following ways:

- CONTROLLED LOAD service contains a *Tspec*.
- GUARANTEED service *flowspec* contains a *Tspec* and an *Rspec*.

### CONTROLLED LOAD Service

Receivers requesting CONTROLLED LOAD service may decide to specify only (but at least) the **ServiceType** parameter in **ReceivingFlowspec**, setting remaining parameters to QOS\_UNSPECIFIED, in which case the RSVP SP copies the *SenderTspec* from the matching PATH message into the *ReceiverFlowspec* sent with the corresponding RESV message. In order to specify additional parameters in the *ReceiverFlowspec* (allowing remaining flow parameters set to QOS\_UNSPECIFIED to be derived from the *SenderTspec*), an application may provide values for other **ReceivingFlowspec** parameters.

### GUARANTEED Service

When a receiver requests GUARANTEED service, the RSVP SP copies *SenderTspec* from the matching PATH message into the *ReceiverFlowspec* sent with the corresponding RESV message.

### RSVP Rspec

The RSVP *Rspec* specifies requested QOS parameters, and is used by the receiver in RESV messages to transmit requested reservation parameters *only* when GUARANTEED service is specified by the application. The *Rspec* derives its parameters from the **ReceivingFlowspec** member of a QOS structure (**ReceivingFlowspec** is of type FLOWSPEC).

The following table explains how parameters in **ReceivingFlowspec** map to *Tspec* parameters.

ReceivingFlowspec parameter	Rspec
<i>TokenRate</i>	<i>Rate</i>
<i>TokenBucketSize</i>	
<i>PeakBandwidth</i>	
<i>MinimumPolicedSize</i>	
<i>MaxSduSize</i>	
<i>DelayVariation</i>	<i>DelaySlackTerm</i>
<i>Latency</i>	

The application specifying GUARANTEED service is expected to provide two of the three parameters, in the following combinations.

- *TokenRate*
- *DelayVariation*
- *DelayVariation*
- *Latency*

If an application fails to specify two of the three required parameters, the RSVP SP infers appropriate values based on the corresponding PATH message(s). The following table explains how the provided parameters are used by the RSVP SP to construct the RSVP *Rspec*.

Application specifies:	RSVP SP constructs <i>Rspec</i> :
<i>TokenRate</i> and <i>DelayVariation</i>	<i>Rate</i> is copied from <i>TokenRate</i> <i>DelaySlackTerm</i> is copied from <i>DelayVariation</i> <i>Latency</i> parameter ignored
<i>DelayVariation</i> and <i>Latency</i>	<i>Rate</i> parameter of <i>Rspec</i> calculated based on <i>DelayVariation</i> and <i>Latency</i> and other parameters obtained from <i>Adspec</i>

### RSVP Adspec

Each RSVP PATH message includes an *Adspec*, which enables QOS-enabled network devices in the path between sender and receiver to advertise the services they support, their resource availability and transmission characteristics. Such information can be useful in helping the receiving application select **FLOWSPEC**.

## Mapping QOS Call Sequences to RSVP

RSVP signaling is invoked once a series of QOS calls that initiate QOS activity are called. Such call sequences tend to differ based on whether an application is a sender or receiver, and for that reason, senders and receivers are discussed separately.

Often, applications will be both sender and receiver, in which case both sender and receiver call sequences may apply.

### Sending Applications

In order for the RSVP SP to invoke RSVP processing and signaling, the following information must be available:

- Peer address (the receiver's address). This enables RSVP to create its session object.
- Source address (the sender's local address). This enables RSVP to create its **SenderTemplate** object.
- QOS parameters in the form of a **SendingFlowspec** (a member of the **QOS** structure). This enables RSVP to generate its sender *Tspec* object.

The means by which the RSVP SP derives information necessary to satisfy those three requirements is outlined in the following table.

<b>Sender type</b>	<b>Session information derived from:</b>	<b>SenderTemplate derived from:</b>
UDP unicast sender	Destination address and port specified in the <i>name</i> parameter of the <b>WSAConnect</b> function.	Local port determined by a RSVP SP call to the <b>getsockname</b> function. For sockets bound explicitly by the application, to a specific address, the RSVP SP calls the <b>getsockname</b> function to get the local address/port. For sockets bound by the application to INADDR_ANY, the RSVP SP gets the local address by issuing a routing interface query on the destination address (as obtained from the application's call to the <b>WSAConnect</b> or <b>WSAJoinLeaf</b> function).
UDP multicast sender	Multicast IP address and port as specified in the <i>name</i> parameter of the <b>WSAJoinLeaf</b> function.	Same as above.
TCP unicast sender	Peer (destination) address and port as determined by a call to the <b>getpeername</b> function, following connection establishment.	Local address and local port determined by call to the <b>getsockname</b> function following connection establishment.

Details of each of these sender types are outlined individually in the following reference pages:

- UDP unicast senders
- UDP multicast senders
- TCP unicast senders

### **UDP Unicast Senders**

The RSVP SP derives information for the initiation of RSVP processing and signaling for UDP unicast senders based on the following table.

Sender type	Session information derived from:	<i>SenderTemplate</i> derived from:
UDP unicast sender	Destination address and port specified in the <i>name</i> parameter of the <b>WSAConnect</b> function.	Local port determined by a RSVP SP call to the <b>getsockname</b> function. For sockets bound explicitly by the application, to a specific address, the RSVP SP calls the <b>getsockname</b> function to get the local address. For sockets bound by the application to <code>INADDR_ANY</code> , the RSVP SP gets the local address by issuing a routing interface query on the destination address (as obtained from the application's call to the <b>WSAConnect</b> or <b>WSAJoinLeaf</b> function).

Unicast UDP senders typically call the **WSAConnect** function to invoke RSVP processing and signaling. For sockets that have been bound using `INADDR_ANY`, the RSVP SP uses the peer address to determine the local address to use *SenderTemplate* by issuing a routing interface query to the underlying transport service provider. This approach returns the address of the local interface used to reach the specified peer.

Typically, the **WSAConnect** function call includes sending QOS parameters (**SendingFlowspec**). Alternatively, the sending application may use the **WSAIoctl(SIO\_SET\_QOS)** function/opcode call to provide sending QOS parameters to the RSVP SP, either before or after the **WSAConnect** function call.

RSVP processing begins as soon as the RSVP SP knows the peer address (from which it may also determine the local bound address) and the sending QOS parameters.

Application programmers can also choose to use unconnected UDP sockets using **Wsalocctl(SIO\_SET\_QOS)** by specifying a **QOS\_OBJECT\_DESTADDR** object in the **ProviderSpecific** buffer of the **QOS** structure. In this scenario, the session information is derived from the **QOS\_OBJECT\_DESTADDR** object.

---

**Note for unconnected UDP sockets** An application may choose to use unconnected UDP sockets by specifying a **QOS\_OBJECT\_DESTADDR** object in the **ProviderSpecific** buffer of the **QOS** structure. In this case, the session information is derived from the **QOS\_OBJECT\_DESTADDR** object.

---

## UDP Multicast Senders

The RSVP SP derives information for the initiation of RSVP signaling for UDP multicast senders based on the following table.

<b>Sender type</b>	<b>Session information derived from:</b>	<b>SenderTemplate derived from:</b>
UDP multicast sender	Multicast IP address and port as specified in the <i>name</i> parameter of the <b>WSAJoinLeaf</b> function.	Local port determined by RSVP SP call to the <b>getsockname</b> function. For sockets bound explicitly by the application, to a specific address, the RSVP SP calls the <b>getsockname</b> function to get the local address. For sockets bound by the application to INADDR_ANY, the RSVP SP gets the local address by issuing a routing interface query on the destination address (as obtained from the application's call to the <b>WSAConnect</b> or <b>WSAJoinLeaf</b> function).

Multicast UDP senders typically call the **WSAJoinLeaf** function to invoke RSVP signaling. The **WSAJoinLeaf** function call provides the destination multicast session address, and may also be used to provide QOS parameters through its LPQOS parameter. If QOS parameters are not provided with the call to **WSAJoinLeaf**, they must be provided separately with a call to the **WSAIoctl(SIO\_SET\_QOS)** function/opcode pair. The RSVP session object included in the corresponding PATH messages is derived from the multicast session address.

For sockets that have been bound using INADDR\_ANY, the RSVP SP uses the multicast session address to determine the local address used in *SenderTemplate* by issuing a routing interface query to the underlying transport service provider. This approach returns the address of the local interface used to reach the specified peer. RSVP processing begins as soon as the RSVP SP knows the peer address and the sending QOS parameters.

### TCP Unicast Senders

The RSVP SP derives information for the initiation of RSVP processing and signaling for TCP unicast senders based on the following table.

<b>Sender type</b>	<b>Session information derived from:</b>	<b>SenderTemplate derived from:</b>
TCP unicast sender	Peer (destination) address and port as determined by a call to the <b>getpeername</b> function, following connection establishment.	Local address and local port determined by call to the <b>getsockname</b> function following connection establishment.

Generally, sockets are connected through the interaction of an active and passive peer; the active peer issues a **WSAConnect** function call, and the passive peer issues a **WSAAccept** function call. In most circumstances the receiver is the active peer, the sender the passive peer. This page assumes the sender is the passive peer; for cases where the sender is the active peer, refer to TCP unicast receivers.

Upon calling the **WSAAccept** function, the TCP unicast sender can gather QOS parameters through the **WSAAccept** function's callback function, but in order to do so, the application must complete the callback function with status `CF_ACCEPT`. The RSVP SP does not use the **ProviderSpecific** buffer when calling the **WSAAccept** function's callback function.

Applications may alternatively set QOS parameters on the socket (or any parameters that require use of the **ProviderSpecific** buffer) through the use of the **WSAIoctl**(`SIO_SET_QOS`) function/opcode call. However, if the application has previously associated QOS parameters with the socket by calling the **WSAIoctl**(`SIO_SET_QOS`) function/opcode pair, completion of the callback function may modify QOS parameters unless the *ServiceType* parameters in the corresponding **FLOWSPEC** structures are set to `SERVICETYPE_NOCHANGE`.

The application may also set QOS parameters on the listening socket prior to the call to **WSAAccept**, and these settings are inherited by the accepted socket, but any parameters set in the **WSAAccept** function's callback function take precedence. RSVP processing begins as soon as the RSVP SP knows the peer address, the address to which the socket is locally bound, and the sending QOS parameters.

## Receiving Applications

For the RSVP SP to invoke RSVP processing and signaling, the receiving application is required to provide the following information:

- QOS parameters, through the use of the **ReceivingFlowspec** member of the **QOS** structure. Receiving applications generally provide such QOS parameters in a call to the **WSAConnect** or **WSAJoinLeaf** functions, or in a call to the **WSAIoctl**(`SIO_SET_QOS`) function/opcode pair.

For receiving applications, the RSVP SP must create a session object, and in certain cases must create a *filterspec* object (*filterspec* selects a subset of the senders by specifying their address) for placement into its RESV messages. These objects are generally matched to the PATH state (which itself is derived from corresponding PATH messages). In order to limit matching of the PATH state, the RSVP SP needs the following information:

- Peer address or addresses (the source address). This enables RSVP to limit RSVP sessions for which RESV messages are sent.
- Local address. The RSVP SP uses the local address to match received PATH messages and to select the interface on which to send out the RESV message.



Transmission of RESV messages begins as soon as the RSVP service has sufficient information.

Details of each of these receiver types are outlined individually in the following sections:

- UDP unicast receivers
- UDP multicast receivers
- TCP unicast receivers

### UDP Unicast Receivers

UDP unicast receivers can initiate a QOS-enabled connection by providing QOS parameters to the RSVP SP using either of the following methods:

- A call to the **WSAConnect** function
- A call using the **WSAIoctl(SIO\_SET\_QOS)** function/opcode pair.

#### Using WSAConnect

A UDP unicast receiver should use the **WSAConnect** function only if it wants to receive from a single sender; using **WSAConnect** causes traffic received from other senders to be discarded. In order to receive traffic from multiple senders using the **WSAConnect** function, a separate socket must be created for each sender. By using the **WSAConnect** function as a UDP unicast receiver, the application provides a peer address and, in doing so, selects a specific sender.

When a UDP unicast receiver uses the **WSAConnect** function, the RSVP SP sends RESV messages only when the PATH state exists for the specified sender, and uses the included peer address to compose fixed filter style (FF) RESV messages.

Note that using the **WSAConnect** function may cause the RSVP SP to cease sending RESV messages that were triggered in a previous call to the **WSAIoctl(SIO\_SET\_QOS)** function/IOCTL pair; this is because PATH state, though it may have existed for some sender, may not have existed for the sender specified in the **WSAConnect** function call.

#### Using WSAIoctl(SIO\_SET\_QOS)

A UDP unicast receiver can use the **WSAIoctl(SIO\_SET\_QOS)** function/IOCTL pair to indicate receiving QOS parameters, after which the RSVP SP begins transmitting RESV messages for any matching PATH state. Since the **WSAIoctl(SIO\_SET\_QOS)** function/IOCTL pair does not associate a peer address with the socket (unless the ProviderSpecific buffer is used to specify a particular sender), the socket will match PATH state of any sender in this RSVP session.

When the **WSAIoctl(SIO\_SET\_QOS)** function/IOCTL pair is used by a UDP unicast receiver, the RSVP SP transmits wildcard filter (WF) style RESV messages. Note that WF RESV messages are used for unconnected UDP sockets only; connected UDP sockets result in Fixed Filter (FF) style reservations.

### Combining **WSAConnect** and **WSAIoctl(SIO\_SET\_QOS)**

A UDP unicast receiver may call both the **WSAIoctl(SIO\_SET\_QOS)** function/IOCTL pair and the **WSAConnect** function in any order. In either case, the RSVP service begins sending RESV messages as soon as the indicated QOS parameters match a PATH state.

If the **WSAIoctl(SIO\_SET\_QOS)** function/IOCTL pair is called before the **WSAConnect** function, the RSVP SP initially sends wildcard filter (WF) RESV messages, presuming a matching PATH state. Subsequently (upon calling of the **WSAConnect** function, its specified peer, and a matching PATH state) the RSVP SP sends a RESVTEAR message for the WF style reservation followed by fixed filter (FF) style RESV messages.

Calling the **WSAIoctl(SIO\_SET\_QOS)** function/IOCTL pair subsequent to a call to the **WSAConnect** function does not negate the selection of a specific sender. Therefore, while the **WSAIoctl(SIO\_SET\_QOS)** function/IOCTL pair is a useful method of altering RESV messages (such as altering QOS parameters, or terminating RESV messages altogether by indicating BEST\_EFFORT), its use does not cause the RSVP SP to send wildcard filter (WF) style RESV messages.

### UDP Multicast Receivers

UDP multicast receivers are expected to do the following:

- Create UDP sockets using the **WSASocket** function.
- Indicate that they are multicast receivers in the accompanying (**WSASocket**) flags.
- Call the **WSAJoinLeaf** function to indicate the multicast session they want to join.

UDP multicast receivers can indicate QOS parameters either through their call to the **WSAJoinLeaf** function, or through the use of the **WSAIoctl(SIO\_SET\_QOS)** function/IOCTL pair.

The RSVP SP does not send RESV messages for UDP multicast receivers until a multicast session address is unambiguously specified through the use of a **WSAJoinLeaf** function call. The RSVP SP does not use parameters with which the socket is bound; since no peer is specified, the RSVP service sends a WF style RESV message only when it has a PATH state from at least one sender.

A UDP unicast receiver may call both the **WSAIoctl(SIO\_SET\_QOS)** function/IOCTL pair and the **WSAJoinLeaf** function in any order. In either case, the RSVP service begins sending RESV messages as soon as there is a sender in the multicast session. However, with multicast receivers the matching PATH state will only be found if a multicast socket has been created with a matching multicast session address.

### Joining Multiple Multicast Groups on a Single Socket

When a UDP multicast receiver joins multiple multicast groups on a single socket, the RSVP SP sends RESV messages for all groups for which there is a matching PATH state. QOS parameters are obtained separately from the **ReceivingFlowspec** included

in individual **WSAJoinLeaf** function calls made for each multicast group. However, if the **WSAIoctl**(SIO\_SET\_QOS) function/IOCTL pair is called, its specified QOS parameters are applied to all presently joined multicast groups.

### TCP Unicast Receivers

TCP receivers are generally the active peer in a TCP connection. TCP unicast receivers are therefore expected to initiate a QOS-enabled connection by providing QOS parameters to the RSVP SP using one of the following methods:

- A call to the **WSAConnect** function
- A call using the **WSAIoctl**(SIO\_SET\_QOS) function/opcode pair.

The RSVP SP uses the address specified in the **WSAConnect** function call to specify the peer sender's address, which enables RSVP to compose its *filterspec* for fixed filter (FF) style RESV messages. The RSVP SP begins RSVP processing.

RSVP will start signaling as soon as the RSVP SP knows the receiving QOS parameters, and those parameters match a PATH state; PATH state only matches if the associated socket's session address matches the bound address and its **SenderTemplate** matches the peer address.

If the receiving socket is bound using INADDR\_ANY, the bound address cannot be determined until the **WSAConnect** function is called. Under this circumstance, RESV messages are delayed until, following connection establishment, the RSVP SP automatically calls the **getsockname** function to determine local address.

## Receiver Reservation Semantics

The process of joining RSVP sessions includes making appropriate function calls particular to the type of RSVP session an application wants to join. Due to their inherent differences, and the difference in the way each handles sockets, unicast (both TCP and UDP) and multicast sessions are best discussed individually. To generalize, the following are usually true:

- Unicast sessions usually include use of the **WSAConnect** function.
- Multicast sessions generally use the **WSAJoinLeaf** function (and **sendto** for multicast senders).
- Both unicast and multicast can use the **WSAIoctl**(SIO\_SET\_QOS) function/opcode pair.

Each of these common methods for joining RSVP sessions is discussed individually.

### Using WSAConnect to Join Unicast RSVP Sessions

The **WSAConnect** function is generally used by unicast receivers to join RSVP sessions. The behavior is somewhat different with TCP sessions and UDP unicast sessions.

## TCP Sessions

The use of the **WSAConnect** function with RSVP-enabled TCP sessions is fairly simple, if somewhat stringent, because parameters match quite well between parameters associated with the **WSAConnect** function and parameters necessary for RSVP to operate. The following table illustrates which conditions of a TCP socket (left column) correspond to RSVP conditions (right column) that enable the TCP socket to join the session.

TCP Socket is:	RSVP session is joined:
Not bound, or is bound and not connected (the <b>WSAConnect</b> function is not issued)	Never
Bound and connected	If both of the following conditions are true: The port and address specified in any TCP session matches the socket's bound port and address. <i>SenderTemplate</i> , which is specified in PATH state associated with the session, matches the connected peer's port and address.

## UDP Unicast Sessions

The use of the **WSAConnect** function with RSVP-enabled UDP unicast sessions is less stringent than its use with TCP sessions, largely due to the fact that it is not always possible to determine unique addresses associated with a UDP socket. The following table illustrates which conditions of a UDP unicast socket (left column) correspond to RSVP conditions (right column) that enable the UDP unicast socket to join the session. These mappings do not apply to UDP multicast sessions.

UDP unicast socket is:	RSVP session is joined:
Not bound and not connected.	Never
Bound using INADDR_ANY, and not connected (the <b>WSAConnect</b> function is not issued).	If the port specified in any UDP session matches the socket's bound port.
Bound using a specific address and not connected.	If the port and address specified in any UDP session matches the socket's bound port and address.
Bound using INADDR_ANY, and connected.	If both of the following conditions are true: The port specified in any UDP session matches the socket's bound port. <i>SenderTemplate</i> , which is specified in PATH state associated with the UDP session, matches the connected peer's port and address.

(continued)

(continued)

**UDP unicast socket is:**

Bound using a specific address and connected.

**RSVP session is joined:**

If both of the following conditions are true:

The port and address specified in any UDP session matches the socket's bound port and address.

*SenderTemplate*, which is specified in PATH state associated with the session, matches the connected peer's port and address.

## Using WSAJoinLeaf to Join Multicast RSVP Sessions

Multicast UDP receivers are expected to create the multicast UDP socket using the **WSASocket** function, and indicate that they are multicast receivers in the accompanying (**WSASocket**) flags. If multicast UDP sockets don't use the **WSASocket** function (and associated multicast flags), the RSVP SP may not be able to determine that the socket is multicast, and therefore may send undesired RESV messages based on unicast matching rules described in the Using **WSAConnect** to Join unicast RSVP Sessions section.

The following table illustrates which conditions of a UDP multicast socket (left column) correspond to RSVP conditions (right column) that enable the UDP multicast socket to join the session (note these mappings do not apply to UDP unicast sessions).

**UDP multicast socket is:**

**RSVP session is joined:**

Not joined to a specific multicast group (the **WSAJoinLeaf** function is not issued).

Never

Joined to a specific multicast group (using the **WSAJoinLeaf** function).

The multicast address specified in any UDP session matches the multicast address specified in the **WSAJoinLeaf** function call.

Applications are required to use the **WSAJoinLeaf** function call before sending or receiving multicast traffic, and are required to set the *dwFlags* parameter of the **WSAJoinLeaf** function to **JL\_SENDER\_ONLY**, **JL\_RECEIVER\_ONLY** or **JL\_BOTH**, to indicate the direction in which QOS service is requested.

It is important to note that alternate multicast semantics, such as simply calling the **sendto** function with a multicast address (for sending) or using **IP\_ADD\_MEMBERSHIP** (for receiving) do not invoke QOS service, even if the **IOCTL** is issued.

## Use of Sendto and WSASendTo by Multicast Senders

Senders that join multicast sessions using the **WSAJoinLeaf** function are required to call the **sendto** or **WSASendTo** function with the correct multicast session address to send data to the multicast session (even though the multicast session address is already provided with the **WSAJoinLeaf** function call). If the sending application calls the **sendto** or **WSASendTo** function specifying a multicast session address other than the address specified with the **WSAJoinLeaf** function call, the data will not receive QOS provisioning. This requirement is due to the fact that the sender's RSVP service generates the RSVP session based on the multicast session address specified in the call to the **WSAJoinLeaf** function.

Also note that QOS-enabled applications should only call the **sendto** or **WSASendTo** function when acting as a multicast sender. Unicast (UDP or TCP) sender applications must specify their destination address using **WSAConnect**, and it is sufficient for that application to use the **send** or **WSASend** function calls, rather than the **sendto** or **WSASendTo** functions.

## Using WSALocctl(SIO\_SET\_QOS) During RSVP Sessions

The use of **WSALocctl(SIO\_SET\_QOS)** is often unnecessary; the use of connection-oriented Windows Sockets function calls (such as the **WSAConnect** function) is generally sufficient for providing the RSVP SP (and therefore RSVP) with requisite QOS parameters.

One exception is when a UDP application receives from multiple senders, in which case the **WSALocctl(SIO\_SET\_QOS)** function/opcode pair must be used to specify QOS parameters to avoid limiting the socket to receive traffic from a single sender (as the use of a connection-oriented function call such as **WSAConnect** would do). This is a limitation of Windows Sockets 2.

Another exception is when a UDP transmit uses the **sendto** function to transmit data to one or more receivers through an unconnected socket (Microsoft® NetMeeting™ version 2.1 is an example of such an application). In this circumstance, the sending application must do the following in order to invoke QOS provisioning:

- Supply the RSVP SP with one or more appropriate **SendingFlowspec(s)** by issuing one or more **SIO\_SET\_QOS** IOCTLs.
- Provide the RSVP SP with the destination address by issuing a **QOS\_OBJECT\_DESTADDR** object in the **ProviderSpecific** buffer.

The **WSALocctl(SIO\_SET\_QOS)** function/opcode pair is also useful for modifying QOS parameters subsequent to the establishment of the connection with a connection-oriented function call. This functionality also enables an application to separate the specification of QOS parameters from the determination of local and peer addresses implicit in making a connection-oriented function call.



---

## CHAPTER 15

# QOS API Reference

## QOS Functions

This section contains an alphabetical list of QOS functions.

- **WPUGetQOSTemplate**
- **WSAGetQOSByName**
- **WSCInstallQOSTemplate**
- **WSCRemoveQOSTemplate**
- **WSPGetQOSByName**

---

## WPUGetQOSTemplate

The **WPUGetQOSTemplate** function retrieves a QOS template for a particular service provider.

```
INT WPUGetQOSTemplate (  
    const LPGUID IpProviderId,  
    LPWSABUF IpQOSName,  
    LPQOS IpQOS  
);
```

### Parameters

*IpProviderId*

[in] Pointer to a provider selected—globally unique identifier (GUID).

*IpQOSName*

[in] Specifies the QOS template name.

*IpQOS*

[out] Pointer to a **QOS** structure.

### Return Values

If **WPUGetQOSTemplate** succeeds, the return value is zero. If the function fails, the return value is **SOCKET\_ERROR**. For extended error information, call **WSAGetLastError**.



**Remarks**

The **WPUGetQOSTemplate** function retrieves a QOS-named template containing the associated **QOS** structure. If *IpProviderId* is NULL, **WPUGetQOSTemplate** attempts to find the QOS-named template in the global list of QOS names. Otherwise, **WPUGetQOSTemplate** searches the template list specific to the service provider indicated by *IpProviderId*.

The *IpQOS* parameter can include a **ProviderSpecific** buffer for retrieval with the basic **QOS** structure. In this case, the **ProviderSpecific** buffer must be large enough to hold the provider-specific information stored in the template; otherwise **WPUGetQOSTemplate** returns WSAENOBUFFS.

**Error Codes**

Error Code	Meaning
WSAEFAULT	The <i>IpQOS</i> or <i>IpQOSName</i> parameter is not a valid part of the user address space.
WSAEINVAL	The specified <i>IpProviderId</i> is invalid, or the <i>IpQOS</i> template is invalid.
WSA_NODATA	The specified QOS name could not be found.
WSAENOBUFFS	The provider-specific buffer is too small.

**! Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in Qosname.h.

**Library:** Use Qosname.lib.

**+ See Also**

**QOS, WSAGetQOSByName, WSCInstallQOSTemplate, WSCRemoveQOSTemplate**

---

## WSAGetQOSByName

The **WSAGetQOSByName** function initializes a **QOS** structure based on a named template, or retrieves an enumeration of the available template names:

```

BOOL WINAPI WSAGetQOSByName (
    SOCKET s,
    LPWSABUF IpQOSName,
    LPQOS IpQOS
);

```

## Parameters

*s*

[in] Descriptor identifying a socket.

*lpQOSName*

[in, out] Specifies the QOS template name or supplies a buffer to retrieve an enumeration of available template names.

*lpQOS*

[out] Pointer to the **QOS** structure to be filled. The **ProviderSpecific** buffer member of the **QOS** structure must be initialized before calling the **WSAGetQOSByName** function.

## Return Values

If **WSAGetQOSByName** succeeds, the return value is TRUE. If the function fails, the return value is FALSE. For extended error information, call **WSAGetLastError**.

The algorithm that **WSAGetQOSByName** applies in its search for a template's name match is:

- The socket's service provider checks for a provider-specific template—a template installed specifically for the socket's service provider—with a name that matches the service provider's name.
- If that fails, the service provider checks its internal table of QOS templates (if it has such a table).
- If that fails, the service provider checks for a list of global QOS templates.
- If any of the preceding steps succeeds, the service provider can modify the QOS template before returning it to Windows Sockets. Otherwise, **WSAGetQOSByName** returns FALSE, and **WSAGetLastError** returns WSA\_NODATA to indicate an invalid name.

## Remarks

Applications can use **WSAGetQOSByName** to initialize a **QOS** structure with a prescribed set of known values appropriate for a particular service class or media type. These known values are stored in a template, and the template is referenced by a well-known name. For example, if the service provider's DLL is named *lpphone.dll*, the template can be referenced by the name *IPPHONE*. Applications can retrieve these values by setting the **buf** member of **WSABUF**, indicated by *lpQOSName*, to point to a string of nonzero length specifying a template name. When doing so, *lpQOSName* is an [in] parameter only, and results are returned through *lpQOS*.

This function can also be used to retrieve an enumeration of available template names. This is done by setting the **buf** member of **WSABUF**, indicated by *lpQOSName*, to a zero-length, null-terminated string. The buffer indicated by **buf** is then overwritten with a sequence of as many null-terminated template names as are available—up to the number of bytes available in **buf**, as provided by the **len** member of **WSABUF**.

The list of names itself is terminated by a zero-length name. When **WSAGetQOSByName** is used to retrieve template names, the *lpQOS* parameter is ignored.

If two templates have the same name, with one template being specific to the service provider and the other being global, they will appear in the list only once.

### Error Codes

Error Code	Meaning
WSANOTINITIALIZED	A successful call to <b>WSAStartup</b> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lpQOSName</i> or <i>lpQOS</i> parameter is not a valid part of the user address space.
WSAENOBUFS	The buffer length for <i>lpQOS</i> is too small.
WSA_NODATA	The specified QOS template name is invalid.

#### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in Winsock2.h.

**Library:** Use Ws2\_32.lib.

#### See Also

**QOS, WSAAccept, WSABUF, WSACConnect, WSALocctl, WSAStartup**

---

## WSCInstallQOSTemplate

The **WSCInstallQOSTemplate** function installs a QOS template, based on a QOS name. This **QOS** structure (and thus its QOS settings) can be retrieved by calling the **WSPGetQOSByName** function and passing in its associated QOS name. Note that the caller of this function must have administrative rights.

```

BOOL WSCInstallQOSTemplate (
    const LPGUID IpProviderId,
    LPWSABUF lpQOSName,
    LPQOS lpQOS
);

```

## Parameters

*IpProviderId*

[in] Pointer to a provider-selected globally unique identifier (GUID).

*IpQOSName*

[in] Specifies the QOS template name.

*IpQOS*

[in] Pointer to a **QOS** structure.

## Return Values

If **WSCInstallQOSTemplate** succeeds, the return value is TRUE. If the function fails, the return value is FALSE. For extended error information, call **WSAGetLastError**.

## Remarks

The **WSCInstallQOSTemplate** function installs a QOS-named template containing the specified and subsequently associated **QOS** structure, or replaces settings of an existing template. Values are stored in nonvolatile storage, so subsequent calls to **WSAGetQOSByName**, passing the same *IpQOSName*, return the **QOS** structure. If *IpProviderId* is set to NULL, the installed QOS-named template applies across all service providers; otherwise the QOS-named template applies only to the provider indicated by the *IpProviderId* parameter.

Windows Sockets 2 includes a base set of QOS templates. You can override and replace any of these QOS templates or change an existing QOS template by simply installing a new template with the existing name. You do not need to delete an existing template before replacing or modifying it. You cannot delete the base set of QOS-named templates included in Windows Sockets 2. However, you can delete templates added subsequently, perhaps by other service providers.

The *IpQOS* parameter, which points to a **QOS** structure, can include a **ProviderSpecific** buffer that will be stored with the basic **QOS** structure and returned in subsequent **WSAGetQOSByName** calls.

You can provide the **ProviderSpecific** buffer—even if *IpProviderId* is set to NULL—to install global name templates that include provider-specific information. Note that this practice may lead the service provider to ignore the **ProviderSpecific** buffer if the service provider does not recognize its contents. The recommended use of **WSCInstallQOSTemplate** is to include a **ProviderSpecific** buffer only if the named template is being installed on a particular service provider, as implemented when *IpProviderId* is not NULL.

**Error Codes**

Error Code	Meaning
WSAEINVAL	The specified QOS template name or provider identifier is invalid, or the contents of the template structure is invalid or incomplete.
WSEFAULT	One or more of the parameters is not a valid part of the user address space.

**! Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in Qosname.h.

**Library:** Use Qosname.lib.

**+ See Also**

**QOS, WPUGetQOSTemplate, WSAGetQOSByName, WSCRemoveQOSTemplate**

---

## WSCRemoveQOSTemplate

The **WSCRemoveQOSTemplate** function removes a QOS template. Note that the caller of this function must have administrative rights.

```

BOOL WSCRemoveQOSTemplate (
    const LPGUID IpProviderId,
    LPWSABUF IpQOSName
);

```

**Parameters**

*IpProviderId*

[in] Pointer to a provider-selected globally unique identifier (GUID).

*IpQOSName*

[in] Specifies the QOS template name.

**Return Values**

If **WSCRemoveQOSTemplate** succeeds, the return value is TRUE. If the function fails, the return value is FALSE. For extended error information, call **WSAGetLastError**.

**Remarks**

The **WSCRemoveQOSTemplate** function deletes a QOS template previously installed with **WSCInstallQOSTemplate**. If *IpProviderId* is NULL, **WSCRemoveQOSTemplate** attempts to find and delete a QOS template from the global list of QOS names. Otherwise, **WSCRemoveQOSTemplate** attempts to find and delete a QOS template

specific to the service provider associated with *IpProviderId*. You cannot delete the base set of QOS names included with Windows Sockets 2. The `WSAEINVAL` error code will be returned if such an attempt is made.

## Error Codes

Error Code	Meaning
<code>WSAEINVAL</code>	The specified QOS template name is invalid.
<code>WSA_NODATA</code>	The specified QOS template could not be found.
<code>WSEFAULT</code>	One or more of the parameters is not a valid part of the user address space.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in `Qosname.h`.

**Library:** Use `Qosname.lib`.

### + See Also

**WPUGetQOSTemplate, WSAGetQOSByName, WSCInstallQOSTemplate**

## WSPGetQOSByName

The QOS **WSPGetQOSByName** function initializes a **QOS** structure based on a named template, or retrieves an enumeration of the available template names.

```

BOOL WINAPI WSPGetQOSByName (
    SOCKET s,
    LPWSABUF lpQOSName,
    LPQOS lpQOS,
    LPINT lpErrno
);

```

### Parameters

*s*

[in] Descriptor identifying a socket.

*lpQOSName*

[in, out] Specifies the QOS template name or supplies a buffer to retrieve an enumeration of available template names.

*lpQOS*

[out] Pointer to the **QOS** structure to be filled.

*lpErrno*

[out] Pointer to the error code.

## Return Values

If **WSPGetQOSByName** succeeds, the return value is TRUE. If the function fails, the return value is FALSE. Error information is available in *lpErrno*.

The algorithm that **WSPGetQOSByName** applies in its search for a template's name match is:

- The service provider checks for a provider-specific template by calling **WPUGetQOSTemplate**, using the service provider's globally unique identifier (GUID) and QOS name as search criteria.
- If that fails, the service provider checks its internal table of QOS templates (if it has such a table).
- If that fails, the service provider calls **WPUGetQOSTemplate** again, using the same QOS name, but using NULL for the GUID to facilitate a query of the global list of QOS names.
- If any of the preceding steps succeeds, the service provider can modify the QOS template before returning it to Windows Sockets. Otherwise, **WSPGetQOSByName** returns FALSE, and *lpErrno* is set to WSA\_NODATA.

## Remarks

Applications can use this function to initialize a **QOS** structure with a prescribed set of known values appropriate for a particular service class or media type. These known values are stored in a template, and the template is referenced by a well-known name. Applications can retrieve these values by setting the **buf** member of **WSABUF**, indicated by *lpQOSName*, to point to a string of nonzero length specifying a template name. When doing so, *lpQOSName* is an [in] parameter only, and results are returned through *lpQOS*.

This function can also be used to retrieve an enumeration of available template names. This is done by setting the **buf** member of **WSABUF**, indicated by *lpQOSName*, to a zero-length, null-terminated Unicode string. The buffer indicated by **buf** is then overwritten with a sequence of as many null-terminated Unicode template names as are available—up to the number of bytes available in **buf**, as provided by the **len** member of **WSABUF**. The list of names itself is terminated by a zero-length Unicode name string. When **WSPGetQOSByName** is used to retrieve template names, the *lpQOS* parameter is ignored.

## Error Codes

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.

Error Code	Meaning
WSAEFAULT	The <i>IpQOSName</i> or <i>IpQOS</i> parameter is not a valid part of the user address space.
WSAENOBUFS	The buffer length for <i>IpQOS</i> is too small.
WSA_NODATA	The specified QOS template name is invalid.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in *Ws2spi.h*.

### + See Also

**QOS, WPUGetQOSTemplate, WSABUF, WSCInstallQOSTemplate, WSCRemoveQOSTemplate**

## QOS Structures

This section describes the following Quality of Service structures:

- **FLOWSPEC**
- **QOS**

## FLOWSPEC

The **FLOWSPEC** structure provides QOS parameters to the RSVP SP. This allows QOS-aware applications to invoke, modify, or remove QOS settings for a given flow.

Some members of **FLOWSPEC** can be set to default values. See *Remarks* for more information.

```
typedef struct _flowspec {
    uint32    TokenRate;           // In bytes/second
    uint32    TokenBucketSize;    // In bytes
    uint32    PeakBandwidth;      // In bytes/second
    uint32    Latency;            // In microseconds
    uint32    DelayVariation;     // In microseconds
    SERVICE_TYPE ServiceType;     // Guaranteed,
                                   // predictive, etc.
    uint32    MaxSduSize;         // In bytes
    uint32    MinimumPolicedSize // In bytes
} FLOWSPEC, *PFLOWSPEC, FAR *LPFLOWSPEC;
```



## Members

### TokenRate

Specifies the permitted rate at which data can be transmitted over the life of the flow. **TokenRate** is similar to other token bucket models seen in such WAN technologies as Frame Relay, in which the token is analogous to a credit. If such tokens are not used immediately, they accrue to allow data transmission up to a certain periodic limit (**PeakBandwidth**, in the case of Windows 2000 QOS). Accrual of credits is limited, however, to a specified amount (**TokenBucketSize**). Limiting total credits (tokens) avoids situations where, for example, flows that are inactive for some time flood the available bandwidth with their large amount of accrued tokens. Because flows may accrue transmission credits over time (at their **TokenRate** value) only up to the maximum of their **TokenBucketSize**, and because they are limited in burst transmissions to their **PeakBandwidth**, traffic control and network-device resource integrity are maintained. Traffic control is maintained because flows cannot send too much data at once, and network-device resource integrity is maintained because such devices are spared from high traffic bursts.

With this model, applications can transmit data only when sufficient credits are available. If sufficient credits are not available, the application must either wait or discard the traffic (based on the value of **QOS\_OBJECT\_SD\_MODE**). Therefore it is important that applications base their **TokenRate** requests on reasonable expectations for transmission requirements. For example, in video applications, **TokenRate** is typically set to the average bit rate from peak to peak.

If **TokenRate** is set to **QOS\_NOT\_SPECIFIED** on the receiver only, the maximum transmission unit (MTU) is used for **TokenRate**, and limits on the transmission rate (the token bucket model) will not be put into effect. **TokenRate** is expressed in bytes per second.

**TokenRate** cannot be set to zero. **TokenRate** cannot be set as a default (that is, set to **QOS\_NOT\_SPECIFIED**) in a sending **FLWSPEC**.

### TokenBucketSize

The maximum amount of credits a given direction of a flow can accrue, regardless of time. In video applications, **TokenBucketSize** will likely be the largest average frame size. In constant rate applications, **TokenBucketSize** should be set to allow for small variations. **TokenBucketSize** is expressed in bytes.

### PeakBandwidth

The upper limit on time-based transmission permission for a given flow, sometimes considered a *burst limit*. **PeakBandwidth** restricts flows that may have accrued a significant amount of transmission credits, or tokens from overburdening network resources with one-time or cyclical data bursts, by enforcing a per-second data transmission ceiling. Some intermediate systems can take advantage of this information, resulting in more efficient resource allocation. **PeakBandwidth** is expressed in bytes per second.

**Latency**

Maximum acceptable delay between transmission of a bit by the sender and its receipt by one or more intended receivers. The precise interpretation of this number depends on the level of guarantee specified in the QOS request. **Latency** is expressed in microseconds.

**DelayVariation**

Difference between the maximum and minimum possible delay a packet will experience. Applications use **DelayVariation** to determine the amount of buffer space needed at the receiving end of the flow. This buffer space information can be used to restore the original data transmission pattern. **DelayVariation** is expressed in microseconds.

**ServiceType**

Specifies the level of service to negotiate for the flow. This member can be one of the following defined service types:

Value	Meaning
SERVICETYPE_NOTRAFFIC	Indicates that no traffic will be transmitted in the specified direction. On duplex-capable media, this value signals underlying software to set up unidirectional connections only. This service type is not valid for the TC API.
SERVICETYPE_BESTEFFORT	Results in no action taken by the RSVP SP. Traffic control does create a BESTEFFORT flow, however, and traffic on the flow will be handled by traffic control similarly to other BESTEFFORT traffic.
SERVICETYPE_CONTROLLEDLOAD	<p>Provides an end-to-end QOS that closely approximates transmission quality provided by best-effort service, as expected under unloaded conditions from the associated network components along the data path.</p> <p>Applications that use <b>SERVICETYPE_CONTROLLEDLOAD</b> may therefore assume the following:</p> <ul style="list-style-type: none"> <li>• The network will deliver a very high percentage of transmitted packets to its intended receivers. In other words, packet loss will closely approximate the basic packet error rate of the transmission medium.</li> <li>• Transmission delay for a very high percentage of the delivered packets will not greatly exceed the minimum transit delay experienced by any successfully delivered packet.</li> </ul>

*(continued)*

*(continued)*

Value	Meaning
SERVICETYPE_GUARANTEED	Guarantees that datagrams will arrive within the guaranteed delivery time and will not be discarded due to queue overflows, provided the flow's traffic stays within its specified traffic parameters. This service is intended for applications that need a firm guarantee that a datagram will arrive no later than a certain time after it was transmitted by its source.
SERVICETYPE_QUALITATIVE	Indicates that the application requires better than BESTEFFORT transmission, but cannot quantify its transmission requirements. Applications that use SERVICETYPE_QUALITATIVE can supply an application ID policy object. The application ID policy object enables policy servers on the network to identify the application, and accordingly, assign an appropriate QOS to the request. For more information on application ID, consult the IETF Internet Draft draft-ietf-rap-rsvp-appid-00.txt, or the Microsoft white paper on Application ID. Traffic control treats flows of this type with the same priority as BESTEFFORT traffic on the local computer. However, application programmers can get boosted priority for such flows by modifying the Layer 2 settings on the associated flow using the QOS_OBJECT_TRAFFIC_CLASS QOS object.
SERVICETYPE_NETWORKCONTROL	Used <i>only</i> for transmission of control packets (such as RSVP signaling messages). This <b>ServiceType</b> has the highest priority.
SERVICETYPE_GENERAL_INFORMATION	Specifies that all service types are supported for a flow. Can be used on sender side only.
SERVICETYPE_NOCHANGE	Indicates that the QOS in the transmission using this <b>ServiceType</b> value is not changed. SERVICETYPE_NOCHANGE can be used when requesting a change in the QOS for one direction only, or when requesting a change only within the ProviderSpecific parameters of a QOS specification, and not in the <b>SendingFlowspec</b> or <b>ReceivingFlowspec</b> .
SERVICE_NO_TRAFFIC_CONTROL	Indicates that traffic control should not be invoked in the specified direction.
SERVICE_NO_QOS_SIGNALING	Suppresses RSVP signaling in the specified direction.

The following list identifies the relative priority of **ServiceType** settings:

SERVICETYPE\_NETWORKCONTROL  
SERVICETYPE\_GUARANTEED  
SERVICETYPE\_CONTROLLED\_LOAD  
SERVICETYPE\_BESTEFFORT, SERVICETYPE\_QUALITATIVE  
Non-conforming traffic

For a simple example, if a given network device were resource-bounded and had to choose among transmitting a packet from one of the above **ServiceType** settings, it would first send a packet of SERVICETYPE\_NETWORKCONTROL, and if there were no packets of that **ServiceType** requiring transmission it would send a packet of **ServiceType** SERVICETYPE\_GUARANTEED, and so on.

### MaxSduSize

Specifies the maximum packet size permitted or used in the traffic flow. This member is expressed in bytes.

### MinimumPolicedSize

Specifies the minimum packet size for which the requested QOS will be provided. Packets smaller than this size are treated by traffic control as **MinimumPolicedSize**. The **MinimumPolicedSize** member is expressed in bytes. When using the **FLOWSPEC** structure in association with RSVP, the value of **MinimumPolicedSize** cannot be zero; however, if you are using the **FLOWSPEC** structure specifically with the TC API, you can set **MinimumPolicedSize** to zero.

### Remarks

Many members of the **FLOWSPEC** structure can be set to default values by setting the member to QOS\_NOT\_SPECIFIED. Note that the members that can be set to default values differ depending on whether the **FLOWSPEC** is a receiving **FLOWSPEC** or a sending **FLOWSPEC**.

There are a handful of considerations you should keep in mind when using **FLOWSPEC** with traffic control:

- **TokenRate** can be QOS\_NOT\_SPECIFIED for SERVICETYPE\_NETWORKCONTROL, SERVICETYPE\_QUALITATIVE, and SERVICETYPE\_BESTEFFORT. **TokenRate** must be valid for all other **ServiceType** values.
- If **PeakBandwidth** is specified, it must be greater than or equal to **TokenRate**.

Many settings can be defaulted in a receiving **FLOWSPEC** except **ServiceType**, with the following considerations:

- For a Controlled Load Service receiver, the default values are derived from the sender **TSPEC**.
- For a Guaranteed Service receiver, **ServiceType** and **TokenRate** must be specified.

The following list specifies the values that are applied when a receiving **FLOWSPEC** sets the corresponding values to default:

**TokenRate**

Set to QOS\_NOT\_SPECIFIED when defaulted in a receiving **FLOWSPEC**. The value is set to the *TokenRate* parameter in the sender's **FLOWSPEC**.

**TokenBucketSize**

Set to QOS\_NOT\_SPECIFIED when defaulted in a receiving **FLOWSPEC**. The value is set to the *TokenBucketSize* parameter in the sender's **FLOWSPEC**.

**PeakBandwidth**

Set to QOS\_NOT\_SPECIFIED when defaulted in a receiving **FLOWSPEC**. The value is set to the *PeakBandwidth* parameter in the sender's **FLOWSPEC**.

**Latency**

Set to QOS\_NOT\_SPECIFIED when defaulted in a receiving **FLOWSPEC**. The value is set to the *Latency* parameter in the sender's **FLOWSPEC**.

**Service Type**

Must be specified.

**DelayVariation**

Set to QOS\_NOT\_SPECIFIED when defaulted in a receiving **FLOWSPEC**. The value is set to the *DelayVariation* parameter in the sender's **FLOWSPEC**.

**MaxSduSize**

Set to QOS\_NOT\_SPECIFIED when defaulted in a receiving **FLOWSPEC**. The value is set to the *MaxSduSize* parameter in the sender's **FLOWSPEC**.

**MinimumPolicedSize**

Set to QOS\_NOT\_SPECIFIED when defaulted in a receiving **FLOWSPEC**. The value is set to the *MinimumPolicedSize* parameter in the sender's **FLOWSPEC**.

When the value of the **ServiceType** is set to **SERVICETYPE\_GUARANTEED**, the following also applies:

- The RATE value in **RSPEC** is set to the value of *TokenRate*.
- The DELAYSLACKTERM value in **RSPEC** is set to *DelayVariation*, which is set to zero if *DelayVariation* is set to QOS\_NOT\_SPECIFIED.
- For receivers requesting **SERVICETYPE\_GUARANTEED**, the receiving *TokenRate* must be specified. This contrasts with a **SERVICETYPE\_CONTROLLEDLOAD** receiver, for which *TokenRate* may be set to QOS\_NOT\_SPECIFIED.

In a sending **FLOWSPEC**, everything can be defaulted except **ServiceType** and **TokenRate**. The following list specifies the values that are applied when a sending **FLOWSPEC** sets the corresponding values to default:

**TokenRate**

Must be specified.

**TokenBucketSize**

Set to 1500 bytes when defaulted in a receiving **FLOWSPEC**.

**PeakBandwidth**

Set to the local link speed, if known, otherwise set to `POSITIVE_INFINITY` when defaulted in a receiving **FLowspec**.

**Latency**

Set to 0 msec when defaulted in a receiving **FLowspec**.

**DelayVariation**

Set to 0 msec when defaulted in a receiving **FLowspec**.

**ServiceType**

Must be specified.

**MaxSduSize**

Set to 1500 bytes when defaulted in a receiving **FLowspec**.

**MinimumPolicedSize**

Set to 128 bytes when defaulted in a receiving **FLowspec**.

---

**Notes for Traffic Control** The following **ServiceTypes** are invalid when specifically working with Traffic Control. If you are unsure whether you are working directly with Traffic Control (and thereby need to be concerned about whether the following **ServiceTypes** are applicable in your situation), you probably are not:

SERVICETYPE\_NOTRAFFIC  
SERVICETYPE\_NETWORK\_UNAVAILABLE  
SERVICETYPE\_GENERAL\_INFORMATION  
SERVICETYPE\_NOCHANGE  
SERVICE\_NO\_TRAFFIC\_CONTROL  
SERVICE\_NO\_QOS\_SIGNALING

---

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in Qos.h.

**+** See Also

**QOS**

---

## QOS

The **QOS** structure provides the means by which QOS-enabled applications can specify quality of service parameters for sent and received traffic on a particular flow.

```
typedef struct _QualityOfService {
    FLOWSPEC SendingFlowspec;    // flowspec for data sending
    FLOWSPEC ReceivingFlowspec; // flowspec for data
                                // receiving
    WSABUF   ProviderSpecific;  // provider-specific
                                // parameters
} QOS, FAR * LPQOS;
```

## Members

### SendingFlowspec

Specifies QOS parameters for the sending direction of a particular flow.

**SendingFlowspec** is sent in the form of a **FLOWSPEC** structure.

### ReceivingFlowspec

Specifies QOS parameters for the receiving direction of a particular flow.

**ReceivingFlowspec** is sent in the form of a **FLOWSPEC** structure.

### ProviderSpecific

Pointer to a structure of type **WSABUF** that can provide additional provider-specific QOS parameters to the RSVP SP for a given flow.

## Remarks

Most applications can fulfill their quality of service requirements without using the ProviderSpecific buffer. However, if the application must provide information not available with standard Windows 2000 QOS parameters, the ProviderSpecific buffer allows the application to provide additional parameters for RSVP and/or traffic control.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in Winsock2.h.

### + See Also

**FLOWSPEC**, **ProviderSpecific**

---

## QOS Objects

Applications that require specific or granular quality of service control, beyond what is available in the QOS API, can use QOS objects. QOS objects implement functionality specific to the Microsoft® Windows 2000® operating system, such as traffic control-related objects, as well as industry-wide functionality such as RSVP-related objects. QOS objects are implemented through the **ProviderSpecific** buffer in the **FLOWSPEC** structure, which is a member of the **QOS** structure.

QOS objects follow a stringent structure. QOS objects always include an information header that specifies the type and length of the QOS object to which it is attached, followed by the object itself.

This section describes the following:

- The **ProviderSpecific** Buffer
- **QOS\_OBJECT\_HDR**
- **QOS\_OBJECT\_DESTADDR**
- **QOS\_OBJECT\_SD\_MODE**
- **QOS\_OBJECT\_SHAPING\_RATE**
- **RSVP\_ADSPEC**
- **RSVP\_RESERVE\_INFO**
- **RSVP\_STATUS\_INFO**

---

## The ProviderSpecific Buffer

The **ProviderSpecific** buffer provides applications that have special QOS needs with a mechanism that enables fine-grained tuning of required QOS parameters. The **ProviderSpecific** buffer is of type **WSABUF** as defined by Windows Sockets 2 and is a member of the **QOS** structure.

The standard mechanisms by which Windows 2000 QOS enables service quality provisioning fulfills QOS requirements for the majority of applications. In some situations, however, service quality parameters not available with standard QOS mechanisms may need to be implemented. The **ProviderSpecific** buffer interface is provided for those situations.

The **ProviderSpecific** buffer specifically includes a length field and a pointer to a buffer, which may contain one or more QOS objects. The format of each object is as follows:

- Each object includes a type field, which specifically identifies the object, followed by:
- A length field, which contains the length of the object inclusive of the header, followed by:
- The object data itself.

See *QOS Objects* for more information.

---

## QOS\_OBJECT\_HDR

The QOS object **QOS\_OBJECT\_HDR** is attached to each QOS object. It specifies the object type and its length.



```
typedef struct {
    ULONG    ObjectType;
    ULONG    ObjectLength;
} QOS_OBJECT_HDR, *LPQOS_OBJECT_HDR;
```

## Members

### ObjectType

Specifies the type of object to which **QOS\_OBJECT\_HDR** is attached.

### ObjectLength

Specifies the length of the attached object, inclusive of **QOS\_OBJECT\_HDR**.

## ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Qos.h.

---

# QOS\_OBJECT\_DESTADDR

The QOS object **QOS\_OBJECT\_DESTADDR** is used during a call to the **WSAIoctl(SIO\_SET\_QOS)** function in order to avoid issuing a **connect** function call for a sending socket.

```
typedef struct _QOS_DESTADDR {
    QOS_OBJECT_HDR    ObjectHdr;
    const struct sockaddr * SocketAddress;
    ULONG              SocketAddressLength;
} QOS_DESTADDR, *LPQOS_DESTADDR;
```

## Members

### ObjectHdr

The QOS object **QOS\_OBJECT\_HDR**. The object type for this QOS object should be **QOS\_OBJECT\_DESTADDR**.

### SocketAddress

Address of the destination socket.

### SocketAddressLength

Length of the **SocketAddress** structure.

## ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Qossp.h.

## QOS\_OBJECT\_SD\_MODE

The QOS object **QOS\_OBJECT\_SD\_MODE** defines the behavior of the traffic control–packet shaper component.

```
typedef struct _QOS_SD_MODE {
    QOS_OBJECT_HDR    ObjectHdr;
    ULONG              ShapeDiscardMode;
} QOS_SD_MODE, *LPQOS_SD_MODE;
```

### Members

#### ObjectHdr

The QOS object **QOS\_OBJECT\_HDR**. The object type for this QOS object should be **QOS\_OBJECT\_SD\_MODE**.

#### ShapeDiscardMode

Specifies the requested behavior of the packet shaper. Note that there are elements of packet handling within these predefined behaviors that depend on the flow settings specified within **FLOWSPEC**.

Value	Meaning
TC_NONCONF_BORROW	Instructs the packet shaper to borrow remaining available resources after all higher priority flows have been serviced. If the <b>TokenRate</b> member of <b>FLOWSPEC</b> is specified for this flow, packets that exceed the value of <b>TokenRate</b> will have their priority demoted to less than <b>SERVICETYPE_BESTEFFECT</b> , as defined in the <b>ServiceType</b> member of the <b>FLOWSPEC</b> structure.
TC_NONCONF_SHAPE	Instructs the packet shaper to retain packets until network resources are available to the flow in sufficient quantity to make such packets conforming. (For example, a 100K packet will be retained in the packet shaper until 100K worth of credit is accrued for the flow, allowing the packet to be transmitted as conforming). Note that <b>TokenRate</b> must be specified if using <b>TC_NONCONF_SHAPE</b> .
TC_NONCONF_DISCARD	Instructs the packet shaper to discard all nonconforming packets. <b>TC_NONCONF_DISCARD</b> should be used with care.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Qos.h.

## QOS\_OBJECT\_SHAPING\_RATE

The QOS object **QOS\_OBJECT\_SHAPING\_RATE** specifies the type of shaping behavior to be applied to a given flow.

```
typedef struct _QOS_SHAPING_RATE {
    QOS_OBJECT_HDR    ObjectHdr;
    ULONG             ShapingRate;
} QOS_SHAPING_RATE, *LPQOS_SHAPING_RATE;
```

### Members

#### ObjectHdr

The QOS object **QOS\_OBJECT\_HDR**. The object type for this QOS object should be **QOS\_OBJECT\_SHAPING\_RATE**.

#### ShapingRate

Value that corresponds to the required shaping rate, as follows:

##### TC\_NONCONF\_BORROW

The flow receives resources that remain after all higher priority flows have been serviced. If a **TokenRate** is specified, packets may be non-conforming and are demoted to less than best-effort priority.

##### TC\_NONCONF\_SHAPE

**TokenRate** must be specified. Nonconforming packets are retained in the packet shaper until they become conforming.

##### TC\_NONCONF\_DISCARD

**TokenRate** must be specified. Nonconforming packets are discarded.

##### TC\_NONCONF\_BORROW\_PLUS

Similar to TC\_NONCONF\_BORROW, but no packets will be marked as non-conforming. Note that the setting **ShapingRate** to value cannot be accomplished with the QOS API; it must be done through the traffic control API (the TC API).

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Qos.h.

---

## RSVP\_ADSPEC

The QOS object **RSVP\_ADSPEC** provides a means by which information describing network devices along the data path between sender and receiver, pertaining to RSVP functionality and available services, is provided or retrieved.

```

typedef struct _RSVP_ADSPEC {
    QOS_OBJECT_HDR    ObjectHdr;           // object header
    AD_GENERAL_PARAMS GeneralParams;       // contains general
                                           // characterization
                                           // parameters
    ULONG             NumberOfServices;    // number of services
    CONTROL_SERVICE   Services[1];        // list of supported services
} RSVP_ADSPEC, *LPRSV_ADSPEC;

```

## Members

### ObjectHdr

The QOS object **QOS\_OBJECT\_HDR**.

### GeneralParams

An **AD\_GENERAL\_PARAMS** structure that provides general characterization parameters for the flow. Information includes RSVP-enabled hop count, bandwidth and latency estimates, and the path's MTU.

### NumberOfServices

Provides a count of the number of services available. (See the following member for more information.)

### Services

A **CONTROL\_SERVICE** array, its element count based on **NumberOfServices**, which provides information about the services available along the data path between the sender and receiver of a given flow.

## ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in Qosp.h.

## RSVP\_RESERVE\_INFO

The QOS object **RSVP\_RESERVE\_INFO**, through the **ProviderSpecific** buffer, allows RSVP behavior for a given flow to be specified or modified at a granular level, and allows default RSVP style settings for a flow to be overridden. Although **RSVP\_RESERVE\_INFO** is technically a structure, its use within Windows 2000 QOS technology—and especially its required inclusion of the **QOS\_OBJECT\_HDR** as its first member—define it as a QOS object.

```

typedef struct _RSVP_RESERVE_INFO {
    QOS_OBJECT_HDR    ObjectHdr;           // type and length of
                                           // object
    ULONG             Style;              // RSVP style

```

(continued)

*(continued)*

```

// (FF,WF,SE)
ULONG          ConfirmRequest; // nonzero for
// confirm request
// (receive only)
ULONG          NumPolicyElement; // number of policy
// elements
LPRSVP_POLICY  PolicyElementList; // set of policy
// elements
ULONG          NumFlowDesc; // number of
// structures
LPFLOWDESCRIPTOR FlowDescList; // structure list
} RSVP_RESERVE_INFO, *LPRSVP_RESERVE_INFO;

```

**Members****ObjectHdr**

The QOS object **QOS\_OBJECT\_HDR**.

**Style**

Specifies the RSVP reservation style for a given flow, and can be used to replace default reservation styles placed on a particular type of flow. More information about RSVP reservation styles, and the default settings for certain QOS-enabled socket sessions, can be found under *Network-Driven QOS Components*. This member can be one of the following values.

Value	Meaning
RSVP_WILDCARD_STYLE	Implements the WF RSVP reservation style. RSVP_WILDCARD_STYLE is the default value for multicast receivers and UDP unicast receivers. Specifying RSVP_WILDCARD_STYLE through RSVP_RESERVE_INFO is useful for overriding the default value (RSVP_FIXED_FILTER_STYLE) applied to connected unicast receivers.
RSVP_FIXED_FILTER_STYLE	Implements the Fixed Filter (FF) RSVP reservation style. RSVP_FIXED_FILTER_STYLE is useful for overriding the default style for multicast receivers or unconnected UDP unicast receivers (RSVP_WILDCARD_STYLE). It may also be used to generate multiple RSVP_FIXED_FILTER_STYLE reservations in instances where only a single RSVP_FIXED_FILTER_STYLE reservation will be generated by default, such as with connected unicast receivers.
RSVP_SHARED_EXPLICIT_STYLE	Implements the Shared Explicit (SE) RSVP reservation style.

---

**Note** It is important to note that the number of senders included in any individual `RSVP_SHARED_EXPLICIT_STYLE` reservation must be less than one hundred senders. If more than one hundred senders attempt to connect to an `RSVP_SHARED_EXPLICIT_STYLE` reservation, the one-hundredth (and above) attempt fails without notice.

---

### ConfirmRequest

Can be used by a receiving application to request notification of its reservation request by setting **ConfirmRequest** to a nonzero value. Such notification is achieved when RSVP-aware devices in the data path between sender and receiver (or vice-versa) transmit an RESV CONFIRMATION message toward the requesting node. Note that an RSVP node is not required to automatically generate RESV CONFIRMATION messages.

### NumPolicyElement

Specifies the number of policy elements.

### PolicyElementList

Pointer to the set of policy elements. Optional policy information, as provided in an **RSVP\_POLICY** structure.

### NumFlowDesc

Specifies the FLOWDESCRIPTOR count.

### FlowDescList

Pointer to the list of FLOWDESCRIPTORS.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in `Qosp.h`.

---

## RSVP\_STATUS\_INFO

The QOS object **RSVP\_STATUS\_INFO** provides information regarding the status of RSVP for a given flow, including event notifications associated with monitoring `FD_QOS` events, as well as error information. **RSVP\_STATUS\_INFO** is useful for storing RSVP-specific status and error information.

```
typedef struct _RSVP_STATUS_INFO {
    QOS_OBJECT_HDR  ObjectHdr;        // object header
    ULONG           StatusCode;       // error or status
                                                // information, see
                                                // Winsock2.h
    ULONG           ExtendedStatus1; // provider-specific
```

(continued)

(continued)

```
                                // status extension
ULONG          ExtendedStatus2; // provider-specific
                                // status extension
} RSVP_STATUS_INFO, *LPRSPV_STATUS_INFO;
```

## Members

### ObjectHdr

The QOS object **QOS\_OBJECT\_HDR**.

### StatusCode

Status information. See *Winsock2.h* for more information.

### ExtendedStatus1

Mechanism for storing or returning provider-specific status information. The *ExtendedStatus1* parameter is used for storing a higher-level, or generalized error code, and is augmented by finer-grained error information provided in *ExtendedStatus2*.

### ExtendedStatus2

Additional mechanism for storing or returning provider-specific status information. Provides finer-grained error information compared to the generalized error information provided in *ExtendedStatus1*.

## Remarks

When applications register their interest in FD\_QOS events (see *QOS Events*), event and error information is associated with the event in the form of the **QOS** structure that is associated with the event. For more detailed information associated with that event, applications can investigate the **RSVP\_STATUS\_INFO** object that is provided in the **ProviderSpecific** buffer of the event-associated **QOS** structure.

## ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 98.

**Header:** Declared in *Qossp.h*.

## CHAPTER 16

# Traffic Control API Reference

## Traffic Control Functions

### TcAddFilter

The **TcAddFilter** function associates a new filter with an existing flow that allows packets matching the filter to be directed to the associated flow.

Filters include a pattern and a mask. The pattern specifies particular parameter values, while the mask specifies which parameters and parameter subfields apply to a given filter. When a pattern and mask combination is applied to a set of packets, matching packets are directed to the flow to which the corresponding filter is associated.

Traffic control returns a handle to the newly added filter, in the *pFilterHandle* parameter, by which clients can refer to the added filter. Pending flows, such as those processing a **TcAddFlow** or **TcModifyFlow** request for which a callback routine has not been completed, cannot have filters associated to them; only flows that have been completed and are stable can apply associated filters.

The relationship between filters and flows is many to one. Multiple filters can be applied to a single flow; however, a filter can only apply to one flow. For example, flow A can have filters X, Y and Z applied to it, but as long as flow A is active, filters X, Y and Z *cannot* apply to any other flows.

```
DWORD TcAddFilter(  
    HANDLE FlowHandle,  
    PTC_GEN_FILTER pGenericFilter,  
    PHANDLE pFilterHandle  
);
```

#### Parameters

##### *FlowHandle*

[in] Handle for the flow, as received from a previous call to the **TcAddFlow** function.

##### *pGenericFilter*

[in] Pointer to a description of the filter to be installed.

##### *pFilterHandle*

[out] Pointer to a buffer where traffic control returns the filter handle. This filter handle is used by the client in subsequent calls to refer to the added filter.



## Return Values

Error codes	Description
NO_ERROR	The function executed without errors.
ERROR_INVALID_HANDLE	The flow handle is invalid.
ERROR_NOT_ENOUGH_MEMORY	The system is out of memory.
ERROR_INVALID_PARAMETER	A parameter is invalid.
ERROR_INVALID_ADDRESS_TYPE	An invalid address type has been provided.
ERROR_DUPLICATE_FILTER	An identical filter exists on a flow on this interface.
ERROR_FILTER_CONFLICT	A conflicting filter exists on a flow on this interface.
ERROR_NOT_READY	The flow is either being installed, modified, or deleted, and is not in a state that accepts filters.

## Remarks

Filters can be of different types. They are typically used to filter packets belonging to different network layers. Filter types installed on an interface generally correspond to the address types of the network layer addresses associated with the interface. The address type should be specified in the filter structure.

Filters may be rejected for various reasons, including possible conflicts with the requested filter and those filters already associated with the flow. Error codes specific to traffic control are provided to help the user diagnose the reason behind a rejection to the **TcAddFilter** function.

---

**Note** Use of the **TcAddFilter** function requires administrative privilege.

---

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

### + See Also

**TcAddFlow**

# TcAddFlow

The **TcAddFlow** function adds a new flow on the specified interface. Note that the successful addition of a flow does not necessarily indicate a change in the way traffic is handled; traffic handling changes are effected by attaching a filter to the added flow, using the **TcAddFilter** function.

Traffic control clients that have registered an **AddFlowComplete** handler (a mechanism for allowing traffic control to call the **CIAddFlowComplete** callback function in order to alert clients of completed flow additions) can expect a return value of **ERROR\_SIGNAL\_PENDING**. For more information, see *Traffic Control Objects*.

```
DWORD TcAddFlow(  
    HANDLE IfcHandle,  
    HANDLE CFlowCtx,  
    ULONG Flags,  
    PTC_GEN_FLOW pGenericFlow,  
    PHANDLE pFlowHandle  
);
```

## Parameters

### *IfcHandle*

[in] Handle associated with the interface on which the flow is to be added. This handle is obtained by a previous call to the **TcOpenInterface** function.

### *CFlowCtx*

[in] Client provided-flow context handle. Used subsequently by traffic control when referring to the added flow.

### *Flags*

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

### *pGenericFlow*

[in] Pointer to a description of the flow being installed.

### *pFlowHandle*

[out] Pointer to a location into which traffic control will return the flow handle. This flow handle should be used in subsequent calls to traffic control to refer to the installed flow.

## Return Values

There are many reasons why a request to add a flow might be rejected. Error codes returned by traffic control from calls to **TcAddFlow** are provided to aid in determining the reason for rejection. For more information on the validation rules applied to flow requests, see traffic control.

Error codes	Description
NO_ERROR	The function executed without errors.
ERROR_SIGNAL_PENDING	The function is being executed asynchronously; the client will be called back through the client-exposed <b>CIAddFlowComplete</b> function when the flow has been added or when the process has been completed.
ERROR_INVALID_HANDLE	The interface handle is invalid.
ERROR_NOT_ENOUGH_MEMORY	The system is out of memory.
ERROR_INVALID_PARAMETER	A parameter is invalid.
ERROR_INVALID_SERVICE_TYPE	An unspecified or bad <b>INTSERV</b> service type has been provided.
ERROR_INVALID_TOKEN_RATE	An unspecified or bad TOKENRATE value has been provided.
ERROR_INVALID_PEAK_RATE	The PEAKBANDWIDTH value is invalid.
ERROR_INVALID_SD_MODE	The SHAPEDISCARDMODE is invalid.
ERROR_INVALID_QOS_PRIORITY	The priority value is invalid.
ERROR_INVALID_TRAFFIC_CLASS	The traffic class value is invalid.
ERROR_NO_SYSTEM_RESOURCES	There are not enough resources to accommodate the requested flow.
ERROR_TC_OBJECT_LENGTH_INVALID	Bad length specified for the <b>TC</b> objects.
ERROR_INVALID_DIFFSERV_FLOW	Applies to Diffserv flows. Indicates that the <b>QOS_OBJECT_DIFFSERV</b> object was passed with an invalid parameter.
ERROR_DS_MAPPING_EXISTS	Applies to Diffserv flows. Indicates that the <b>QOS_DIFFSERV_RULE</b> specified in <b>TC_GEN_FLOW</b> already applies to an existing flow on the interface.
ERROR_INVALID_SHAPE_RATE	The <b>QOS_OBJECT_SHAPING_RATE</b> object was passed with an invalid <i>ShapeRate</i> .
ERROR_INVALID_DS_CLASS	The <b>QOS_OBJECT_DS_CLASS</b> is invalid.
ERROR_NETWORK_UNREACHABLE	The network cable is not plugged into the adapter.

## Remarks

If the **TcAddFlow** function returns `ERROR_SIGNAL_PENDING`, the **CiAddFlowComplete** function will be called on a different thread than the thread that called the **TcAddFlow** function.

Only the addition of a filter will affect traffic control. However, the addition of a flow will cause resources to be committed within traffic control components. This enables traffic control to prepare for handling traffic on the added flow.

Traffic control may delete a flow for various reasons, including the inability to accommodate the flow due to bandwidth restrictions, and adjusted policy requirements. Clients are notified of deleted flows through the **CiNotifyHandler** callback function, with the `TC_NOTIFY_FLOW_CLOSE` event.

---

**Note** Use of the **TcAddFlow** function requires administrative privilege.

---

## ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in `Traffic.h`.

**Library:** Use `Traffic.lib`.

# TcCloseInterface

The **TcCloseInterface** function closes an interface previously opened with a call to **TcOpenInterface**. All flows and filters on a particular interface should be closed before closing the interface with a call to **TcCloseInterface**.

```
DWORD TcCloseInterface (
    HANDLE IfcHandle
);
```

## Parameters

### *IfcHandle*

[in] Handle associated with the interface to be closed. This handle is obtained by a previous call to the **TcOpenInterface** function.

## Return Values

Error code	Description
<code>NO_ERROR</code>	The function executed without errors.
<code>ERROR_INVALID_HANDLE</code>	The interface handle is invalid.
<code>ERROR_TC_SUPPORTED_OBJECTS_EXIST</code>	Not all flows have been deleted for this interface.

**Remarks**

Regardless of whether **TcCloseInterface** is called, an interface will be closed following a TC\_NOTIFY\_IFC\_CLOSE notification event. If the **TcCloseInterface** function is called with the handle of an interface that has already been closed, the handle will be invalidated and **TcCloseInterface** will return ERROR\_INVALID\_HANDLE.

---

**Note** Use of **TcCloseInterface** requires administrative privilege.

---

**! Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

**+ See Also**

**TcOpenInterface**

---

## TcDeleteFilter

The **TcDeleteFilter** function deletes a filter previously added with the **TcAddFilter** function:

```
DWORD TcDeleteFilter(
    HANDLE FilterHandle
);
```

**Parameters**

*FilterHandle*

[in] Handle to the filter to be deleted, as received in a previous call to the **TcAddFilter** function.

**Return Values****Error codes****Description**

NO\_ERROR

The function executed without errors.

ERROR\_INVALID\_HANDLE

The filter handle is invalid.

---

**Note** Use of the **TcDeleteFilter** function requires administrative privilege.

---

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

**+** See Also

**TcAddFilter**

---

## TcDeleteFlow

The **TcDeleteFlow** function deletes a flow that has been added with the **TcAddFlow** function. Clients should delete all filters associated with a flow before deleting it, otherwise, an error will be returned and the function will not delete the flow.

Traffic control clients that have registered a **DeleteFlowComplete** handler (a mechanism for allowing traffic control to call the **CIDeleteFlowComplete** callback function in order to alert clients of completed flow deletions) can expect a return value of **ERROR\_SIGNAL\_PENDING**.

```
DWORD TcDeleteFlow(
    HANDLE FlowHandle
);
```

### Parameters

*FlowHandle*

[in] Handle for the flow, as received from a previous call to the **TcAddFlow** function.

### Return Values

#### Error codes

#### Description

**NO\_ERROR**

The function executed without errors.

**ERROR\_SIGNAL\_PENDING**

The function is being executed asynchronously; the client will be called back through the client-exposed **CIDeleteFlowComplete** function when the flow has been added, or when the process has been completed.

**ERROR\_INVALID\_HANDLE**

The flow handle is invalid or NULL.

**ERROR\_TC\_SUPPORTED\_OBJECTS\_EXIST**

At least one filter associated with this flow exists.

## Remarks

If the **TcDeleteFlow** function returns `ERROR_SIGNAL_PENDING`, the **CIDeleteFlowComplete** function will be called on a different thread than the thread that called the **TcDeleteFlow** function.

---

**Note** Use of the **TcDeleteFlow** function requires administrative privilege.

---

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in `Traffic.h`.

**Library:** Use `Traffic.lib`.

### + See Also

**TcEnumerateFlows**, **TcAddFlow**, **CIDeleteFlowComplete**

---

## TcDeregisterClient

The **TcDeregisterClient** function deregisters a client with the Traffic Control Interface (TCI). Before deregistering, a client must delete each installed flow and filter with the **TcDeleteFlow** and **TcDeleteFilter** functions, and close all open interfaces with the **TcCloseInterface** function, respectively.

```
DWORD TcDeregisterClient(
    HANDLE ClientHandle
);
```

### Parameters

*ClientHandle*

[in] Handle assigned to the client through the previous call to the **TcRegisterClient** function.

### Return Values

#### Error codes

#### Description

`NO_ERROR`

The function executed without errors.

`ERROR_INVALID_HANDLE`

Invalid interface handle, or the handle was set to `NULL`.

`ERROR_TC_SUPPORTED_OBJECTS_EXIST`

Interfaces are still open for this client. all interfaces must be closed to deregister a client.

## Remarks

Once a client calls **TcDeregisterClient**, the only traffic control function the client is allowed to call is **TcRegisterClient**.

---

**Note** Use of the **TcDeregisterClient** function requires administrative privilege.

---

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

### + See Also

**TcRegisterClient**, **TcCloseInterface**, **TcDeleteFlow**, **TcDeleteFilter**

---

## TcEnumerateFlows

The **TcEnumerateFlows** function enumerates installed flows and their associated filters on an interface.

The process of returning flow enumeration often consists of multiple calls to the **TcEnumerateFlows** function. The process of receiving flow information from **TcEnumerateFlows** can be compared to reading through a book in multiple sittings—a certain number of pages are read at one time, a bookmark is placed where the reading stopped, and reading is resumed from the bookmark until the book is finished.

The **TcEnumerateFlows** function fills the *Buffer* parameter with as many flow enumerations as the buffer can hold, then returns a handle in the *pEnumToken* parameter that internally bookmarks where the enumeration stopped. Subsequent calls to **TcEnumerateFlows** must then pass the returned *pEnumToken* value to instruct traffic control where to resume flow enumeration information. When all flows have been enumerated, *pEnumToken* will be NULL.

```
DWORD TcEnumerateFlows(  
    HANDLE IfcHandle,  
    PHANDLE pEnumToken,  
    PULONG pFlowCount,  
    PULONG pBufSize,  
    PENUMERATION_BUFFER Buffer  
);
```



## Parameters

### *IfcHandle*

[in] Handle associated with the interface on which flows are to be enumerated. This handle is obtained by a previous call to the **TcOpenInterface** function.

### *pEnumToken*

[in, out] Pointer to the enumeration token, used internally by traffic control to maintain returned flow information state.

For input of the initial call to **TcEnumerateFlows**, *pEnumToken* should be set to NULL. For input on subsequent calls, *pEnumToken* must be the value returned as the *pEnumToken* OUT parameter from the immediately preceding call to **TcEnumerateFlows**.

For output, *pEnumToken* is the refreshed enumeration token that must be used in the following call to **TcEnumerateFlows**.

### *pFlowCount*

[in, out] Pointer to the number of requested or returned flows. For input, this parameter designates the number of requested flows or it can be set to (-1) to request all flows. For output, *pFlowCount* returns the number of flows actually returned in *Buffer*.

### *pBufSize*

[in, out] Pointer to the size of the client-provided buffer or the number of bytes used by traffic control. For input, points to the size of *Buffer*, in bytes. For output, points to the actual amount of buffer space, in bytes, written or needed with flow enumerations.

### *Buffer*

[out] Pointer to the buffer containing flow enumerations. See **ENUMERATION\_BUFFER** for more information about flow enumerations.

## Return Values

Error codes	Description
NO_ERROR	The function executed without errors.
ERROR_INVALID_HANDLE	Invalid interface handle.
ERROR_INVALID_PARAMETER	One of the pointers is NULL, or <i>pFlowCount</i> or <i>pBufSize</i> are set to zero.
ERROR_INSUFFICIENT_BUFFER	The buffer is too small to store even a single flow's information and attached filters.
ERROR_NOT_ENOUGH_MEMORY	The system is out of memory.
ERROR_INVALID_DATA	The enumeration token is no longer valid.

## Remarks

Do not request zero flows, or pass a buffer with a size equal to zero or pointer to a NULL.

If an enumeration token pointer has been invalidated by traffic control (due to the deletion of a flow), continuing to enumerate flows is not allowed, and the call will return `ERROR_INVALID_DATA`. Under this circumstance, the process of enumeration must start over. This circumstance can occur when the next flow to be enumerated is deleted while enumeration is in progress.

To get the total number of flows for a given interface, call **TcQueryInterface** and specify `GUID_QOS_FLOW_COUNT`.

---

**Note** Use of the **TcEnumerateFlows** function requires administrative privilege.

---

#### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in `Traffic.h`.

**Library:** Use `Traffic.lib`.

#### + See Also

**TcOpenInterface**, **TcQueryInterface**

---

## TcEnumerateInterfaces

The **TcEnumerateInterfaces** function enumerates all traffic control–enabled network interfaces. Clients are notified of interface changes through the **CINotifyHandler** function.

```
DWORD TcEnumerateInterfaces(  
    HANDLE ClientHandle,  
    OUT PULONG pBufferSize,  
    PTC_IFC_DESCRIPTOR InterfaceBuffer  
);
```

### Parameters

#### *ClientHandle*

[in] Handle used by traffic control to identify the client. Clients receive handles when registering with traffic control through the **TcRegisterClient** function.

#### *pBufferSize*

[in, out] Pointer to a value indicating the size of the buffer. For input, this value is the size of the buffer, in bytes, allocated by the caller. For output, this value is the actual size of the buffer, in bytes, used or needed by traffic control. A value of zero on output indicated that no traffic control interfaces are available, indicating that the QOS Packet Scheduler is not installed.

*InterfaceBuffer*

[out] Pointer to the buffer containing the returned list of interface descriptors.

**Return Values**

Successful completion returns the device name of the interface.

<b>Error code</b>	<b>Description</b>
NO_ERROR	The function executed without errors.
ERROR_INVALID_HANDLE	The client handle is invalid.
ERROR_INVALID_PARAMETER	One of the parameters is NULL.
ERROR_INSUFFICIENT_BUFFER	The buffer is too small to enumerate all interfaces. If this error is returned, the correct (required) size of the buffer is passed back in <i>pBufferSize</i> .
ERROR_NOT_ENOUGH_MEMORY	The system is out of memory.

**Remarks**

The client calling the **TcEnumerateInterfaces** function must first allocate a buffer, then pass the buffer to traffic control through *InterfaceBuffer*. Traffic control returns a pointer to an array of interface descriptors in *InterfaceBuffer*. Each interface descriptor contains two elements:

- The traffic control interface's identifying text string.
- The network address list descriptor currently associated with the interface.


The network address list descriptor includes the media type, as well as a list of network addresses. The media type determines how the network address list should be interpreted:

- For connectionless media such as a LAN, the network address list contains all the protocol-specific addresses associated with the interface.
- For connection-oriented media such as a WAN, the network address list contains an even number of network addresses:
  - The first address in each pair represents the local (source) address of the interface.
  - The second address in each pair represents the remote (destination) address of the interface.

---

**Note** Use of the **TcEnumerateInterfaces** function requires administrative privilege.

---

 **Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

**+** See Also

**TcRegisterClient, CInotifyHandler**

## TcGetFlowName

The **TcGetFlowName** function provides the name of a flow that has been created by the calling client. Flow properties and other characteristics of flows are provided based on the name of a flow. Flow names can also be retrieved by a call to the **TcEnumerateFlows** function.

```
DWORD TcGetFlowName(
    HANDLE FlowHandle,
    ULONG StrSize,
    LPTSTR pFlowName
);
```

### Parameters

*FlowHandle*

[in] Handle for the flow.

*StrSize*

[in] Size of the string buffer provided in *pFlowName*.

*pFlowName*

[out] Pointer to the output buffer holding the flow name.

### Return Values

#### Error codes

#### Description

NO_ERROR	The function executed without errors.
ERROR_INVALID_HANDLE	The flow handle is invalid.
ERROR_INVALID_PARAMETER	One of the parameters is invalid.
ERROR_INSUFFICIENT_BUFFER	The buffer is too small to contain the results.

**Note** Use of the **TcGetFlowName** function requires administrative privilege.


### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

 See Also

**TcEnumerateFlows**

## TcModifyFlow

The **TcModifyFlow** function modifies an existing flow. When calling **TcModifyFlow**, new *flowspec* parameters and any traffic control objects should be filled.

Traffic control clients that have registered a **ModifyFlowComplete** handler (a mechanism for allowing traffic control to call the **CIModifyFlowComplete** callback function in order to alert clients of completed flow modifications) can expect a return value of **ERROR\_SIGNAL\_PENDING**.

```
DWORD TcModifyFlow(
    HANDLE FlowHandle,
    PTC_GEN_FLOW pGenericFlow
);
```

### Parameters

*FlowHandle*

[in] Handle for the flow, as received from a previous call to the **TcAddFlow** function.

*pGenericFlow*

[in] Pointer to a description of the flow modifications.

### Return Values

#### Error codes

#### Description

**NO\_ERROR**

The function executed without errors.

**ERROR\_SIGNAL\_PENDING**

The function is being executed asynchronously; the client will be called back through the client-exposed **CIModifyFlowComplete** function when the flow has been added, or when the process has been completed.

**ERROR\_INVALID\_HANDLE**

The interface handle is invalid.

**ERROR\_NOT\_ENOUGH\_MEMORY**

The system is out of memory.

**ERROR\_INVALID\_PARAMETER**

A parameter is invalid.

**ERROR\_INVALID\_SERVICE\_TYPE**

An unspecified or bad **intserv** service type has been provided.

**ERROR\_INVALID\_TOKEN\_RATE**

An unspecified or bad *TokenRate* value has been provided.

Error codes	Description
ERROR_INVALID_PEAK_RATE	The <i>PeakBandwidth</i> value is invalid.
ERROR_INVALID_SD_MODE	The <i>ShapeDiscardMode</i> is invalid.
ERROR_INVALID_QOS_PRIORITY	The priority value is invalid.
ERROR_INVALID_TRAFFIC_CLASS	The traffic class value is invalid.
ERROR_NO_SYSTEM_RESOURCES	There are not enough resources to accommodate the requested flow.
ERROR_TC_OBJECT_LENGTH_INVALID	Bad length specified for the TC objects.
ERROR_INVALID_DIFFSERV_FLOW	Applies to Diffserv flows. Indicates that the QOS_DIFFSERV object was passed with an invalid parameter.
ERROR_DS_MAPPING_EXISTS	Applies to Diffserv flows. Indicates that the QOS_DIFFSERV_RULE specified in <b>TC_GEN_FLOW</b> already applies to an existing flow on the interface.
ERROR_INVALID_SHAPE_RATE	The <b>QOS_OBJECT_SHAPING_RATE</b> was passed with an invalid <i>ShapeRate</i> .
ERROR_INVALID_DS_CLASS	QOS_OBJECT_DS_CLASS is invalid.
ERROR_NETWORK_UNREACHABLE	The network cable is not plugged into the adapter.

### Remarks

If the **TcModifyFlow** function returns ERROR\_SIGNAL\_PENDING, the **CIModifyFlowComplete** function will be called on a different thread than the thread that called the **TcModifyFlow** function.

---

**Note** Use of the **TcModifyFlow** function requires administrative privilege.

---

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

### + See Also

**TcEnumerateFlows, TcAddFlow, CIModifyFlowComplete**

# TcOpenInterface

The **TcOpenInterface** function opens an interface. The **TcOpenInterface** function identifies and opens an interface based on its text string, which is available from a call to **TcEnumerateInterfaces**. Once an interface is opened, the client must be prepared to receive notification regarding the open interface, through traffic control's use of the interface context.

```
DWORD TcOpenInterface(
    LPTSTR pInterfaceName,
    HANDLE ClientHandle,
    HANDLE CIfcCtx,
    PHANDLE pIfcHandle
);
```

## Parameters

### *pInterfaceName*

[in] Pointer to the text string identifying the interface to be opened. This text string is part of the information returned in a previous call to **TcEnumerateInterfaces**.

### *ClientHandle*

[in] Handle used by traffic control to identify the client, obtained through the *pClientHandle* parameter of the client's call to **TcRegisterClient**.

### *CIfcCtx*

[in] Client's interface–context handle for the opened interface. Used as a callback parameter by traffic control when communicating with the client about the opened interface. This can be a container to hold an arbitrary client-defined context for this instance of the interface.

### *pIfcHandle*

[out] Pointer to the buffer where traffic control can return an interface handle. The interface handle returned to *pIfcHandle* must be used by the client to identify the interface in subsequent calls to traffic control.

## Return Values

Error code	Description
NO_ERROR	The function executed without errors.
ERROR_INVALID_PARAMETER	One of the parameters is NULL.
ERROR_NOT_ENOUGH_MEMORY	The system is out of memory.
ERROR_NOT_FOUND	Traffic control failed to find an interface with the name provided in <i>pInterfaceName</i> .
ERROR_INVALID_HANDLE	The client handle is invalid.

---

**Note** Use of the **TcOpenInterface** function requires administrative privilege.

---

#### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in `Traffic.h`.

**Library:** Use `Traffic.lib`.

#### + See Also

**TcEnumerateInterfaces**, **TcRegisterClient**, **CINotifyHandler**

---

## TcQueryFlow

The **TcQueryFlow** function queries traffic control for the value of a specific flow parameter based on the name of the flow. The name of a flow can be retrieved from the **TcEnumerateFlows** function or from the **TcGetFlowName** function.

```
DWORD TcQueryFlow (  
    LPTSTR FlowName,  
    PGUID pGuidParam,  
    OUT PULONG pBufferSize,  
    PVOID Buffer  
);
```

### Parameters

*FlowName*

[in] Name of the flow being queried.

*pGuidParam*

[in] Pointer to the globally unique identifier (GUID) that corresponds to the flow parameter of interest. A list of traffic control's GUIDs can be found in **GUID**.

*pBufferSize*

[in] Pointer to the size of the client-provided buffer or the number of bytes used by traffic control. For input, points to the size of *Buffer*, in bytes. For output, points to the actual amount of buffer space written with returned flow-parameter data, in bytes.

*Buffer*

[out] Pointer to the client-provided buffer in which the returned flow parameter is written.



## Return Values

Error codes	Description
NO_ERROR	The function executed without errors.
ERROR_INVALID_PARAMETER	A parameter is invalid.
ERROR_INSUFFICIENT_BUFFER	The provided buffer is too small to hold the results.
ERROR_NOT_SUPPORTED	The requested GUID is not supported.
ERROR_WMI_GUID_NOT_FOUND	The device did not register for this GUID.
ERROR_WMI_INSTANCE_NOT_FOUND	The instance name was not found, likely because the flow or the interface is in the process of being closed.

**Note** Use of the **TcQueryFlow** function requires administrative privilege.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

### + See Also

**TcEnumerateFlows, TcGetFlowName, GUID**

## TcQueryInterface

The **TcQueryInterface** function queries traffic control for related per-interface parameters. A traffic control parameter is queried by providing its Globally Unique Identifier (GUID). Setting the *NotifyChange* parameter to TRUE enables event notification on the specified GUID, after which notification events are sent to a client whenever the queried parameter changes. GUIDs for which clients can request notification are found in the **GUID** entry; the column titled "Notification" denotes which GUIDs are available for notification.

```
DWORD TcQueryInterface (
    HANDLE IfcHandle,
    PGUID pGuidParam,
    BOOLEAN NotifyChange,
    OUT PULONG BufferSize,
    PVOID Buffer
);
```

## Parameters

### *IfcHandle*

[in] Handle associated with the interface to be queried. This handle is obtained by a previous call to the **TcOpenInterface** function.

### *pGuidParam*

[in] Pointer to the globally unique identifier (GUID) that corresponds to the traffic control parameter being queried.

### *NotifyChange*

[in] Used to request notifications from traffic control for the parameter being queried. If TRUE, traffic control will notify the client, through the **CINotifyHandler** function, upon changes to the parameter corresponding to the GUID provided in *pGuidParam*. Notifications are off by default.

### *BufferSize*

[in, out] Indicates the size of the buffer. For input, this value is the size of the buffer allocated by the caller. For output, this value is the actual size of the buffer, in bytes, used by traffic control.

### *Buffer*

[out] Pointer to a client-allocated buffer into which returned data will be written.

## Return Values

Note that, with regard to a requested notification state, only a return value of NO\_ERROR will result in the application of the requested notification state. If a return value other than NO\_ERROR is returned from a call to the **TcQueryInterface** function, the requested change in notification state will not be accepted.

Error code	Description
NO_ERROR	The function executed without errors.
ERROR_INVALID_HANDLE	Invalid interface handle.
ERROR_INVALID_PARAMETER	Invalid or NULL parameter.
ERROR_INSUFFICIENT_BUFFER	The buffer is too small to store the results.
ERROR_NOT_SUPPORTED	Querying for the GUID provided is not supported on the provided interface.
ERROR_WMI_GUID_NOT_FOUND	The device did not register for this GUID.
ERROR_WMI_INSTANCE_NOT_FOUND	The instance name was not found, likely because the interface is in the process of being closed.

---

**Note** Use of the **TcQueryInterface** function requires administrative privilege.

---

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

**+** See Also

**TcEnumerateInterfaces, TcRegisterClient, CNotifyHandler**

---

## TcRegisterClient

The **TcRegisterClient** function is used to register a client with the Traffic Control Interface (TCI). The **TcRegisterClient** function must be the first function call a client makes to the TCI.

Client registration provides callback routines that allow the TCI to complete either client-initiated operations or asynchronous events. Upon successful registration, the caller of the **TcRegisterClient** function must be ready to have any of its TCI handlers called. See **Entry Points Exposed by Clients of the Traffic Control Interface** for more information.

```
DWORD TcRegisterClient(  
    ULONG TciVersion,  
    HANDLE CRegCtx,  
    PTCI_CLIENT_FUNC_LIST pClientHandlerList,  
    PHANDLE pClientHandle  
);
```

### Parameters

**TciVersion**

[in] Traffic control version expected by the client, included to ensure compatibility between traffic control and the client. Clients can pass **CURRENT\_TCI\_VERSION**, defined in Traffic.h.

**CRegCtx**

[in] Client registration context. **CRegCtx** is returned when the client's notification handler function is called. This can be a container to hold an arbitrary client-defined context for this instance of the interface.

**pClientHandlerList**

[in] Pointer to a list of client-supplied handlers. Client-supplied handlers are used for notification events and asynchronous completions. Each completion routine is optional, with the exception of the notification handler. Setting the notification handler to **NULL** will return an **ERROR\_INVALID\_PARAMETER**.

*pClientHandle*

[out] Pointer to the buffer that traffic control uses to return a registration handle to the client.

## Return Values

Error code	Description
NO_ERROR	The function executed without errors.
ERROR_NOT_ENOUGH_MEMORY	The system is out of memory.
ERROR_INVALID_PARAMETER	One of the parameters is NULL.
ERROR_INCOMPATIBLE_TCI_VERSION	The TCI version is wrong.
ERROR_OPEN_FAILED	Traffic control failed to open a system device. The likely cause is insufficient privileges.

**Note** Use of the **TcRegisterClient** function requires administrative privilege.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

## TcSetFlow

The **TcSetFlow** function sets individual parameters for a given flow:

```
DWORD TcSetFlow(
    LPWSTR pFlowName,
    PGUID pGuidParam,
    ULONG BufferSize,
    PVOID Buffer
);
```

### Parameters

*pFlowName*

[in] Name of the flow being set. The value for this parameter is obtained by a previous call to the **TcEnumerateFlows** function or the **TcGetFlowName** function.

*pGuidParam*

[in] Pointer to the Globally Unique Identifier (GUID) that corresponds to the parameter to be set. A list of available GUIDs can be found in **GUID**.

*BufferSize*

[in] Size of the client-provided buffer.

*Buffer*


[in] Pointer to a client-provided buffer. Buffer must contain the value to which the traffic control parameter provided in *pGuidParam* should be set.

**Return Values**

The **TcSetFlow** function has the following return values.

<b>Error codes</b>	<b>Description</b>
NO_ERROR	The function executed without errors.
ERROR_NOT_READY	The flow is currently being modified.
ERROR_NOT_ENOUGH_MEMORY	The buffer size was insufficient for the GUID.
ERROR_INVALID_PARAMETER	Invalid parameter.
ERROR_NOT_SUPPORTED	Setting the GUID for the provided flow is not supported.
ERROR_WMI_INSTANCE_NOT_FOUND	The instance name was not found, likely due to the flow or the interface being in the process of being closed.
ERROR_WMI_GUID_NOT_FOUND	The device did not register for this GUID.

**Note** Use of the **TcSetFlow** function requires administrative privilege.

 **Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

## TcSetInterface

The **TcSetInterface** function sets individual parameters for a given interface.

```

DWORD TcSetInterface(
    HANDLE IfcHandle,
    PGUID pGuidParam,
    ULONG BufferSize,
    PVOID Buffer
);

```

## Parameters

### *IfcHandle*

[in] Handle associated with the interface to be set. This handle is obtained by a previous call to the **TcOpenInterface** function.

### *pGuidParam*

[in] Pointer to the globally unique identifier (GUID) that corresponds to the parameter to be set. A list of available GUIDs can be found in **GUID**.

### *BufferSize*

[in] Size of the client-provided buffer.

### *Buffer*

[in] Pointer to a client-provided buffer. *Buffer* must contain the value to which the traffic control parameter provided in *pGuidParam* should be set.

## Return Values

Error codes	Description
NO_ERROR	The function executed without errors.
ERROR_INVALID_HANDLE	Invalid interface handle.
ERROR_INVALID_PARAMETER	Invalid parameter.
ERROR_NOT_SUPPORTED	Setting the GUID for the provided interface is not supported.
ERROR_WMI_INSTANCE_NOT_FOUND	The GUID is not available.
ERROR_WMI_GUID_NOT_FOUND	The device did not register for this GUID.

**Note** Use of the **TcSetInterface** function requires administrative privilege. The list of GUIDS that can be set is explained in **GUID**.

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**Library:** Use Traffic.lib.

## Entry Points Exposed by Clients of the Traffic Control Interface

When registering as a client of the traffic control interface, the user is expected to provide a list of entry points, which can be called by the TCI. These are defined in this section.

- **CIAddFlowComplete**
- **CIDeleteFlowComplete**
- **CIModifyFlowComplete**
- **CINotifyHandler**

---

## CIAddFlowComplete

The **CIAddFlowComplete** function is used by traffic control to notify the client of the completion of its previous call to the **TcAddFlow** function.

The **CIAddFlowComplete** callback function is optional. If this function is not specified, **TcAddFlow** will block until it completes.

```
VOID CIAddFlowComplete(  
    HANDLE CFlowCtx,  
    DWORD Status  
);
```

### Parameters

#### *CFlowCtx*

[in] Client provided-flow context handle. This can be the container used to hold an arbitrary client-defined context for this instance of the client. This value will be the same as the value provided by the client during its corresponding call to **TcAddFlow**.

#### *Status*

[in] Completion status for the **TcAddFlow** request. This value may be any of the return values possible for the **TcAddFlow** function, with the exception of **ERROR\_SIGNAL\_PENDING**.

---

**Note** Use of the **CIAddFlowComplete** function requires administrative privilege.

---

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

### See Also

**TcAddFlow**

## CIDeleteFlowComplete

The **CIDeleteFlowComplete** function is used by traffic control to notify the client of the completion of its previous call to the **TcDeleteFlow** function.

The **CIDeleteFlowComplete** callback function is optional. If this function is not specified, **TcDeleteFlow** will block until it completes.

```
VOID CIDeleteFlowComplete(  
    HANDLE CFlowCtx,  
    DWORD Status  
);
```

### Parameters

#### *CFlowCtx*

[in] Client provided–flow context handle. This can be the container used to hold an arbitrary client-defined context for this instance of the client. This value will be the same as the value provided by the client during its corresponding call to **TcDeleteFlow**.

#### *Status*

[in] Completion status for the **TcDeleteFlow** request. This value may be any of the return values possible for the **TcDeleteFlow** function, with the exception of **ERROR\_SIGNAL\_PENDING**.

---

**Note** Use of the **CIDeleteFlowComplete** function requires administrative privilege.

---

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

### + See Also

**TcDeleteFlow**

---

## CIModifyFlowComplete

The **CIModifyFlowComplete** function is used by traffic control to notify the client of the completion of its previous call to the **TcModifyFlow** function.

The **CIModifyFlowComplete** callback function is optional. If this function is not specified, **TcModifyFlow** will block until it completes.



```

VOID CModifyFlowComplete(
    HANDLE CIFlowCtx,
    DWORD Status
);

```

### Parameters

#### *CIFlowCtx*

[in] Client provided–flow context handle. This can be the container used to hold an arbitrary client-defined context for this instance of the client. This value will be the same as the value provided by the client during its corresponding call to

**TcModifyFlow**.

#### *Status*

[in] Completion status for the **TcModifyFlow** request. This value may be any of the return values possible for the **TcModifyFlow** function, with the exception of **ERROR\_SIGNAL\_PENDING**.

---

**Note** Use of the **CIModifyFlowComplete** function requires administrative privilege.

---

#### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

#### + See Also

**TcModifyFlow**

---

## CINotifyHandler

The **CINotifyHandler** function is used by traffic control to notify the client of various traffic control–specific events, including the deletion of flows, changes in filter parameters, or the closing of an interface.

The **CINotifyHandler** callback function should be exposed by all clients using traffic control services.

```

VOID CINotifyHandler(
    HANDLE CRegCtx,
    HANDLE CIIfcCtx,
    ULONG Event,
    ULONG SubCode,
    ULONG BufSize,
    PVOID Buffer
);

```

## Parameters

### *ClRegCtx*

[in] Client registration context, provided to traffic control by the client with the client's call to the **TcRegisterClient** function.

### *ClIfcCtx*

[in] Client interface context, provided to traffic control by the client with the client's call to the **TcOpenInterface** function. Note that during a TC\_NOTIFY\_IFC\_UP event, *ClIfcCtx* is not available and will be set to NULL.

### *Event*

[in] Describes the notification event. See the Remarks section for a list of notification events.

### *SubCode*

[in] Value used to further qualify a notification event.

### *BufSize*

[in] Size of the buffer included with the notification event.

### *Buffer*

[in] Buffer containing the detailed event information associated with *Event* and *SubCode*.

## Remarks

Notification events may require the traffic control client to take particular action or respond appropriately, for example, removing filters or enumerating flows for affected interfaces.

The following table describes the various notification events.

<b>Event</b>	<b>SubCode</b>	<b>Buffer contents</b>	<b>Remarks</b>
TC_NOTIFY_IFC_UP	None	InterfaceName of the new interface	A new traffic control interface is coming up, and the list of addresses is indicated.
TC_NOTIFY_IFC_CLOSE	Reason for close	InterfaceName of the new interface	Indicates an interface that was opened by the client is being closed by the system. Note that the interface and its supported flows and filters are closed by the system upon return from the notification handler. The client does not need to close the interface, delete flows, or delete filters.
TC_NOTIFY_IFC_CHANGE	None	New parameter value	Used to notify clients that have registered for interface change notification through the <i>NotifyChange</i> parameter of the <b>TcQueryInterface</b> function.

(continued)

*(continued)*

Event	SubCode	Buffer contents	Remarks
TC_NOTIFY_PARAM_CHANGED	Pointer to the GUID for a traffic control parameter queried using the <b>TcQueryInterface</b> function.	New parameter value	This event is notified as a result of a change in a parameter previously queried with the <i>NotifyChange</i> flag set.
TC_NOTIFY_FLOW_CLOSE	<i>CIFlowCtx</i>	InterfaceName of the closed interface	Indicates system closure of a client-created flow. The system deletes all associated filters.

---

**Note** Use of the **CINotifyHandler** function requires administrative privilege.

---

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

---

## Traffic Control Structures

This section describes the following traffic control structures:

- **ADDRESS\_LIST\_DESCRIPTOR**
- **ENUMERATION\_BUFFER**
- **IP\_PATTERN**
- **QOS\_DIFFSERV\_RULE**
- **TC\_GEN\_FILTER**
- **TC\_GEN\_FLOW**
- **TC\_IFC\_DESCRIPTOR**
- **TCI\_CLIENT\_FUNC\_LIST**

There is also a collection of structures associated with traffic control that enable application developers to query or gather statistical information regarding the packet scheduler and its traffic control faculties. That collection of statistic-enabling structures is the following:

- **GUID**
- **PS\_COMPONENT\_STATS**

- **PS\_ADAPTER\_STATS**
- **PS\_FLOW\_STATS**
- **PS\_CONFORMER\_STATS**
- **PS\_SHAPER\_STATS**
- **PS\_DRRSEQ\_STATS**

Many structures explained in this section work closely with the **FLOWSPEC** structure.

---

## ADDRESS\_LIST\_DESCRIPTOR

The **ADDRESS\_LIST\_DESCRIPTOR** structure provides network address descriptor information for a given interface. For point-to-point media such as WAN connections, the list is a pair of addresses, the first of which is always the local or source address, the second of which is the remote or destination address. Note that the members of **ADDRESS\_LIST\_DESCRIPTOR** are defined in `Ntddndis.h`.

```
typedef struct _ADDRESS_LIST_DESCRIPTOR {  
  
    NDIS_MEDIUM          MediaType;  
    NETWORK_ADDRESS_LIST AddressList;  
  
} ADDRESS_LIST_DESCRIPTOR, *PADDRESS_LIST_DESCRIPTOR;
```

### Members

#### MediaType

Pointer to the media type of the interface. **NDIS\_MEDIUM** is a defined type from definitions provided in `Ntddndis.h`.

#### AddressList

Pointer to the address list for the interface. **NETWORK\_ADDRESS\_LIST** is defined in `Ntddndis.h`.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in `Traffic.h`.

---

## ENUMERATION\_BUFFER

The **ENUMERATION\_BUFFER** structure contains information specific to a given flow, including flow name, the number of filters associated with the flow, and an array of filters associated with the flow.

```
typedef struct _ENUMERATION_BUFFER {  
  
    ULONG          Length;  
    ULONG          OwnerProcessId;  
    USHORT         FlowNameLength;  
    WCHAR          FlowName[MAX_STRING_LENGTH];  
    PTC_GEN_FLOW   pFlow;  
    ULONG          NumberOfFilters;  
    TC_GEN_FILTER  GenericFilter[1];  
  
} ENUMERATION_BUFFER, *PENUMERATION_BUFFER;
```

## Members

### Length

Number of bytes from the beginning of the **ENUMERATION\_BUFFER** to the next **ENUMERATION\_BUFFER**.

### OwnerProcessId

Identifies the owner of the process.

### FlowNameLength

Specifies the length of the **FlowName** member.

### FlowName

Array of **WCHAR** characters, of length **MAX\_STRING\_LENGTH**, that specifies the flow name.

### pFlow

Pointer to the corresponding **TC\_GEN\_FLOW** structure. This structure is placed immediately after the array of **TC\_GEN\_FILTERS** and is included in **Length**.

### NumberOfFilters

Specifies the number of filters associated with the flow.

### GenericFilter

Array of **TC\_GEN\_FILTER** structures. The number of elements in the array corresponds to the number of filters attached to the specified flow. Note that in order to enumerate through the array of **TC\_GEN\_FILTER** structures, you need to increment the pointer to the current **TC\_GEN\_FILTER** by using the following:

`sizeof(TC_GEN_FILTER) + 2 * [the pattern size of the current TC_GEN_FILTER structure].`

## Example

The following example shows how to use the **ENUMERATION\_BUFFER** with traffic control. It also provides an example of enumerating through the array of **TC\_GEN\_FILTER** structures found in **GenericFilter**.

```
#define UNICODE

#include <winsock2.h>
#include <windows.h>
#include <qos.h>
#include <ntddndis.h>
#include <traffic.h>
#include <tcerror.h>
#include <stdlib.h>
#include <stdio.h>

#define INITIAL_IFC_BUFFER_SIZE    4096
#define INITIAL_FLOW_BUFFER_SIZE  1024
#define FLOW_NAME_BUFFER_SIZE     1024
#define QUERY_FLOW_BUFFER_SIZE    2048
#define QUERY_IFC_BUFFER_SIZE     2048

void ExamineFlow(int nFlow, PENUMERATION_BUFFER pEnumBuffer)
{
    PTC_GEN_FILTER pCurrentFilter = pEnumBuffer->GenericFilter;

    printf("    Flow %d:\n",nFlow);
    printf("        Name:                %ws\n",pEnumBuffer->FlowName);
    printf("        OwnerProcessID:       %d\n",pEnumBuffer->OwnerProcessId);
    printf("        Number of Filters:    %d\n",pEnumBuffer->NumberOfFilters);

    for (ULONG i=0; i < pEnumBuffer->NumberOfFilters; i++)
    {
        printf("            Filter %d: ",i);
        if (pCurrentFilter->AddressType == NDIS_PROTOCOL_ID_TCP_IP)
            printf("TCP/IP Filter\n");
        else if (pCurrentFilter->AddressType == NDIS_PROTOCOL_ID_IPX)
            printf("IPX Filter\n");
        else
            printf("Other Filter Type\n");
        pCurrentFilter = (PTC_GEN_FILTER)((PBYTE)pCurrentFilter +
            sizeof(TC_GEN_FILTER) + (pCurrentFilter->PatternSize * 2));
    }
}

void EnumFlows(HANDLE hIfc)
{
    PENUMERATION_BUFFER pEnumBuffer, pCurrentEnumBuffer;
    ULONG EnumBufferSize = INITIAL_FLOW_BUFFER_SIZE;
```

(continued)



(continued)

```
ULONG FilledEnumBufferSize;
HANDLE hEnum;

ULONG FlowCount, err, i;
int nFlow = 0;

pEnumBuffer = (PENUMERATION_BUFFER) malloc ( EnumBufferSize );

if (pEnumBuffer == NULL) {
    printf("Error: malloc");
    return;
}

hEnum = NULL;

do
{
    FilledEnumBufferSize = EnumBufferSize;
    FlowCount = -1; // Request All Flows

    err = TcEnumerateFlows ( hIfc,
                            &hEnum,
                            &FlowCount,
                            &FilledEnumBufferSize,
                            pEnumBuffer );

    if (err == ERROR_INSUFFICIENT_BUFFER)
    {
        //
        // The buffer was not large enough to hold any entries.
        // Make it bigger.
        //

        EnumBufferSize *= 2;
        pEnumBuffer = (PENUMERATION_BUFFER) realloc( pEnumBuffer,
EnumBufferSize );
    }
    else if (err == NO_ERROR)
    {
        pCurrentEnumBuffer = pEnumBuffer;

        for (ULONG i = 0; i < FlowCount; i++)
        {
```

```

        nFlow++;
        ExamineFlow(nFlow, pCurrentEnumBuffer);

        //
        // Advance to the next buffer
        //

        pCurrentEnumBuffer =
(PENUMERATION_BUFFER)((PBYTE)pCurrentEnumBuffer + pCurrentEnumBuffer->Length);
    }
}
} while ( (hEnum != NULL && err == NO_ERROR) || (err ==
ERROR_INSUFFICIENT_BUFFER) );

free(pEnumBuffer);

if (err != NO_ERROR && err != ERROR_INVALID_DATA)
{
    printf("Error: TcEnumerateFlow");
    return;
}
}

void ExamineInterface(HANDLE hClient, int nIfcs, PTC_IFC_DESCRIPTOR pCurrentIfc)
{
    HANDLE hIfc;
    ULONG err;

    printf("Interface %d:\n    Name: %ws\n\n",nIfcs,pCurrentIfc->pInterfaceName);

    err = TcOpenInterface(
        pCurrentIfc->pInterfaceName,
        hClient,
        NULL,
        &hIfc);

    if (err != NO_ERROR)
    {
        printf("Error: TcOpenInterface");
        return;
    }

    EnumFlows(hIfc);
}

```

*(continued)*



*(continued)*

```
err = TcCloseInterface(hIfc);
if (err != NO_ERROR)
{
    printf("Error: TcCloseInterface");
    return;
}
}

void EnumInterfaces(HANDLE hClient)
{
    PTC_IFC_DESCRIPTOR pIfcs, pCurrentIfc;
    ULONG BufferSize = INITIAL_IFC_BUFFER_SIZE;
    ULONG ActualBufferSize, RemainingBufferSize;
    int nIfcs = 0;

    while (true)
    {
        pIfcs = (PTC_IFC_DESCRIPTOR) malloc(BufferSize);

        if (pIfcs == NULL) {
            printf("Error: malloc");
            return;
        }

        ULONG err = TcEnumerateInterfaces(
            hClient,
            &ActualBufferSize,
            pIfcs);

        if (err == NO_ERROR)
        {
            // We've got the data
            break;
        }
        else if (err == ERROR_INSUFFICIENT_BUFFER)
        {
            free(pIfcs);
            BufferSize *= 2;
        }
        else
        {
            printf("Error: TcEnumerateInterfaces");
            return;
        }
    }
}
```

```
    }

    RemainingBufferSize = ActualBufferSize;
    pCurrentIfc = pIfcs;
    while (RemainingBufferSize > 0)
    {
        nIfcs++;
        ExamineInterface(hClient, nIfcs, pCurrentIfc);

        RemainingBufferSize -= pCurrentIfc->Length;
        pCurrentIfc = (PTC_IFC_DESCRIPTOR) ((PBYTE) pCurrentIfc + pCurrentIfc-
>Length);
    }

    free(pIfcs);
}

VOID CALLBACK NotifyHandler(
    HANDLE ClRegCtx,
    HANDLE ClIfcCtx,
    ULONG Event,
    ULONG SubCode,
    ULONG BufSize,
    PVOID Buffer)
{
    switch (Event) {
        case TC_NOTIFY_IFC_UP:
        case TC_NOTIFY_IFC_CLOSE:
        case TC_NOTIFY_IFC_CHANGE:
        case TC_NOTIFY_PARAM_CHANGED:
        case TC_NOTIFY_FLOW_CLOSE:
            break;
    }
}

void main()
{
    ULONG err;
    TCI_CLIENT_FUNC_LIST Handlers;
    HANDLE hClient;

    Handlers.ClNotifyHandler = NotifyHandler;

```

*(continued)*

*(continued)*

```

    Handlers.CIAddFlowCompleteHandler = NULL;
    Handlers.CIModifyFlowCompleteHandler = NULL;
    Handlers.CIDeleteFlowCompleteHandler = NULL;

    err = TcRegisterClient(CURRENT_TCI_VERSION, (HANDLE) NULL, &Handlers,
&hClient);
    if (err != NO_ERROR)
    {
        printf("Error: TcRegisterClient");
        exit(1);
    }

    EnumInterfaces(hClient);

    err = TcDeregisterClient(hClient);
    if (err != NO_ERROR)
    {
        printf("Error: TcDeregisterClient");
        exit(1);
    }
}

```

**!** Requirements**Windows NT/2000:** Requires Windows 2000.**Windows 95/98:** Unsupported.**Header:** Declared in Traffic.h.**+** See Also**TC\_GEN\_FLOW, TC\_GEN\_FILTER**

## IP\_PATTERN

The **IP\_PATTERN** structure applies a specific pattern or corresponding mask for the IP protocol. The **IP\_PATTERN** structure designation is used by the traffic control interface in the application of packet filters:

```

typedef struct _IP_PATTERN {
    ULONG    Reserved1;
    ULONG    Reserved2;

    ULONG    SrcAddr;

```



```
    ULONG    DstAddr;

    union {
        struct { USHORT s_srcport,s_dstport; } S_un_ports;
        struct { UCHAR s_type,s_code; USHORT filler; } S_un_icmp;
        ULONG S_Spi;
    } S_un;

    UCHAR    ProtocolId;
    UCHAR    Reserved3[3];

#define tcSrcPort      S_un.S_un_ports.s_srcport
#define tcDstPort      S_un.S_un_ports.s_dstport
#define tcIcmpType     S_un.S_un_icmp.s_type
#define tcIcmpCode     S_un.S_un_icmp.s_code
#define tcSpi          S_un.S_Spi

} IP_PATTERN, *PIP_PATTERN;
```

## Members

### Reserved1

Reserved for future use.

### Reserved2

Reserved for future use.

### SrcAddr

Source address.

### DstAddr

Destination address.

### ProtocolId

Protocol identifier.

### Reserved3

Reserved for future use.

### tcSrcPort

Source port.

### tcDstPort

Destination port.

### tcIcmpType

ICMP message type

### tcIcmpCode

ICMP message code.

### tcSpi

Service provider interface.

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

---

## QOS\_DIFFSERV\_RULE

The **QOS\_DIFFSERV\_RULE** structure is used in conjunction with the traffic control object **QOS\_OBJECT\_DIFFSERV** to provide Diffserv rules for a given flow.

```
typedef struct _QOS_DIFFSERV_RULE {
    UCHAR InboundDSField;
    UCHAR ConformingOutboundDSField;
    UCHAR NonConformingOutboundDSField;
    UCHAR ConformingUserPriority;
    UCHAR NonConformingUserPriority;
} QOS_DIFFSERV_RULE, *LPQOS_DIFFSERV_RULE;
```

### Members

#### InboundDSField

Diffserv code point (DSCP) on the inbound packet. **InboundDSField** must be unique for the interface, otherwise the flow addition will fail.

Valid range is 0x00–0x3F.

#### ConformingOutboundDSField

Diffserv code point (DSCP) marked on all conforming packets on the flow. This member can be used to re-mark the packet before it is forwarded.

Valid range is 0x00–0x3F.

#### NonConformingOutboundDSField

Diffserv code point (DSCP) marked on all nonconforming packets on the flow. This member can be used to remark the packet before it is forwarded.

Valid range is 0x00–0x3F.

#### ConformingUserPriority

UserPriority value marked on all conforming packets on the flow. This member can be used to remark the packet before it is forwarded.

Valid range is 0–7

#### NonConformingUserPriority

UserPriority value marked on all nonconforming packets on the flow. This member can be used to remark the packet before it is forwarded.

Valid range is 0–7

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

**+** See Also

**QOS\_OBJECT\_DIFFSERV**

---

## TC\_GEN\_FILTER

The **TC\_GEN\_FILTER** structure creates a filter that matches a certain set of packet attributes or criteria, which can subsequently be used to associate packets that meet the attribute criteria with a particular flow. The **TC\_GEN\_FILTER** structure uses its **AddressType** member to indicate a specific filter type to apply to the filter.

```
typedef struct _TC_GEN_FILTER {
    USHORT    AddressType,    // defines specific filter type,
                          // Defined in ntddndis.h:
                          // NDIS_PROTOCOL_ID_TCP_IP
                          // NDIS_PROTOCOL_ID_IPX, etc.
    ULONG    PatternSize,    // sizeof specific pattern
    PVOID    Pattern,        // specific format, e.g.
                          // IP_PATTERN
    PVOID    Mask            // same type as Pattern
} TC_GEN_FILTER, *TC_GEN_FILTER;
```

### Members

#### AddressType

Defines the filter type to be applied with the filter, as defined in Ntddndis.h. With the designation of a specific filter in **AddressType**, the generic filter structure **TC\_GEN\_FILTER** provides a specific filter type.

#### PatternSize

Size of the **Pattern** member.

#### Pattern

Indicates the specific format of the pattern to be applied to the filter, such as **IP\_PATTERN**. The pattern specifies which bits of a given packet should be evaluated when determining whether a packet is included in the filter.

#### Mask

A bitmask applied to the bits designated in the **Pattern** member. The application of the **Mask** member to the **Pattern** member determines which bits in the **Pattern** member are significant (should be applied to the filter criteria). Note that the **Mask** member must be of the same type as the **Pattern** member.

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

## TC\_GEN\_FLOW

The **TC\_GEN\_FLOW** structure creates a generic flow for use with the traffic control interface. The flow is customized through the use of the structure's members.

```
typedef struct _TC_GEN_FLOW {
    FLOWSPEC        SendingFlowspec,
    FLOWSPEC        ReceivingFlowspec,
    ULONG           TcObjectsLength, // number of optional bytes
    UCHAR           TcObjects[1]
} TC_GEN_FLOW, *PTC_GEN_FLOW;
```

### Members

#### **SendingFlowspec**

**FLOWSPEC** structure for the sending direction of the flow.

#### **ReceivingFlowspec**

**FLOWSPEC** structure for the receiving direction of the flow.

#### **TcObjectsLength**

Length of **TcObjects**.

#### **TcObjects**

Buffer that contains an array of traffic control objects specific to the given flow. The **TcObjects** member can contain objects from the list of currently supported objects. Definitions of these objects can be found in Qos.h and Traffic.h:

**QOS\_OBJECT\_DS\_CLASS**

**QOS\_OBJECT\_TRAFFIC\_CLASS**

**QOS\_OBJECT\_DIFFSERV**

**QOS\_OBJECT\_SD\_MODE**

**QOS\_OBJECT\_SHAPING\_RATE**

**QOS\_OBJECT\_END\_OF\_LIST**

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

 See Also

FLOWSPEC

---

## TC\_IFC\_DESCRIPTOR

The **TC\_IFC\_DESCRIPTOR** structure is an interface identifier used to enumerate interfaces.

```
typedef struct _TC_IFC_DESCRIPTOR {
    ULONG                Length;
    LPWSTR               pInterfaceName;
    LPWSTR               pInterfaceID;
    ADDRESS_LIST_DESCRIPTOR AddressListDesc;
} TC_IFC_DESCRIPTOR, *PTC_IFC_DESCRIPTOR;
```

### Members

#### Length

Number of bytes from the beginning of the **TC\_IFC\_DESCRIPTOR** to the next **TC\_IFC\_DESCRIPTOR**.

#### pInterfaceName

Pointer to a zero-terminated Unicode string representing the name of the packet shaper interface. This name is used in subsequent TC API calls to reference the interface.

#### pInterfaceID

Pointer to a zero-terminated Unicode string naming the **DeviceName** of the interface.

#### AddressListDesc

Network address list descriptor.

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

---

## TCI\_CLIENT\_FUNC\_LIST

The **TCI\_CLIENT\_FUNC\_LIST** structure is used by the traffic control interface to register and then access client-callback functions. Each member of **TCI\_CLIENT\_FUNC\_LIST** is a pointer to the client provided-callback function.



```
typedef struct _TCI_CLIENT_FUNC_LIST {  
  
    TCI_NOTIFY_HANDLER          CINotifyHandler,  
    TCI_ADD_FLOW_COMPLETE_HANDLER C1AddFlowCompleteHandler,  
    TCI_MOD_FLOW_COMPLETE_HANDLER C1ModifyFlowCompleteHandler,  
    TCI_DEL_FLOW_COMPLETE_HANDLER C1DeleteFlowCompleteHandler,  
  
} TCI_CLIENT_FUNC_LIST, *PTCI_CLIENT_FUNC_LIST;
```

## Members

### **CINotifyHandler**

Pointer to the client-callback function **CINotifyHandler**.

### **CIAddFlowCompleteHandler**

Pointer to the client-callback function **CIAddFlowCompleteHandler**.

### **CIModifyFlowCompleteHandler**

Pointer to the client-callback function **CIModifyFlowCompleteHandler**.

### **CIDeleteFlowCompleteHandler**

Pointer to the client-callback function **CIDeleteFlowCompleteHandler**.

## Remarks

Any member of the **TCI\_CLIENT\_FUNC\_LIST** structure can be NULL except **TCI\_NOTIFY\_HANDLER**.

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

### See Also

**CINotifyHandler**, **CIAddFlowComplete**, **CIModifyFlowComplete**,  
**CIDeleteFlowComplete**

---

# GUID

Traffic control uses **GUIDs** to convey information about the status of packet shaper interfaces, to report errors, and to provide statistics.

The **GUID\_QOS\_STATISTICS\_BUFFER** structure is actually an array of **PS\_COMPONENT\_STATS** structures, which in turn serve as a container structure, holding one of the five following structures in its **Stats** member.

- **PS\_ADAPTER\_STATS**
- **PS\_FLOW\_STATS**
- **PS\_CONFORMER\_STATS**
- **PS\_SHAPER\_STATS**
- **PS\_DRRSEQ\_STATS**

The following table lists GUIDs used with packet shaper on Windows 2000 QOS-enabled computers. In the scope column, P/I indicates that the GUID is based per-interface, and P/F indicates it is based per-flow.

GUID	Size	Scope	Settable?	Notification?	Description
GUID_QOS_REMAINING_BANDWIDTH	ULONG	P/I	No	Yes	The remaining amount of reservable bandwidth on this interface.
GUID_QOS_LATENCY	ULONG	P/I	No	No	The latency of this interface.
GUID_QOS_FLOW_COUNT	ULONG	P/I	No	Yes	Number of active flows.
GUID_QOS_NON_BESTEFFECTORT_LIMIT	ULONG	P/I	No	No	The total amount of reservable bandwidth.
GUID_QOS_MAX_OUTSTANDING_SENDS	ULONG	P/I	No	No	Maximum number of sends that can be outstanding on this interface.
GUID_QOS_STATISTICS_BUFFER	See the following.	P/I, P/F	Yes	No	Statistics collected for flow or interface. The <b>GUID_QOS_STATISTICS_BUFFER</b> structure is actually an array of <b>PS_COMPONENT_STATS</b> structures, as described in the preceding.
GUID_QOS_FLOW_MODE	ULONG	P/I	Yes	No	Mode of operation for this interface: STANDARD or DIFFSERV. Note that you must specify the constants rather than Standard or Diffserv when using this GUID.
GUID_QOS_ISSLOW_FLOW	ULONG	P/F	No	No	Indicates an ISSLOW flow.

(continued)

*(continued)*

GUID	Size	Scope	Set- table?	Notifi- cation?	Description
GUID_QOS_ TIMER_ RESOLUTION	ULONG	none	No	No	Timer resolution, in microseconds.
GUID_QOS_ FLOW_IP_ CONFORMING	ULONG	P/F	No	No	The conforming DSCP value for this flow.
GUID_QOS_FLOW_ IP_ NONCONFORMING	ULONG	P/F	No	No	The nonconforming SCP value for this flow.
GUID_QOS_FLOW_ 8021P_ CONFORMING	ULONG	P/F	No	No	The conforming 802.1p value for this flow
GUID_QOS_FLOW_ 8021P_ NONCONFORMING	ULONG	P/F	No	No	The nonconforming 802.1p value for this flow.

**Remarks**

In the **GUID\_QOS\_STATISTICS\_BUFFER** GUID described in the table above, you must pass in a buffer that is large enough to hold all returned **PS\_COMPONENT\_STATS** structures.

**! Requirements**

**Version:** Requires MAPI 1.0 or later.

**Header:** Declared in Mapiguide.h.

## PS\_COMPONENT\_STATS

The **PS\_COMPONENT\_STATS** structure enables applications to get statistical information regarding their TC-enabled flow. This structure obtains information from **GUID\_QOS\_STATISTICS\_BUFFER** GUID. This GUID actually is an array of **PS\_COMPONENT\_STATS**, with each element of that array (each **PS\_COMPONENT\_STATS** structure) containing one of the five **PS\_\*** structure types explained subsequently.

```
typedef struct _PS_COMPONENT_STATS
{
    #define PS_COMPONENT_ADAPTER        1
    #define PS_COMPONENT_FLOW          2
    #define PS_COMPONENT_CONFORMER    3
```

```
#define PS_COMPONENT_SHAPER      4
#define PS_COMPONENT_DRRSEQ     5

    ULONG Type;
    ULONG Length;
    UCHAR Stats[1];
} PS_COMPONENT_STATS, *PPS_COMPONENT_STATS;
```

## Members

### Type

Indicates one of the following types of **PS\_\*** structure contained in the **Stats** member:

- **PS\_ADAPTER\_STATS**
- **PS\_FLOW\_STATS**
- **PS\_CONFORMER\_STATS**
- **PS\_SHAPER\_STATS**
- **PS\_DRRSEQ\_STATS**

### Length

Length of the **Stats** member, in bytes.

### Stats

Array of structures of the type indicated in the **Type** member.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Ntddpsch.h.

---

## PS\_ADAPTER\_STATS

The **PS\_ADAPTER\_STATS** structure provides statistical packet shaper information about a specified adapter. Note that the **PS\_ADAPTER\_STATS** structure is used in conjunction with the **PS\_COMPONENT\_STATS** structure.

```
typedef struct _PS_ADAPTER_STATS {

    ULONG OutOfPackets;
    ULONG FlowsOpened;
    ULONG FlowsClosed;
    ULONG FlowsRejected;
    ULONG FlowsModified;
    ULONG FlowModsRejected;
    ULONG MaxSimultaneousFlows;

} PS_ADAPTER_STATS, *PPS_ADAPTER_STATS;
```

## Members

### OutOfPackets

Number of instances in which the adapter had no packets to transmit on the specified adapter.

### FlowsOpened

Number of flows opened on the adapter.

### FlowsClosed

Number of flows closed on the adapter.

### FlowsRejected

Number of flows that were rejected due to packet shaper constraints on the adapter.

### FlowsModified

Number of flows that were modified on the adapter.

### FlowModsRejected

Number of flow modifications that were rejected on the adapter due to packet shaper constraints.

### MaxSimultaneousFlows

Maximum number of simultaneous flows.

## ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Ntddpsch.h.

---

## PS\_FLOW\_STATS

The **PS\_FLOW\_STATS** structure provides statistical packet shaper information about a particular flow. Note that the **PS\_FLOW\_STATS** structure is used in conjunction with the **PS\_COMPONENT\_STATS** structure.

```
typedef struct _PS_FLOW_STATS {
    ULONG           DroppedPackets;
    ULONG           PacketsScheduled;
    ULONG           PacketsTransmitted;
    LARGE_INTEGER   BytesScheduled;
    LARGE_INTEGER   BytesTransmitted;
} PS_FLOW_STATS, *PPS_FLOW_STATS;
```

## Members

### DroppedPackets

Number of packets that have been dropped from the flow.

**PacketsScheduled**

Number of packets that have been scheduled for transmission on the flow.

**PacketsTransmitted**

Number of packets that have been transmitted on the flow.

**BytesScheduled**

Number of bytes that have been scheduled for transmission on the flow.

**BytesTransmitted**

Number of bytes that have been transmitted on the flow.

**! Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Ntddpsch.h.

---

## PS\_CONFORMER\_STATS

The **PS\_CONFORMER\_STATS** structure provides statistical packet shaper information about a particular flow. Note that the **PS\_CONFORMER\_STATS** structure is used in conjunction with the **PS\_COMPONENT\_STATS** structure.

```
typedef struct _PS_CONFORMER_STATS {
    ULONG        NonconformingPacketsScheduled;
} PS_CONFORMER_STATS, *PPS_CONFORMER_STATS;
```

**Members****NonconformingPacketsScheduled**

Number of nonconforming packets that have been scheduled on the flow or interface.

**! Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Ntddpsch.h.

---

## PS\_SHAPER\_STATS

The **PS\_SHAPER\_STATS** structure provides statistical packet shaper information about the computer's Windows 2000 packet shaper component. Note that the **PS\_SHAPER\_STATS** structure is used in conjunction with the **PS\_COMPONENT\_STATS** structure.

```
typedef struct _PS_SHAPER_STATS {
    ULONG MaxPacketsInShaper;
    ULONG AveragePacketsInShaper;
} PS_SHAPER_STATS, *PPS_SHAPER_STATS;
```

## Members

### MaxPacketsInShaper

Maximum number of packets that have been in the packet shaper for the flow or interface.

### AveragePacketsInShaper

Average number of packets that have been in the packet shaper for the flow or interface.

## ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Ntddpsch.h.

---

# PS\_DRRSEQ\_STATS

The **PS\_DRRSEQ\_STATS** structure provides network interface card (NIC) and packet sequencer–packet shaper statistics. Note that the **PS\_DRRSEQ\_STATS** structure is used in conjunction with the **PS\_COMPONENT\_STATS** structure.

```
typedef struct _PS_DRRSEQ_STATS {
    ULONG MaxPacketsInNetcard;
    ULONG AveragePacketsInNetcard;
    ULONG MaxPacketsInSequencer;
    ULONG AveragePacketsInSequencer;
    ULONG NonconformingPacketsTransmitted;
} PS_DRRSEQ_STATS, *PPS_DRRSEQ_STATS;
```

## Members

### MaxPacketsInNetcard

Maximum number of packets that have been queued in the network interface card for the flow or interface.

### AveragePacketsInNetcard

Average number of packets queued in the network interface card for the flow or interface.

### MaxPacketsInSequencer

Maximum number of packets that have been queued in the packet sequencer for the flow or interface.

**AveragePacketsInSequencer**

Average number of packets that have been queued in the packet sequencer for the flow or interface.

**NonconformingPacketsTransmitted**

Number of nonconforming packets that have been transmitted for the flow or interface.

**! Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Ntddpsch.h.

## Traffic Control Objects

Applications that require specific or granular traffic control, beyond what is available in the TC API, can use traffic control objects. The use of traffic control objects with the TC API is similar to the use of QOS Objects with the QOS API.

Like QOS objects, traffic control object follow a stringent structure. Traffic control objects always include an information header that specifies the type and length of the traffic control object to which it is attached, followed by the object itself.

Some objects are shared by QOS and traffic control. Those objects are defined in the QOS Objects section, and are the following:

- **QOS\_OBJECT\_SD\_MODE**
- **QOS\_OBJECT\_SHAPING\_RATE**

This section describes the following traffic control objects:

- **QOS\_OBJECT\_TRAFFIC\_CLASS**
- **QOS\_OBJECT\_DS\_CLASS**
- **QOS\_OBJECT\_DIFFSERV**

As with QOS objects, traffic control objects begin each object with the **QOS\_OBJECT\_HDR**, which is used by QOS components that interrogate QOS objects to ascertain the type of object that follows the header.



## QOS\_OBJECT\_TRAFFIC\_CLASS

The traffic control object **QOS\_OBJECT\_TRAFFIC\_CLASS** is used to override the default UserPriority value ascribed to packets that classify to (are associated with) a given flow. By default, the UserPriority value of a flow is derived from the ServiceType; the capability to override the default UserPriority is necessary because packets can be tagged in their Layer 2 headers (such as an 802.1p header) to specify their priority to Layer-2 devices. Using **QOS\_OBJECT\_TRAFFIC\_CLASS** enables application developers to override the default UserPriority setting.

```
typedef struct _QOS_TRAFFIC_CLASS
{
    QOS_OBJECT_HDR  ObjectHdr;
    ULONG           TrafficClass;
} QOS_TRAFFIC_CLASS, *LPQOS_TRAFFIC_CLASS;
```

### Members

#### ObjectHdr

The QOS object **QOS\_OBJECT\_HDR**. The object type for this traffic control object should be **QOS\_OBJECT\_TRAFFIC\_CLASS**.

#### TrafficClass

User priority value of the flow. The valid range is zero through seven. The following settings are chosen (by default) when the **QOS\_OBJECT\_TRAFFIC\_CLASS** traffic control object is not used.

Service Types	Traffic Class Default Value
ServiceTypeBestEffort, ServiceTypeQualitative	0
ServiceTypeControlledLoad	4
ServiceTypeGuaranteed	5
ServiceTypeNetworkControl	7
Non Conformant traffic	1

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

### + See Also

**QOS\_DIFFSERV\_RULE, QOS\_OBJECT\_DS\_CLASS, QOS\_OBJECT\_DIFFSERV**

## QOS\_OBJECT\_DS\_CLASS

The traffic control object **QOS\_OBJECT\_DS\_CLASS** enables application developers to override the default Diffserv code point (DSCP) value for the IP packets associated with a given flow. By default, the DSCP value is derived from the flow's ServiceType.

```
typedef struct _QOS_DS_CLASS
{
    QOS_OBJECT_HDR  ObjectHdr;
    ULONG           DSField;
} QOS_DS_CLASS, *LPQOS_DS_CLASS;
```

### Members

#### ObjectHdr

The QOS object **QOS\_OBJECT\_HDR**. The object type for this traffic control object should be **QOS\_OBJECT\_DS\_CLASS**.

#### DSField

User priority value for the flow. The valid range is 0x00 through 0x3F. The following settings are chosen (by default) when the QOS\_OBJECT\_DS\_CLASS traffic control object is not used.

Service types	Traffic class default value
ServiceTypeBestEffort, ServiceTypeQualitative	0
ServiceTypeControlledLoad	0x18
ServiceTypeGuaranteed	0x28
ServiceTypeNetworkControl	0x30
Non Conformant traffic	0x00

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

### + See Also

**QOS\_DIFFSERV\_RULE, QOS\_OBJECT\_DIFFSERV,  
QOS\_OBJECT\_TRAFFIC\_CLASS**

## QOS\_OBJECT\_DIFFSERV

The **QOS\_OBJECT\_DIFFSERV** traffic control object is used to specify filters for the packet scheduler when it operates in Differentiated Services Mode.

```
typedef struct _QOS_DIFFSERV {
    QOS_OBJECT_HDR  ObjectHdr;
    ULONG           DSFieldCount;
    UCHAR           DiffservRule[1];
} QOS_DIFFSERV, *LPQOS_DIFFSERV;
```

### Members

#### ObjectHdr

The QOS object **QOS\_OBJECT\_HDR**. The object type for this traffic control object should be **QOS\_OBJECT\_DIFFSERV**.

#### DSFieldCount

Number of Diffserv Rules in on this object.

#### DiffservRule

Array of **QOS\_DIFFSERV\_RULE** structures.

### Remarks

The **QOS\_OBJECT\_DIFFSERV** object is used to specify the set of Diffserv rules that apply to the specified flow, all of which are specified in the **DiffservRule** member. Each Diffserv rule has an InboundDSField, which signifies the DSCP on the Inbound packet. The Diffserv Rules also have an OutboundDSCP and UserPriority values for conforming and non conforming packets. These indicate the DSCP and 802.1p values that go out on the forwarded packet. Note that the DSCP or UserPriority mapping based on ServiceType or **QOS\_OBJECT\_DS\_CLASS** or **QOS\_OBJECT\_TRAFFIC\_CLASS** is - not- used in this mode.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Traffic.h.

### + See Also

**QOS\_DIFFSERV\_RULE**, **QOS\_OBJECT\_DS\_CLASS**,  
**QOS\_OBJECT\_TRAFFIC\_CLASS**

---

## CHAPTER 17

# Local Policy Module API Reference

The Local Policy Module (LPM) API is a set of functions that the Policy Control Module (PCM) uses to interact with one or more LPMs. LPM API functions must be exported by LPMs to allow the PCM to invoke the LPM. This approach enables multiple LPMs to interact with the PCM, and subsequently, allows the PCM to return policy based–admission control decisions from one or more LPMs to the Admission Control Service (ACS). This concept is similar to a three-tiered approach, and enables multiple LPMs to easily coexist on a single system. Note, too, that LPMs may selectively accept or reject individual flows within a given policy based–decision request.

## LPM Functions

The following functions are exposed by the Microsoft-provided LPM, Msidlpm.dll.

- **cbpAdmitRsvpMsg**
- **cbpGetRsvpObjects**

The following functions must be exposed by each LPM; they allow the PCM to communicate requests from the Subnet Bandwidth Manager (SBM) (through the PCM). The Microsoft-supplied LPM, implemented as a DLL in Msidlpm.dll, also uses this API. Note that callback functions, identified by their **cbp** prefix, should not be exposed by LPMs; these callback functions, which are exposed by the PCM, facilitate deferred response from LPMs to PCM requests.

- **LPM\_AdmitRsvpMsg**
- **LPM\_CommitResv**
- **LPM\_Deinitialize**
- **LPM\_DeleteState**
- **LPM\_GetRsvpObjects**
- **LPM\_Initialize**
- **Lpm\_IpAddressTable**

## cbpAdmitRsvpMsg

The **cbpAdmitRsvpMsg** function is used by LPMs to return results for the **LPM\_AdmitRsvpMsg** request. LPMs should only use this function if they have returned **LPM\_RESULT\_DEFER** to the **LPM\_AdmitRsvpMsg** function call. The PCM will only accept results from this function within the result-time limit established by each LPM through the *ResultTimeLimit* parameter of the **LPM\_Initialize** function.

```
void
cbpAdmitRsvpMsg (
    LPM_HANDLE LpmHandle,
    RHANDLE RequestHandle,
    LPV LpmPriorityValue,
    int PolicyErrorCode,
    int PolicyErrorValue,
    VOID *Reserved
);
```

### Parameters

#### *LpmHandle*

[in] Unique handle for the LPM, as supplied in **LPM\_Initialize**. The PCM will ignore any result that is not accompanied by a valid LPM handle.

#### *RequestHandle*

[in] Unique handle that distinguishes this request from all other requests. LPMs must pass this handle to the PCM when returning results asynchronously for an individual request by calling **cbpAdmitRsvpMsg**. *RequestHandles* become invalid once results are returned, requiring each request to get its own unique *RequestHandle* from the PCM.

#### *LpmPriorityValue*

[in] LPM Priority Value assigned to the request. The PCM assumes **LPV\_DONT\_CARE** if *LpmPriorityValue* is invalid.

#### *PolicyErrorCode*

[in] Policy error code value. *PolicyErrorCode* must be a nonzero value; the SBM will copy this value, in combination with *PolicyErrorValue*, into the RSVP Error Object when sending **PATHERR** or **RESERR** messages (as the result of policy based–admission control failure, to provide a reason for rejecting the request).

#### *PolicyErrorValue*

[in] Policy error value. *PolicyErrorValue* must be a nonzero value; the SBM will copy this value, in combination with *PolicyErrorCode*, into the RSVP Error Object when sending **PATHERR** or **RESERR** messages (as the result of policy based–admission control failure, to provide a reason for rejecting the request).

#### *Reserved*

[in] This parameter is reserved for future use.

## Remarks

When a request has been rejected, the PCM will call the LPM to instruct it to delete the request's state. The LPM can choose to delete the request's state at any time during the rejection process. If the LPM deletes a request's state shortly after its rejection of the request, the LPM must be prepared to handle subsequent calls (by the PCM, through the **LPM\_DeleteState** function) to delete the (already deleted) state.

The LPM does not need to maintain state for requests to which it returns LPV\_DONT\_CARE. However, the LPM must be prepared to handle **LPM\_DeleteState** requests for this (nonexisting) state.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

## cbpGetRsvpObjects

The **cbpGetRsvpObjects** function is a callback function for LPMs to asynchronously return results for **LPM\_GetRsvpObjects** requests. LPMs call the **cbpGetRsvpObjects** function to asynchronously return policy data objects to the PCM for an **LPM\_GetRsvpObjects** request. An LPM should only use the **cbpGetRsvpObjects** function if it returned LPM\_RESULTS\_DEFER to the PCM's **LPM\_GetRsvpObjects** request.

```
ULONG  
cbpGetRsvpObjects (  
    LPM_HANDLE LpmHandle,  
    RHANDLE RequestHandle,  
    ULONG LpmError,  
    int RsvpObjectsCount,  
    POLICY_DATA *RsvpObjects[]  
);
```

### Parameters

#### *LpmHandle*

[in] Unique handle for the LPM, as supplied in **LPM\_Initialize**. The PCM will ignore any result that is not accompanied by a valid handle.

#### *RequestHandle*

[in] Unique handle that distinguishes this request from all other requests, provided from the corresponding **LPM\_GetRsvpObjects** request.

#### *LpmError*

[in] Error value, used by the PCM to determine whether the policy data objects returned with this function should be used. Any value other than LPM\_OK will result in the PCM ignoring the contents of *\*RsvpObjects*.

Note that if an LPM is returning an error, it should free buffers allocated during the **LPM\_GetRsvpObjects** request processing; these buffers should have been allocated using the **MemoryAllocator** function, supplied within the **LPM\_Initialize** function as its *FreeMemory* parameter.

If no policy data objects are being returned, *LpmError* must be set to LPM\_OK, *RsvpObjectsCount* must be set to zero, and *\*RsvpObjects* must be set to NULL. The LPM can force the SBM to stop sending out the RSVP message by setting the value of *LpmError* to LPV\_DROP\_MSG.

#### *RsvpObjectsCount*

[in] Number of policy data objects being returned. If no policy data objects are being returned, *LpmError* must be set to LPM\_OK, *RsvpObjectsCount* must be set to zero, and *\*RsvpObjects* must be set to NULL.

#### *RsvpObjects*

[in] Array of pointers to policy data object. The buffer containing the policy data objects should be allocated using the *MemoryAllocator* function supplied within the **LPM\_Initialize** function. The Subnet Bandwidth Manager (SBM) will free the policy data objects when they are no longer needed.

If no policy data objects are being returned, *LpmError* must be set to LPM\_OK, *RsvpObjectsCount* must be set to zero, and *\*RsvpObjects* must be set to NULL.

### Remarks

LPMs do not need to send policy data options if only default options are required. Since the content of policy data objects are opaque to the PCM, no host-to-network order conversion of policy element headers and contents will be done by the PCM; the PCM expects LPMs to generate policy elements in the network order such that the receiver of the policy elements can correctly parse them. However, the policy data object header must be in host order to allow the PCM to merge policy elements (if possible or applicable).

From LPMs that support all PE types, the PCM expects complete policy data objects and their required policy data options. Furthermore, the PCM expects the policy data object header to be in host order; it is the responsibility of the LPM to process the host-to-network order conversions of policy options and policy elements.

If any LPM returns LPV\_DROP\_MSG, the SBM will not send out an RSVP refresh message, but will free the policy data objects returned by other LPMs (those that did not return LPV\_DROP\_MSG, if any). By not sending out RSVP refresh messages, a flow's RSVP state both upstream and downstream will begin to age, and eventually get deleted.

---

**Note** The SBM will send out the RSVP refresh message even if some or all LPMs fail to return policy data objects in a timely fashion, even though such an outgoing RSVP message may not contain all policy data objects it should.

---

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

## LPM\_AdmitRsvpMsg

The **LPM\_AdmitRsvpMsg** function is called by the PCM to pass RSVP messages to the LPM for policy based-admission control decisions. Results from calling **LPM\_AdmitRsvpMsg** can be passed back to the PCM either synchronously or asynchronously by setting the return value appropriately. Asynchronous results should be returned by calling the **cbpAdmitRsvpMsg** function.

```
ULONG
APIENTRY
LPM_AdmitRsvpMsg (
    RHANDLE PcmReqHandle,
    RSVP_HOP *pRcvdIntf,
    RSVP_MSG_OBJS *pRsvpMsgObjs,
    int RcvdRsvpMsgLength,
    UCHAR *RcvdRsvpMsg,
    ULONG *pulPcmActionFlags,
    POLICY_DECISION pPolicyDecisions,
    void *Reserved
);
```

### Parameters

#### *PcmReqHandle*

[in] Unique handle that identifies this request from all other requests. LPMs must pass this handle to the PCM when returning results asynchronously for an individual request by calling **cbpAdmitRsvpMsg**. *PcmReqHandles* become invalid once results are returned, requiring each request to get its own unique *PcmReqHandle* from the PCM.

#### *pRcvdIntf*

[in] Pointer to the interface on which the message was received. The received interface IP address is supplied as the RSVP HOP object, and the Logical Interface Handle is set to the SNMP Index. Note that interface index numbers can change with the addition and deletion of interfaces, due to the Plug and Play features of Windows 2000.

#### *pRsvpMsgObjs*

[in] Objects received from RSVP. The SBM unpacks received RSVP messages into individual objects and converts the contents of such RSVP objects into host order, and supplies them in the **RSVP\_MSG\_OBJS** structure, which is defined in *lpmapi.h*. The following objects are supplied.



*RcvdRsvpMsgLength*

Value	Meaning
<i>RsvpMsgType</i>	RSVP message type, as defined by the RSVP protocol.
<i>RsvpSession</i>	Pointer to the RSVP session, as defined by the RSVP protocol. Note that contents are in host order.
<i>RsvpFromHop</i>	Pointer to the hop from which the RSVP message was received. Note that contents are in host order.
<i>RsvpScope</i>	Pointer to the RSVP scope object.
<i>RsvpStyle</i>	Pointer to the RSVP reservation style, as defined by the RSVP protocol. Note that contents are in host order.
<i>FlowDescListCount</i>	Number of flow descriptors.
<i>FlowDescList</i>	Array of flow descriptor pointers.
<i>PolicyDataCount</i>	Number of policy data objects.
<i>PolicyDataObjects</i>	Array of policy data object pointers. Note that only the RSVP object header and the policy options are converted to host order, but policy element headers as well as contents are left in network order; the PCM cannot convert the latter to host order, because the PCM cannot parse policy elements. Note that the Microsoft-provided LPM, Msidlpn.dll, does reorder policy element content into host order.
<i>ErrorSpec</i>	Pointer to the received <b>RSVP_ERROR_SPEC</b> object.

*RcvdRsvpMsgLength*

[in] Length of the received RSVP message, in bytes.

*RcvdRsvpMsg*

[in] RSVP message, in network order.

*pulPcmActionFlags*

[out] Flags used to specify an action requested of the PCM. The LPM can currently set this parameter to **FORCE\_IMMEDIATE\_REFRESH** to request an immediate refresh of the message being admitted. An LPM can set this flag if a change in policy data is detected that it wants to forward immediately. Before sending, the SBM asks the LPM to supply policy information for the outgoing refresh message.

Note that LPMs do not need to set this flag when a new **PATH** message is being accepted; SBMs automatically send the new **PATH** message toward receivers.

*pPolicyDecisions*

[out] Pointer to policy decisions. An LPM must allocate this buffer using the memory allocator supplied in the **LPM\_Initialize** function call; the SBM frees the buffer after acting on *pPolicyDecisions*. The PCM looks at *pPolicyDecisions* only when the function returns **LPM\_RESULT\_READY**. Synchronous policy decisions must be returned for each flow in *FlowDescList*, and the number of entries in the *pPolicyDecisions* array must be equal to *FlowDescListCount*. Each policy decision consists of the following.

Value	Meaning
<i>LpmPriorityValue</i>	Pointer to a buffer to receive the LPM Priority Value from the LPM. Note that the PCM will only look at this parameter if the return value of <b>LPM_AdmitRsvpMsg</b> is set to LPM_RESULT_READY. If the LPM is returning results synchronously, this parameter must be set to a valid priority value. See <i>Local Policy Module</i> for more information.
<i>PolicyErrorCode</i>	Pointer to a policy error code. If the request is being rejected synchronously, LPMs must provide a nonzero value for this parameter; the SBM will copy this value, in combination with <i>PolicyErrorValue</i> , into the RSVP error object when sending PATHERR or RESVERR messages (as the result of policy based-admission control failure, to provide a reason for rejecting the request).
<i>PolicyErrorValue</i>	Pointer to a policy error value. If the request is being rejected synchronously, LPMs must provide a nonzero value for this parameter; the SBM will copy this value, in combination with <i>PolicyErrorCode</i> , into the RSVP error object when sending PATHERR or RESVERR messages (as the result of policy based-admission control failure, to provide a reason for rejecting the request).

Since an LPMs return POLICY\_DECISION is an array, an LPM can accept a subset of flows in *FlowDescList* and reject the rest of them, if appropriate. For example, since FF style RESV messages can contain multiple flows, when an LPM rejects some flows and accepts others, the SBM generates a separate RESVERR message for each rejected flow; before sending the RESVERR message, PCM calls each LPM to supply policy data objects for each outgoing RESVERR message.

#### *Reserved*

[out] Reserved for future use.

## Return Values

### LPM\_RESULT\_READY

The LPM has made a policy decision. The result is available in *LpmPriorityValue*.

### LPM\_RESULT\_DEFER

The LPM was unable to return a decision immediately, but will do so using the callback function **cbpAdmitRsvpMsg**. The LPM must call the callback within the time period specified by the *ResultTimeLimit* parameter of **LPM\_Initialize**, otherwise the PCM will assume LPV\_REJECT.

### Any other value

The PCM assumes LPV\_DONT\_CARE. In addition, the PCM will assume the return value to be synchronous, and thus will not expect the LPM to call the **cbpAdmitRsvpMsg** callback function; even if the LPM calls the function later, the result will be rejected.

## Remarks

The Subnet Bandwidth Manager (SBM) forwards RSVP PATH, RESV, PATHERR, RESVERR, PATH\_TEAR, and RESV\_TEAR messages to the PCM. If a request passes LPM policy-based admission (in which case the success status is passed up through the PCM to the SBM), the SBM performs resource based-admission control as part of its RSVP processing; if resource based-admission control fails, the SBM will instruct the PCM to instruct each LPM to delete its state through the **LPM\_CommitResv** function. In such circumstances, the SBM (and not the LPMs) will create the requisite RSVP error message.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Lpmapi.h.

## LPM\_CommitResv

The **LPM\_CommitResv** function is called by the PCM to obtain reservation commitment decisions from the LPM.

```
VOID
LPM_CommitResv (
    RSVP_SESSION *RsvpSession,
    RSVP_HOP *FlowInstalledIntf,
    RESV_STYLE *RsvpStyle,
    int FilterSpecCount,
    FILTER_SPEC **ppFilterSpecList,
    IS_FLOWSPEC *pMergedFlowSpec,
    ULONG CommitDecision
);
```

### Parameters

#### *RsvpSession*

[in] Pointer to the RSVP session object for which the reservation commitment is being requested.

#### *FlowInstalledIntf*

[in] Pointer to the interface on which the message was received. The received interface IP address is supplied as the RSVP HOP object, and the Logical Interface Handle is set to the SNMP Index. Note that interface index numbers can change with the addition and deletion of interfaces, due to the Plug and Play features of Windows 2000.

#### *RsvpStyle*

[in] RSVP reservation style being requested.

*FilterSpecCount*

[in] Number of filter specs in *ppFilterSpecList*.

*ppFilterSpecList*

[in] Array of filter specs, listing the senders for whom the flow is created.

*pMergedFlowSpec*

[in] Flow spec installed on the specified interface. The flow spec is a merged flow for all receivers that can be reached by *FlowInstalledIntf*.

*CommitDecision*

[in] Value of the commitment decision reached by the LPM. The following list indicates possible values:

RESOURCES\_ALLOCATED  
RESOURCES\_MODIFIED

**Remarks**

When the resources are allocated by the SBM for a new reservation, it calls LPMs with *CommitDecision* set to RESOURCES\_ALLOCATED. When resources allocated for an existing reservation are changed, the SBM calls the **LPM\_CommitResv** function with *CommitDecision* set to RESOURCES\_MODIFIED.

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Lpmapi.h.

## LPM\_Deinitialize

The **LPM\_Deinitialize** function allows the PCM to instruct LPMs to deinitialize, whether due to system shutdown or a change in Designated Subnet Bandwidth Manager (DSBM) status. This occurs when the Admission Control Service no longer needs to do policy based-admission control, such as when a demotion from DSBM status occurs. LPMs should free resources, close connections to external entities such as policy servers, directory services, and perform any other cleanup necessary to properly relinquish LPM activities. The PCM will unload the DLL after **LPM\_Deinitialize** returns.

```
int  
LPM_Deinitialize (  
    LPM_HANDLE LpmHandle  
);
```

**Parameters***LpmHandle*

Unique handle to the LPM, as supplied through **LPM\_Initialize** during initialization.

## Return Values

### LPM\_OK

The LPM deinitialized successfully.

If another value is returned from **LPM\_Deinitialize**, the PCM will record the name of this DLL (implementations of LPMs are always in the form of a DLL), as well as this return value, in the Event Log.

## Remarks

LPMs do not need to return errors for outstanding requests when **LPM\_Deinitialize** is called; PCM assumes LPV\_REJECT for outstanding requests. LPMs should deinitialize synchronously before returning. If an LPM has been loaded and initialized multiple times to facilitate the handling of multiple PE types, the PCM will call **LPM\_Deinitialize** multiple times as well.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Lpmapi.h.

---

## LPM\_DeleteState

The **LPM\_DeleteState** function is called by the PCM to delete LPMs' RSVP state information. RSVP states are deleted on various occasions, including when the SBM receives RSVP TEAR/ERR messages, or when an RSVP state times out. The **LPM\_DeleteState** function call is synchronous. The PCM does not expect any results from the LPM for this request.

```
void
LPM_DeleteState (
    SESSION *RsvpSession,
    HOP *ReceivedIntfAddress,
    MSG_TYPE RsvpMsgType,
    HOP *RsvpHop,
    STYLE *RsvpStyle,
    int FilterSpecCount,
    FILTER_SPEC **FilterSpecList,
    int DeleteReason
);
```

## Parameters

### *RsvpSession*

[in] Pointer to the RSVP session object for which the LPM should delete its state. This value is never NULL.

### *ReceivedIntfAddress*

[in] Pointer to the interface on which the RSVP TEAR message was received. The received interface IP address is supplied as the **RSVP HOP** object, and the Logical Interface Handle is set to the SNMP Index. If the PCM is calling the **LPM\_DeleteState** function for any reason other than an RSVP TEAR message, this argument can be NULL. Note that interface index numbers can change with the addition and deletion of interfaces, due to the Plug and Play features of Windows 2000.

### *RsvpMsgType*

[in] RSVP message type for which the LPM should delete its state.

### *RsvpHop*

[in] Pointer to an **RSVP HOP** object identifying the node that sent the TEAR message. LPMs can use this argument to locate state information.

### *RsvpStyle*

[in] Pointer to an argument that specifies the RSVP reservation style for RSVP RESV\_TEAR messages. LPMs can use this argument to locate state information.

### *FilterSpecCount*

[in] Specifies the number of *FilterSpecs* in *FilterSpecList*. For RESV messages, *FilterSpecCount* is dependent on *RsvpStyle*. For PATH messages, this value will always be 1.

### *FilterSpecList*

[in] Array of *FilterSpec* pointers. Note that the contents of *FilterSpecList* is dependent on *RsvpStyle*; if *RsvpMsgType* is RSVP\_PATH then *FilterSpecList* specifies the *SenderTemplate*, if *RsvpMsgType* is RSVP\_RESV then *FilterSpecList* is the list of filters for which the RESV state needs to be deleted.

### *DeleteReason*

[in] Reason for deleting the state. Possible values are:

RCVD\_PATH\_TEAR  
RCVD\_RESV\_TEAR  
ADM\_CTRL\_FAILED  
STATE\_TIMEOUT  
FLOW\_DURATION

LPMs can use *DeleteReason* for statistic gathering or any other use.

## Remarks

The PCM will call the **LPM\_DeleteState** function for each LPM; LPMs should be prepared to handle **LPM\_DeleteState** for a nonexistent state, as described further in the Remarks section of the **cbpAdmitRsvpMsg** function.

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Lpmapi.h.

---

## LPM\_GetRsvpObjects

The **LPM\_GetRsvpObjects** function allows the PCM to query LPMs for policy data. The data is forwarded by the PCM to the SBM for inclusion in RSVP refresh messages that require policy data.

Results from the **LPM\_GetRsvpObjects** function can be returned synchronously or asynchronously. Asynchronous results are returned by calling the **cbpGetRsvpObjects** callback function.

```
ULONG
APIENTRY
LPM_GetRsvpObjects (
    RHANDLE PcmReqHandle,
    ULONG MaxPdSize,
    RSVP_HOP *SendingIntfAddr,
    RSVP_MSG_OBJS *pRsvpMsgObjs,
    int *pRsvpObjectsCount,
    RsvpObjHdr ***pppRsvpObjects,
    void *Reserved
);
```

### Parameters

#### *PcmReqHandle*

[in] Unique handle that distinguishes this request from all other requests. LPMs should use this *PcmReqHandle* when returning results asynchronously using the **cbpGetRsvpObjects** callback function.

#### *MaxPdSize*

[in] Maximum allowable size of the returned policy data.

#### *SendingIntfAddr*

[in] Pointer to the interface on which the RSVP message will be sent out. The sending interface IP address is supplied as the **RSVP HOP** object, which equates to PHOP for PATH messages and NHOP for RESV messages. The Logical Interface Handle is set to the SNMP Index. Note that interface index numbers can change with the addition and deletion of interfaces, due to the Plug and Play features of Windows 2000.

#### *pRsvpMsgObjs*

[in] RSVP objects generated by the SBM. All RSVP objects are in host order. The following objects are supplied.

Value	Meaning
<i>RsvpMsgType</i>	RSVP message type, as defined in the RSVP protocol. This can be used by an LPM to locate the state from which it can generate policy data objects.
<i>RsvpSession</i>	RSVP session for which the SBM requires policy information. This can be used by an LPM to locate the state from which it can generate policy data objects.
<i>RsvpHop</i>	The <b>HOP</b> to which the RSVP message is being forwarded. Since a PATH message is sent directory to the session address, this HOP pointer is NULL for PATH messages. For all other messages, the address in the HOP object is the node address and the LIH is unused.
<i>RsvpStyle</i>	RSVP reservation style, as defined in the RSVP protocol. If an RESV message is being sent out by the SBM, <i>RsvpStyle</i> specifies the reservation style. If a PATH message is being sent, <i>RsvpStyle</i> is NULL.
<i>RsvpScope</i>	The RSVP scope of an outgoing RESV message, as long as the SCOPE object is not NULL. Used only for WF-style reservations. For all other RSVP reservation styles, <i>RsvpScope</i> is NULL.
<i>FlowDescCount</i>	Number of flow descriptors.
<i>FlowDescList</i>	Array of flow descriptor pointers in the outgoing RSVP message. For PATH messages, there will be only one FlowDescriptor containing sender template and sender TSPEC.

#### *pRsvpObjectsCount*

[out] Pointer to the number of policy objects being returned. When an LPM is immediately returning results, the *pRsvpObjectsCount* and *pppRsvpObjects* parameters should be used to return policy data objects. Note that the buffer containing the policy data objects should be allocated using the memory allocation function **PALLOCMEM**, supplied within the **LPM\_Initialize** function.

#### *pppRsvpObjects*

[out] Pointer to an array of policy data object pointers returned in response to the request. Note that the buffer containing the policy data objects, and this array of policy data object pointers, should be allocated using the memory allocation function **PALLOCMEM**, supplied within the **LPM\_Initialize** function.

#### *Reserved*

[out] Reserved for future use.

## Return Values

### LPM\_RESULT\_READY

The LPM has returned the policy data using *pRsvpObjectsCount* and *pppRsvpObjects* parameters.



**LPM\_RESULT\_DEFER**

The LPM is unable to return the policy data objects synchronously, and will return the policy data objects with a subsequent call to **cbpGetRsvpObjects**.

**LPM\_DROP\_MSG**

The LPM can return this value when it does not want to refresh the outgoing message.

**LPM\_ERROR**

The LPM has encountered an error.

**Remarks**

If an LPM doesn't have policy data to return from the **LPM\_GetRsvpObjects** function call, it should synchronously return **LPM\_RESULT\_READY**, set *pppRsvpObjects* to **NULL**, and set *pRsvpObjectsCount* to zero. If a synchronous return isn't possible, an LPM should return **LPM\_RESULT\_DEFER**, and return the result by calling the **cbpGetRsvpObjects** callback function. If the LPM does not have any policy data objects to return, it can set *pppRsvpObjects* to **NULL** and *pRsvpObjectsCount* to zero.

If any LPM returns **LPM\_DROP\_MSG**, the SBM will not send out an RSVP refresh message, and will free the policy data objects returned by other LPMs (those that did not return **LPM\_DROP\_MSG**, if any). By not sending out RSVP refresh messages, a flow's RSVP state both upstream and downstream will begin to age, and eventually get deleted.

---

**Note** The SBM will send out the RSVP refresh message even if some or all LPMs fail to return policy data objects in a timely fashion, even though such an outgoing RSVP message may not contain all policy data objects it should.

---

**! Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Lpmapi.h.

---

## LPM\_Initialize

The **LPM\_Initialize** function initializes a local policy module (LPM). This occurs when the Admission Control Service needs to do policy based-admission control, such as when an SBM becomes the Designated Subnet Bandwidth Manager (DSBM). LPMs should initialize themselves, synchronously, before returning.

```
int
LPM_Initialize (
    LPM_HANDLE LpmHandle,
    int ResultTimeLimit,
    void MemoryAllocator,
    void FreeMemory,
    void *(cbpAdmitRsvpMsg)(...),
    void *(cbpGetRsvpObjects)(...),
    int ConfiguredLpmCount,
    int *SupportedPeType,
    int Reserved
);
```

## Parameters

### *LpmHandle*

[in] Unique handle for the LPM, assigned by the PCM.

### *ResultTimeLimit*

[in] Represents the value, in seconds, within which the LPM must respond to PCM requests. An LPM's failure to respond to PCM requests within *ResultTimeLimit* results in assumed rejection of the PCM request. The PCM will ignore responses provided after *ResultTimeLimit*.

### *MemoryAllocator*

[in] PCM provided—memory allocation routine **PALLOCMEM**. This function call must be used by LPMs when returning policy information to the PCM. This function, combined with the **PFREEMEM** function, allows the SBM to use different memory management schemes without requiring the recompilation of LPMs. Its use is recommended. LPMs do not need to use this function to manage their local buffers.

### *FreeMemory*

[in] PCM provided—memory freeing routine **PFREEMEM**. This function, combined with the function **PALLOCMEM**, allows the SBM to use different memory-management schemes without requiring the recompilation of LPMs. Its use is recommended. If memory allocated with *MemoryAllocator* is not returned to PCM, nor freed with this call, a memory leak will result. LPMs do not need to use this function to manage their local buffers. The SBM will free this buffer when it is no longer needed. This function call must be used by LPMs when returning policy information to the PCM.

### *cbpAdmitRsvpMsg*

[in] Pointer to the function used to asynchronously return results to the **LPM\_AdmitRsvpMsg** request.

### *cbpGetRsvpObjects*

[in] Pointer to the function used to asynchronously return policy data objects from calls to the **LPM\_GetRsvpObjects** request.

### *ConfiguredLpmCount*

[in] Number of LPMs configured for use with the PCM. Note that this value does not indicate whether LPMs have been successfully loaded or initialized. This value is a useful indication that the PCM will attempt to load multiple LPMs on the system.

### *SupportedPeType*

[out] Valid Policy Element (PE) type that the LPM uses to make policy based-admission control decisions. Each LPM can only support one PE type, though future versions may allow an LPM to support multiple PE types. Reserved PE types are defined in `Lpmapi.h`.

It is possible for a single DLL to support multiple PE types by having the DLL name entered multiple times in the PCM configuration data. Under such circumstances, the PCM will load and call the same **LPM\_Initialize** routine multiple times; it is the LPM's responsibility to return different PE types for these additional calls.

LPMs can return a special PE type, **LPM\_ALL\_PE\_TYPES**, to indicate that it will make policy based-admission control decisions based on all policy data objects. In this scenario, the PCM will assume that this LPM understands how to generate policy data objects for outgoing messages that the PCM is not able to understand.

### *Reserved*

[in, out] Reserved for future use.

## Return Values

If the LPM is initialized successfully, and a valid PE type is returned in *SupportedPeType*, the return value will be `LPM_OK`. The PCM treats any value other than `LPM_OK` as an error, and unloads the DLL (LPMs are always implemented as DLLs). If a value other than `LPM_OK` is returned or *SupportedPeType* is invalid, the PCM writes a record to the Event Log and includes the name of the DLL and the returned error value.

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in `Lpmapi.h`.

---

## Lpm\_IpAddressTable

The **Lpm\_IpAddressTable** function is used by the PCM to pass a list of IP addresses assigned to the Windows 2000 Server upon which the LPM is initialized. The PCM calls this routine after the LPM has successfully initialized, but before making any requests. The PCM also uses the **Lpm\_IpAddressTable** function to update LPMs regarding IP address changes. LPMs are expected to detect IP address changes and update their states appropriately.

```
void  
Lpm_IpAddressTable (  
    ULONG cIpAddrTable,  
    LPMIPTABLE *pIpAddrTable  
);
```

### Parameters

#### *cIpAddrTable*

[in] Number of addresses in the IP table.

#### *pIpAddrTable*

[in] Pointer to an **LPMIPTABLE** structure that contains the IP addresses assigned to the Windows 2000 Server on which the LPM resides.

### ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Lpmapi.h.

## LPM Structures

The following structures are exposed by the Microsoft-provided LPM:

- **LPMIPTABLE**
- **PALLOCMEM**
- **PFREEMEM**

## LPMIPTABLE

The **LPMIPTABLE** structure contains IP information, including the SNMP index, IP address, and subnet mask for each interface. The **LPMIPTABLE** structure is supplied as an argument for the **LPM\_IpAddressTable** function.

```
typedef struct lpmiptable {  
    ULONG        UlIfIndex; // SNMP index for this interface  
    IN_ADDR      IfIpAddr;  // Interface IP address  
    IN_ADDR      IfNetMask; // Interface subnet mask  
} LPMIPTABLE;
```

### Members

#### **UlIfIndex**

SNMP index for the interface.

#### **IfIpAddr**

IP address for the interface.

**IfNetMask**

IP subnet mask for the interface.

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Lpmapi.h.

---

## PALLOCMEM

The **PALLOCMEM** function is a memory allocation function provided by the PCM, used for allocating memory when returning policy information to the PCM. The **PALLOCMEM** function is supplied as a parameter of the **LPM\_Initialize** function, and allows the SBM to experiment with different memory-management schemes without requiring recompilation of LPMs.

```
void  
PALLOCMEM (  
    IN DWORD Size  
);
```

**Parameters***Size*

Size of the memory buffer required by the LPM.

**Remarks**

LPMs do not need to use this function to manage their local buffers.

**!** Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Lpmapi.h.

---

## PFREEMEM

The **PFREEMEM** function is a memory-freeing function provided by the PCM. **PFREEMEM** frees memory buffers that were allocated using **PALLOCMEM**. The **PFREEMEM** function is supplied as a parameter of the **LPM\_Initialize** function. The combination of **PALLOCMEM** and **PFREEMEM** allows the SBM to experiment with different memory-management schemes without requiring recompilation of LPMs.

```
void  
PFREEMEM (  
    IN void *pv  
);
```

## Parameters

*pv*

Pointer to the memory buffer to free.

## Remarks

LPMs do not need to use this function to manage their local buffers. LPMs need to use this function to free buffers that were allocated, but were not sent to the PCM. For example, if a buffer is allocated in anticipation of a PCM's response to a request, but a response is never returned (perhaps the remote policy store is unavailable or unresponsive), that buffer *must* be freed with this function, or a memory leak will ensue.

## ! Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Unsupported.

**Header:** Declared in Lpmapi.h.



## I N D E X

# Networking Services Programming Elements – Alphabetical Listing

This final part, found in each volume in the Networking Services Library, provides a comprehensive programming element index that has been designed to make your life easier.

Rather than cluttering the TOCs of each individual volume in this library with the names of programming elements, I've relegated such per-element information to a central location: the back of each volume. This index points you to the volume that has the information you need, and organizes the information in a way that lends itself to easy use.

Also, to keep you as informed and up-to-date as possible about Microsoft technologies, I've created (and maintain) a live Web-based document that maps Microsoft technologies to the locations where you can get more information about them. The following link gets you to the live index of technologies:

**[www.iseminger.com/winprs/technologies](http://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](mailto:winprs@microsoft.com). While I can't guarantee a reply, I'll read the input, and if others can benefit, I will incorporate the idea into future libraries.

Locators are arranged by Volume Number followed by Page Number.

## A

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
CIAddFlowComplete .....	Vol. 1, 830
CIDeleteFlowComplete .....	Vol. 1, 831
CIModifyFlowComplete .....	Vol. 1, 831
CINotifyHandler .....	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
GetFriendlyIIndex.....	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
getprotobyname.....	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
- IEAPPProviderConfig ..... Vol. 4, 426  
 IEAPPProviderConfig::  
   RouterInvokeConfigUI ..... Vol. 4, 430  
 IEAPPProviderConfig::  
   RouterInvokeCredentialsUI ..... Vol. 4, 432  
 IEAPPProviderConfig::  
   ServerInvokeConfigUI ..... Vol. 4, 429  
 IEAPPProviderConfig::Initialize ..... Vol. 4, 426  
 IEAPPProviderConfig::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
ISyncMgrSynchronizelInvoke .....	Vol. 2, 191
ISyncMgrSynchronizelInvoke::	
UpdateAll .....	Vol. 2, 192
ISyncMgrSynchronizelInvoke::	
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

MACYIELD CALLBACK .....	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
MgmDeRegisterMPProtocol .....	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
MgmRegisterMPProtocol .....	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  
MprAdminConnectionHangupNotification ..... Vol. 4, 344  
MprAdminConnectionHangupNotification2 ..... Vol. 4, 345  
MprAdminDeregisterConnectionNotification ..... 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  
MprAdminInterfaceGetCredentialsEx ..... Vol. 5, 82  
MprAdminInterfaceGetHandle ..... Vol. 5, 83  
MprAdminInterfaceGetInfo ..... Vol. 5, 84  
MprAdminInterfaceQueryUpdateResult ..... Vol. 5, 86  
MprAdminInterfaceSetCredentials ..... Vol. 5, 87  
MprAdminInterfaceSetCredentialsEx ..... Vol. 5, 89  
MprAdminInterfaceSetInfo ..... Vol. 5, 90  
MprAdminInterfaceTransportGetInfo ..... Vol. 5, 93  
MprAdminInterfaceTransportRemove ..... Vol. 5, 94  
MprAdminInterfaceTransportSetInfo ..... Vol. 5, 95  
MprAdminInterfaceTransportAdd ..... Vol. 5, 91  
MprAdminInterfaceUpdatePhonebookInfo ..... 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  
MprAdminRegisterConnectionNotification ..... 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
NdrGetUserMarshallInfo .....	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
PfADDRESSSTYPE .....	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 Interface .....	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  
 RASAUTODIALENTY..... Vol. 4, 207  
 RasClearConnectionStatistics..... Vol. 4, 107  
 RasClearLinkStatistics..... Vol. 4, 107  
 RASCONN..... Vol. 4, 208

RasConnectionNotification .....	Vol. 4, 109	RasGetEntryProperties .....	Vol. 4, 160
RASCONNSTATE .....	Vol. 4, 258	RasGetErrorString .....	Vol. 4, 162
RASCONNSTATUS .....	Vol. 4, 210	RasGetLinkStatistics .....	Vol. 4, 164
RasCreatePhonebookEntry .....	Vol. 4, 110	RasGetProjectionInfo .....	Vol. 4, 165
RASCREREDENTIALS .....	Vol. 4, 211	RasGetSubEntryHandle .....	Vol. 4, 167
RASCTRYINFO .....	Vol. 4, 212	RasGetSubEntryProperties .....	Vol. 4, 168
RasCustomDeleteEntryNotify .....	Vol. 4, 111	RasHangUp .....	Vol. 4, 170
RasCustomDial .....	Vol. 4, 112	RasInvokeEapUI .....	Vol. 4, 171
RasCustomDialDlg .....	Vol. 4, 114	RASIPADDR .....	Vol. 4, 239
RasCustomEntryDlg .....	Vol. 4, 116	RasMonitorDlg .....	Vol. 4, 173
RasCustomHangUp .....	Vol. 4, 118	RASMONITORDLG .....	Vol. 4, 240
RasCustomScriptExecute .....	Vol. 4, 197	RASNOUSER .....	Vol. 4, 241
RasDeleteEntry .....	Vol. 4, 119	RASPBDLG .....	Vol. 4, 243
RASDEVINFO .....	Vol. 4, 214	RasPBDlgFunc .....	Vol. 4, 174
RasDial .....	Vol. 4, 120	RasPhonebookDlg .....	Vol. 4, 176
RasDialDlg .....	Vol. 4, 123	RASPPCCP .....	Vol. 4, 245
RASDIALDLG .....	Vol. 4, 215	RASPPPIP .....	Vol. 4, 247
RASDIALEXTENSIONS .....	Vol. 4, 217	RASPPPIPX .....	Vol. 4, 251
RasDialFunc .....	Vol. 4, 125	RASPPPLCP .....	Vol. 4, 248
RasDialFunc1 .....	Vol. 4, 127	RASPPPNBF .....	Vol. 4, 252
RasDialFunc2 .....	Vol. 4, 129	RASPROJECTION .....	Vol. 4, 263
RASDIALPARAMS .....	Vol. 4, 219	RasReceiveBuffer .....	Vol. 4, 201
RasEapBegin .....	Vol. 4, 389	RasRenameEntry .....	Vol. 4, 178
RasEapEnd .....	Vol. 4, 391	RasRetrieveDialogBuffer .....	Vol. 4, 203
RasEapFreeMemory .....	Vol. 4, 391	RasSecurityDialogBegin .....	Vol. 4, 284
RasEapGetIdentity .....	Vol. 4, 392	RasSecurityDialogComplete .....	Vol. 4, 286
RasEapGetInfo .....	Vol. 4, 395	RasSecurityDialogEnd .....	Vol. 4, 287
RASEAPINFO .....	Vol. 4, 222	RasSecurityDialogGetInfo .....	Vol. 4, 288
RasEapInitialize .....	Vol. 4, 396	RasSecurityDialogReceive .....	Vol. 4, 289
RasEapInvokeConfigUI .....	Vol. 4, 397	RasSecurityDialogSend .....	Vol. 4, 291
RasEapInvokeInteractiveUI .....	Vol. 4, 399	RasSendBuffer .....	Vol. 4, 200
RasEapMakeMessage .....	Vol. 4, 401	RasSetAutodialAddress .....	Vol. 4, 179
RASEAPUSERIDENTITY .....	Vol. 4, 222	RasSetAutodialEnable .....	Vol. 4, 181
RasEditPhonebookEntry .....	Vol. 4, 131	RasSetAutodialParam .....	Vol. 4, 182
RASENTRY .....	Vol. 4, 223	RasSetCredentials .....	Vol. 4, 184
RasEntryDlg .....	Vol. 4, 133	RasSetCustomAuthData .....	Vol. 4, 186
RASENTRYDLG .....	Vol. 4, 236	RasSetEapUserData .....	Vol. 4, 187
RASENTRYNAME .....	Vol. 4, 238	RasSetEntryDialParams .....	Vol. 4, 189
RasEnumAutodialAddresses .....	Vol. 4, 135	RasSetEntryProperties .....	Vol. 4, 191
RasEnumConnections .....	Vol. 4, 136	RasSetSubEntryProperties .....	Vol. 4, 193
RasEnumDevices .....	Vol. 4, 137	RASSLIP .....	Vol. 4, 253
RasEnumEntries .....	Vol. 4, 139	RASSUBENTRY .....	Vol. 4, 254
RasFreeBuffer .....	Vol. 4, 199	RasValidateEntryName .....	Vol. 4, 195
RasFreeEapUserIdentity .....	Vol. 4, 142	recv .....	Vol. 1, 196
RasGetAutodialAddress .....	Vol. 4, 143	recvfrom .....	Vol. 1, 199
RasGetAutodialEnable .....	Vol. 4, 144	RegisterProtocol .....	Vol. 5, 290
RasGetAutodialParam .....	Vol. 4, 145	REMOTE_NAME_INFO .....	Vol. 3, 665
RasGetBuffer .....	Vol. 4, 198	Route Flags .....	Vol. 5, 501
RasGetConnectionStatistics .....	Vol. 4, 147	ROUTER_CONNECTION_STATE .....	Vol. 5, 167
RasGetConnectStatus .....	Vol. 4, 148	ROUTER_INTERFACE_TYPE .....	Vol. 5, 168
RasGetCountryInfo .....	Vol. 4, 149	Routing Table Query Flags .....	Vol. 5, 504
RasGetCredentials .....	Vol. 4, 151	ROUTING_PROTOCOL_CONFIG .....	Vol. 5, 562
RasGetCustomAuthData .....	Vol. 4, 153	RPC_ASYNC_EVENT .....	Vol. 3, 315
RasGetEapUserData .....	Vol. 4, 155	RPC_ASYNC_STATE .....	Vol. 3, 298
RasGetEapUserIdentity .....	Vol. 4, 156	RPC_AUTH_IDENTITY_HANDLE .....	Vol. 3, 318
RasGetEntryDialParams .....	Vol. 4, 158		

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	RpcStringBindingParse.....	Vol. 3, 559
RpcNsMgmtSetExpAge.....	Vol. 3, 480	RpcStringFree .....	Vol. 3, 561
RpcNsProfileDelete .....	Vol. 3, 481	RpcTestCancel .....	Vol. 3, 562
RpcNsProfileEltAdd .....	Vol. 3, 482	RpcTryExcept .....	Vol. 3, 590
RpcNsProfileEltInqBegin .....	Vol. 3, 484	RpcTryFinally .....	Vol. 3, 590
RpcNsProfileEltInqDone.....	Vol. 3, 488	RpcWinSetYieldInfo.....	Vol. 3, 563
RpcNsProfileEltInqNext.....	Vol. 3, 488	RpcWinSetYieldTimeout.....	Vol. 3, 566
RpcNsProfileEltRemove.....	Vol. 3, 490	RSVP_ADSPEC .....	Vol. 1, 802
RpcObjectInqType.....	Vol. 3, 492	RSVP_RESERVE_INFO .....	Vol. 1, 803
RpcObjectSetInqFn .....	Vol. 3, 493	RSVP_STATUS_INFO .....	Vol. 1, 805
RpcObjectSetType .....	Vol. 3, 494	RTM_DEST_INFO.....	Vol. 5, 480
RpcProtseqVectorFree .....	Vol. 3, 496	RTM_ENTITY_EXPORT_	
RpcRaiseException .....	Vol. 3, 497	METHOD .....	Vol. 5, 477
RpcRevertToSelf.....	Vol. 3, 501	RTM_ENTITY_EXPORT_	
RpcRevertToSelfEx.....	Vol. 3, 502	METHODS .....	Vol. 5, 481
RpcServerInqBindings.....	Vol. 3, 503	RTM_ENTITY_ID.....	Vol. 5, 482
RpcServerInqDefaultPrincName .....	Vol. 3, 504	RTM_ENTITY_INFO.....	Vol. 5, 483
RpcServerInqIf .....	Vol. 3, 505	RTM_ENTITY_METHOD_	
RpcServerListen .....	Vol. 3, 506	OUTPUT .....	Vol. 5, 484
RpcServerRegisterAuthInfo .....	Vol. 3, 508	RTM_ENTITY_METHOD_INPUT.....	Vol. 5, 483
RpcServerRegisterIf .....	Vol. 3, 511	RTM_EVENT_CALLBACK .....	Vol. 5, 478
RpcServerRegisterIf2 .....	Vol. 3, 512	RTM_EVENT_TYPE.....	Vol. 5, 506
RpcServerRegisterIfEx .....	Vol. 3, 514	RTM_IP_ROUTE .....	Vol. 5, 357
RpcServerTestCancel .....	Vol. 3, 516	RTM_IPV4_GET_ADDR_AND_	
RpcServerUnregisterIf.....	Vol. 3, 517	LEN .....	Vol. 5, 492
RpcServerUseAllProtseqs.....	Vol. 3, 519	RTM_IPV4_GET_ADDR_AND_	
RpcServerUseAllProtseqsEx .....	Vol. 3, 521	MASK.....	Vol. 5, 493
RpcServerUseAllProtseqsIf .....	Vol. 3, 523	RTM_IPV4_LEN_FROM_MASK .....	Vol. 5, 494
RpcServerUseAllProtseqsIfEx .....	Vol. 3, 524	RTM_IPV4_MAKE_NET_	
RpcServerUseProtseq.....	Vol. 3, 526	ADDRESS.....	Vol. 5, 495
RpcServerUseProtseqEp.....	Vol. 3, 530	RTM_IPV4_MASK_FROM_LEN .....	Vol. 5, 496
RpcServerUseProtseqEpEx.....	Vol. 3, 532	RTM_IPV4_SET_ADDR_AND_	
RpcServerUseProtseqEx .....	Vol. 3, 528	LEN .....	Vol. 5, 497
RpcServerUseProtseqIf.....	Vol. 3, 534	RTM_IPV4_SET_ADDR_AND_	
RpcServerUseProtseqIfEx .....	Vol. 3, 536	MASK.....	Vol. 5, 498
RpcSmAllocate .....	Vol. 3, 538	RTM_IPX_ROUTE.....	Vol. 5, 358
RpcSmClientFree .....	Vol. 3, 539	RTM_NET_ADDRESS .....	Vol. 5, 485
RpcSmDestroyClientContext .....	Vol. 3, 540	RTM_NEXTHOP_INFO .....	Vol. 5, 486
RpcSmDisableAllocate .....	Vol. 3, 541	RTM_NEXTHOP_LIST .....	Vol. 5, 487
RpcSmEnableAllocate.....	Vol. 3, 542	RTM_PREF_INFO .....	Vol. 5, 488
RpcSmFree .....	Vol. 3, 543	RTM_REGN_PROFILE .....	Vol. 5, 488
RpcSmGetThreadHandle .....	Vol. 3, 544	RTM_ROUTE_INFO.....	Vol. 5, 489
RpcSmSetClientAllocFree .....	Vol. 3, 545	RTM_SIZE_OF_DEST_INFO.....	Vol. 5, 499
RpcSmSetThreadHandle .....	Vol. 3, 546	RTM_SIZE_OF_ROUTE_INFO.....	Vol. 5, 500
RpcSmSwapClientAllocFree .....	Vol. 3, 547	RtmAddNextHop.....	Vol. 5, 405
RpcSsAllocate .....	Vol. 3, 548	RtmAddRoute .....	Vol. 5, 335
RpcSsDestroyClientContext.....	Vol. 3, 549	RtmAddRouteToDest.....	Vol. 5, 406
RpcSsDisableAllocate .....	Vol. 3, 550	RtmBlockDeleteRoutes.....	Vol. 5, 347
RpcSsDontSerializeContext.....	Vol. 3, 550	RtmBlockMethods.....	Vol. 5, 409
RpcSsEnableAllocate.....	Vol. 3, 551	RtmCloseEnumerationHandle .....	Vol. 5, 346
RpcSsFree .....	Vol. 3, 552	RtmCreateDestEnum.....	Vol. 5, 410
RpcSsGetThreadHandle .....	Vol. 3, 553	RtmCreateEnumerationHandle .....	Vol. 5, 343
RpcSsSetClientAllocFree .....	Vol. 3, 554	RtmCreateNextHopEnum .....	Vol. 5, 413
RpcSsSetThreadHandle .....	Vol. 3, 555	RtmCreateRouteEnum .....	Vol. 5, 414
RpcSsSwapClientAllocFree .....	Vol. 3, 556	RtmCreateRouteList .....	Vol. 5, 417
RpcStringBindingCompose .....	Vol. 3, 558	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	SnmpUtilIdsToA.....	Vol. 2, 319
SnmpDuplicateVbl.....	Vol. 2, 395	SnmpUtilMemAlloc.....	Vol. 2, 321
SnmpEncodeMsg.....	Vol. 2, 396	SnmpUtilMemFree.....	Vol. 2, 321
SnmpEntityToStr.....	Vol. 2, 398	SnmpUtilMemReAlloc.....	Vol. 2, 322
SnmpExtensionClose.....	Vol. 2, 290	SnmpUtilOctetsCmp.....	Vol. 2, 323
SnmpExtensionInit.....	Vol. 2, 291	SnmpUtilOctetsCpy.....	Vol. 2, 324
SnmpExtensionInitEx.....	Vol. 2, 293	SnmpUtilOctetsFree.....	Vol. 2, 325
SnmpExtensionMonitor.....	Vol. 2, 294	SnmpUtilOctetsNCmp.....	Vol. 2, 325
SnmpExtensionQuery.....	Vol. 2, 295	SnmpUtilOidAppend.....	Vol. 2, 326
SnmpExtensionQueryEx.....	Vol. 2, 298	SnmpUtilOidCmp.....	Vol. 2, 327
SnmpExtensionTrap.....	Vol. 2, 302	SnmpUtilOidCpy.....	Vol. 2, 328
SnmpFreeContext.....	Vol. 2, 399	SnmpUtilOidFree.....	Vol. 2, 329
SnmpFreeDescriptor.....	Vol. 2, 401	SnmpUtilOidNCmp.....	Vol. 2, 330
SnmpFreeEntity.....	Vol. 2, 402	SnmpUtilOidToA.....	Vol. 2, 331
SnmpFreePdu.....	Vol. 2, 403	SnmpUtilPrintAsnAny.....	Vol. 2, 331
SnmpFreeVbl.....	Vol. 2, 404	SnmpUtilPrintOid.....	Vol. 2, 332
SnmpGetLastError.....	Vol. 2, 406	SnmpUtilVarBindCpy.....	Vol. 2, 333
SnmpGetPduData.....	Vol. 2, 407	SnmpUtilVarBindFree.....	Vol. 2, 335
SnmpGetRetransmitMode.....	Vol. 2, 411	SnmpUtilVarBindListCpy.....	Vol. 2, 334
SnmpGetRetry.....	Vol. 2, 412	SnmpUtilVarBindListFree.....	Vol. 2, 335
SnmpGetTimeout.....	Vol. 2, 414	SnmpVarBind.....	Vol. 2, 340
SnmpGetTranslateMode.....	Vol. 2, 416	SnmpVarBindList.....	Vol. 2, 341
SnmpGetVb.....	Vol. 2, 417	sockaddr.....	Vol. 1, 396
SnmpGetVendorInfo.....	Vol. 2, 420	SOCKADDR_IRDA.....	Vol. 1, 397
SnmpListen.....	Vol. 2, 421	socket.....	Vol. 1, 225
SnmpMgrClose.....	Vol. 2, 304	SOCKET_ADDRESS.....	Vol. 1, 397
SnmpMgrGetTrap.....	Vol. 2, 305	SOURCE_GROUP_ENTRY.....	Vol. 5, 563
SnmpMgrOidToStr.....	Vol. 2, 307	StartComplete.....	Vol. 5, 293
SnmpMgrOpen.....	Vol. 2, 308	StartProtocol.....	Vol. 5, 294
SnmpMgrRequest.....	Vol. 2, 309	StopProtocol.....	Vol. 5, 295
SnmpMgrStrToOid.....	Vol. 2, 311	String Binding.....	Vol. 3, 324
SnmpMgrTrapListen.....	Vol. 2, 312	String UUID.....	Vol. 3, 329
SnmpOidCompare.....	Vol. 2, 423	SUPPORT_FUNCTIONS.....	Vol. 5, 305
SnmpOidCopy.....	Vol. 2, 425	SYNCMGRFLAG.....	Vol. 2, 196
SnmpOidToStr.....	Vol. 2, 427	SYNCMGRHANDLERFLAGS.....	Vol. 2, 197
SnmpOpen.....	Vol. 2, 428	SYNCMGRHANDLERINFO.....	Vol. 2, 201
SnmpRecvMsg.....	Vol. 2, 430	SYNCMGRINVOKEFLAGS.....	Vol. 2, 200
SnmpRegister.....	Vol. 2, 433	SYNCMGRITEM.....	Vol. 2, 203
SnmpSendMsg.....	Vol. 2, 436	SYNCMGRITEMFLAGS.....	Vol. 2, 199
SnmpSetPduData.....	Vol. 2, 438	SYNCMGRLOGERRORINFO.....	Vol. 2, 202
SnmpSetPort.....	Vol. 2, 440	SYNCMGRLOGLEVEL.....	Vol. 2, 199
SnmpSetRetransmitMode.....	Vol. 2, 442	SYNCMGRPROGRESSITEM.....	Vol. 2, 201
SnmpSetRetry.....	Vol. 2, 444	SYNCMGRSTATUS.....	Vol. 2, 198
SnmpSetTimeout.....	Vol. 2, 445		
SnmpSetTranslateMode.....	Vol. 2, 446	<b>T</b>	
SnmpSetVb.....	Vol. 2, 448	TC_GEN_FILTER.....	Vol. 1, 845
SnmpStartup.....	Vol. 2, 450	TC_GEN_FLOW.....	Vol. 1, 846
SnmpStrToContext.....	Vol. 2, 453	TC_IFC_DESCRIPTOR.....	Vol. 1, 847
SnmpStrToEntity.....	Vol. 2, 455	TcAddFilter.....	Vol. 1, 807
SnmpStrToOid.....	Vol. 2, 456	TcAddFlow.....	Vol. 1, 809
SnmpSvcGetUptime.....	Vol. 2, 314	TcCloseInterface.....	Vol. 1, 811
SnmpSvcSetLogLevel.....	Vol. 2, 315	TcDeleteFilter.....	Vol. 1, 812
SnmpSvcSetLogType.....	Vol. 2, 316	TcDeleteFlow.....	Vol. 1, 813
SnmpUtilAsnAnyCpy.....	Vol. 2, 317	TcDeregisterClient.....	Vol. 1, 814
SnmpUtilAsnAnyFree.....	Vol. 2, 317	TcEnumerateFlows.....	Vol. 1, 815
SnmpUtilDbgPrint.....	Vol. 2, 318		

- 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
<b>Y</b>	
YieldFunctionName .....	Vol. 3, 580

# Windows® Sockets and QOS



*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.*

## **Windows Socket and QOS**

This volume provides vital programmatic information about Windows Sockets 2 (Winsock) and Quality of Service (QOS), two networking standards that Windows 2000 supports. Winsock provides easy access to multiple transport protocols, enabling programmers to create advanced Web and network-aware applications that transmit data regardless of the protocol being used. The industry-wide QOS initiative enables developers to create or retrofit mission-critical applications that can operate as if network traffic conditions were ideal, even when the network is clogged.



**Included on DVD-ROM:**

An MSDN™ Quarterly Snapshot