Contents

AllocateAndGetTcpExTableFromStack

AllocateAndGetUdpExTableFromStack

CancelIPChangeNotify

CancelMibChangeNotify2

CancelSecurityHealthChangeNotify

ConvertInterfaceAliasToLuid

ConvertInterfaceGuidToLuid

ConvertInterfaceIndexToLuid

ConvertInterfaceLuidToAlias

ConvertInterfaceLuidToGuid

ConvertInterfaceLuidToIndex

ConvertInterfaceLuidToNameA

ConvertInterfaceLuidToNameW

ConvertInterfaceNameToLuidA

ConvertInterfaceNameToLuidW

ConvertIpv4MaskToLength

ConvertLengthToIpv4Mask

CreateAnycastIpAddressEntry

CreateIpForwardEntry

CreateIpForwardEntry2

CreateIpNetEntry

CreateIpNetEntry2

CreatePersistentTcpPortReservation

CreatePersistentUdpPortReservation

CreateProxyArpEntry

CreateSortedAddressPairs

CreateUnicastIpAddressEntry

DeleteAnycastIpAddressEntry

DeleteIPAddress

DeleteIpForwardEntry

DeleteIpForwardEntry2

DeleteIpNetEntry

DeleteIpNetEntry2

DeletePersistentTcpPortReservation

DeletePersistentUdpPortReservation

DeleteProxyArpEntry

DeleteUnicastIpAddressEntry

DisableMediaSense

EnableRouter

FlushIpNetTable

FlushIpNetTable2

FlushIpPathTable

FreeMibTable

GetAdapterIndex

GetAdapterOrderMap

GetAdaptersAddresses

GetAdaptersInfo

GetAnycastIpAddressEntry

GetAnycastIpAddressTable

GetBestInterface

GetBestInterfaceEx

GetBestRoute

GetBestRoute2

GetDefaultCompartmentId

GetExtendedTcpTable

GetExtendedUdpTable

GetFriendlyIfIndex

GetIcmpStatistics

GetIcmpStatisticsEx

GetIfEntry

GetIfEntry2

GetIfEntry2Ex

GetIfStackTable

GetIfTable

GetIfTable2

GetIfTable2Ex

GetInterfaceInfo

GetInvertedIfStackTable

GetIpAddrTable

GetIpErrorString

GetIpForwardEntry2

GetIpForwardTable

GetIpForwardTable2

GetIpInterfaceEntry

GetIpInterfaceTable

GetIpNetEntry2

GetIpNetTable

GetIpNetTable2

GetIpNetworkConnectionBandwidthEstimates

GetIpPathEntry

GetIpPathTable

GetIpStatistics

GetIpStatisticsEx

GetMulticastIpAddressEntry

GetMulticastIpAddressTable

GetNetworkParams

GetNumberOfInterfaces

GetOwnerModuleFromTcpEntry

GetOwnerModuleFromTcp6Entry

GetOwnerModuleFromUdpEntry

GetOwnerModuleFromUdp6Entry

GetPerAdapterInfo

GetPerTcp6ConnectionEStats

GetPerTcpConnectionEStats

GetRTTAndHopCount

GetTcpStatistics

GetTcpStatisticsEx

GetTcpStatisticsEx2

GetTcp6Table

GetTcp6Table2

GetTcpTable

GetTcpTable2

GetTeredoPort

GetUdpStatistics

GetUdpStatisticsEx

GetUdpStatisticsEx2

GetUdpTable

GetUdp6Table

GetUnicastIpAddressEntry

GetUnicastIpAddressTable

GetUniDirectionalAdapterInfo

Icmp6CreateFile

Icmp6ParseReplies

Icmp6SendEcho2

IcmpCloseHandle

IcmpCreateFile

IcmpParseReplies

IcmpSendEcho

IcmpSendEcho2

IcmpSendEcho2Ex

if_indextoname

if_nametoindex

InitializeIpForwardEntry

InitializeIpInterfaceEntry

InitializeUnicastIpAddressEntry

IpReleaseAddress

IpRenewAddress

LookupPersistentTcpPortReservation

LookupPersistentUdpPortReservation

NhpAllocateAndGetInterfaceInfoFromStack

NotifyAddrChange

NotifyIpInterfaceChange

NotifyRouteChange

NotifyRouteChange2

NotifySecurityHealthChange

NotifyStableUnicastIpAddressTable

NotifyTeredoPortChange

NotifyUnicastIpAddressChange

ParseNetworkString

ResolveIpNetEntry2

ResolveNeighbor

RestoreMediaSense

RtlEthernetAddressToString

RtlEthernetStringToAddress

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

SendARP

SetIfEntry

SetIpForwardEntry

SetIpForwardEntry2

SetIpInterfaceEntry

SetIpNetEntry

SetIpNetEntry2

SetIpStatistics

IP_DAD_STATE

IP_PREFIX_ORIGIN

IP_SUFFIX_ORIGIN

NET_ADDRESS_FORMAT

SCOPE_LEVEL

TCP_BOOLEAN_OPTIONAL

TCP_ESTATS_TYPE

TCP_SOFT_ERROR

TCP_TABLE_CLASS

TCPIP_OWNER_MODULE_INFO_CLASS

UDP_TABLE_CLASS

IP Helper

1/7/2020 • 2 minutes to read • Edit Online

Purpose

The Internet Protocol Helper (IP Helper) API enables the retrieval and modification of network configuration settings for the local computer.

Where applicable

The IP Helper API is applicable in any computing environment where programmatically manipulating network and TCP/IP configuration is useful. Typical applications include IP routing protocols and Simple Network Management Protocol (SNMP) agents.

Developer audience

The IP Helper API is designed for use by C/C++ programmers. Programmers should also be familiar with Windows networking and TCP/IP networking concepts.

Run-time requirements

The IP Helper API can be used on all Windows platforms. Not all operating systems support all functions. Where certain implementations or capabilities of Windows Sockets 2 platform restrictions do exist, they are clearly noted in the documentation. If an IP Helper function is called on a platform that does not support the function, ERROR_NOT_SUPPORTED is returned. For more specific information about which operating systems support a particular function, refer to the Requirements sections in the documentation.

In this section

TOPIC	DESCRIPTION
What's New in IP Helper	Information on new features for IP Helper.
About IP Helper	General information about IP Helper programming considerations and capabilities available to developers.
Using IP Helper	Procedures and programming techniques used with IP Helper. This section includes basic IP Helper programming techniques, such as Getting Started With IP Helper.
IP Helper Reference	Reference documentation for the IP Helper functions, structures, and enumerations.

Related topics

W
i
n
d

owsSockets2

RoutingandRemoteAccessService

What's New in IP Helper

1/7/2020 • 2 minutes to read • Edit Online

Windows 8 and Windows Server 2012

The following features have been added to the IP Helper APIs on Windows 8 and Windows Server 2012.

A function that retrieves historical bandwidth estimates for a network connection. For more information, see:

- **GetIpNetworkConnectionBandwidthEstimates**

A structure that contains information on the available bandwidth estimates and associated variance as determined by the TCP/IP stack. For more information, see:

- **NL_BANDWIDTH_INFORMATION**

Windows 7 and Windows Server 2008 R2

The following features have been added to the IP Helper APIs on Windows 7 and Windows Server 2008 R2.

Functions that convert an Ethernet address between a binary format and string format for the Ethernet MAC address. For more information, see:

- **RtlEthernetAddressToString**
- **RtlEthernetStringToAddress**

Windows Server 2008 and Windows Vista SP1

The following functions have been added to the IP Helper APIs on Windows Server 2008 and Windows Vista with Service Pack 1 (SP1).

A function that works with IPv4 and the Internet Control Message Protocol (ICMP). For more information, see:

- **IcmpSendEcho2Ex**

Windows Vista

The following groups of functions have been added to the IP Helper APIs on Windows Vista and later.

Functions that work with both IPv6 and IPv4 for interface conversion. For more information, see:

- **ConvertInterfaceAliasToLuid**
- **ConvertInterfaceGuidToLuid**
- **ConvertInterfaceIndexToLuid**
- **ConvertInterfaceLuidToGuid**
- **ConvertInterfaceLuidToIndex**
- **ConvertInterfaceLuidToNameA**
- **ConvertInterfaceLuidToNameW**
- **ConvertInterfaceNameToLuidA**
- **ConvertInterfaceNameToLuidW**
- **if_indextoname**
- **if_nametoindex**

Functions that work with both IPv6 and IPv4 for interface management. For more information, see:

- **GetIfEntry2**
- **GetIfStackTable**
- **GetIfTable2**
- **GetIfTable2Ex**
- **GetInvertedIfStackTable**
- **GetIpInterfaceEntry**
- **GetIpInterfaceTable**
- **InitializeIpInterfaceEntry**
- **SetIpInterfaceEntry**

Functions that work with both IPv6 and IPv4 for IP address management. For more information, see:

- **CreateAnycastIpAddressEntry**
- **CreateUnicastIpAddressEntry**
- **DeleteAnycastIpAddressEntry**
- **DeleteUnicastIpAddressEntry**
- **GetAnycastIpAddressEntry**
- **GetAnycastIpAddressTable**
- **GetMulticastIpAddressEntry**
- **GetMulticastIpAddressTable**
- **GetUnicastIpAddressEntry**
- **GetUnicastIpAddressTable**
- **InitializeUnicastIpAddressEntry**
- **NotifyStableUnicastIpAddressTable**
- **SetUnicastIpAddressEntry**

A function that works with both IPv6 and IPv4 for IP table memory management. For more information, see:

- **FreeMibTable**

Functions that work with both IPv6 and IPv4 for IP neighbor address management. For more information, see:

- **CreateIpNetEntry2**
- **DeleteIpNetEntry2**
- **FlushIpNetTable2**
- **GetIpNetEntry2**
- **GetIpNetTable2**
- **ResolveIpNetEntry2**
- **ResolveNeighbor**
- **SetIpNetEntry2**

Functions that work with both IPv6 and IPv4 for IP path management. For more information, see:

- **FlushIpPathTable**
- **GetIpPathEntry**
- **GetIpPathTable**

Functions that work with both IPv6 and IPv4 for IP route management. For more information, see:

- **CreateIpForwardEntry2**
- **DeleteIpForwardEntry2**

- **GetBestRoute2**
- **GetIpForwardEntry2**
- **GetIpForwardTable2**
- **InitializeIpForwardEntry**
- **SetIpForwardEntry2**
- **SetIpStatisticsEx**

Functions that work with both IPv6 and IPv4 for notification. For more information, see:

- **CancelMibChangeNotify2**
- **NotifyIpInterfaceChange**
- **NotifyRouteChange2**
- **NotifyUnicastIpAddressChange**

Utility functions that work with IP addresses. For more information, see:

- **ConvertIpv4MaskToLength**
- **ConvertLengthToIpv4Mask**
- **CreateSortedAddressPairs**
- **ParseNetworkString**

Functions that work with Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) to retrieve the IPv6 or IPv4 TCP connection table or UDP listener table. For more information, see:

- **GetTcp6Table**
- **GetTcp6Table2**
- **GetTcpTable2**
- **GetUdp6Table**

Functions that work with Transmission Control Protocol (TCP) to retrieve extended TCP statistics on a connection. For more information, see:

- **GetPerTcp6ConnectionEStats**
- **GetPerTcpConnectionEStats**
- **SetPerTcp6ConnectionEStats**
- **SetPerTcpConnectionEStats**

New functions that work for Teredo IPv6 client management. For more information, see:

- **GetTeredoPort**
- **NotifyTeredoPortChange**
- **NotifyStableUnicastIpAddressTable**

Utility functions that provide conversions between IP addresses and string representations of IP addresses. For more information, see:

- **RtlIpv4AddressToString**
- **RtlIpv4AddressToStringEx**
- **RtlIpv4StringToAddress**
- **RtlIpv4StringToAddressEx**
- **RtlIpv6AddressToString**
- **RtlIpv6AddressToStringEx**
- **RtlIpv6StringToAddress**
- **RtlIpv6StringToAddressEx**

Functions that provide persistent reservations for a consecutive block of TCP or UDP ports on the local computer. For more information, see:

- **CreatePersistentTcpPortReservation**
- **CreatePersistentUdpPortReservation**
- **DeletePersistentTcpPortReservation**
- **DeletePersistentUdpPortReservation**
- **LookupPersistentTcpPortReservation**
- **LookupPersistentUdpPortReservation**

Windows Server 2003

The following functions have been added to the IP Helper APIs on Windows Server 2003 and later:

- **CancelSecurityHealthChangeNotify**
- **NotifySecurityHealthChange**

Windows XP SP2

The following functions have been added to the IP Helper APIs on Windows XP with Service Pack 2 (SP2) and later:

- **GetOwnerModuleFromTcpEntry**
- **GetOwnerModuleFromTcp6Entry**
- **GetOwnerModuleFromUdpEntry**
- **GetOwnerModuleFromUdp6Entry**

About IP Helper

1/7/2020 • 2 minutes to read • Edit Online

Internet Protocol Helper (IP Helper) assists network administration of the local computer by enabling applications to retrieve information about the network configuration of the local computer, and to modify that configuration. IP Helper also provides notification mechanisms to ensure that an application is notified when certain aspects of the local computer network configuration change.

Many of the IP Helper functions pass structure parameters that represent data types associated with the Management Information Base technology. The IP Helper functions use these structures to represent various networking information, such as ARP cache entries. Since these structures are also used by the MIB API, they are described in the Management Information Base Reference. Although the IP Helper API uses these structures, IP Helper is distinct from the Management Information Base (MIB) and Simple Network Management Protocol (SNMP).

The IP Helper documentation uses the terms "adapter" and "interface" extensively. An "adapter" is a legacy term that is an abbreviated form of network adapter which originally referred to some form of network hardware. An adapter is a datalink-level abstraction.

An "interface" is a newer term used in the IETF RFC documents. The RFCs define an interface as an abstract concept that represents a node's attachment to a link. An interface is an IP-level abstraction.

The adapter and interface terms are used somewhat interchangeably in the IP Helper documentation. In IP Helper documentation, a list of adapters could include a software WAN interface as well as the loopback interface (neither of these refer to a hardware device). In the IP Helper APIs, there is a one-to-one mapping of adapters to interfaces.

IP Helper provides capabilities in the following areas:

- Retrieving Information About Network Configuration
- Managing Network Adapters
- Managing Interfaces
- Managing IP Addresses
- Using the Address Resolution Protocol
- Retrieving Information on the Internet Protocol and the Internet Control Message Protocol
- Managing Routing
- Receiving Notification of Network Events
- Retrieving Information About the Transmission Control Protocol and the User Datagram Protocol

Retrieving Information About Network Configuration

1/7/2020 • 2 minutes to read • Edit Online

IP Helper provides information about the network configuration of the local computer. To retrieve general configuration information, use the **GetNetworkParams** function. This function returns information that is not specific to a particular adapter or interface. For example, **GetNetworkParams** returns a list of the DNS servers that are used by the local computer.

- For code samples involving **GetNetworkParams** see Retrieving Information Using GetNetworkParams.

Managing Network Adapters

1/7/2020 • 2 minutes to read • Edit Online

IP Helper provides capabilities for managing network adapters. There is a one-to-one correspondence between the interfaces and adapters on a given computer. An interface is an IP-level abstraction, whereas an adapter is a datalink-level abstraction.

Use the functions described in the following paragraphs to retrieve information about the network adapters in the local computer.

The **GetAdaptersInfo** function returns an array of **IP_ADAPTER_INFO** structures, one for each adapter in the local computer. The **GetPerAdapterInfo** function returns additional information about a specific adapter. The **GetPerAdapterInfo** function requires the caller to specify the index of the adapter. To obtain the adapter index from the adapter name, use the **GetAdapterIndex** function.

Some applications use adapters that receive datagrams, but cannot transmit them. To obtain information about these adapters, use the **GetUniDirectionalAdapterInfo** function.

The **GetAdaptersAddresses** function enables you to retrieve the IP addresses associated with a particular adapter. This function supports both IPv4 and IPv6 addressing.

- For code samples involving **GetAdaptersInfo** see Managing Network Adapters Using GetAdaptersInfo.

Managing Interfaces

1/7/2020 • 2 minutes to read • Edit Online

IP Helper extends your abilities to manage network interfaces. There is a one-to-one correspondence between the interfaces and adapters on a given computer. An interface is an IP-level abstraction, whereas an adapter is a datalink-level abstraction.

Use the functions described in the following paragraphs to manage interfaces on the local computer.

The **GetNumberOfInterfaces** function returns the number of interfaces on the local computer.

The **GetInterfaceInfo** function returns a table that contains the names and corresponding indexes for the interfaces on the local computer. For code samples involving **GetInterfaceInfo** see Managing Interfaces Using GetInterfaceInfo.

The **GetFriendlyIfIndex** function takes an interface index and returns a backward-compatible interface index, that is, one that uses only the lower 24 bits. This type of index is sometimes referred to as a "friendly" interface index.

The **GetIfEntry** function returns a **MIB_IFROW** structure that contains information about a particular interface on the local computer. This function requires the caller to supply the index of the interface.

The **GetIfTable** function returns a table of **MIB_IFROW** entries, one for each interface on the computer.

Use the **SetIfEntry** function to modify the configuration of a particular interface.

Managing IP Addresses

1/7/2020 • 2 minutes to read • Edit Online

IP Helper can assist in the management of IP addresses associated with interfaces on the local computer. Use the functions described in the following paragraphs for IP address management.

The **GetIpAddrTable** function retrieves a table that contains the mapping of IP addresses to interfaces. More than one IP address may be associated with the same interface.

Use the **AddIPAddress** function to add an IP address to a particular interface. To remove IP addresses that were previously added using **AddIPAddress**, use the **DeleteIPAddress** function.

The **IpReleaseAddress** and **IpRenewAddress** functions require the local computer to be using Dynamic Host Configuration Protocol (DHCP). The **IpReleaseAddress** function releases an IP address that was previously obtained from DHCP. The **IpRenewAddress** function renews a DHCP lease on a particular IP address. A DHCP lease allows a computer to use an IP address for a specified period of time.

- For code samples involving **GetIpAddrTable** see Managing IP Addresses Using GetIpAddrTable.
- For code samples involving **AddIPAddress** and **DeleteIPAddress**, see Managing IP Addresses Using AddIPAddress and DeleteIPAddress.
- For code samples involving **IpReleaseAddress** and **IpRenewAddress** see Managing DHCP Leases Using IpReleaseAddress and IpRenewAddress.

Using the Address Resolution Protocol

1/7/2020 • 2 minutes to read • Edit Online

You can use IP Helper to perform Address Resolution Protocol (ARP) operations for the local computer. Use the following functions to retrieve and modify the ARP table.

The **GetIpNetTable** retrieves the ARP table. The ARP table contains the mapping of IP addresses to physical addresses. Physical addresses are sometimes referred to as Media Access Controller (MAC) addresses.

Use the **CreateIpNetEntry** and **DeleteIpNetEntry** functions to add or remove particular ARP entries to or from the table. The **FlushIpNetTable** function deletes all entries from the table.

To create or delete proxy ARP entries, use the **CreateProxyArpEntry** and **DeleteProxyArpEntry** functions.

The **SendARP** function sends an ARP request to the local network.

Retrieving Information on the Internet Protocol and the Internet Control Message Protocol

1/7/2020 • 2 minutes to read • Edit Online

IP Helper provides information retrieval capabilities that are useful for the network administration of the local computer. The following functions retrieve statistics for the Internet Protocol (IP) and the Internet Control Message Protocol (ICMP). You can also use these functions to set certain configuration parameters for IP.

The **GetIpStatistics** function retrieves the current IP statistics for the local computer. For code samples involving **GetIpStatistics** see Retrieving Information Using GetIpStatistics.

The **GetIcmpStatistics** function retrieves the current ICMP statistics.

Use the **SetIpStatistics** function to enable or disable IP forwarding. This function also makes it possible for you to set the default time-to-live (TTL) for IP datagrams. Alternatively, you can set the TTL by using the **SetIpTTL** function.

Managing Routing

1/7/2020 • 2 minutes to read • Edit Online

IP Helper provides features to manage network routing. Use the following functions to manage the IP routing table, and to obtain other routing information.

Retrieve the contents of the IP routing table by making a call to the **GetIpForwardTable** function. Manipulate specific entries in the IP routing table using the **CreateIpForwardEntry**, **DeleteIpForwardEntry**, and **SetIpForwardEntry** functions. Use the **CreateIpForwardEntry** function to add a new routing table entry. Use the **DeleteIpForwardEntry** function to remove an existing entry. The **SetIpForwardEntry** function modifies an existing entry.

The router management capabilities of IP Helper can be used to retrieve information about how datagrams are routed over the network. The **GetBestRoute** function retrieves the best route to a specified destination address. The **GetBestInterface** function retrieves the index of the interface used by the best route to a specified destination address. Lastly, the **GetRTTAndHopCount** function retrieves the round-trip time (RTT) and number of hops to a specified destination address.

Receiving Notification of Network Events

1/7/2020 • 2 minutes to read • Edit Online

Use the following functions to ensure that an application receives notification of certain network events.

Use the **NotifyAddrChange** function to request notification of any change that occurs in the mapping between IP addresses and interfaces on the local computer.

Similarly, the **NotifyRouteChange** function enables an application to request notification of any change that occurs in the IP routing table.

The notifications provided by these functions do not specify what changed; they simply specify that something changed. Use other IP Helper functions, such as **GetIPAddrTable** and **GetBestRoute**, to determine the exact nature of the change.

Retrieving Information About the Transmission Control Protocol and the User Datagram Protocol

1/7/2020 • 2 minutes to read • Edit Online

IP Helper makes it possible to access information about network protocols that are used on the local computer. Use the functions described in the following paragraphs to retrieve information about the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) on the local computer.

The **GetTcpStatistics** function retrieves the current statistics for TCP. For code samples involving **GetTcpStatistics** see Retrieving Information Using GetTcpStatistics.

The **GetUdpStatistics** function retrieves the current statistics for UDP.

The **GetTcpTable** function retrieves the TCP connection table. The **GetUdpTable** retrieves the UDP listener table.

The **SetTcpEntry** function enables a developer to set the state of a specified TCP connection to MIB_TCP_STATE_DELETE_TCB.

Using IP Helper

1/7/2020 • 2 minutes to read • Edit Online

This section describes procedures and programming techniques employed with IP Helper. It includes basic IP Helper programming techniques, such as Getting Started with IP Helper, as well as advanced techniques useful for more experienced IP Helper developers.

The following list describes the topic in this section:

Getting Started with IP Helper

Getting Started with IP Helper

1/7/2020 • 2 minutes to read • Edit Online

The following is a step-by-step guide to getting started programming using the IP Helper application programming interface (API). It is designed to provide an understanding of basic IP Helper functions and data structures, and how they work together.

The application that is used for illustration is a very basic IP Helper application. More advanced code examples are included in the samples included with the Microsoft Windows Software Development Kit (SDK).

The first step is the same for most IP Helper applications.

- Creating a Basic IP Helper Application

The following sections describe the remaining steps for creating this basic IP Helper application.

- Retrieving Information Using GetNetworkParams
- Managing Network Adapters Using GetAdaptersInfo
- Managing Interfaces Using GetInterfaceInfo
- Managing IP Addresses Using GetIpAddrTable
- Managing DHCP Leases Using IpReleaseAddress and IpRenewAddress
- Managing IP Addresses Using AddIPAddress and DeleteIPAddress
- Retrieving Information Using GetIpStatistics
- Retrieving Information Using GetTcpStatistics

The complete source code for this basic IP Helper example.

- Complete IP Helper Application Source Code

Advanced IP Helper Samples

Several more advanced IP Helper samples are included with the Microsoft Windows Software Development Kit (SDK). By default, the IP Helper sample source code is installed by the Windows SDK released for Windows 7 in the following directory:

C:\Program Files\Microsoft SDKs\Windows\v7.0\Samples\NetDs\IPHelp

The more advanced samples listed below are found in the following directories:

- EnableRouter

  This directory contains a sample that demonstrates how to use the **EnableRouter** and **UnenableRouter** IP Helper functions to enable and disable IPv4 forwarding on the local computer.

- iparp

  This directory contains a sample program that demonstrates how to use the IP Helper functions to display and manipulate entries in the IPv4 ARP table on the local computer.

- ipchange

  This directory contains a sample program that demonstrates how to use IP Helper functions to programmatically change an IP address for a specific network adapter on your machine. This program also demonstrates how to retrieve existing network adapter IP configuration information.

- IPConfig

  This directory contains a sample program that demonstrates how to programmatically retrieve IPv4 configuration information similar to the IPCONFIG.EXE utility. It demonstrates how to use the **GetNetworkParams** and **GetAdaptersInfo** functions. Note that the **GetAdaptersInfo** function only retrieves IPv4 information.

- IPRenew

  This directory contains a sample program that demonstrates how to programmatically release and renew IPv4 addresses obtained through DHCP. This program also demonstrates how to retrieve existing network adapter configuration information.

- IPRoute

  This directory contains a sample program that demonstrates how to use the IP Helper functions to manipulate the IPv4 routing table.

- ipstat

  This directory contains a sample program that demonstrates how to use the IP Helper functions to show IPv4 connections for a protocol. By default, statistics are shown for IP, ICMP, TCP and UDP.

- Netinfo

  This directory contains a sample program that demonstrates how to the use the new IP Helper APIs introduced on Windows Vista and later to display/change address and interface information for IPv4 and IPv6.

Creating a Basic IP Helper Application

1/7/2020 • 2 minutes to read • Edit Online

To create a basic IP Helper application

1. Create a new empty project.
2. Add an empty C++ source file to the project.
3. Ensure that the build environment refers to the Include, Lib, and Src directories of the Platform Software Development Kit (SDK).
4. Ensure that the build environment links to the IP Helper Library file Iphlpapi.lib and the Winsock Library file WS2_32.lib.

   > **NOTE**
   > Some basic Winsock functions are used to return IP address values and other information.

5. Begin programming the IP Helper application. Use the IP Helper API by including the IP Helper header file.

```
#include <winsock2.h>
#include <iphlpapi.h>
#include <stdio.h>

int main() {
  return 0;
}
```

NOTE

The *Iphlpapi.h* header file is required for applications that use the IP Helper functions. The *Iphlpapi.h* header file automatically includes other headers files with structures and enumerations used by the IP Helper functions.

The new IP Helper functions introduced in Windows Vista and later are defined in the *Netioapi.h* header file which is automatically included by the *Iphlpapi.h* header file. The *Netioapi.h* header file should never be used directly.

Many of the structures and enumerations used by IP Helper functions are defined in the *Iprtrmib.h*, *Ipexport.h*, and *Iptypes.h* header files. These header files are automatically included in the *Iphlpapi.h* header file and should never be used directly.

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed. Some of the structures are now defined in the *Ipmib.h*, *Tcpmib.h*, and *Udpmib.h* header files, not in the *Iprtrmib.h* header file. The *Ipmib.h* header file automatically includes the *Ifmib.h* header file. Note that these header files are automatically included in *Iprtrmib.h*, which is automatically included in the *Iphlpapi.h* header file.

The *Winsock2.h* header file for Windows Sockets 2.0 is required by most applications using the IP Helper APIs. When the *Winsock2.h* header file is required, the #include line for this file should be placed before the #include line for the *Iphlpapi.h* header file.

The *Winsock2.h* header file internally includes core elements from the *Windows.h* header file, so there is not usually an #include line for *Windows.h* header file in IP Helper applications. If an #include line is needed for the *Windows.h* header file, this should be preceded with the #define WIN32_LEAN_AND_MEAN macro. For historical reasons, the *Windows.h* header defaults to including the *Winsock.h* header file for Windows Sockets 1.1. The declarations in the *Winsock.h* header file for Windows Sockets 1.1 will conflict with the declarations in the *Winsock2.h* header file required by Windows Sockets 2.0. The WIN32_LEAN_AND_MEAN macro prevents the *Winsock.h* header file from being included by the *Windows.h* header file. An example illustrating this is shown below.

```
#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#include <windows.h>

#include <winsock2.h>
#include <iphlpapi.h>
#include <stdio.h>

int main() {
  return 0;
}
```

NOTE

This basic IP Helper application only uses some IP address data structures and IP address to string conversion functions from Windows Sockets 2.0. These Windows Sockets functions can be used without calling **WSAStartup** to initialize Windows Sockets resources and **WSACleanup** when done using these resources.

In IP Helper applications that use other Winsock functions other than these IP address to string functions, the **WSAStartup** function must be called to initialize Windows Sockets resources before calling any Windows Sockets functions and **WSACleanup** should be called when the application is done using Windows Sockets resources.

NOTE

The *Stdio.h* header file is required for the use of various standard C functions in this basic IP Helper application.

Next Step: Retrieving Information Using GetNetworkParams

Retrieving Information Using GetNetworkParams

1/7/2020 • 2 minutes to read • Edit Online

The **GetNetworkParams** function fills a pointer to a **FIXED_INFO** structure with data about the current network settings.

To use GetNetworkParams

1. Declare a pointer to a **FIXED_INFO** object called *pFixedInfo*, and a **ULONG** object called *ulOutBufLen*. These variables are passed as parameters to the **GetNetworkParams** function. Also create a **DWORD** variable *dwRetVal* (used for error checking).

```
FIXED_INFO *pFixedInfo;
IP_ADDR_STRING *pIPAddr;

ULONG ulOutBufLen;
DWORD dwRetVal;
```

2. Allocate memory for the structures.

> **NOTE**
> The size of *ulOutBufLen* is not sufficient to hold the information. See the next step.

```
pFixedInfo = (FIXED_INFO *) malloc(sizeof (FIXED_INFO));
ulOutBufLen = sizeof (FIXED_INFO);
```

3. Make an initial call to **GetNetworkParams** to get the size required for the *ulOutBufLen* variable.

> **NOTE**
> This function function will fail, and is used to ensure that the *ulOutBufLen* variable specifies a size sufficient for holding all the data returned to *pFixedInfo*. This is a common programming model for data structures and functions of this type.

```
if (GetNetworkParams(pFixedInfo, &ulOutBufLen) == ERROR_BUFFER_OVERFLOW) {
    free(pFixedInfo);
    pFixedInfo = (FIXED_INFO *) malloc(ulOutBufLen);
    if (pFixedInfo == NULL) {
        printf("Error allocating memory needed to call GetNetworkParams\n");
    }
}
```

4. Make a second call to **GetNetworkParams** using general error checking and returning its value to the **DWORD** variable *dwRetVal*; used for more advanced error checking.

```
if (dwRetVal = GetNetworkParams(pFixedInfo, &ulOutBufLen) != NO_ERROR) {
    printf("GetNetworkParams failed with error %d\n", dwRetVal);
    if (pFixedInfo) {
        free(pFixedInfo);
    }
}
```

5. If the call was successful, access the data from the *pFixedInfo* data structure.

```
    printf("\tHost Name: %s\n", pFixedInfo->HostName);
    printf("\tDomain Name: %s\n", pFixedInfo->DomainName);
    printf("\tDNS Servers:\n");
    printf("\t\t%s\n", pFixedInfo->DnsServerList.IpAddress.String);

    pIPAddr = pFixedInfo->DnsServerList.Next;
    while (pIPAddr) {
        printf("\t\t%s\n", pIPAddr->IpAddress.String);
        pIPAddr = pIPAddr->Next;
    }

    printf("\tNode Type: ");
    switch (pFixedInfo->NodeType) {
    case 1:
        printf("%s\n", "Broadcast");
        break;
    case 2:
        printf("%s\n", "Peer to peer");
        break;
    case 4:
        printf("%s\n", "Mixed");
        break;
    case 8:
        printf("%s\n", "Hybrid");
        break;
    default:
        printf("\n");
    }

    printf("\tNetBIOS Scope ID: %s\n", pFixedInfo->ScopeId);

    if (pFixedInfo->EnableRouting)
        printf("\tIP Routing Enabled: Yes\n");
    else
        printf("\tIP Routing Enabled: No\n");

    if (pFixedInfo->EnableProxy)
        printf("\tWINS Proxy Enabled: Yes\n");
    else
        printf("\tWINS Proxy Enabled: No\n");

    if (pFixedInfo->EnableDns)
        printf("\tNetBIOS Resolution Uses DNS: Yes\n");
    else
        printf("\tNetBIOS Resolution Uses DNS: No\n");
```

6. Free any memory allocated for the *pFixedInfo* structure.

```
if (pFixedInfo) {
    free(pFixedInfo);
    pFixedInfo = NULL;
}
```

Next Step: Managing Network Adapters Using GetAdaptersInfo

Previous Step: Creating a Basic IP Helper Application

Managing Network Adapters Using GetAdaptersInfo

1/7/2020 • 2 minutes to read • Edit Online

The **GetAdaptersInfo** function fills a pointer to an **IP_ADAPTER_INFO** structure with information about the network adapters associated with the system.

To use GetAdaptersInfo

1. Declare a pointer to an **IP_ADAPTER_INFO** variable called *pAdapterInfo*, and a **ULONG** variable called *ulOutBufLen*. These variables are passed as parameters to the **GetAdaptersInfo** function. Also create a **DWORD** variable called *dwRetVal* (for error checking).

```
IP_ADAPTER_INFO  *pAdapterInfo;
ULONG            ulOutBufLen;
DWORD            dwRetVal;
```

2. Allocate memory for the structures.

```
pAdapterInfo = (IP_ADAPTER_INFO *) malloc( sizeof(IP_ADAPTER_INFO) );
ulOutBufLen = sizeof(IP_ADAPTER_INFO);
```

3. Make an initial call to **GetAdaptersInfo** to get the size needed into the *ulOutBufLen* variable.

> **NOTE**
> This call to the function is meant to fail, and is used to ensure that the *ulOutBufLen* variable specifies a size sufficient for holding all the information returned to *pAdapterInfo*. This is a common programming model for data structures and functions of this type.

```
if (GetAdaptersInfo( pAdapterInfo, &ulOutBufLen) != ERROR_SUCCESS) {
    free (pAdapterInfo);
    pAdapterInfo = (IP_ADAPTER_INFO *) malloc ( ulOutBufLen );
}
```

4. Make a second call to **GetAdaptersInfo**, passing *pAdapterInfo* and *ulOutBufLen* as parameters and doing general error checking. Return its value to the **DWORD** variable *dwRetVal* (for more extensive error checking).

```
if ((dwRetVal = GetAdaptersInfo( pAdapterInfo, &ulOutBufLen)) != ERROR_SUCCESS) {
    printf("GetAdaptersInfo call failed with %d\n", dwRetVal);
}
```

5. If the call was successful, access some of the data in the *pAdapterInfo* structure.

```
PIP_ADAPTER_INFO pAdapter = pAdapterInfo;
while (pAdapter) {
    printf("Adapter Name: %s\n", pAdapter->AdapterName);
    printf("Adapter Desc: %s\n", pAdapter->Description);
    printf("\tAdapter Addr: \t");
    for (UINT i = 0; i < pAdapter->AddressLength; i++) {
        if (i == (pAdapter->AddressLength - 1))
            printf("%.2X\n",(int)pAdapter->Address[i]);
        else
            printf("%.2X-",(int)pAdapter->Address[i]);
    }
    printf("IP Address: %s\n", pAdapter->IpAddressList.IpAddress.String);
    printf("IP Mask: %s\n", pAdapter->IpAddressList.IpMask.String);
    printf("\tGateway: \t%s\n", pAdapter->GatewayList.IpAddress.String);
    printf("\t***\n");
    if (pAdapter->DhcpEnabled) {
        printf("\tDHCP Enabled: Yes\n");
        printf("\t\tDHCP Server: \t%s\n", pAdapter->DhcpServer.IpAddress.String);
    }
    else
      printf("\tDHCP Enabled: No\n");

  pAdapter = pAdapter->Next;
}
```

6. Free any memory allocated for the *pAdapterInfo* structure.

```
if (pAdapterInfo)
        free(pAdapterInfo);
```

Next Step: Managing Interfaces Using GetInterfaceInfo

Previous Step: Retrieving Information Using GetNetworkParams

Managing Interfaces Using GetInterfaceInfo

1/7/2020 • 2 minutes to read • Edit Online

The **GetInterfaceInfo** function fills a pointer to an **IP_INTERFACE_INFO** structure with information about the interfaces associated with the system.

To use GetInterfaceInfo

1. Declare a pointer to an **IP_INTERFACE_INFO** object called `pInfo`, and a **ULONG** object called `ulOutBufLen`. Also declare a **DWORD** object called `dwRetVal` (used for error checking).

```
ULONG               ulOutBufLen;
DWORD               dwRetVal;
unsigned int       i;

IP_INTERFACE_INFO*  pInterfaceInfo;
```

2. Allocate memory for the structures.

> **NOTE**
> The size of `ulOutBufLen` is not sufficient to hold the information. See the next step.

```
pInterfaceInfo = (IP_INTERFACE_INFO *) malloc(sizeof (IP_INTERFACE_INFO));
ulOutBufLen = sizeof(IP_INTERFACE_INFO);
```

3. Make an initial call to **GetInterfaceInfo** to get the size needed into the `ulOutBufLen` variable.

> **NOTE**
> This call to the function is meant to fail, and is used to ensure that the `ulOutBufLen` variable specifies a size sufficient for holding all the information returned to `pInfo`. This is a common programming model in IP Helper for data structures and functions of this type.

```
if (GetInterfaceInfo(pInterfaceInfo, &ulOutBufLen) ==
    ERROR_INSUFFICIENT_BUFFER) {
    free(pInterfaceInfo);
    pInterfaceInfo = (IP_INTERFACE_INFO *) malloc(ulOutBufLen);
}
```

4. Make a second call to **GetInterfaceInfo** with general error checking and return its value to the **DWORD** variable `dwRetVal` (for more advanced error checking).

```
    if ((dwRetVal = GetInterfaceInfo(pInterfaceInfo, &ulOutBufLen)) != NO_ERROR) {
        printf("  GetInterfaceInfo failed with error: %d\n", dwRetVal);
    }
```

5. If the call was successful, access the data from the `pInfo` data structure.

```
        printf("  GetInterfaceInfo succeeded.\n");

        printf("  Num Adapters: %ld\n\n", pInterfaceInfo->NumAdapters);
        for (i = 0; i < (unsigned int) pInterfaceInfo->NumAdapters; i++) {
            printf("  Adapter Index[%d]: %ld\n", i,
                   pInterfaceInfo->Adapter[i].Index);
            printf("  Adapter Name[%d]:  %ws\n\n", i,
                   pInterfaceInfo->Adapter[i].Name);
        }
    }
```

> **NOTE**
>
> The %ws in the first line denotes a wide string. This is used because the **Name** attribute of the **IP_ADAPTER_INDEX_MAP** structure `Adapter` is a **WCHAR**, which is a Unicode string.

6. Free any memory allocated for the *pInfo* structure.

```
    if (pInterfaceInfo) {
        free(pInterfaceInfo);
        pInterfaceInfo = NULL;
    }
```

Next Step: Managing IP Addresses Using GetIpAddrTable

Previous Step: Managing Network Adapters Using GetAdaptersInfo

Managing IP Addresses Using GetIpAddrTable

1/7/2020 • 2 minutes to read • Edit Online

The **GetIpAddrTable** function fills a pointer to an **MIB_IPADDRTABLE** structure with information about the current IP addresses associated with the system.

To use GetIpAddrTable

1. Declare a pointer to an **MIB_IPADDRTABLE** object called *pIPAddrTable*, and a **DWORD** object called *dwSize*. These variables are passed as parameters to the **GetIpAddrTable** function. Also create a **DWORD** variable called *dwRetVal* (used for error checking).

```
MIB_IPADDRTABLE  *pIPAddrTable;
DWORD            dwSize = 0;
DWORD            dwRetVal;
```

2. Allocate memory for the structure.

> **NOTE**
> The size of *dwSize* is not sufficient to hold the information. See the next step.

```
pIPAddrTable = (MIB_IPADDRTABLE*) malloc( sizeof(MIB_IPADDRTABLE) );
```

3. Make an initial call to **GetIpAddrTable** to get the size needed into the *dwSize* variable.

> **NOTE**
> This call to the function is meant to fail, and is used to ensure that the *dwSize* variable specifies a size sufficient for holding all the information returned to *pIPAddrTable*. This is a common programming model for data structures and functions of this type.

```
if (GetIpAddrTable(pIPAddrTable, &dwSize, 0) == ERROR_INSUFFICIENT_BUFFER) {
    free( pIPAddrTable );
    pIPAddrTable = (MIB_IPADDRTABLE *) malloc ( dwSize );
}
```

4. Make a second call to **GetIpAddrTable** with general error checking and return its value to the **DWORD** variable *dwRetVal* (for more advanced error checking).

```
if ( (dwRetVal = GetIpAddrTable( pIPAddrTable, &dwSize, 0 )) != NO_ERROR ) {
    printf("GetIpAddrTable call failed with %d\n", dwRetVal);
}
```

5. If the call was successful, access the data from the *pIPAddrTable* data structure.

```
printf("IP Address:         %ld\n", pIPAddrTable->table[0].dwAddr);
printf("IP Mask:            %ld\n", pIPAddrTable->table[0].dwMask);
printf("IF Index:           %ld\n", pIPAddrTable->table[0].dwIndex);
printf("Broadcast Addr:     %ld\n", pIPAddrTable->table[0].dwBCastAddr);
printf("Re-assembly size:   %ld\n", pIPAddrTable->table[0].dwReasmSize);
```

6. Free any memory allocated for the *pIPAddrTable* structure.

```
if (pIPAddrTable)
        free(pIPAddrTable);
```

NOTE

The **DWORD** objects *dwAddr* and *dwMask* are returned as numerical values in host byte order, not network byte order. These values are not dotted IP addresses.

Next Step: Managing DHCP Leases Using IpReleaseAddress and IpRenewAddress

Previous Step: Managing Interfaces Using GetInterfaceInfo

Manage DHCP leases with IpReleaseAddress, IpRenewAddress

1/7/2020 • 2 minutes to read • Edit Online

The **IpReleaseAddress** and **IpRenewAddress** functions are used to release and renew the current Dynamic Host Configuration Protocol (DHCP) lease. The **IpReleaseAddress** function releases an IPv4 address previously obtained through DHCP. The **IpRenewAddress** function renews a lease on an IPv4 address previously obtained through DHCP. It is common to use these two functions together, first releasing the lease with a call to **IpReleaseAddress**, and then renewing the lease with a call to the **IpRenewAddress** function.

When a DHCP client has previously obtained a DHCP lease and **IpReleaseAddress** is not called before the **IpRenewAddress** function, the DHCP client request is sent to the DHCP server that issued the initial DHCP lease. This DHCP server may not available or the DHCP request may fail. When a host has previously obtained a DHCP lease and **IpReleaseAddress** is called before the **IpRenewAddress** function, the DHCP client first releases the IP address obtained and sends a DHCP client request for a response from any available DHCP server.

> **NOTE**
> The **IpReleaseAddress** and **IpRenewAddress** functions require that DHCP be enabled to perform correctly.

The **IpReleaseAddress** function takes a pointer to an **IP_ADAPTER_INDEX_MAP** structure as its only parameter. To obtain this parameter, first call **GetInterfaceInfo**. For help with the **GetInterfaceInfo** function, see Managing Interfaces Using GetInterfaceInfo.

To use IpReleaseAddress

1. Obtain a pointer to an **IP_ADAPTER_INDEX_MAP** structure using the **GetInterfaceInfo** function. (For help with the GetInterfaceInfo function, see Managing Interfaces Using GetInterfaceInfo). Create a **DWORD** object `dwRetVal` (used for error checking). It is assumed that the variable returned by **GetInterfaceInfo** is called `pInfo`.

   ```
   DWORD dwRetVal;
   ```

2. If DHCP is enabled, call the **IpReleaseAddress** function, passing the **IP_ADAPTER_INDEX_MAP** variable `Adapter` as its parameter. Check for general errors and return its value to the **DWORD** variable `dwRetVal` (for more extensive error checking).

   > **NOTE**
   > The **GetAdaptersInfo** function returns a parameter that can be used to check whether DHCP is enabled before calling these functions. For help with **GetAdaptersInfo**, see Managing Network Adapters Using GetAdaptersInfo.

```
if ((dwRetVal = IpReleaseAddress(&pInfo->Adapter[0])) == NO_ERROR) {
    printf("Ip Release succeeded.\n");
}
```

NOTE

It is common to use these two functions together, calling the **IpReleaseAddress** function and then calling the **IpRenewAddress** function, passing the same structure as the parameter to both functions. The following procedure assumes that the functions are not used together; however, if the functions are used together, skip step 1.

To use IpRenewAddress

1. Obtain a pointer to an **IP_ADAPTER_INDEX_MAP** structure using the **GetInterfaceInfo** function. (For help with the **GetInterfaceInfo** function, see Managing Interfaces Using GetInterfaceInfo). Declare a **DWORD** object `dwRetVal` (used for error checking) if this variable has not been declared. It is assumed that the variable returned by **GetInterfaceInfo** is called `pInfo`.

   ```
   DWORD dwRetVal;
   ```

2. Call the **IpRenewAddress** function, passing the **IP_ADAPTER_INDEX_MAP** variable `Adapter` as its parameter. Check for general errors and return its value to the **DWORD** variable `dwRetVal` (for more extensive error checking).

   ```
   if ((dwRetVal = IpRenewAddress(&pInfo->Adapter[0])) == NO_ERROR) {
       printf("Ip Renew succeeded.\n");
   }
   ```

Next Step: Managing IP Addresses Using AddIPAddress and DeleteIPAddress

Previous Step: Managing IP Addresses Using GetIpAddrTable

Manage IP addresses with AddIPAddress, DeleteIPAddress

1/7/2020 • 2 minutes to read • Edit Online

The **AddIPAddress** function adds the specified IPv4 address to the specified adapter. The **DeleteIPAddress** function deletes the specified IPv4 address from the specified adapter. These functions can be used to add and delete IPv4 addresses to a network adapter.

An IPv4 address added by the **AddIPAddress** function is not persistent. The IPv4 address exists only as long as the adapter object exists. Restarting the computer destroys the IPv4 address, as does manually resetting the network interface card (NIC).

After **AddIPAddress** has been called successfully, DHCP will be disabled for the IP address that was added. Therefore, functions such as **IpReleaseAddress**, which require DHCP to be enabled, will not be functional on the added IP address. The **DeleteIPAddress** function can be used to delete the added IPv4 address.

> **NOTE**
> Group policies, enterprise policies, and other restrictions on the network may prevent these functions from completing successfully. Ensure that the application has the necessary network permissions before attempting to use these functions.

To use AddIPAddress

1. Declare **ULONG** variables named `NTEContext` and `NTEInstance`, both initialized to zero.

   > **NOTE**
   > The `NTEContext` variable is the only parameter to the **DeleteIPAddress** function; to delete the IP address that is added, `NTEContext` must be stored and unchanged.

   ```
   ULONG NTEContext = 0;
   ULONG NTEInstance = 0;
   ```

   > **NOTE**

2. Declare variables for the IPAddr and IPMask structures named `iaIPAddress` and `iaIPMask`, respectively. These values are simply unsigned integers. Initialize the `iaIPAddress` and `iaIPMask` variables using the **inet_addr** function.

```
UINT iaIPAddress;
UINT iaIPMask;

iaIPAddress = inet_addr("192.168.0.5");
iaIPMask    = inet_addr("255.255.255.0");
```

3. Call the **AddIPAddress** function to add the IPv4 address. Check for errors and return the error value to the **DWORD** variable `dwRetVal` (for more extensive error checking).

```
dwRetVal = AddIPAddress(iaIPAddress, iaIPMask, pIPAddrTable->table[0].dwIndex,
                        &NTEContext, &NTEInstance);
if (dwRetVal != NO_ERROR) {
    printf("AddIPAddress call failed with %d\n", dwRetVal);
}
```

> **NOTE**
>
> The third parameter is the adapter index, which can be obtained by calling the **GetIpAddrTable** function. It is assumed that the variable returned by this function is named `pIPAddrTable`. For help with the **GetIpAddrTable** function, see Managing IP Address Using GetIpAddrTable.

To use DeleteIpAddress

- Call the **DeleteIPAddress** function, passing the `NTEContext` variable as its parameter. Check for errors and return the error value to the **DWORD** variable `dwRetVal` (for more extensive error checking).

```
dwRetVal = DeleteIPAddress(NTEContext);
if (dwRetVal != NO_ERROR) {
        printf("\tDeleteIPAddress failed with error: %d\n", dwRetVal);
}
```

> **NOTE**
>
> To use **DeleteIPAddress**, **AddIPAddress** must first be called to get the handle `NTEContext`. The previous procedure assumes that **AddIPAddress** has already been called somewhere in the code, and `NTEContext` has been saved and remains uncorrupted.

Next Step: Retrieving Information Using GetIpStatistics

Previous Step: Managing DHCP Leases Using IpReleaseAddress and IpRenewAddress

Retrieving Information Using GetIpStatistics

1/7/2020 • 2 minutes to read • Edit Online

The **GetIpStatistics** function fills a pointer to an **MIB_IPSTATS** structure with information about the current IP statistics associated with the system.

To use GetIpStatistics

1. Declare some variables that are needed.

   Declare a **DWORD** variable `dwRetval` that will be using for error checking function calls. Declare a pointer to an **MIB_IPSTATS** variable called *pStats*, and allocate memory for the structure. Check that memory could be allocated.

   ```
   MIB_IPSTATS  *pStats;
   DWORD        dwRetVal = 0;

   pStats = (MIB_IPSTATS*) malloc(sizeof(MIB_IPSTATS));

   if (pStats == NULL) {
       printf("Unable to allocate memory for MIB_IPSTATS\n");
   }
   ```

2. Call the **GetIpStatistics** function with the *pStats* parameter to retrieve IP statistics for the local computer. Check for errors and return the error value in the **DWORD** variable `dwRetval`. If an error occurs, the `dwRetval` variable can be used for more extensive error checking and reporting.

   ```
   dwRetVal = GetIpStatistics(pStats);
   if (dwRetVal != NO_ERROR) {
       printf("GetIpStatistics call failed with %d\n", dwRetVal);
   }
   ```

3. If the call to **GetIpStatistics** was successful, print out some of the data in the **MIB_IPSTATS** structure pointed to by the *pStats* parameter.

   ```
   printf("Number of interfaces:   %ld\n", pStats->dwNumIf);
   printf("Number of IP addresses: %ld\n", pStats->dwNumAddr);
   printf("Number of received datagrams:  %ld\n", pStats->dwInReceives);
   printf("NUmber of outgoing datagrams requested to transmit:  %ld\n", pStats->dwOutRequests);
   ```

4. Free the memory allocated for the **MIB_IPSTATS** structure pointed to by the *pStats* parameter. This should be done once the application no longer needs the data returned by the *pStats* parameter.

   ```
   if (pStats)
       free(pStats);
   ```

Next Step: Retrieving Information Using GetTcpStatistics

Previous Step: Managing IP Addresses Using AddIPAddress and DeleteIPAddress

Retrieving Information Using GetTcpStatistics

1/7/2020 • 2 minutes to read • Edit Online

The **GetTcpStatistics** function fills a pointer to an **MIB_TCPSTATS** structure with information on the TCP protocol statistics for the local computer.

To use GetTcpStatistics

1. Declare some variables that are needed.

   Declare a **DWORD** variable `dwRetVal` that will be using for error checking function calls. Declare a pointer to an **MIB_TCPSTATS** variable called *pTCPStats*, and allocate memory for the structure. Check that memory could be allocated.

   ```
   DWORD dwRetVal = 0;
   PMIB_TCPSTATS pTCPStats;

   pTCPStats = (MIB_TCPSTATS *) malloc(sizeof (MIB_TCPSTATS));
   if (pTCPStats == NULL) {
       printf("Error allocating memory\n");
   }
   ```

2. Call the **GetTcpStatistics** function with the *pTCPStats* parameter to retrieve TCP statistics for IPv4 on the local computer. Check for errors and return the error value in the **DWORD** variable `dwRetVal`. If an error occurs, the `dwRetVal` variable can be used for more extensive error checking and reporting.

   ```
       if ((dwRetVal = GetTcpStatistics(pTCPStats)) != NO_ERROR) {
           printf("GetTcpStatistics failed with error: %ld\n", dwRetVal);
       }
   ```

3. If the call was successful, access the data returned in the **MIB_TCPSTATS** pointed to by the *pTCPStats* parameter.

   ```
   printf("\tNumber of active opens:  %u\n", pTCPStats->dwActiveOpens);
   printf("\tNumber of passive opens: %u\n", pTCPStats->dwPassiveOpens);
   printf("\tNumber of segments received: %u\n", pTCPStats->dwInSegs);
   printf("\tNumber of segments transmitted: %u\n", pTCPStats->dwOutSegs);
   printf("\tNumber of total connections: %u\n", pTCPStats->dwNumConns);
   ```

4. Free the memory allocated for the **MIB_TCPSTATS** structure pointed to by the *pTCPStats* parameter. This should be done once the application no longer needs the data returned by the *pTCPStats* parameter.

   ```
   if (pTCPStats)
       free(pTCPStats);
   ```

Next Step: Retrieving Information Using GetIpStatistics

Previous Step: Retrieving Information Using GetIpStatistics

Complete Source Code

- Complete IP Helper Application Source Code

Complete IP Helper Application Source Code

1/7/2020 • 6 minutes to read • Edit Online

The following is the complete source code for the IP Helper application. Additional error checking has been added to source code.

```
#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#include <windows.h>

#include <winsock2.h>
#include <ws2tcpip.h>

#include <iphlpapi.h>

#include <stdio.h>
#include <time.h>

// Need to link with Iphlpapi.lib and Ws2_32.lib
#pragma comment(lib, "iphlpapi.lib")
#pragma comment(lib, "ws2_32.lib")

#define MALLOC(x) HeapAlloc(GetProcessHeap(), 0, (x))
#define FREE(x) HeapFree(GetProcessHeap(), 0, (x))
/* Note: could also use malloc() and free() */

int main()
{

    /* Some general variables */
    ULONG ulOutBufLen;
    DWORD dwRetVal;
    unsigned int i;

    /* variables used for GetNetworkParams */
    FIXED_INFO *pFixedInfo;
    IP_ADDR_STRING *pIPAddr;

    /* variables used for GetAdapterInfo */
    IP_ADAPTER_INFO *pAdapterInfo;
    IP_ADAPTER_INFO *pAdapter;

    /* variables used to print DHCP time info */
    struct tm newtime;
    char buffer[32];
    errno_t error;

    /* variables used for GetInterfaceInfo */
    IP_INTERFACE_INFO *pInterfaceInfo;

    /* variables used for GetIpAddrTable */
    MIB_IPADDRTABLE *pIPAddrTable;
    DWORD dwSize;
    IN_ADDR IPAddr;
    char *strIPAddr;

    /* variables used for AddIpAddress */
    UINT iaIPAddress;
    UINT imIPMask;
    ULONG NTEContext;
```

```
    ULONG NTEInstance;

    /* variables used for GetIpStatistics */
    MIB_IPSTATS *pStats;

    /* variables used for GetTcpStatistics */
    MIB_TCPSTATS *pTCPStats;

    printf("------------------------\n");
    printf("This is GetNetworkParams\n");
    printf("------------------------\n");

    pFixedInfo = (FIXED_INFO *) MALLOC(sizeof (FIXED_INFO));
    if (pFixedInfo == NULL) {
        printf("Error allocating memory needed to call GetNetworkParams\n");
        return 1;
    }
    ulOutBufLen = sizeof (FIXED_INFO);

    if (GetNetworkParams(pFixedInfo, &ulOutBufLen) == ERROR_BUFFER_OVERFLOW) {
        FREE(pFixedInfo);
        pFixedInfo = (FIXED_INFO *) MALLOC(ulOutBufLen);
        if (pFixedInfo == NULL) {
            printf("Error allocating memory needed to call GetNetworkParams\n");
            return 1;
        }
    }

    if (dwRetVal = GetNetworkParams(pFixedInfo, &ulOutBufLen) != NO_ERROR) {
        printf("GetNetworkParams failed with error %d\n", dwRetVal);
        if (pFixedInfo)
            FREE(pFixedInfo);
        return 1;
    } else {
        printf("\tHost Name: %s\n", pFixedInfo->HostName);
        printf("\tDomain Name: %s\n", pFixedInfo->DomainName);
        printf("\tDNS Servers:\n");
        printf("\t\t%s\n", pFixedInfo->DnsServerList.IpAddress.String);

        pIPAddr = pFixedInfo->DnsServerList.Next;
        while (pIPAddr) {
            printf("\t\t%s\n", pIPAddr->IpAddress.String);
            pIPAddr = pIPAddr->Next;
        }

        printf("\tNode Type: ");
        switch (pFixedInfo->NodeType) {
        case 1:
            printf("%s\n", "Broadcast");
            break;
        case 2:
            printf("%s\n", "Peer to peer");
            break;
        case 4:
            printf("%s\n", "Mixed");
            break;
        case 8:
            printf("%s\n", "Hybrid");
            break;
        default:
            printf("\n");
        }

        printf("\tNetBIOS Scope ID: %s\n", pFixedInfo->ScopeId);

        if (pFixedInfo->EnableRouting)
            printf("\tIP Routing Enabled: Yes\n");
        else
            printf("\tIP Routing Enabled: No\n");
```

```
        printf("\tIP Routing Enabled: No\n");

    if (pFixedInfo->EnableProxy)
        printf("\tWINS Proxy Enabled: Yes\n");
    else
        printf("\tWINS Proxy Enabled: No\n");

    if (pFixedInfo->EnableDns)
        printf("\tNetBIOS Resolution Uses DNS: Yes\n");
    else
        printf("\tNetBIOS Resolution Uses DNS: No\n");
}

/* Free allocated memory no longer needed */
if (pFixedInfo) {
    FREE(pFixedInfo);
    pFixedInfo = NULL;
}

printf("------------------------\n");
printf("This is GetAdaptersInfo\n");
printf("------------------------\n");

pAdapterInfo = (IP_ADAPTER_INFO *) MALLOC(sizeof (IP_ADAPTER_INFO));
if (pAdapterInfo == NULL) {
    printf("Error allocating memory needed to call GetAdapterInfo\n");
    return 1;
}
ulOutBufLen = sizeof (IP_ADAPTER_INFO);

if (GetAdaptersInfo(pAdapterInfo, &ulOutBufLen) == ERROR_BUFFER_OVERFLOW) {
    FREE(pAdapterInfo);
    pAdapterInfo = (IP_ADAPTER_INFO *) MALLOC(ulOutBufLen);
    if (pAdapterInfo == NULL) {
        printf("Error allocating memory needed to call GetAdapterInfo\n");
        return 1;
    }
}

if ((dwRetVal = GetAdaptersInfo(pAdapterInfo, &ulOutBufLen)) != NO_ERROR) {
    printf("GetAdaptersInfo failed with error %d\n", dwRetVal);
    if (pAdapterInfo)
        FREE(pAdapterInfo);
    return 1;
}

pAdapter = pAdapterInfo;
while (pAdapter) {
    printf("\tAdapter Name: \t%s\n", pAdapter->AdapterName);
    printf("\tAdapter Desc: \t%s\n", pAdapter->Description);
    printf("\tAdapter Addr: \t");
    for (i = 0; i < (int) pAdapter->AddressLength; i++) {
        if (i == (pAdapter->AddressLength - 1))
            printf("%.2X\n", (int) pAdapter->Address[i]);
        else
            printf("%.2X-", (int) pAdapter->Address[i]);
    }
    printf("\tIP Address: \t%s\n",
           pAdapter->IpAddressList.IpAddress.String);
    printf("\tIP Mask: \t%s\n", pAdapter->IpAddressList.IpMask.String);

    printf("\tGateway: \t%s\n", pAdapter->GatewayList.IpAddress.String);
    printf("\t***\n");

    if (pAdapter->DhcpEnabled) {
        printf("\tDHCP Enabled: \tYes\n");
        printf("\tDHCP Server: \t%s\n",
               pAdapter->DhcpServer.IpAddress.String);

        printf("\tLease Obtained: ");
```

```
            printf("\tLease Obtained: ");
            /* Display local time */
            error = _localtime32_s(&newtime, (__time32_t*) &pAdapter->LeaseObtained);
            if (error)
                printf("\tInvalid Argument to _localtime32_s\n");

            else {
                // Convert to an ASCII representation
                error = asctime_s(buffer, 32, &newtime);
                if (error)
                    printf("Invalid Argument to asctime_s\n");
                else
                    /* asctime_s returns the string terminated by \n\0 */
                    printf("%s", buffer);
            }

            printf("\tLease Expires:  ");
            error = _localtime32_s(&newtime, (__time32_t*) &pAdapter->LeaseExpires);
            if (error)
                printf("Invalid Argument to _localtime32_s\n");
            else {
                // Convert to an ASCII representation
                error = asctime_s(buffer, 32, &newtime);
                if (error)
                    printf("Invalid Argument to asctime_s\n");
                else
                    /* asctime_s returns the string terminated by \n\0 */
                    printf("%s", buffer);
            }
        } else
            printf("\tDHCP Enabled: \tNo\n");

        if (pAdapter->HaveWins) {
            printf("\tHave Wins: \tYes\n");
            printf("\tPrimary Wins Server: \t%s\n",
                   pAdapter->PrimaryWinsServer.IpAddress.String);
            printf("\tSecondary Wins Server: \t%s\n",
                   pAdapter->SecondaryWinsServer.IpAddress.String);
        } else
            printf("\tHave Wins: \tNo\n");

        printf("\n");
        pAdapter = pAdapter->Next;
    }

    printf("------------------------\n");
    printf("This is GetInterfaceInfo\n");
    printf("------------------------\n");

    pInterfaceInfo = (IP_INTERFACE_INFO *) MALLOC(sizeof (IP_INTERFACE_INFO));
    if (pInterfaceInfo == NULL) {
        printf("Error allocating memory needed to call GetInterfaceInfo\n");
        return 1;
    }
    ulOutBufLen = sizeof (IP_INTERFACE_INFO);
    if (GetInterfaceInfo(pInterfaceInfo, &ulOutBufLen) ==
        ERROR_INSUFFICIENT_BUFFER) {
        FREE(pInterfaceInfo);
        pInterfaceInfo = (IP_INTERFACE_INFO *) MALLOC(ulOutBufLen);
        if (pInterfaceInfo == NULL) {
            printf("Error allocating memory needed to call GetInterfaceInfo\n");
            return 1;
        }
        printf("\t The size needed for the output buffer ulLen = %ld\n",
               ulOutBufLen);
    }

    if ((dwRetVal = GetInterfaceInfo(pInterfaceInfo, &ulOutBufLen)) == NO_ERROR) {
        printf("\tNum Adapters: %ld\n\n", pInterfaceInfo->NumAdapters);
        for (i = 0; i < (unsigned int) pInterfaceInfo->NumAdapters; i++) {
```

```
        for (i = 0; i < (unsigned int) pInterfaceInfo->NumAdapters; i++) {
            printf("\tAdapter Index[%d]: %ld\n", i,
                   pInterfaceInfo->Adapter[i].Index);
            printf("\tAdapter Name[%d]:  %ws\n\n", i,
                   pInterfaceInfo->Adapter[i].Name);
        }
        printf("GetInterfaceInfo call succeeded.\n");
    } else {
        LPVOID lpMsgBuf = NULL;

        if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
FORMAT_MESSAGE_IGNORE_INSERTS, NULL, dwRetVal, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),       // Default
language
                          (LPTSTR) & lpMsgBuf, 0, NULL)) {
            printf("\tError: %s", lpMsgBuf);
        }
        LocalFree(lpMsgBuf);
    }

    /* If DHCP enabled, release and renew the IP address */
    /* THIS WORKS BUT IT TAKES A LONG TIME AND INTERRUPTS NET CONNECTIONS */
    if (pAdapterInfo->DhcpEnabled && pInterfaceInfo->NumAdapters) {
        printf("Calling IpReleaseAddress for Adapter[%d]\n", 0);
        if ((dwRetVal =
             IpReleaseAddress(&pInterfaceInfo->Adapter[0])) == NO_ERROR) {
            printf("Ip Release succeeded.\n");
        }
        if ((dwRetVal =
             IpRenewAddress(&pInterfaceInfo->Adapter[0])) == NO_ERROR) {
            printf("Ip Renew succeeded.\n");
        }
    }

    /* Free allocated memory no longer needed */
    if (pAdapterInfo) {
        FREE(pAdapterInfo);
        pAdapterInfo = NULL;
    }
    if (pInterfaceInfo) {
        FREE(pInterfaceInfo);
        pInterfaceInfo = NULL;
    }

    printf("----------------------\n");
    printf("This is GetIpAddrTable\n");
    printf("----------------------\n");

    pIPAddrTable = (MIB_IPADDRTABLE *) MALLOC(sizeof (MIB_IPADDRTABLE));
    if (pIPAddrTable == NULL) {
        printf("Error allocating memory needed to call GetIpAddrTable\n");
        return 1;
    }
    dwSize = 0;
    IPAddr.S_un.S_addr = ntohl(pIPAddrTable->table[1].dwAddr);
    strIPAddr = inet_ntoa(IPAddr);

    if (GetIpAddrTable(pIPAddrTable, &dwSize, 0) == ERROR_INSUFFICIENT_BUFFER) {
        FREE(pIPAddrTable);
        pIPAddrTable = (MIB_IPADDRTABLE *) MALLOC(dwSize);
        if (pIPAddrTable == NULL) {
            printf("Error allocating memory needed to call GetIpAddrTable\n");
            return 1;
        }
    }

    if ((dwRetVal = GetIpAddrTable(pIPAddrTable, &dwSize, 0)) != NO_ERROR) {
        printf("GetIpAddrTable failed with error %d\n", dwRetVal);
        if (pIPAddrTable)
            FREE(pIPAddrTable);
```

```
        return 1;
    }

    printf("\tNum Entries: %ld\n", pIPAddrTable->dwNumEntries);
    for (i = 0; i < (unsigned int) pIPAddrTable->dwNumEntries; i++) {
        printf("\n\tInterface Index[%d]:\t%ld\n", i,
               pIPAddrTable->table[i].dwIndex);
        IPAddr.S_un.S_addr = (u_long) pIPAddrTable->table[i].dwAddr;
        printf("\tIP Address[%d]:     \t%s\n", i, inet_ntoa(IPAddr));
        IPAddr.S_un.S_addr = (u_long) pIPAddrTable->table[i].dwMask;
        printf("\tSubnet Mask[%d]:    \t%s\n", i, inet_ntoa(IPAddr));
        IPAddr.S_un.S_addr = (u_long) pIPAddrTable->table[i].dwBCastAddr;
        printf("\tBroadCast[%d]:      \t%s (%ld%)\n", i, inet_ntoa(IPAddr),
               pIPAddrTable->table[i].dwBCastAddr);
        printf("\tReassembly size[%d]:\t%ld\n", i,
               pIPAddrTable->table[i].dwReasmSize);
        printf("\tAddress Index[%d]:  \t%ld\n", i,
               pIPAddrTable->table[i].dwIndex);
        printf("\tType and State[%d]:", i);
        if (pIPAddrTable->table[i].wType & MIB_IPADDR_PRIMARY)
            printf("\tPrimary IP Address");
        if (pIPAddrTable->table[i].wType & MIB_IPADDR_DYNAMIC)
            printf("\tDynamic IP Address");
        if (pIPAddrTable->table[i].wType & MIB_IPADDR_DISCONNECTED)
            printf("\tAddress is on disconnected interface");
        if (pIPAddrTable->table[i].wType & MIB_IPADDR_DELETED)
            printf("\tAddress is being deleted");
        if (pIPAddrTable->table[i].wType & MIB_IPADDR_TRANSIENT)
            printf("\tTransient address");
        printf("\n");
    }

    iaIPAddress = inet_addr("192.168.0.27");
    imIPMask = inet_addr("255.255.255.0");

    NTEContext = 0;
    NTEInstance = 0;

    if ((dwRetVal = AddIPAddress(iaIPAddress,
                                 imIPMask,
                                 pIPAddrTable->table[0].
                                 dwIndex,
                                 &NTEContext, &NTEInstance)) != NO_ERROR) {

        LPVOID lpMsgBuf;
        printf("\tError adding IP address.\n");

        if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
FORMAT_MESSAGE_IGNORE_INSERTS, NULL, dwRetVal, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),       // Default
language
                          (LPTSTR) & lpMsgBuf, 0, NULL)) {
            printf("\tError: %s", lpMsgBuf);
        }
        LocalFree(lpMsgBuf);
    }

    if ((dwRetVal = DeleteIPAddress(NTEContext)) != NO_ERROR) {
        printf("DeleteIPAddress failed with error %d\n", dwRetVal);
    }

    /* Free allocated memory no longer needed */
    if (pIPAddrTable) {
        FREE(pIPAddrTable);
        pIPAddrTable = NULL;
    }

    printf("------------------------\n");
    printf("This is GetIPStatistics()\n");
    printf("------------------------\n");
```

```
    pStats = (MIB_IPSTATS *) MALLOC(sizeof (MIB_IPSTATS));
    if (pStats == NULL) {
        printf("Error allocating memory needed to call GetIpStatistics\n");
        return 1;
    }

    if ((dwRetVal = GetIpStatistics(pStats)) != NO_ERROR) {
        printf("GetIPStatistics failed with error %d\n", dwRetVal);
        if (pStats)
            FREE(pStats);
        return 1;
    }

    printf("\tNumber of IP addresses: %ld\n", pStats->dwNumAddr);
    printf("\tNumber of Interfaces: %ld\n", pStats->dwNumIf);
    printf("\tReceives: %ld\n", pStats->dwInReceives);
    printf("\tOut Requests: %ld\n", pStats->dwOutRequests);
    printf("\tRoutes: %ld\n", pStats->dwNumRoutes);
    printf("\tTimeout Time: %ld\n", pStats->dwReasmTimeout);
    printf("\tIn Delivers: %ld\n", pStats->dwInDelivers);
    printf("\tIn Discards: %ld\n", pStats->dwInDiscards);
    printf("\tTotal In: %ld\n", pStats->dwInDelivers + pStats->dwInDiscards);
    printf("\tIn Header Errors: %ld\n", pStats->dwInHdrErrors);

    /* Free allocated memory no longer needed */
    if (pStats) {
        FREE(pStats);
        pStats = NULL;
    }

    printf("-------------------------\n");
    printf("This is GetTCPStatistics()\n");
    printf("-------------------------\n");

    pTCPStats = (MIB_TCPSTATS *) MALLOC(sizeof (MIB_TCPSTATS));
    if (pTCPStats == NULL) {
        printf("Error allocating memory needed to call GetTcpStatistics\n");
        return 1;
    }

    if ((dwRetVal = GetTcpStatistics(pTCPStats)) != NO_ERROR) {
        printf("GetTcpStatistics failed with error %d\n", dwRetVal);
        if (pTCPStats)
            FREE(pTCPStats);
        return 1;
    }

    printf("\tActive Opens: %ld\n", pTCPStats->dwActiveOpens);
    printf("\tPassive Opens: %ld\n", pTCPStats->dwPassiveOpens);
    printf("\tSegments Recv: %ld\n", pTCPStats->dwInSegs);
    printf("\tSegments Xmit: %ld\n", pTCPStats->dwOutSegs);
    printf("\tTotal # Conxs: %ld\n", pTCPStats->dwNumConns);

    /* Free allocated memory no longer needed */
    if (pTCPStats) {
        FREE(pTCPStats);
        pTCPStats = NULL;
    }

    return 0;
}
```

IP Helper Reference

1/7/2020 • 2 minutes to read • Edit Online

Use the following functions and structures to retrieve and modify configuration settings for the Transmission Control Protocol/Internet Protocol (TCP/IP) transport on the local computer:

- IP Helper Functions
- IP Helper Structures
- IP Helper Enumerations

IP Helper Functions

1/7/2020 • 2 minutes to read • Edit Online

The following functions retrieve and modify configuration settings for the TCP/IP transport on the local computer. The following categorical listing can help determine which collection of functions is best suited for a given task:

Adapter Management

- **GetAdapterIndex**
- **GetAdaptersAddresses**
- **GetAdaptersInfo**
- **GetPerAdapterInfo**
- **GetUniDirectionalAdapterInfo**

Address Resolution Protocol (ARP) Management

- **CreateIpNetEntry**
- **CreateProxyArpEntry**
- **DeleteIpNetEntry**
- **DeleteProxyArpEntry**
- **FlushIpNetTable**
- **GetIpNetTable**
- **SendARP**
- **SetIpNetEntry**

Interface Conversion

- **ConvertInterfaceAliasToLuid**
- **ConvertInterfaceGuidToLuid**
- **ConvertInterfaceIndexToLuid**
- **ConvertInterfaceLuidToAlias**
- **ConvertInterfaceLuidToGuid**
- **ConvertInterfaceLuidToIndex**
- **ConvertInterfaceLuidToNameA**
- **ConvertInterfaceLuidToNameW**
- **ConvertInterfaceNameToLuidA**
- **ConvertInterfaceNameToLuidW**
- **if_indextoname**
- **if_nametoindex**

Interface Management

- **GetFriendlyIfIndex**
- **GetIfEntry**
- **GetIfEntry2**
- **GetIfEntry2Ex**

- GetIfStackTable
- GetIfTable
- GetIfTable2
- GetIfTable2Ex
- GetInterfaceInfo
- GetInvertedIfStackTable
- GetIpInterfaceEntry
- GetIpInterfaceTable
- GetNumberOfInterfaces
- InitializeIpInterfaceEntry
- SetIfEntry
- SetIpInterfaceEntry

Internet Protocol (IP) and Internet Control Message Protocol (ICMP)

- GetIcmpStatistics
- GetIpStatistics
- Icmp6CreateFile
- Icmp6ParseReplies
- Icmp6SendEcho2
- IcmpCloseHandle
- IcmpCreateFile
- IcmpParseReplies
- IcmpSendEcho
- IcmpSendEcho2
- IcmpSendEcho2Ex
- SetIpTTL

IP Address Management

- AddIPAddress
- CreateAnycastIpAddressEntry
- CreateUnicastIpAddressEntry
- DeleteIPAddress
- DeleteAnycastIpAddressEntry
- DeleteUnicastIpAddressEntry
- GetAnycastIpAddressEntry
- GetAnycastIpAddressTable
- GetIpAddrTable
- GetMulticastIpAddressEntry
- GetMulticastIpAddressTable
- GetUnicastIpAddressEntry
- GetUnicastIpAddressTable
- InitializeUnicastIpAddressEntry
- IpReleaseAddress
- IpRenewAddress
- NotifyStableUnicastIpAddressTable
- SetUnicastIpAddressEntry

IP Address String Conversion

- RtlIpv4AddressToString
- RtlIpv4AddressToStringEx
- RtlIpv4StringToAddress
- RtlIpv4StringToAddressEx
- RtlIpv6AddressToString
- RtlIpv6AddressToStringEx
- RtlIpv6StringToAddress
- RtlIpv6StringToAddressEx

IP Neighbor Address Management

- CreateIpNetEntry2
- DeleteIpNetEntry2
- FlushIpNetTable2
- GetIpNetEntry2
- GetIpNetTable2
- ResolveIpNetEntry2
- ResolveNeighbor
- SetIpNetEntry2

IP Path Management

- FlushIpPathTable
- GetIpPathEntry
- GetIpPathTable

IP Route Management

- CreateIpForwardEntry
- CreateIpForwardEntry2
- DeleteIpForwardEntry
- DeleteIpForwardEntry2
- EnableRouter
- GetBestInterface
- GetBestInterfaceEx
- GetBestRoute
- GetBestRoute2
- GetIpForwardEntry2
- GetIpForwardTable
- GetIpForwardTable2
- GetRTTAndHopCount
- InitializeIpForwardEntry
- SetIpForwardEntry
- SetIpForwardEntry2
- SetIpStatistics
- SetIpStatisticsEx
- UnenableRouter

IP Table Memory Management

- **FreeMibTable**

IP Utility

- **ConvertIpv4MaskToLength**
- **ConvertLengthToIpv4Mask**
- **CreateSortedAddressPairs**
- **ParseNetworkString**

Network Configuration

- **GetNetworkParams**

Notification

- **CancelMibChangeNotify2**
- **NotifyAddrChange**
- **NotifyIpInterfaceChange**
- **NotifyRouteChange**
- **NotifyRouteChange2**
- **NotifyUnicastIpAddressChange**

Persistent Port Reservarion

- **CreatePersistentTcpPortReservation**
- **CreatePersistentUdpPortReservation**
- **DeletePersistentTcpPortReservation**
- **DeletePersistentUdpPortReservation**
- **LookupPersistentTcpPortReservation**
- **LookupPersistentUdpPortReservation**

Security Health

- **CancelSecurityHealthChangeNotify**
- **NotifySecurityHealthChange**

These functions are defined only on Windows Server 2003.

> **NOTE**
>
> These functions are not available Windows Vista and Windows Server 2008.

Teredo IPv6 Client Management

- **GetTeredoPort**
- **NotifyTeredoPortChange**
- **NotifyStableUnicastIpAddressTable**

Transmission Control Protocol (TCP) and User Datagram Protocol (UDP)

- **GetExtendedTcpTable**
- **GetExtendedUdpTable**
- **GetOwnerModuleFromTcp6Entry**
- **GetOwnerModuleFromTcpEntry**
- **GetOwnerModuleFromUdp6Entry**
- **GetOwnerModuleFromUdpEntry**
- **GetPerTcp6ConnectionEStats**
- **GetPerTcpConnectionEStats**
- **GetTcpStatistics**
- **GetTcpStatisticsEx**
- **GetTcpStatisticsEx2**
- **GetTcp6Table**
- **GetTcp6Table2**
- **GetTcpTable**
- **GetTcpTable2**
- **SetPerTcp6ConnectionEStats**
- **SetPerTcpConnectionEStats**
- **SetTcpEntry**
- **GetUdp6Table**
- **GetUdpStatistics**
- **GetUdpStatisticsEx**
- **GetUdpStatisticsEx2**
- **GetUdpTable**

Deprecated APIs

> **NOTE**
> These functions are deprecated and not supported by Microsoft.

- **AllocateAndGetTcpExTableFromStack**
- **AllocateAndGetUdpExTableFromStack**

IP Helper Structures

1/7/2020 • 2 minutes to read • Edit Online

The following structures and unions are used with IP Helper:

- **ARP_SEND_REPLY**
- **FIXED_INFO**
- **ICMP_ECHO_REPLY**
- **ICMP_ECHO_REPLY32**
- **ICMPV6_ECHO_REPLY**
- **in_addr**
- **IP_ADAPTER_ADDRESSES**
- **IP_ADAPTER_ANYCAST_ADDRESS**
- **IP_ADAPTER_DNS_SERVER_ADDRESS**
- **IP_ADAPTER_DNS_SUFFIX**
- **IP_ADAPTER_GATEWAY_ADDRESS**
- **IP_ADAPTER_INDEX_MAP**
- **IP_ADAPTER_INFO**
- **IP_ADAPTER_MULTICAST_ADDRESS**
- **IP_ADAPTER_ORDER_MAP**
- **IP_ADAPTER_PREFIX**
- **IP_ADAPTER_UNICAST_ADDRESS**
- **IP_ADAPTER_WINS_SERVER_ADDRESS**
- **IP_ADDR_STRING**
- **IP_ADDRESS_PREFIX**
- **IP_ADDRESS_STRING**
- **IP_INTERFACE_INFO**
- **IP_INTERFACE_NAME_INFO**
- **IP_MCAST_COUNTER_INFO**
- **IP_OPTION_INFORMATION**
- **IP_OPTION_INFORMATION32**
- **IP_PER_ADAPTER_INFO**
- **IP_UNIDIRECTIONAL_ADAPTER_ADDRESS**
- **IPV6_ADDRESS_EX**
- **NET_ADDRESS_INFO**
- **NET_LUID**
- **SOCKADDR_IN6_PAIR**
- **SOCKADDR_INET**
- **TCP_ESTATS_BANDWIDTH_ROD_v0**
- **TCP_ESTATS_BANDWIDTH_RW_v0**
- **TCP_ESTATS_DATA_ROD_v0**
- **TCP_ESTATS_DATA_RW_v0**
- **TCP_ESTATS_FINE_RTT_ROD_v0**
- **TCP_ESTATS_FINE_RTT_RW_v0**

- TCP_ESTATS_OBS_REC_ROD_v0
- TCP_ESTATS_OBS_REC_RW_v0
- TCP_ESTATS_PATH_ROD_v0
- TCP_ESTATS_PATH_RW_v0
- TCP_ESTATS_REC_ROD_v0
- TCP_ESTATS_REC_RW_v0
- TCP_ESTATS_SEND_BUFF_ROD_v0
- TCP_ESTATS_SEND_BUFF_RW_v0
- TCP_ESTATS_SND_CONG_ROD_v0
- TCP_ESTATS_SND_CONG_ROS_v0
- TCP_ESTATS_SND_CONG_RW_v0
- TCP_ESTATS_SYN_OPTS_ROS_v0
- TCPIP_OWNER_MODULE_BASIC_INFO
- TCP_RESERVE_PORT_RANGE

IP Helper Enumerations

1/7/2020 • 2 minutes to read • Edit Online

The following enumerations are used with IP Helper:

- **IF_OPER_STATUS**
- **IP_DAD_STATE**
- **IP_PREFIX_ORIGIN**
- **IP_SUFFIX_ORIGIN**
- **NET_ADDRESS_FORMAT**
- **SCOPE_LEVEL**
- **TCP_BOOLEAN_OPTIONAL**
- **TCP_ESTATS_TYPE**
- **TCP_SOFT_ERROR**
- **TCP_TABLE_CLASS**
- **TCPIP_OWNER_MODULE_INFO_CLASS**
- **UDP_TABLE_CLASS**