# Winsock2 and Internet Protocol 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| IPv4 is commonly known as the **network protocol** that the Internet uses. IP is widely available on most computer operating systems and can be used on most LANs, such as a small network in your office, and on WANs, such as the Internet. With the explosion in the number of computers on the Internet, the limitations of IPv4 are becoming apparent, and as a result, the next generation IP was developed, which is known as IPv6.  In this chapter, we will discuss the background, addressing scheme, name resolution, and Winsock specifics for both IPv4 and IPv6. Then, we'll discuss how to write applications that seamlessly operate over either version of IP.   IPv4   IPv4 was developed by the U.S. Department of Defense's Advanced Research Project Agency (ARPA), which built an experimental packet switching network in the 1960s. The initial network protocols were cumbersome, which led to the development of a better protocol in the mid 1970s. This research eventually led to IPv4 as well as TCP.   Addressing   In IPv4, computers are assigned an address that is represented as a 32-bit number, formally known as an IPv4 address. IPv4 addresses are typically represented in a dotted decimal format in which each octet (8 bits) of the address is converted to a decimal number and separated by a period (“dots”).  IPv4 addresses are divided into classes that describe the portion of the address assigned to the network and the portion assigned to endpoints. Table 3-1 lists the different classes.   |  |  |  |  | | --- | --- | --- | --- | | **Class** | **Network Portion** | **First Number** | **Number of Endpoints** | | A | 8 bits | 0–127 | 16,777,216 | | B | 16 bits | 128–191 | 65,536 | | C | 24 bits | 192–223 | 256 | | D | N/A | 224–239 | N/A | | E | N/A | 240–255 | N/A | |

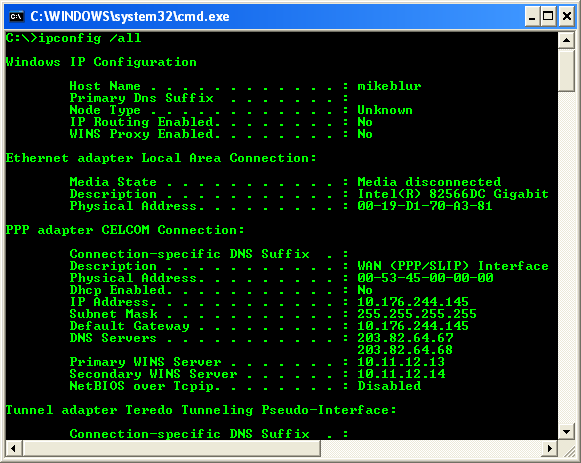
In a slash notation, when specifying an IP address, the number of bits indicating the network portion can be appended to the dotted decimal address after a back slash (/). For example, the address 172.31.28.120/16 indicates that the first 16 bits make up the network portion of the address. This is equivalent to a subnet mask of 255.255.0.0.

The last two entries in Table 3-1 are special classes of IPv4 addresses. Class D addresses are reserved for IPv4 multicasting and class E addresses are experimental. Also, there are several blocks of addresses that have been reserved for private use and cannot be used by a system on the Internet. They are the following:

1. 10.0.0.0–10.255.255.255 (10.0.0.0/8)
2. 172.16.0.0–172.31.255.255 (172.16.0.0/12)
3. 192.168.0.0–192.168.255.255 (192.168.0.0/16)

Finally, there is the loopback address (127.0.0.1), which is a special address that refers to the local computer.

To list the IPv4 addresses assigned to the local interfaces, the IPCONFIG.EXE command can be used to list each network adapter and the IPv4 address(es) assigned to it.



If an application needs to programmatically obtain a list of its IPv4 addresses, it can call WSAIoctl() with the SIO\_ADDRESS\_LIST\_QUERY command. In addition, the IP Helper APIs also provide this function. We've discussed the breakdown of the IPv4 address space, and from within these different address classes there are three types of IPv4 addresses: **unicast**, **multicast**, and **broadcast**. Each address type will be covered in the next sections.

# Unicast

Unicast addresses are those addresses that are assigned to an individual computer interface. Only one interface may be assigned that address. If another computer is configured with the same address on the network, then that is an error that will result in data being delivered incorrectly. Classes A, B, and C comprise the unicast address space for IPv4.

Typically, an interface on a host is assigned an IPv4 (unicast) address either statically or by a configuration protocol like Dynamic Host Configuration Protocol (DHCP). If a DHCP server cannot be reached, the system automatically assigns an address in the range of 169.254.0.0/16 using Automatic Private IP Addressing (APIPA).

To prevent having to memorize numeric IP addresses, an IPv4 address can be associated to the host computer name by using the Domain Name System (DNS). Later, we will discuss how to resolve the host name to its IPv4 address (and its IPv6 address as well).

# Multicast

Multicast addresses are not assigned to a specific interface. Instead, multiple computers may “join” a multicast group listening on a particular multicast address. Everyone joined to that group will receive any data destined to that address. Multicast addresses are class D addresses. One of the greatest benefits to multicasting is the capability to deliver multicast data to only those machines that are interested in that data.

# Broadcast

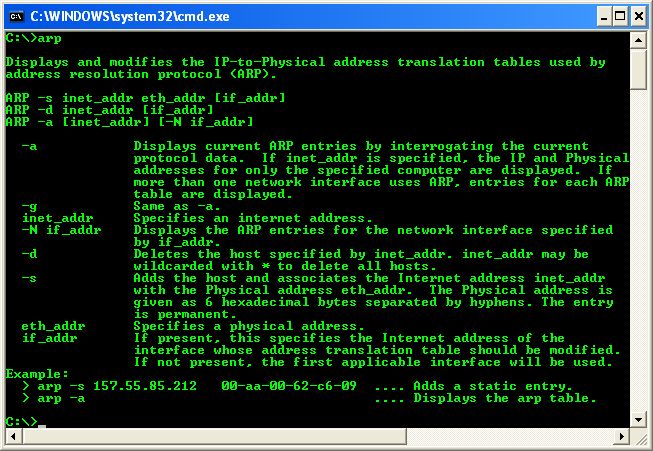
IPv4 supports broadcasting data. This means that data sent to the limited broadcast address, 255.255.255.255, will be received and processed by every machine on the local network. This is generally considered a bad practice because even those computers that are not interested in the broadcast data must process the packet.

If applications require broadcasting, it is better to use subnet directed broadcasts. This is still broadcasting data, but as the name implies it is directed to machines on a specific subnet only. For example, a datagram sent to 172.31.28.255 will be received by every machine on only that same subnet.

**IPv4 Management Protocols**

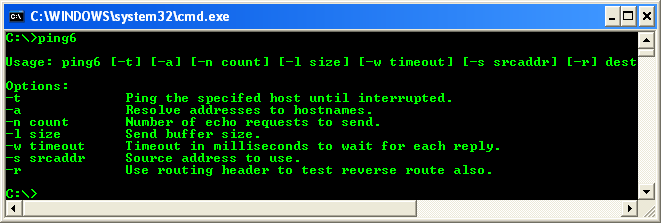
 The IPv4 protocol relies on several other protocols to function. The three support protocols we are most interested in is the **Address Resolution Protocol** (ARP), the **Internet Control Message Protocol** (ICMP), and the **Internet Group Management Protocol** (IGMP).

ARP is used to resolve the 32-bit IPv4 address into a physical or hardware address so the IPv4 packet can be wrapped in the appropriate media frame (such as an Ethernet frame). In Windows you can invoke the ARP tool using arp command.



 A host must resolve the next-hop IPv4 address to its corresponding hardware address before sending data on the wire. If the destination address is on the local network, the ARP request is made for the destination's physical address. If one or more routers separate the source from the destination, an ARP request is made for the default gateway and the packet is forwarded to it. The IP Helper API contains some ARP routines that we can use.

ICMP is designed to send status and error messages between IPv4 hosts. The types of messages include echo requests and replies, destination unreachable, and time exceeded. ICMP is also used to discover nearby routers. We will go into more detail on ICMP and will illustrate how to send your own ICMP messages in another chapter. As an example, the ping command is based on the ICMP protocol.



 IGMP is used to manage multicast group membership. When applications on a host join multicast group, the host sends out IGMP membership reports, which inform routers on the network segment which multicast groups data is to be received on. Routers need this information to forward multicast packets destined to these multicast groups to network segments only when there are receivers interested in it.

**Addressing IPv4 from Winsock**

In Winsock, applications specify IPv4 addresses and service port information through the SOCKADDR\_IN structure, which is defined as:

 struct sockaddr\_in

{

    short           sin\_family;

    u\_short         sin\_port;

    struct in\_addr  sin\_addr;

    char            sin\_zero[8];

};

 The sin\_family field must be set to AF\_INET, which tells Winsock you are using the IP address family. The sin\_port field defines which TCP or UDP communication port will be used to identify a server service. Note that the port number does not actually apply to the IPv4 protocol but is a property of the transport layer protocol(s) encapsulated within an IPv4 header, such as TCP or UDP.

Applications should be particularly careful in choosing a port because some of the available port numbers are reserved for well-known services, such as FTP and HTTP. The ports that well-known services use are controlled and assigned by the Internet Assigned Numbers Authority (IANA) and are listed on its Web page at [IANA port number assignment](http://www.iana.org/assignments/port-numbers). Essentially, the port numbers are divided into the following three ranges: well-known, registered, and dynamic and/or private ports.

1. 0–1023 are controlled by IANA and are reserved for well-known services.
2. 1024–49151 are registered ports listed by IANA and can be used by ordinary user processes or programs executed by ordinary users.
3. 49152–65535 are dynamic and/or private ports.

Ordinary user applications should choose the registered ports in the range 1024–49151 to avoid the possibility of using a port already in use by another application or a system service. Ports in the range 49152–65535 can also be used freely because no services are registered on these ports with IANA. If, when using the bind() API function, your application binds to a port that is already in use by another application on your host, the system will return the Winsock error WSAEADDRINUSE. Also, it is valid for clients to send or connect without explicitly binding to a local address and port. In this case, the system will implicitly bind the socket to a local port from the range of 1024 to 5000. This is the same behavior that occurs if an application explicitly binds the socket but specifies a local port of zero.

The sin\_addr field of the SOCKADDR\_IN structure is used for storing an IPv4 address as a four-byte, network-byte-ordered quantity, which is an unsigned long integer data type. Depending on how this field is used, **it can represent a local or a remote IP address**. IP addresses are normally specified in Internet standard dotted notation as “a.b.c.d.” Each letter represents a number for each byte and is assigned, from left to right, to the four bytes of the unsigned long integer. The final field, sin\_zero, functions only as padding to make the SOCKADDR\_IN structure the same size as the SOCKADDR structure.

All fields of this and every other socket address structure need to be in network byte order. However, if applications use the name resolution and assignment APIs discussed later in this chapter, the necessary conversion is automatically performed. It is only when an application explicitly assigns or retrieves values from the structure members that the byte order conversion is required.

**IPv6**

 With the explosion in the number of computers on the Internet, the limitations of IPv4 are becoming apparent. First and foremost, the number of available IPv4 addresses is being exhausted. This has led to the use of network address translators (NATs), which map multiple private addresses to a single public IP addresses. NATs are useful for client-server applications but can be problematic when connecting two organizations that use the private address space. Also, NATs must sometimes be aware of the underlying protocols to perform the appropriate address translation.

Second, IPv4 addressing is not entirely hierarchical, which means that the Internet backbone routers must maintain vast routing tables to deliver IPv4 packets correctly to any location on the Internet.

Another incentive for developing IPv6 is to provide simpler configuration. With IPv4, addresses must be assigned statically or via a configuration protocol such as DHCP. Ideally, hosts would not have to rely upon the administration of a DHCP infrastructure. Instead, they will be able to auto configure themselves based on the network segment on which they are located.

A developer-release version of IPv6 is provided with Windows XP. For Windows 2000, a technology preview IPv6 protocol is available for download from [Microsoft IPv6 info](http://technet.microsoft.com/en-us/network/bb530961.aspx). For Windows NT 4.0, a Microsoft Research IPv6 protocol may also be obtained from [IPv6 protocol](http://technet.microsoft.com/en-us/network/bb530961.aspx).

In this section, we will cover the different types of IPv6 addresses, the support protocols that IPv6 uses, and how IPv6 addresses are handled from Winsock. Although we will discuss addressing and name resolution, we will not cover all aspects of IPv6, such as routing or setting up an IPv6 network. More info on this thing can be found at [Internet protocol version 6 info](http://www.tenouk.com/internetprotocolversion6ipv6.html).

**Addressing**

 The most noticeable difference between IPv4 and IPv6 addresses is that an IPv6 address is 128 bits, which is four times larger than an IPv4 address. One reason for such a large address space is to subdivide the available addresses into a hierarchy of routing domains that reflect the Internet's topology. Table 3-2 lists a portion of how the address space is allocated and lists the address prefix for each portion. The address prefix denotes the high order bits of an IPv6 address. IPv6 addressing is described in RFC 2373.

|  |  |  |
| --- | --- | --- |
| **Allocation** | **Address Prefix** | **Fraction of Address Space** |
| Reserved | 0000 0000 | 1/256 |
| Reserved for NSAP allocation | 0000 001 | 1/128 |
| Aggregatable global unicast addresses | 001 | 1/8 |
| Link-local unicast addresses | 1111 1110 10 | 1/1024 |
| Site-local unicast addresses | 1111 1110 11 | 1/1024 |
| Multicast addresses | 1111 1111 | 1/256 |

An IPv6 address is typically expressed in 16-bit chunks displayed as hexadecimal numbers separated by colons. The following is an example of an IPv6 address:

21DA:00D3:0000:2F3B:02AA:00FF:FE28:9C5A

 Leading zeroes within each 16-bit block may be removed, as seen here:

21DA:D3:0:2F3B:2AA:FF:FE28:9C5A

 Many IPv6 addresses contain long sequences of zeroes, which may be compressed by substituting two colons for the block of zeros. For example, the following address:

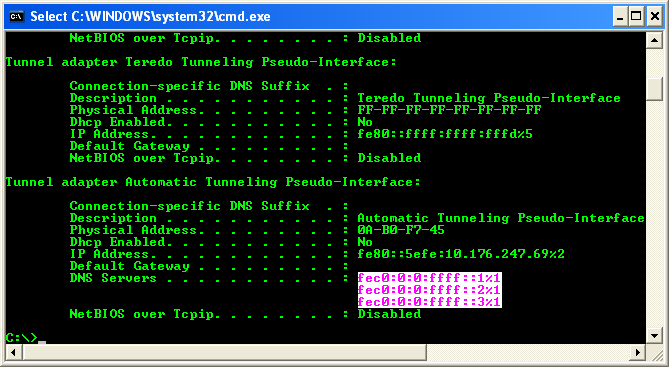
FE80:0:0:0:12:0:34:56

 can be compressed to:

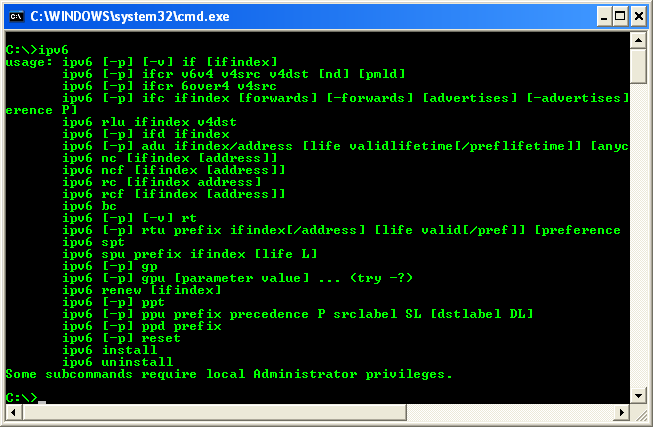
FE80::12:0:34:56

 Note that only a single contiguous sequence of 16-bit zero blocks may be compressed.

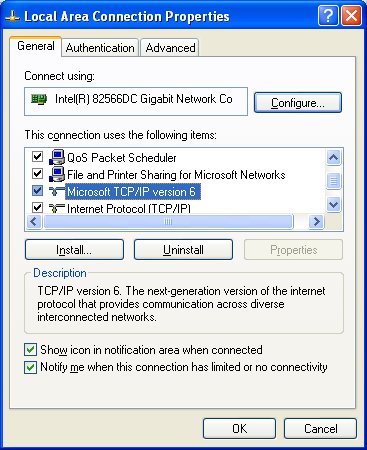
If you computer is IPv6 enabled, you can view the information using the IPCONFIG command.



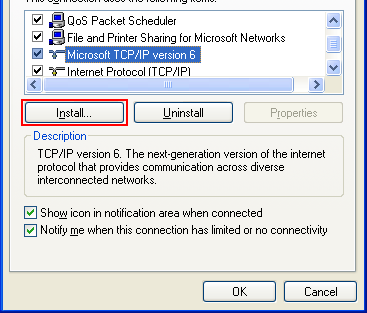
 Depending on the platform, you can use other two methods to obtain a list of the IPv6 addresses assigned to a computer's interfaces. For the Microsoft Research and Windows 2000 Technology Preview stacks downloaded from the Web as well as Windows XP Home Edition and Windows XP Professional, the IPV6 command is used. To enumerate the IPv6 interfaces, execute IPV6 if at the command prompt.



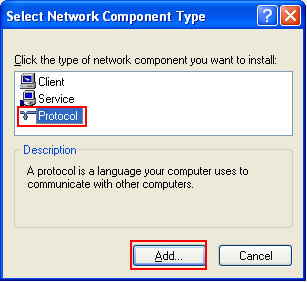
If there is no ipv6 command at your command prompt, you need to check whether the protocol has been installed or not. This can be accessed through the Local Area Connection property page.



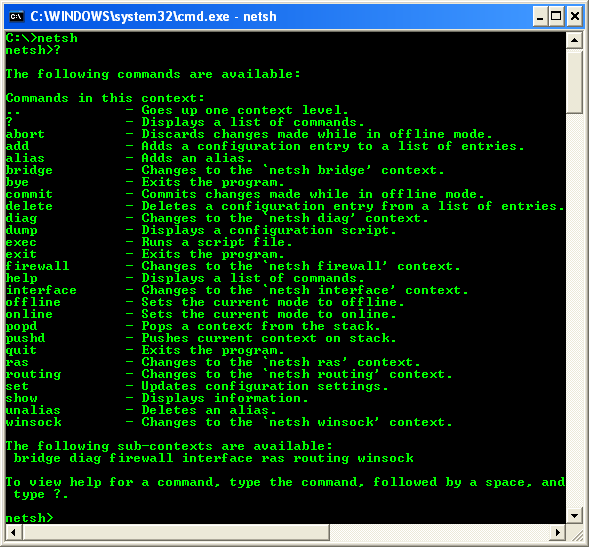
If there is no Microsoft TCP/IP version 6 item, then you can install it using the Install button.



 Select Protocol as the type of network component to install. Then click Add button and select Microsoft TCP/IP version 6. If you cannot find the item, then you need to install the Windows networking component.



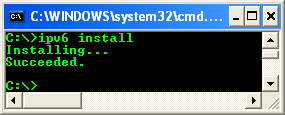
For all versions of Windows 2000 and Windows XP (including the latest versions of Windows releases), the NETSH command may also be used.



 The command syntax is: NETSH interface IPv6 show interface.



 You can also use the ipv6 tool to install/uninstall the ipv6 protocol.



 To programmatically obtain the configuration of local interfaces, the SIO\_ADDRESS\_LIST\_QUERY ioctl and the IP Helper API can be used. There are three basic types of IPv6 addresses: **unicast**, **anycast**, and **multicast**. Note that IPv6 does not define a **broadcast** address (multicasting is used instead). In the following sections, we will discuss each address type.

**Unicast**

A unicast address identifies a single interface. With IPv6, however, an interface will most likely have more than one unicast address assigned to it. There are four types of unicast addresses that you will likely encounter:

1. Link-local addresses.
2. Site-local addresses.
3. Global addresses.
4. Compatibility addresses.

An interface will always have a link-local address assigned to it, each physical network interface is auto configured with one. A link-local address is used to communicate only with other nodes on the same link. Link-local address always begins with an **fe80::/64** prefix. Also, because no routing information is kept for link-local addresses, the interface index is often displayed with the address. Every physical interface on the system is assigned an **adapter index number** (also known as a **scope ID**). When a link-local address is assigned to an interface, the link number is appended to the address. The following address is the link-local address assigned to the physical adapter whose interface index number is five.

fe80::250:8bff:fea0:92ed%5

 In Winsock, if a connection is being established using link-local addresses, then the interface index must be present to indicate which link the remote host is reachable from. An IPv6 link-local address is synonymous with an IPv4 APIPA address discussed earlier in the chapter.

For example, consider host A, which has the link-local address fe80::250:8bff:fea0:92ed%5 and host B, which has the link-local address fe80::250:daff:fec3:9e34%4. If host A issues a connect to host B, it would use the destination address of B with its own scope ID that can reach host B. The address to connect to would be fe80::250:daff:fec3:9e34%5.

Site-local addresses are IPv6 addresses that are reachable only on the local network environment, such as the corporate network at a particular site. These addresses are comparable to the IPv4 private address space because they cannot be reached from other sites or the Internet and routers on the private network do not forward this traffic beyond the local site. Site-local addresses use the prefix **fec0::/48**. Site-local addresses must be assigned from either an IPv6 router or via [DHCPv6](http://msdn.microsoft.com/en-us/library/cc202589.aspx). IPv6-enabled routers will send **Router Advertisement** (**RA**) messages, which advertise the network portion of the address (such as the first 64 bits of the address consisting of the 48-bit site-local prefix and a 16-bit subnet ID), which the host will then use to assign a site-local address to the interface on which the RA was received.

Global addresses are just that: globally reachable on IPv6 Internet. Global addresses begin with 001. The remaining 61 bits of the first 64 bits are used to establish a routing hierarchy, and the last 64 bits comprise the interface identifier that uniquely identifies a network interface on a subnet. Global addresses are also assigned via router advertisements or by using DHCPv6.

The last type of unicast addresses are compatibility addresses, which are designed to aid in the transition from IPv4 to IPv6. There are four kinds of compatibility addresses that Windows supports:

1. Intrasite Automatic Tunnel Addressing Protocol (ISATAP)
2. 6to4
3. 6over4 and
4. IPv4 compatible

ISATAP addresses can be derived from any IPv6 unicast address, such as link-local, site-local, and global addresses. Most often you will see an ISATAP address derived from a link-local address. These addresses also contain an embedded IPv4 address. For example, the ISATAP address fe80::5efe:172.17.7.2 is a link-local address and contains the IPv4 address of the host (172.17.7.2). When data is sent from this interface, the IPv6 packet is encapsulated within an IPv4 header. The IPv4 destination address is obtained from the v4 address embedded within the IPv6 ISATAP destination address. The v4 address must be globally reachable for two endpoints to communicate via automatic tunneling. ISATAP addresses are currently an Internet Engineering Task Force (IETF) draft which is [RFC 5214](http://www.rfc-archive.org/getrfc.php?rfc=5214).

The second type of compatibility address is called 6to4 and is described in [RFC 3056](http://www.ietf.org/rfc/rfc3056.txt). 6to4 addresses use the global prefix 2002:WWXX:YYZZ::/48, in which WWXX:YYZZ is the hexadecimal-colon representation of w.x.y.z, a public IPv4 address. 6to4 allows IPv6/IPv4 hosts to communicate over an IPv4 routing infrastructure.

Windows XP provides a 6to4 service. This service allows hosts to communicate with other 6to4 hosts within the same site, 6to4 hosts connected to the Internet, 6to4 hosts in other sites across the IPv4 Internet, as well as with hosts on the IPv6 Internet using a 6to4 relay router. On Windows XP, the 6to4 service is configured to run automatically. If there is a public IPv4 address assigned to an interface, a 6to4 Tunneling Interface (interface index 3) is created and assigned the 6to4 address(es).

The third type of compatibility address is 6over4, which is a tunneling technique using IPv4 multicasting. It allows IPv4 and IPv6 nodes to communicate using IPv6 over an IPv4 infrastructure. This technique is described in [RFC 2529](http://www.rfc-archive.org/getrfc.php?rfc=2529).

The last type of compatibility address is the IPv4 compatible address. These addresses take the form of 0:0:0:0:0:0:w.x.y.z (or ::w.x.y.z) in which w.x.y.z is the dotted decimal representation of a public IPv4 address. When an IPv4 compatible address is used by an application as the destination, the IPv6 traffic is automatically encapsulated within an IPv4 header and sent to the destination over the IPv4 network.

# Anycast

Anycast is an address that identifies multiple interfaces. The purpose of these addresses is to route packets destined to an anycast address to the nearest interface assigned that anycast address. A good scenario for anycast addresses is when there are several nodes on the network that provide a certain service. Each machine can be assigned the same anycast address and clients interested in contacting that service will be routed to the nearest member. This is different from multicast because this communication is one to one of many instead of one to many. Currently however, anycast addresses are assigned to routers only.

**Multicast**

 Multicasting in IPv6 is similar to IPv4 multicasting. A process joins a multicast group on a particular interface and data destined to that multicast address is received. IPv6 multicast addresses begin with 1111 1111 (FF).

 IPv6 Management Protocols

 IPv6 requires only a single helper protocol: Internet Control Message Protocol for IPv6 (ICMPv6), which is defined in [RFC 2463](http://www.rfc-archive.org/getrfc.php?rfc=2463). ICMPv6 provides the same types of services that ICMP does, such as destination unreachable, echo and echo reply, but also provides a mechanism for Multicast Listener Discovery (MLD) and Neighbor Discovery (ND). **MLD replaces IGMP and ND replaces ARP**.

**Addressing IPv6 from Winsock**

 To specify IPv6 addresses in Winsock applications, the following structure is used.

 struct sockaddr\_in6 {

                        short                sin6\_family;

                        u\_short                       sin6\_port;

                        u\_long             sin6\_flowinfo;

                        struct in6\_addr sin6\_addr;

                        u\_long             sin6\_scope\_id;

};

 The first field simply identifies the address family, which is AF\_INET6, and the second is the port number. All fields within this structure must be in network byte order. Note that all the information discussed about port numbers in the IPv4 section apply equally to IPv6 because the port number is a property of the encapsulated protocols, such as TCP and UDP, which are also available from IPv6. The third field, sin6\_flowinfo, is used to mark the traffic for the connection but is not implemented in the Microsoft IPv6 stack. The fourth field is a 16-byte structure that contains the binary IPv6 address. The last member, sin6\_scope\_id, indicates the interface index (or scope ID) on which the address is located. Remember that for link-local addresses, the local scope ID on which the destination is located must be specified and the sin6\_scope\_id field is used for this. Site-local addresses may reference the site number as the scope ID. Global addresses do not contain a scope ID. One last item to note is that the SOCKADDR\_IN6 structure is 28 bytes in length and the SOCKADDR and SOCKADDR\_IN structures are only 16 bytes long.

**Address and Name Resolution**

In this section, we'll cover how to assign both literal string addresses and resolve names to the address specific structures for both IP protocols. First, we will cover the new name resolution APIs: getaddrinfo() and getnameinfo(). These APIs have replaced the IPv4 specific routines. Then we'll cover the generic Winsock APIs for converting between string literal addresses and socket address structure. These APIs are WSAAddressToString() and WSAStringToAddress(). Note that these functions perform only address conversion and assignment, not name resolution.

Next, the IPv4 specific legacy routines will be described. We include the legacy API descriptions in case legacy code needs to be maintained, but any new projects should use the newer API functions. By using the newer functions it will be trivial to write an application that can seamlessly operate over both IPv4 and IPv6, which is the topic of the last section in this chapter.

Finally, note that all the name resolution functions covered in this chapter deal only with resolving names and not registering a name with an address. This is accomplished by the Winsock Registration and Name Resolution (RNR) APIs, discussed in other chapter.

|  |
| --- |
| Name Resolution Routines    Along with IPv6, several new name resolution functions were introduced that could handle both IPv4 and IPv6 addresses. The legacy functions like gethostbyname() and inet\_addr() work with IPv4 addresses only. The replacement functions are named getnameinfo() and getaddrinfo().  These new name resolution routines are defined in WS2TCPIP.H. Also, note that although these functions are new for Windows XP, they can be made available to work on all Winsock 2 enabled platforms. This is done by including the header file WSPIAPI.H before including WS2TCPIP.H. The compiled binary will then run on all Winsock 2 enabled platforms, such as Windows 95, Windows 98, Windows Me, Windows NT 4.0, and Windows 2000. The getaddrinfo() function provides protocol-independent name resolution. The function prototype is:    int getaddrinfo(                          const char FAR \*nodename,                          const char FAR \*servname,                          const struct addrinfo FAR \*hints,                          struct addrinfo FAR \*FAR \*res  );    The nodename parameter specifies the NULL-terminated host name or literal address. The servname is a NULL-terminated string containing the port number or a service name such as “ftp” or “telnet.” The third parameter, hints, is a structure that can pass one or more options that affect how the name resolution is performed. Finally, the res parameter returns a linked list of addrinfo structure containing the addresses the string name was resolved to. If the operation succeeds, zero is returned; otherwise the Winsock error code is returned. The addrinfo structure is defined as:    struct addrinfo {                          int                    ai\_flags;                          int                    ai\_family;                          int                    ai\_socktype;                          int                    ai\_protocol;                          size\_t              ai\_addrlen;                          char                 \*ai\_canonname;                          struct sockaddr \*ai\_addr;                          struct addrinfo \*ai\_next;  }; |

When passing hints into the API, the structure should be zeroed out beforehand, and the first four fields are relevant:

1. ai\_flags indicates one of three values: AI\_PASSIVE, AI\_CANONNAME, or AI\_NUMERICHOST. AI\_CANONNAME indicates that nodename is a computer name like www.microsoft.com and AI\_NUMERICHOST indicates that it is a literal string address such as “10.10.10.1”. AI\_PASSIVE will be discussed later.
2. ai\_family can indicate AF\_INET, AF\_INET6, or AF\_UNSPEC. If you wish to resolve to a specific address, type the supply AF\_INET or AF\_INET6. Otherwise, if AF\_UNSPEC is given, then the addresses returned could be either IPv4 or IPv6 or both.
3. ai\_socktype specifies the desired socket type, such as SOCK\_DGRAM, SOCK\_STREAM. This field is used when servname contains the name of a service. That is, some services have different port numbers depending on whether UDP or TCP is used.
4. ai\_protocol specifies the desired protocol, such as IPPROTO\_TCP. Again, this field is useful when servname indicates a service.

If no hints are passed into getaddrinfo(), the function behaves as if a zeroed hints structure was provided with an ai\_family of AF\_UNSPEC.

If the function succeeds, then the resolved addresses are returned via res. If the name resolved to more than one address, then the result is a linked list chained by the ai\_next field. Every address resolved from the name is indicated in ai\_addr with the length of that socket address structure given in ai\_addrlen. These two fields may be passed directly into bind(), connect(), sendto(), etc. The following code snippet example shows the use of the addrinfo structure.

// Declare and initialize variables.

char\* ip = "127.0.0.1";

char\* port = "7777";

struct addrinfo aiHints;

struct addrinfo \*aiList = NULL;

int retVal;

// Setup the hints address info structure

// which is passed to the getaddrinfo() function

memset(&aiHints, 0, sizeof(aiHints));

aiHints.ai\_family = AF\_INET;

aiHints.ai\_socktype = SOCK\_STREAM;

aiHints.ai\_protocol = IPPROTO\_TCP;

// Call getaddrinfo(). If the call succeeds, the aiList variable

// will hold a linked list f addrinfo structures containing

// response information about the host

if ((retVal = getaddrinfo(ip, port, &aiHints, &aiList)) != 0)

{

  printf("getaddrinfo() failed with error code %d.\n", retVal);

}

The following code sample illustrates how to resolve a hostname along with the port number before making a TCP connection to the server.

SOCKET                                s;

struct addrinfo                        hints, \*result;

int                                            rc;

memset(&hints, 0, sizeof(hints));

hints.ai\_flags = AI\_CANONNAME;

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

rc = getaddrinfo("bodopiang", "5001", &hints, &result);

if (rc != 0)

{

            // unable to resolve the name

}

s = socket(result->ai\_family, result->ai\_socktype, result->ai\_protocol);

if (s == INVALID\_SOCKET)

{

            // socket API failed

}

rc = connect(s, result->ai\_addr, result->ai\_addrlen);

if (rc == SOCKET\_ERROR)

{

            // connect API failed

}

freeaddrinfo(result);

In this example, the application is resolving the hostname “bodopiang” and wants to establish a TCP connection to a service on port 5001. You'll also notice that this code doesn't care if the name resolved to an IPv4 or an IPv6 address. It is possible that “bodopiang” has both IPv4 and IPv6 addresses registered, in which case result will contain additional addrinfo structures linked by the ai\_next field. If an application wanted only IPv4 addresses registered to “bodopiang,” the hints.ai\_family should be set to AF\_INET. Finally, note that the information returned via res is dynamically allocated and needs to be freed by calling the freeaddrinfo() API once the application is finished using the information.

Another common action that applications perform is assigning a literal string address such as “172.17.7.1” or “fe80::1234” into a socket address structure of the appropriate type. The getaddrinfo() function does this by setting the AI\_NUMERICHOST flag within the hints. The following code illustrates this.

struct addrinfo                        hints, \*result;

int                                            rc;

memset(&hints, 0, sizeof(hints));

hints.ai\_flags = AI\_NUMERICHOST;

hints.ai\_family = AI\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

rc = getaddrinfo("172.17.7.1", "5001", &hints, &result);

if (rc != 0)

{

            // invalid literal address

}

// Use the result

freeaddrinfo(result);

The following program example tries to demonstrate the working version. Create a new empty Win32 console mode application and add the project/solution name.

#include <winsock2.h>

// For addrinfo structure

#include <ws2tcpip.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

            WSADATA wsaData;

            SOCKET                    s;

            int                                rc, iResult, RetCode;

            struct addrinfo \*result = NULL;

            struct addrinfo hints;

            // Initialize Winsock

            iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

            if (iResult != 0)

            {

                printf("Client: WSAStartup() failed with error code %d\n", iResult);

                return 1;

            }

            else

                printf("Client: WSAStartup() is OK...\n");

            // Setup the hints address info structure

            // which is passed to the getaddrinfo() function

            ZeroMemory( &hints, sizeof(hints) );

            hints.ai\_flags = AI\_CANONNAME;

            hints.ai\_family = AF\_INET;

            hints.ai\_socktype = SOCK\_STREAM;

            hints.ai\_protocol = IPPROTO\_TCP;

            // rc = getaddrinfo("206.190.60.37", "7171", &hints, &result);

            rc = getaddrinfo("www.ipv6tf.org", "80", &hints, &result);

            if (rc != 0)

            {

                        printf("Client: getaddrinfo() failed with error code %ld\n", WSAGetLastError());

                        WSACleanup();

                        return 1;

            }

            printf("Client: getaddrinfo() looks OK!\n");

            // Create a new socket to make a client connection.

            s = socket(result->ai\_family, result->ai\_socktype, result->ai\_protocol);

            if(s == INVALID\_SOCKET)

            {

                        printf("Client: socket() failed! Error code: %ld\n", WSAGetLastError());

                        // Do the clean up

                        WSACleanup();

                        // Exit with error

                        return 1;

            }

            else

                        printf("Client: socket() is OK!\n");

             // Make a connection to the server with socket SendingSocket.

            RetCode = connect(s, result->ai\_addr, result->ai\_addrlen);

            if(RetCode != 0)

            {

                        printf("Client: connect() failed! Error code: %ld\n", WSAGetLastError());

                        // Close the socket

                        closesocket(s);

                        // Do the clean up

                        WSACleanup();

                        // Exit with error

                        return 1;

            }

            else

            {

                        printf("Client: connect() is OK, got connected...\n");

                        printf("Client: Ready for sending and/or receiving data...\n");

            }

            // Return the allocated resource to the heap

            freeaddrinfo(result);

            // When your application is finished handling the connection, call WSACleanup.

            if(WSACleanup() != 0)

                        printf("Client: WSACleanup() failed!...\n");

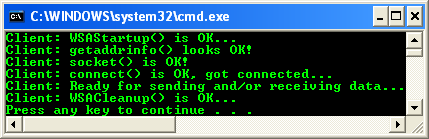
            else

                        printf("Client: WSACleanup() is OK...\n");

            return 0;

}

Build and run the project. The following screenshot shows a sample output.



The literal address “172.17.7.1” will be converted to the necessary socket address structure and returned via result. Because AF\_UNSPEC is passed, the API will determine the correct socket address structure (SOCKADDR\_IN or SOCKADDR\_IN6) required and convert the address accordingly. As before, the port field of the resulting socket address structure will be initialized to 5001.

Note that if no flags are passed as part of the hints and a literal string address is resolved, the returned structure addrinfo containing the converted address will have the AI\_NUMERICHOST flags set. Likewise, if a hostname is resolved but no hints are passed, the returned structure addrinfo flag will contain AI\_CANONNAME.

The last flag that can be used with getaddrinfo() is AI\_PASSIVE, which is used to obtain an address that can be passed to the bind() function. For IPv4, this would be INADDR\_ANY (0.0.0.0) and for IPv6 it would be IN6ADDR\_ANY (::). To obtain the bind address, the hints should indicate which address family the passive address is to be obtained for (via ai\_family), nodename should be NULL, and servname should be non-NULL - indicating the port number the application will bind to (which can be “0”). If AF\_UNSPEC is passed in the hints, then two addrinfo structures will be returned, one with the IPv4 bind address and the other with the IPv6 bind address.

The AI\_PASSIVE flag is useful after resolving a hostname via getaddrinfo(). Once the resolved address is returned, the original result's ai\_family can be used in another call to getaddrinfo() to obtain the appropriate bind address for that address family. This prevents applications from having to touch the internal socket address structure's fields and also removes the need for two separate code paths for binding the socket depending on which address family the address was resolved to. Take note that another name that you will find for getaddrinfo() is getaddrinfoA().

The following is the program example. Create a new empty Win32 console mode application and add the project/solution name.

Add the following code.

// link with Ws2\_32.lib

#include <winsock2.h>

// For addrinfo structure

#include <ws2tcpip.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

    // Declare and initialize variables

    WSADATA wsaData;

    int iResult;

    DWORD dwRetval;

    int i = 1;

    char \*port = "7171";

    struct addrinfo \*result = NULL;

    struct addrinfo \*ptr = NULL;

    struct addrinfo hints;

    // Validate the parameters

    if (argc != 2)

    {

        printf("Usage: %s <hostname>\n", argv[0]);

        printf("  getaddrinfo() provides protocol-independent translation\n");

        printf("  from an ANSI host name to an IP address\n");

        printf("Example: %s www.contoso.com\n", argv[0]);

        return 1;

    }

    // Initialize Winsock

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    if (iResult != 0)

    {

        printf("WSAStartup() failed with error code %d\n", iResult);

        return 1;

    }

    else

        printf("WSAStartup() is OK...\n");

    // Setup the hints address info structure

    // which is passed to the getaddrinfo() function

    ZeroMemory( &hints, sizeof(hints) );

    hints.ai\_family = AF\_UNSPEC;

    hints.ai\_socktype = SOCK\_STREAM;

    hints.ai\_protocol = IPPROTO\_TCP;

    // Call getaddrinfo(). If the call succeeds,

    // the result variable will hold a linked list

    // of addrinfo structures containing response information

    dwRetval = getaddrinfo(argv[1], port, &hints, &result);

    if ( dwRetval != 0 )

    {

        printf("getaddrinfo() failed with error: %d\n", dwRetval);

        WSACleanup();

        return 1;

    }

    printf("getaddrinfo() returned success!\n");

    // Retrieve each address and print out the hex bytes

    for(ptr=result; ptr!=NULL; ptr=ptr->ai\_next)

    {

        printf("\ngetaddrinfo() response %d\n", i++);

        printf("\tFlags: 0x%x\n", ptr->ai\_flags);

        printf("\tFamily: ");

        switch (ptr->ai\_family)

         {

            case AF\_UNSPEC:

                printf("Unspecified\n");

                break;

            case AF\_INET:

                printf("AF\_INET (IPv4)\n");

                break;

            case AF\_INET6:

                printf("AF\_INET6 (IPv6)\n");

                break;

            case AF\_NETBIOS:

                printf("AF\_NETBIOS (NetBIOS)\n");

                break;

            default:

                printf("Other %ld\n", ptr->ai\_family);

                break;

        }

        printf("\tSocket type: ");

        switch (ptr->ai\_socktype)

        {

            case 0:

                printf("Unspecified\n");

                break;

            case SOCK\_STREAM:

                printf("SOCK\_STREAM (stream)\n");

                break;

            case SOCK\_DGRAM:

                printf("SOCK\_DGRAM (datagram) \n");

                break;

            case SOCK\_RAW:

                printf("SOCK\_RAW (raw) \n");

                break;

            case SOCK\_RDM:

                printf("SOCK\_RDM (reliable message datagram)\n");

                break;

            case SOCK\_SEQPACKET:

                printf("SOCK\_SEQPACKET (pseudo-stream packet)\n");

                break;

            default:

                printf("Other %ld\n", ptr->ai\_socktype);

                break;

        }

        printf("\tProtocol: ");

        switch (ptr->ai\_protocol) {

            case 0:

                printf("Unspecified\n");

                break;

            case IPPROTO\_TCP:

                printf("IPPROTO\_TCP (TCP)\n");

                break;

            case IPPROTO\_UDP:

                printf("IPPROTO\_UDP (UDP) \n");

                break;

            default:

                printf("Other %ld\n", ptr->ai\_protocol);

                break;

        }

        printf("\tLength of this sockaddr: %d\n", ptr->ai\_addrlen);

        printf("\tCanonical name: %s\n", ptr->ai\_canonname);

    }

    // Release the allocated resource

    freeaddrinfo(result);

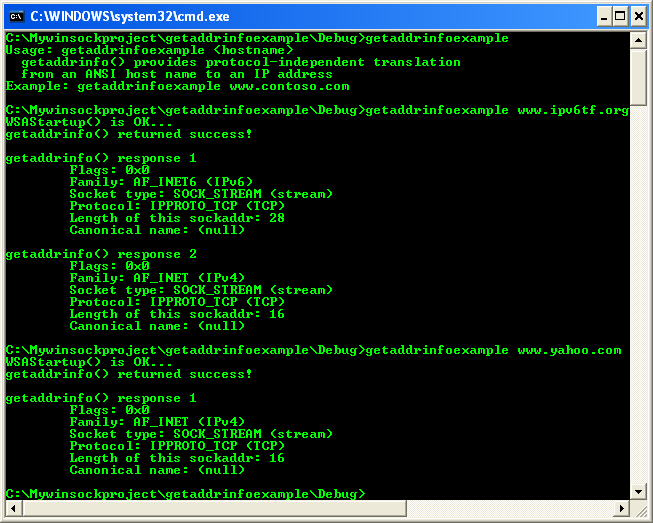
    // WSA clean up

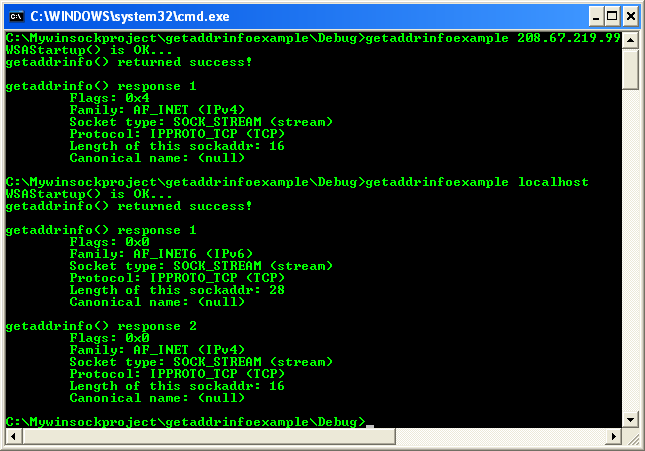
    WSACleanup();

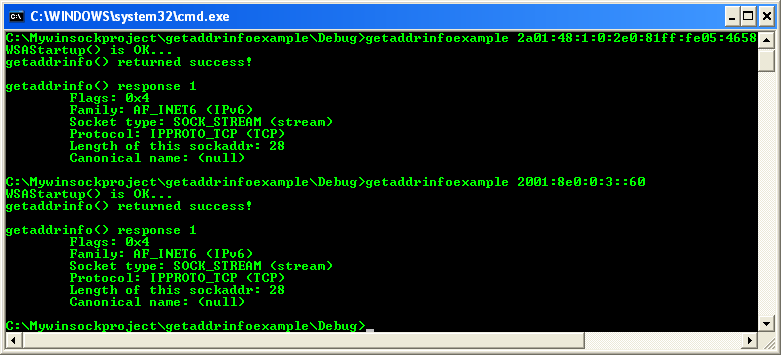
    return 0;

}

The following screenshots show sample outputs.







The Unicode Version, GetAddrInfoW()

The getaddrinfow() function provides protocol-independent translation from a Unicode host name to an address. Create a new empty Win32 console mode application and add the project/solution name.

Add the code given below.

// link with Ws2\_32.lib

#include <winsock2.h>

#include <ws2tcpip.h>

#include <stdio.h>

int wmain(int argc, wchar\_t \*\*argv)

{

    // Declare and initialize variables

    WSADATA wsaData;

    int iResult;

    DWORD dwRetval;

    int i = 1;

    // The L modifier can be discarded

    // for VS 2005/2008. Unicode supported by default

    // which can be verified through project property page

    WCHAR \*port = L"7172";

    ADDRINFOW \*result = NULL;

    ADDRINFOW \*ptr = NULL;

    ADDRINFOW hints;

    // Validate the parameters

    if (argc != 2)

    {

        wprintf(L"Usage: %ws <hostname>\n", argv[0]);

        wprintf(L"  getaddrinfow provides protocol-independent translation\n");

        wprintf(L"  from a Unicode host name to an IP address\n");

        wprintf(L"Example: %ws www.contoso.com\n", argv[0]);

        return 1;

    }

    // Initialize Winsock

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    if (iResult != 0)

    {

        wprintf(L"WSAStartup() failed with error code %d\n", iResult);

        return 1;

    }

    wprintf(L"WSAStartup() looks OK...\n");

    // Setup the hints address info structure

    // which is passed to the getaddrinfo() function

    ZeroMemory( &hints, sizeof(hints) );

    hints.ai\_family = AF\_UNSPEC;

    hints.ai\_socktype = SOCK\_STREAM;

    hints.ai\_protocol = IPPROTO\_TCP;

    // Call GetAddrinfoW(). If the call succeeds,

    // the result variable will hold a linked list

    // of addrinfow structures containing response information

    dwRetval = GetAddrInfoW(argv[1], port, &hints, &result);

    if ( dwRetval != 0 )

    {

        wprintf(L"GetAddrInfoW() failed with error code %d\n", dwRetval);

        WSACleanup();

        return 1;

    }

    wprintf(L"GetAddrInfoW() returned success!\n");

    // Retrieve each address and print out the hex bytes

    for(ptr=result; ptr != NULL ;ptr=ptr->ai\_next)

    {

        wprintf(L"GetAddrInfoW() response %d\n", i++);

        wprintf(L"\tFlags: 0x%x\n", ptr->ai\_flags);

        wprintf(L"\tFamily: ");

        switch (ptr->ai\_family)

        {

            case AF\_UNSPEC:

                wprintf(L"Unspecified\n");

                break;

            case AF\_INET:

                wprintf(L"AF\_INET (IPv4)\n");

                break;

            case AF\_INET6:

                wprintf(L"AF\_INET6 (IPv6)\n");

                break;

            default:

                wprintf(L"Other %ld\n", ptr->ai\_family);

                break;

        }

        wprintf(L"\tSocket type: ");

        switch (ptr->ai\_socktype)

        {

            case 0:

                wprintf(L"Unspecified\n");

                break;

            case SOCK\_STREAM:

                wprintf(L"SOCK\_STREAM (stream)\n");

                break;

            case SOCK\_DGRAM:

                wprintf(L"SOCK\_DGRAM (datagram) \n");

                break;

            case SOCK\_RAW:

                wprintf(L"SOCK\_RAW (raw) \n");

                break;

            case SOCK\_RDM:

                wprintf(L"SOCK\_RDM (reliable message datagram)\n");

                break;

            case SOCK\_SEQPACKET:

                wprintf(L"SOCK\_SEQPACKET (pseudo-stream packet)\n");

                break;

            default:

                wprintf(L"Other %ld\n", ptr->ai\_socktype);

                break;

        }

        wprintf(L"\tProtocol: ");

        switch (ptr->ai\_protocol)

        {

            case 0:

                wprintf(L"Unspecified\n");

                break;

            case IPPROTO\_TCP:

                wprintf(L"IPPROTO\_TCP (TCP)\n");

                break;

            case IPPROTO\_UDP:

                wprintf(L"IPPROTO\_UDP (UDP) \n");

                break;

            default:

                wprintf(L"Other %ld\n", ptr->ai\_protocol);

                break;

        }

        wprintf(L"\tLength of this sockaddr: %d\n", ptr->ai\_addrlen);

        wprintf(L"\tCanonical name: %s\n", ptr->ai\_canonname);

    }

    // Return the allocated resource back to the heap

    FreeAddrInfoW(result);

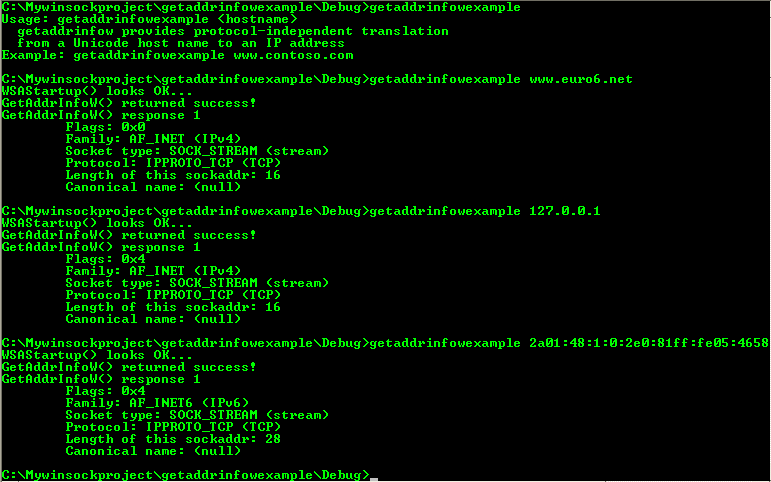
    // WSA clean up

    WSACleanup();

    return 0;

}

Build and run the project. The following screenshot shows a sample output.



|  |
| --- |
| Program Example Using AI\_NUMERICHOST    The following code example shows how to use the getaddrinfo() function to convert a text string representation of an IP address to an addrinfo structure that contains a sockaddr structure for the IP address and other information. Create a new empty Win32 console mode application and add the project/solution name.    The Windows socket/winsock2 IPv4, IPv6 Internet Protocol programming: another new getaddrinfo() project |

Add the following code.

// link with Ws2\_32.lib

#include <winsock2.h>

#include <ws2tcpip.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

    // Declare and initialize variables

    WSADATA wsaData;

    int iResult;

    DWORD dwRetval;

    int i = 1;

    struct addrinfo \*result = NULL;

    struct addrinfo \*ptr = NULL;

    struct addrinfo hints;

    // Validate the parameters

    if (argc != 2) {

        printf("Usage: %s <IP Address String>\n", argv[0]);

        printf("  getaddrinfo() determines the IP binary network address\n");

        printf("Example: %s 206.190.60.37\n", argv[0]);

        return 1;

    }

    // Initialize Winsock

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    if (iResult != 0)

    {

        printf("WSAStartup() failed with error code %d\n", iResult);

        return 1;

    }

    printf("WSAStartup() looks OK!\n");

    // Setup the hints address info structure which is passed to the getaddrinfo() function

    ZeroMemory( &hints, sizeof(hints) );

    hints.ai\_flags = AI\_NUMERICHOST;

    hints.ai\_family = AF\_UNSPEC;

    // Call getaddrinfo(). If the call succeeds,

    // the result variable will hold a linked list

    // of addrinfo structures containing response information

    dwRetval = getaddrinfo(argv[1], NULL, &hints, &result);

    if ( dwRetval != 0 )

    {

        printf("getaddrinfo failed with error: %d\n", dwRetval);

        WSACleanup();

        return 1;

    }

    printf("getaddrinfo() returned success!\n");

    // Retrieve each address and print out the hex bytes

    for(ptr=result; ptr != NULL ;ptr=ptr->ai\_next)

    {

        printf("getaddrinfo() response %d\n", i++);

        printf("\tFlags: 0x%x\n", ptr->ai\_flags);

        printf("\tFamily: ");

        switch (ptr->ai\_family)

        {

            case AF\_UNSPEC:

                printf("Unspecified\n");

                break;

            case AF\_INET:

                printf("AF\_INET (IPv4)\n");

                break;

            case AF\_INET6:

                printf("AF\_INET6 (IPv6)\n");

                break;

            case AF\_NETBIOS:

                printf("AF\_NETBIOS (NetBIOS)\n");

                break;

            default:

                printf("Other %ld\n", ptr->ai\_family);

                break;

        }

        printf("\tSocket type: ");

        switch (ptr->ai\_socktype)

        {

            case 0:

                printf("Unspecified\n");

                break;

            case SOCK\_STREAM:

                printf("SOCK\_STREAM (stream)\n");

                break;

            case SOCK\_DGRAM:

                printf("SOCK\_DGRAM (datagram) \n");

                break;

            case SOCK\_RAW:

                printf("SOCK\_RAW (raw) \n");

                break;

            case SOCK\_RDM:

                printf("SOCK\_RDM (reliable message datagram)\n");

                break;

            case SOCK\_SEQPACKET:

                printf("SOCK\_SEQPACKET (pseudo-stream packet)\n");

                break;

            default:

                printf("Other %ld\n", ptr->ai\_socktype);

                break;

        }

        printf("\tProtocol: ");

        switch (ptr->ai\_protocol)

        {

            case 0:

                printf("Unspecified\n");

                break;

            case IPPROTO\_TCP:

                printf("IPPROTO\_TCP (TCP)\n");

                break;

            case IPPROTO\_UDP:

                printf("IPPROTO\_UDP (UDP) \n");

                break;

            default:

                printf("Other %ld\n", ptr->ai\_protocol);

                break;

        }

        printf("\tLength of this sockaddr: %d\n", ptr->ai\_addrlen);

        printf("\tCanonical name: %s\n", ptr->ai\_canonname);

    }

    // Release the allocated resource

    freeaddrinfo(result);

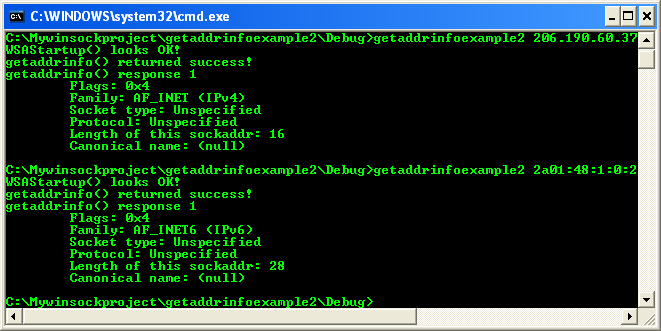
    // WSA clean up

    WSACleanup();

    return 0;

}

The following screenshot shows a sample output.



The other new name resolution API is getnameinfo(), which performs the reverse of the getaddrinfo() function. It takes a socket address structure already initialized and returns the host and service name corresponding to the address and port information. The getnameinfo() function is prototyped as the following:

int getnameinfo(

                        const struct sockaddr FAR \*sa,

                        socklen\_t salen,

                        char FAR \*host,

                        DWORD hostlen,

                        char FAR \*serv,

                        DWORD servlen,

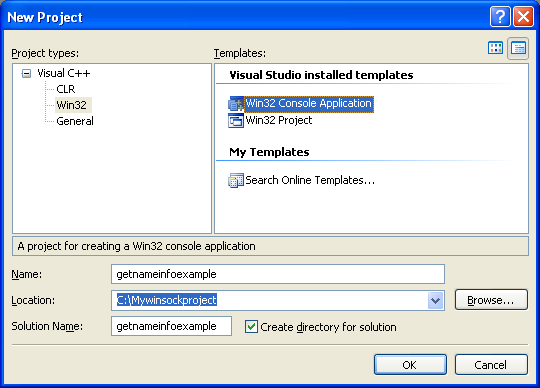
                        int flags

);

The parameters are fairly self-explanatory. sa is the socket address structure on which the name information will be obtained for and salen is the size of that structure. host is the character buffer to receive the host's name. By default, the fully qualified domain name (FQDN) will be returned. hostlen simply indicates the size of the host buffer. serv is the character buffer to receive the service/port information and servlen is the length of that buffer. Finally, the flags parameter indicates how the socket address should be resolved. The possible flag values are the following:

1. NI\_NOFQDN indicates that only the relative distinguished name (RDN) returned. For example, with this flag set the host named “mist.microsoft.com” would return only “mist”.
2. NI\_NUMERICHOST indicates to return the string representation of the address and not the hostname.
3. NI\_NAMEREQD indicates if the address cannot be resolved to a FQDN to return an error.
4. NI\_NUMERICSERV indicates to return the port information as a string instead of resolving to a well-known service name, such as “ftp.” Note that if the serv buffer is supplied with this flag absent and the port number cannot be resolved to a well-known service, genameinfo() will fail with error WSANO\_DATA (11004).
5. NI\_DGRAM is used to differentiate datagram services from stream services. This is necessary for those few services that define different port numbers for UDP and TCP.

The following code example shows how to use the getnameinfo() function. Create a new empty Win32 console mode application and add the project/solution name.



Add the following code.

#include <winsock2.h>

#include <ws2tcpip.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

    // Declare and initialize variables

    WSADATA wsaData;

    int iResult;

    DWORD dwRetval;

    struct sockaddr\_in saGNI;

    char hostname[NI\_MAXHOST]; // Set to the max value

    char servInfo[NI\_MAXSERV]; // Set to the max value

    // Assume we plan to use TCP port 7777

    u\_short port = 7777;

    // Validate the parameters

    if (argc != 2)

    {

        printf("Usage: %s <IPv4 address>\n", argv[0]);

        printf("  This program return hostname\n");

        printf("Example: %s 127.0.0.1\n", argv[0]);

        return 1;

    }

    // Initialize Winsock

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    if (iResult != 0)

    {

        printf("WSAStartup() failed with error code %d\n", iResult);

        return 1;

    }

    printf("WSAStartup() looks OK...\n");

    // Set up sockaddr\_in structure which is passed to the getnameinfo function

    saGNI.sin\_family = AF\_INET; // AF\_INET6

    saGNI.sin\_addr.s\_addr = inet\_addr(argv[1]);

    saGNI.sin\_port = htons(port);

    // Call getnameinfo()

    dwRetval = getnameinfo((struct sockaddr \*) &saGNI,

                           sizeof (struct sockaddr),

                           hostname,

                           NI\_MAXHOST, servInfo,

                           NI\_MAXSERV, NI\_NUMERICSERV);

    if (dwRetval != 0)

    {

        printf("getnameinfo() failed with error code %ld\n", WSAGetLastError());

    }

    else

    {

        printf("getnameinfo() returned hostname \"%s\"\n", hostname);

    }

    // Terminate the use of Winsock

    WSACleanup();

    if (dwRetval != 0)

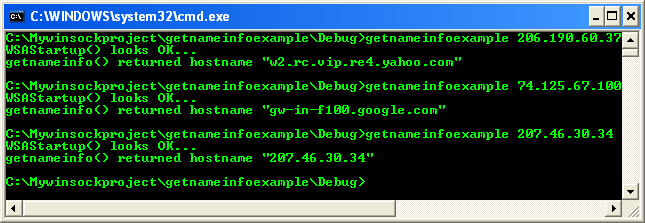
        return 1;

    else

        return 0;

}

The following screenshot shows a sample output.



The getnameinfoW() function is the Unicode version of getnameinfo(). The getnameinfoW() function was added to the Ws2\_32.dll in Windows XP with SP2 and function cannot be used on versions of Windows earlier than Windows XP with SP2.

getaddrinfo() and getnameinfo() are address independent variant that hides the detail in name-to-address translation, or vice versa. It implements functionalities for the following functions: gethostbyname(), gethostbyaddr(), inet\_ntop(), inet\_pton(), getservbyname(), getservbyport() and many more.

The [rfc2553](http://www.ietf.org/rfc/rfc2553.txt) has proposed the [struct sockaddr\_storage](http://msdn.microsoft.com/en-us/library/ms740504.aspx) which is a placeholder for all sockaddr-variant structures. This RFC has been adopted by implementers including Microsoft. The SOCKADDR structure is large enough to contain a transport address for most address families. For a structure that is guaranteed to be large enough to contain a transport address for all possible address families. This is implemented like follows:

struct sockaddr\_storage {

  short   ss\_family;

  char    \_\_ss\_pad1[\_SS\_PAD1SIZE];

  \_\_int64 \_\_ss\_align;

  char    \_\_ss\_pad2[\_SS\_PAD2SIZE];

};

You should use this structure to hold any of sockaddr-variant structures.

Simple Address Conversion

When an application needs only to convert between string literal addresses and socket address structures, the WSAStringToAddress() and WSAAddressToString() helper APIs are available. WSAStringToAddress() is not as “smart” as getaddrinfo() because you must specify the address family that the string address belongs to. The API syntax is:

INT WSAStringToAddress(

                        LPTSTR AddressString,

                        INT AddressFamily,

                        LPWSAPROTOCOL\_INFO lpProtocolInfo,

                        LPSOCKADDR lpAddress,

                        LPINT lpAddressLength

);

The first parameter is the string to convert and the second indicates the address family the string belongs to (such as AF\_INET, AF\_INET6, or AF\_IPX). The third parameter, lpProtocolInfo, is an optional pointer to the WSAPROTOCOL\_INFO structure that defines the protocol provider to use when performing the conversion. If there are multiple providers implementing a protocol, this parameter can be used to specify an explicit provider. The fourth parameter is the appropriate socket address structure to which the string address will be converted and assigned into.

Note that this API will convert string addresses that contain port numbers. For example, the IPv4 string notation allows a colon followed by the port number at the end of the address. For example, “157.54.126.42:1200” indicates the IPv4 address using port 1200. In IPv6, the IPv6 address string must be enclosed in square brackets after which the colon and port notation may be used. For example, [fe80::250:8bff:fea0:92ed%5]:80 indicates a link-local address with its scope ID followed by port 80. Note that only port numbers will be resolved and not service names (such as “ftp”). For both these examples, if these strings were converted with WSAStringToAddress(), then the returned socket address structure will be initialized with the appropriate binary IP address, port number, and address family. For IPv6, the scope ID field will also be initialized if the string address contains “%scope\_ID” after the address portion. The WSAAddressToString() provides a mapping from a socket address structure to a string representation of that address. The prototype is:

INT WSAAddressToString(

                        LPSOCKADDR lpsaAddress,

                        DWORD dwAddressLength,

                        LPWSAPROTOCOL\_INFO lpProtocolInfo,

                        LPTSTR lpszAddressString,

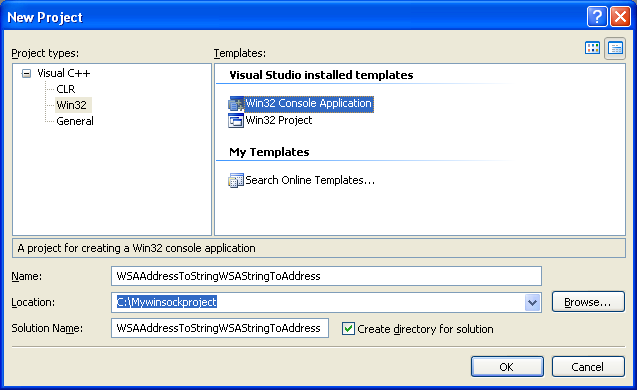
                        LPDWORD lpdwAddressStringLength

);

This function takes a SOCKADDR structure and formats the binary address to a string indicated by the buffer lpszAddressString. Again, if there is more than one transport provider for a given protocol, a specific one may be selected by passing its WSAPROTOCOL\_INFO structure as lpProtocolInfo. Note that the address family is a field of the SOCKADDR structure passed as lpsaAddress. Support for IPv6 addresses using the WSAAddressToString() function was added on Windows XP with Service Pack 1 (SP1) and later. IPv6 must also be installed on the local computer for the WSAAddressToString() function to support IPv6 addresses. While the inet\_ntoa() function works only with IPv4 addresses, the WSAAddressToString() function works with any socket address supported by a Winsock provider on the local computer including IPv6 addresses.

WSAAddressToString() and WSAStringToAddress() Program Examples

The following program example tries to demonstrate the use of WSAAddressToString() and WSAStringToAddress() functions. Create a new empty Win32 console mode application and add the project/solution name.



Add the following code.

// All in wide character/Unicode and IPv6

// Uninitialized socket parameters will be assigned the

// default values. The comment parts are for IPv4

#include <winsock2.h>

#include <ws2tcpip.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

            WSADATA                wsaData;

            int                                iResult, RetVal, RetVal2;

            // SOCKADDR\_IN  MyAddr;

            SOCKADDR\_IN6  MyAddrValue;

            // Storage buffer, using LPWSTR pointer failed lor!

            // WCHAR  MyAddrString[256] = L"For IPv4...";

            WCHAR  MyAddrString[256] = L"Some dummy initializer";

            // Take note that if the address is in FQDN/Name/hostname

            // we need to convert it to a dotted IP address. For example:

            // WCHAR IPAddr[256] = L"localhost"; // this should fail lor!

            // WCHAR IPAddr[256] = L"209.131.36.158";

            WCHAR IPAddr[256] = L"2a01:48:1:0:2e0:81ff:fe05:4658";

            // These buffer sizes must be allocated properly else

            // the WSAStringToAddress() and WSAAddressToString() will

            // fail miserably!

            // int MyAddrSize = sizeof(struct sockaddr\_storage);

            int MyAddrSize = sizeof(struct sockaddr\_storage);

            // DWORD MyAddrSize2 = sizeof(struct sockaddr\_storage);

            DWORD MyAddrSize2 = sizeof(struct sockaddr\_storage);

    // Initialize Winsock

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    if (iResult != 0)

    {

        printf("WSAStartup() failed with error code %d\n", WSAGetLastError());

        // Exit with error

        return 1;

    }

    else

        printf("WSAStartup() is OK...\n");

            // Initialize the address family

            // MyAddrValue.sin\_family = AF\_INET;

            MyAddrValue.sin6\_family = AF\_INET6;

            // This part should convert the server/hostname/FQDN to a standard dotted IP address

            // string first (if any) using other helper functions such as getaddressinfo()

            //  ==============Convert address string to value using WSAStringToAddress()==============

            printf("\nWSAStringToAddress() - string to address value\n");

            RetVal = WSAStringToAddress(

                        IPAddr,                                               // Pointer to address string to be converted

                        AF\_INET6 /\* AF\_INET\*/,                  // address family

                        NULL,                                                 // Protocol info structure, not useable here

                        (LPSOCKADDR)&MyAddrValue,  // Socket address string buffer [out]

                        &MyAddrSize                                    // Length of socket structure

                        );

            // Something wrong

            if( RetVal != 0)

            {

                        printf("WSAStringToAddress() failed with error code %ld\n", WSAGetLastError());

                        // Clean-up

                        WSACleanup();

                        // Exit with error

                        return 1;

            }

            else

                        printf("WSAStringToAddress() looks fine! Return value is %ld\n", RetVal);

            // printf("The IP Address part of the buffer filled with %ul\n", MyAddr.sin\_addr);

            printf("The IP address value is %ul\n", MyAddrValue.sin6\_addr);

            // and other info

            printf("The address family value is %ul\n", MyAddrValue.sin6\_family);

            printf("The port value is %ul\n", MyAddrValue.sin6\_port);

            printf("The scopeID is %ul\n", MyAddrValue.sin6\_scope\_id);

            // =========Do the re-conversion using WSAAddressToString()===============

            printf("\nWSAAddressToString() - address value to string\n");

            RetVal2 = WSAAddressToString(

                        (LPSOCKADDR)&MyAddrValue,  // Pointer to address value to be converted

                        MyAddrSize,              // Length of pointer to address value, in bytes

                        NULL,                         // A pointer to the WSAPROTOCOL\_INFO structure for a particular provider

                        MyAddrString,           // A pointer to the buffer that receives the human-readable address string

                        &MyAddrSize2          // length of the buffer pointed to by the MyAddrString parameter

                        );

            // Something wrong

            if( RetVal2 != 0)

            {

                        printf("WSAAddressToString() failed with error code %ld\n", WSAGetLastError());

                        // Clean-up

                        WSACleanup();

                        // Exit with error

                        return 1;

            }

            else

                        printf("WSAAddressToString() looks fine! Return value is %ld\n", RetVal2);

            // printf("The IP Address is %ul\n", MyAddr.sin\_addr);

            printf("The address string is %S\n", MyAddrString);

            // and other info

            printf("The address family is %ul\n", MyAddrValue.sin6\_family);

            printf("The port value is %ul\n", MyAddrValue.sin6\_port);

            printf("The scopeID is %ul\n", MyAddrValue.sin6\_scope\_id);

            // Call WSACleanup for WSA\* cleanup

            if(WSACleanup() != 0)

                printf("\nWSACleanup() failed!...\n");

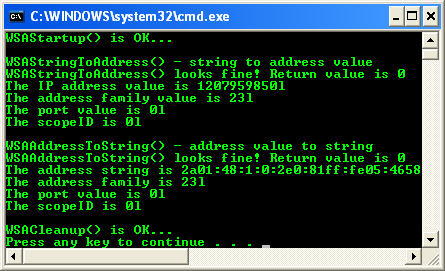
            else

                printf("\nWSACleanup() is OK...\n");

            return 0;

}

Build and run the project. The following screenshot shows a sample output.



The following is another simple example for WSAAddressToString() and WSAStringToAddress() functions.

#include <winsock2.h>

#include <stdio.h>

int main(void)

{

            WSADATA wsaData;

            SOCKADDR\_IN  addr;

            char   AddrValue[256] = "216.239.61.104";

            char   AddrString[256]  = "Some dummy value";

            DWORD   dwSizeOfStr = sizeof(AddrValue);

            int    nSizeOfInput = sizeof(AddrString), iResult;

    // Initialize Winsock

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    if (iResult != 0)

    {

        printf("WSAStartup() failed with error code %d\n", WSAGetLastError());

        // Exit with error

        return 1;

    }

    else

        printf("WSAStartup() is OK...\n");

            // Initialize address family

            addr.sin\_family = AF\_INET;

            if(WSAStringToAddressA(AddrValue, AF\_INET, NULL, (LPSOCKADDR)&addr, &nSizeOfInput) != 0)

            {

                        printf("\nWSAStringToAddressA failed with error num %ld\n", WSAGetLastError());

                        WSACleanup();

                        return 1;

            }

            else

            {

                        printf("\nAddress in value = %ul\n", addr.sin\_addr);

            }

            if(WSAAddressToStringA((LPSOCKADDR)&addr, sizeof(addr), NULL, AddrString, &dwSizeOfStr) != 0)

            {

                        printf("WSAAddressToStringA() failed miserably with error code %ld\n", WSAGetLastError());

                        WSACleanup();

                        return 1;

            }

            else

                        printf("Address string = %s\n", AddrString);

            // Do the WSACleanup...

            if(WSACleanup() != 0)

                        printf("\nWSACleanup() failed!...\n");

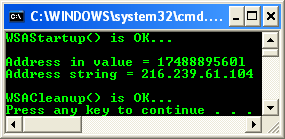
            else

                        printf("\nWSACleanup() is OK...\n");

            return 0;

}

Build and run the project. The following screenshot shows a sample output.



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| The InetNtop()/inet\_ntop()   The InetNtop() function converts an IPv4 or IPv6 Internet network address into a string in Internet standard format. The ANSI version of this function is inet\_ntop(). The syntax is:   PCTSTR WSAAPI InetNtop(                          INT  Family,                          PVOID pAddr,                          PTSTR pStringBuf,                          size\_t StringBufSize  );    The Family is the address family. The possible values for the address family are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Winsock2.h, and should never be used directly. Note that the values for the AF\_ address family and PF\_ protocol family constants are identical (for example, AF\_INET and PF\_INET), so either constant can be used. The values currently supported are AF\_INET and AF\_INET6.     |  |  | | --- | --- | | **Value** | **Meaning** | | AF\_INET (2) | The Internet Protocol version 4 (IPv4) address family. When this parameter is specified, this function returns an IPv4 address string. | | AF\_INET6 (23) | The Internet Protocol version 6 (IPv6) address family. When this parameter is specified, this function returns an IPv6 address string. |     The pAddr is a pointer to the IP address in network byte to convert to a string. When the Family parameter is AF\_INET, then the pAddr parameter must point to an IN\_ADDR structure with the IPv4 address to convert. When the Family parameter is AF\_INET6, then the pAddr parameter must point to an IN6\_ADDR structure with the IPv6 address to convert.  The third parameter, pStringBuf is a pointer to a buffer in which to store the NULL-terminated string representation of the IP address. For an IPv4 address, this buffer should be large enough to hold at least 16 characters and for an IPv6 address, this buffer should be large enough to hold at least 46 characters.  For the StringBufSize, on input, the length, in characters, of the buffer pointed to by the pStringBuf parameter while on output, this parameter contains the number of characters actually written to the buffer pointed to by the pStringBuf parameter. |

If no error occurs, InetNtop function returns a pointer to a buffer containing the string representation of IP address in standard format. Otherwise, a value of NULL is returned, and a specific error code can be retrieved by calling the WSAGetLastError() for extended error information. If the function fails, the extended error code returned by WSAGetLastError() can be one of the following values.

|  |  |
| --- | --- |
| **Error code** | **Meaning** |
| WSAEAFNOSUPPORT | The address family specified in the Family parameter is not supported. This error is returned if the Family parameter specified was not AF\_INET or AF\_INET6. |
| ERROR\_INVALID\_PARAMETER | An invalid parameter was passed to the function. This error is returned if a NULL pointer is passed in the pStringBuf or the StringBufSize parameter is zero. This error is also returned if the length of the buffer pointed to by the pStringBuf parameter is not large enough to receive the string representation of the IP address. |

The InetNtop() function is supported on **Windows Vista and later** which provides a protocol-independent address-to-string translation. It takes an Internet address structure specified by the pAddr parameter and returns a NULL-terminated string that represents the IP address. While the inet\_ntoa() function works only with IPv4 addresses, the InetNtop() function works with either IPv4 or IPv6 addresses.

The ANSI version of this function is inet\_ntop() as defined in RFC 2553. The InetNtop() function does not require that the Windows Sockets DLL be loaded to perform IP address to string conversion. If the Family parameter specified is AF\_INET, then the pAddr parameter must point to an IN\_ADDR structure with the IPv4 address to convert. The address string returned in the buffer pointed to by the pStringBuf parameter is in dotted-decimal notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation. If the Family parameter specified is AF\_INET6, then the pAddr parameter must point to an IN6\_ADDR structure with the IPv6 address to convert. The address string returned in the buffer pointed to by the pStringBuf parameter is in Internet standard format. The basic string representation consists of 8 hexadecimal numbers separated by colons. A string of consecutive zero numbers is replaced with a double-colon. There can only be one double-colon in the string representation of the IPv6 address. The last 32 bits are represented in IPv4-style dotted-octet notation if the address is an IPv4-compatible address.

If the length of the buffer pointed to by the pStringBuf parameter is not large enough to receive the string representation of the IP address, InetNtop returns ERROR\_INVALID\_PARAMETER.

When UNICODE or \_UNICODE is defined, InetNtop() is defined to InetNtopW(), the Unicode version of this function. The pStringBuf parameter is defined to the PSTR data type. When UNICODE or \_UNICODE is not defined, InetNtop() is defined to InetNtopA(), the ANSI version of this function. The ANSI version of this function is always defined as inet\_ntop(). The pStringBuf parameter is defined to the PWSTR data type.

The IN\_ADDR structure is defined in the Inaddr.h header file. The IN6\_ADDR structure is defined in the In6addr.h header file. On Windows Vista and later, the RtlIpv4AddressToString() and RtlIpv4AddressToStringEx() functions can be used to convert an IPv4 address represented as an IN\_ADDR structure to a string representation of an IPv4 address in Internet standard dotted-decimal notation. Also for the similar Windows platform, the RtlIpv6AddressToString() and RtlIpv6AddressToStringEx() functions can be used to convert an IPv6 address represented as an IN6\_ADDR structure to a string representation of an IPv6 address. The RtlIpv6AddressToStringEx() function is more flexible since it also converts an IPv6 address, scope ID, and port to a IPv6 string in standard format.

**The InetPton()/inet\_pton()**

 The InetPton() function converts an IPv4 or IPv6 Internet network address in its standard text presentation form into its numeric binary form. The ANSI version of this function is inet\_pton(). The syntax is:

 PCTSTR WSAAPI inet\_pton(

  INT  Family,

  PCTSTR pszAddrString,

  PVOID pAddrBuf

);

The Family parameter is the address family. The possible values for the address family are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Winsock2.h, and should never be used directly. Note that the values for the AF\_ address family and PF\_ protocol family constants are identical (for example, AF\_INET and PF\_INET), so either constant can be used. The values currently supported are AF\_INET and AF\_INET6.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| AF\_INET (2) | The Internet Protocol version 4 (IPv4) address family. When this parameter is specified, the pszAddrString parameter must point to a text representation of an IPv4 address and the pAddrBuf parameter returns a pointer to an IN\_ADDR structure that represents the IPv4 address. |
| AF\_INET6 (23) | The Internet Protocol version 6 (IPv6) address family. When this parameter is specified, the pszAddrString parameter must point to a text representation of an IPv6 address and the pAddrBuf parameter returns a pointer to an IN6\_ADDR structure that represents the IPv6 address. |

The pszAddrString is a pointer to the NULL-terminated string that contains the text representation of the IP address to convert to numeric binary form. When the Family parameter is AF\_INET, then the pszAddrString parameter must point to a text representation of an IPv6 address in standard notation. When the Family parameter is AF\_INET6, then the pszAddrString parameter must point to a text representation of an IPv4 address in standard dotted-decimal notation.

The third parameter, pAddrBuf is a pointer to a buffer in which to store the numeric binary representation of the IP address. The IP address is returned in network byte order. When the Family parameter is AF\_INET, this buffer should be large enough to hold an IN\_ADDR structure.

When the Family parameter is AF\_INET6, this buffer should be large enough to hold an IN6\_ADDR structure.

If no error occurs, the InetPton() function returns a value of 1 and the buffer pointed to by the pAddrBuf parameter contains the binary numeric IP address in network byte order. The InetPton() function returns a value of 0 if the pAddrBuf parameter points to a string that is not a valid IPv4 dotted-decimal string or a valid IPv6 address string. Otherwise, a value of -1 is returned, and a specific error code can be retrieved by calling the WSAGetLastError() for extended error information. If the function has an error, the extended error code returned by WSAGetLastError() can be one of the following values.

|  |  |
| --- | --- |
| **Error code** | **Meaning** |
| WSAEAFNOSUPPORT | The address family specified in the Family parameter is not supported. This error is returned if the Family parameter specified was not AF\_INET or AF\_INET6. |
| WSAEFAULT | The pszAddrString or pAddrBuf parameters are NULL or are not part of the user address space. |

The InetPton() function is supported on **Windows Vista and later** which provides a protocol-independent conversion of an Internet network address in its standard text presentation form into its numeric binary form. The InetPton() function takes a text representation of an Internet address pointed to by the pszAddrString parameter and returns a pointer to the numeric binary IP address in the pAddrBuf parameter. While the inet\_addr() function works only with IPv4 address strings, the InetPton function works with either IPv4 or IPv6 address strings. The ANSI version of this function is inet\_pton() as defined in RFC 2553.

The InetPton() function does not require that the Windows Sockets DLL be loaded to perform conversion of a text string that represents an IP address to a numeric binary IP address. If the Family parameter specified is AF\_INET, then the pszAddrString parameter must point a text string of an IPv4 address in dotted-decimal notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation.

If the Family parameter specified is AF\_INET6, then the pszAddrString parameter must point a text string of an IPv6 address in Internet standard format. The basic string representation consists of 8 hexadecimal numbers separated by colons. A string of consecutive zero numbers may be replaced with a double-colon. There can only be one double-colon in the string representation of the IPv6 address. The last 32 bits may be represented in IPv4-style dotted-octet notation if the address is an IPv4-compatible address.

When UNICODE or \_UNICODE is defined, InetPton() is defined to InetPtonW(), the Unicode version of this function. The pszAddrString parameter is defined to the PCWSTR data type. When UNICODE or \_UNICODE is not defined, InetPton() is defined to InetPtonA(), the ANSI version of this function. The ANSI version of this function is always defined as inet\_pton(). The pszAddrString parameter is defined to the PCSTR data type. The IN\_ADDR structure is defined in the Inaddr.h header file. The IN6\_ADDR structure is defined in the In6addr.h header file. The following code snippet shows a sample inet\_pton() and inet\_ntop() use. Take note that inet\_pton() and inet\_ntop() functions already available and used in BSD socket long time ago.

 #include <winsock2.h>

#include <ws2tcpip.h>

#include <stdlib.h>

#include <stdio.h>

// Minimum system required are

// Windows Server 2008 or Windows Vista

// http://msdn.microsoft.com/en-us/library/aa383745.aspx

// #define \_WIN32\_WINNT 0x0600

int main(int argc, char \*argv[])

{

            WSADATA wsaData;

            struct addrinfo hints, \*res, \*p;

            int status;

            // Provide big enough buffer, ipv6 should be the biggest

            char ipstr[INET6\_ADDRSTRLEN];

            char ipstr2[INET6\_ADDRSTRLEN];

            if (argc != 2)

            {

                        printf("Usage: %s <hostname>\n", argv[0]);

                        printf("Example: %s www.yahoo.com\n", argv[0]);

                        return 1;

            }

            // Initialization

            if (WSAStartup(MAKEWORD(1,1), &wsaData) != 0)

            {

                        printf("WSAStartup() failed miserably! With error code %ld\n", WSAGetLastError());

                        return 1;

            }

            else

                        printf("WSAStartup() looks fine!\n");

            memset(&hints, 0, sizeof hints);

            hints.ai\_family = AF\_UNSPEC; // AF\_INET or AF\_INET6 to force version

            hints.ai\_socktype = SOCK\_STREAM;

            if ((status = getaddrinfo(argv[1], NULL, &hints, &res)) != 0)

            {

                        printf("getaddrinfo() failed lor! with error code %ld\n", WSAGetLastError());

                        WSACleanup();

                        return 1;

            }

            printf("The IP addresses for %s:\n", argv[1]);

            for(p = res;p != NULL; p = p->ai\_next)

            {

                        void \*addr;

                        char \*ipver;

                        // Get the pointer to the address itself, different fields in IPv4 and IPv6

                        if (p->ai\_family == AF\_INET)

                        {

                                    // IPv4

                                    struct sockaddr\_in \*ipv4 = (struct sockaddr\_in \*)p->ai\_addr;

                                    addr = &(ipv4->sin\_addr);

                                    ipver = "IPv4";

                        }

                        else

                        {

                                    // IPv6

                                    struct sockaddr\_in6 \*ipv6 = (struct sockaddr\_in6 \*)p->ai\_addr;

                                    addr = &(ipv6->sin6\_addr);

                                    ipver = "IPv6";

                        }

                        // Convert the IP to a string and print it

                        printf("String format: ");

                        // InetNtop(address\_family, IP\_address\_in\_network\_byte\_to\_convert\_to\_a\_string,

                        //         buffer\_to\_store\_the\_IP\_address\_string, the\_IP\_string\_length\_in\_character);

                        inet\_ntop(p->ai\_family, addr, (PSTR)ipstr, sizeof(ipstr));

                        printf(" %s: %s\n", ipver, ipstr);

                        printf("Value format: ");

                        // InetPton(address\_family, string\_to\_be\_converted, buffer\_to\_store\_the\_converted\_string);

                        inet\_pton(p->ai\_family, (PCSTR)ipstr, ipstr2);

                        printf(" %s: %ul\n", ipver, ipstr2);

            }

            // Deallocate the resource...

            freeaddrinfo(res);

            // Cleanup the WSA...

            WSACleanup();

            return 0;

}

 You can compile on the older Windows OS but target the minimum Windows Vista or Server 2008 to run this program. Running on the older platform will generate a message similar to the following crap:

The Windows socket/winsock2 IPv4, IPv6 Internet Protocol programming: The procedure entry point inet_ntop could not be located in the dynamic link library ws2_32.dll


# Legacy Name Resolution Routines

This section covers the legacy name resolution and is included only for the sake of code maintenance, because new applications should be using getaddrinfo() and getnameinfo(). The other feature you will notice is that the two new API calls replace eight legacy functions.

The function inet\_addr() converts a dotted IPv4 address to a 32-bit unsigned long integer quantity. The inet\_addr() function is defined as:

 unsigned long inet\_addr(const char FAR \*cp);

 The cp field is a null-terminated character string that accepts an IP address in dotted notation. Note that this function returns an IPv4 address as a 32-bit unsigned long integer in network-byte order, which can be assigned into the SOCKADDR\_IN field sin\_addr. The reverse of inet\_addr() is inet\_ntoa(), which takes an IPv4 network address and converts it to a string. This function is declared as:

char FAR \*inet\_ntoa(Struct in\_addr in);

The following code sample demonstrates how to create a SOCKADDR\_IN structure using the inet\_addr() and htons functions.

 SOCKADDR\_IN InternetAddr;

INT nPortId = 5150;

InternetAddr.sin\_family = AF\_INET;

// Convert the proposed dotted Internet address 136.149.3.29

// to a 4-byte integer, and assign it to sin\_addr

InternetAddr.sin\_addr.s\_addr = inet\_addr("136.149.3.29");

// The nPortId variable is stored in host-byte order. Convert

// nPortId to network-byte order, and assign it to sin\_port.

InternetAddr.sin\_port = htons(nPortId);

The following code example demonstrates the use of the inet\_addr() and inet\_ntoa() functions.

 #include <winsock2.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

    // Declare and initialize variables

    WSADATA wsaData;

    int iResult;

    unsigned long ulAddr = INADDR\_NONE;

    ULONG TempBuff = 0;

    struct sockaddr\_in saSample;

    // Validate the parameters

    if (argc != 2)

    {

        printf("Usage: %s <IPv4 address>\n", argv[0]);

        printf("  inet\_addr() converts a string containing an\n");

        printf("  IPv4 address in one of the supported formats\n");

        printf("  to a unsigned long representing an IN\_ADDR\n");

                        printf("Example: %s 209.131.36.158\n", argv[0]);

        return 1;

    }

    // Initialize Winsock

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    if (iResult != 0)

    {

        printf("WSAStartup() failed with error code %ld\n", WSAGetLastError());

        return 1;

    }

    // Just for demonstration, call inet\_addr(). If the call

    // succeeds, the result variable will hold a IN\_ADDR structure

    ulAddr = inet\_addr(argv[1]);

    if ( ulAddr == INADDR\_NONE )

    {

        printf("inet\_addr() failed and returned INADDR\_NONE\n");

        WSACleanup();

        return 1;

    }

    if (ulAddr == INADDR\_ANY) {

        printf("inet\_addr() failed and returned INADDR\_ANY\n");

        WSACleanup();

        return 1;

    }

            printf("inet\_addr() returned success!\n");

            // Retrieve address and print out the hex/dec/octal bytes

            printf("  Hex:\t\t0X%0X\n", ulAddr);

            printf("  Decimal:\t%ul\n", ulAddr);

            printf("  Octal:\t0%o\n", ulAddr);

            // Then, ready to be used

            saSample.sin\_family = AF\_INET;

            // In typical app. we can be directly use:

            // usesaSample.sin\_addr.s\_addr = inet\_addr(argv[1]);

            saSample.sin\_addr.s\_addr = ulAddr;

            saSample.sin\_port = htons(1234);

            // More codes here, create socket, bind, connect etc.

            // Revert the process using inet\_ntoa() just for the address part

            if(inet\_ntoa(saSample.sin\_addr) != NULL)

            {

                        printf("inet\_ntoa() return success!\n");

                        printf("The IP address is %s\n", inet\_ntoa(saSample.sin\_addr));

            }

            else

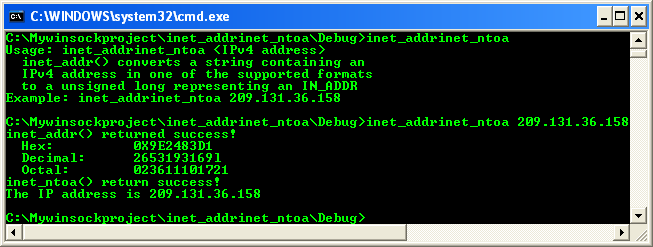
                        printf("inet\_ntoa() failed with error code %ld\n", WSAGetLastError());

    WSACleanup();

    return 0;

}

Build and run the project. The following screenshot shows a sample output.



 The Winsock functions gethostbyname(), WSAAsyncGetHostByName(), gethostbyaddr(), and WSAAsyncGetHostByAddr() retrieve host information corresponding to a host name or host address from a host database. The first two functions translate a hostname to its network IPv4 addresses and the second two do the reverse - map an IPv4 network address back to a hostname. These functions return a HOSTENT structure that is defined as:

struct hostent

{

    char FAR \*       h\_name;

    char FAR \* FAR \* h\_aliases;

    short            h\_addrtype;

    short            h\_length;

    char FAR \* FAR \* h\_addr\_list;

};

 The h\_name field is the official name of the host. If your network uses the DNS, it is the FQDN that causes the name server to return a reply. If your network uses a local “hosts” file, it is the first entry after the IP address. The h\_aliases field is a null-terminated array of alternative names for the host. The h\_addrtype represents the address family being returned. The h\_length field defines the length in bytes of each address in the h\_addr\_list field, which will be four bytes for IPv4 addresses. The h\_addr\_list field is a null-terminated array of IP addresses for the host. (A host can have more than one IP address assigned to it.) Each address in the array is returned in network-byte order.

Normally, applications use the first address in the array. However, if more than one address is returned, applications should randomly choose an available address rather than always use the first address. The prototypes for these functions are:

|  |
| --- |
| struct hostent FAR \* gethostbyname (const char FAR \* name); |
| HANDLE WSAAsyncGetHostByName(      HWND hWnd,      unsigned int wMsg,      const char FAR \* name,      char FAR \* buf,      int buflen  ); |
| struct HOSTENT FAR \* gethostbyaddr(      const char FAR \* addr,      int len,      int type  ); |
| HANDLE WSAAsyncGetHostByAddr(                          HWND hWnd,                          unsigned int wMsg,                          const char FAR \*addr,                          int len,                          int type,                          char FAR \*buf,                          int buflen  ); |

 Take note that gethostbyname() and gethostbyaddr() functions considered deprecated, instead use getaddrinfo() function to provide smooth transition to IPv6 compatibility

For the first two functions, the name parameter represents a friendly name of the host you are looking for, and the latter two functions take an IPv4 network address in the addr parameter. The length of the address is specified as len. Also, type indicates the address family of the network address passed, which would be AF\_INET. All four functions return the results via a HOSTENT structure. For the two synchronous functions, the HOSTENT is a system-allocated buffer that the application should not rely on being static. The two asynchronous functions will copy the HOSTENT structure to the buffer indicated by the buf parameter. This buffer size should be equal to MAXGETHOSTSTRUCT.

Finally, these and the rest of the asynchronous name and service resolution functions return a HANDLE identifying the operation issued. Upon completion, a window message indicated by wMsg is posted to the window given by hWnd. If at some point the application wishes to cancel the asynchronous request, the WSACancelAsyncRequest() function is used. This function is declared as:

 int WSACancelAsyncRequest(HANDLE hAsyncTaskHandle);

 Keep in mind that the synchronous API calls will block until the query completes or times out, which could take several seconds. The following program example demonstrates the use of gethostbyname() and gethostbyaddr() functions.

 #include <winsock2.h>

#include <ws2tcpip.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

    // Declare and initialize variables

    WSADATA wsaData;

    DWORD dwError;

    int i = 0;

    struct hostent \*remoteHost;

    char \*host\_name;

    struct in\_addr addr;

    char \*\*pAlias;

    // Validate the parameters

    if (argc != 2)

    {

        printf("Usage: %s <ipv4address> or <hostname>\n", argv[0]);

        printf("Example: %s 127.0.0.1\n", argv[0]);

        printf("  to return the host\n");

        printf("Example: %s www.palerbutuh.com\n", argv[0]);

        printf("  to return the IP addresses for a host\n");

        return 1;

    }

    // Initialize Winsock

    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)

    {

        printf("WSAStartup() failed with error code %ld\n", WSAGetLastError());

        return 1;

    }

    else

        printf("WSAStartup() looks fine!\n");

            host\_name = argv[1];

            // If the user input is an alpha name for the host, use gethostbyname()

            // If not, gethostbyaddr()

            // We are assuming the IPv4 here

            if (isalpha(host\_name[0]))    // host address is a name instead og IP

            {

                        printf("Calling gethostbyname() with %s\n", host\_name);

                        remoteHost = gethostbyname(host\_name);

            }

            else

            {

                        // host address is an IP

                        printf("Calling gethostbyaddr() with %s\n", host\_name);

                        addr.s\_addr = inet\_addr(host\_name);

                        if (addr.s\_addr == INADDR\_NONE)

                        {

                                    printf("The IPv4 address entered must be a legal address!\n");

                                    WSACleanup();

                                    return 1;

                        }

                        else

                        {

                                    // gethostbyaddr() is OK

                                    // Should provide both the Ipv6 and IPv4. Just IPv4 shown here

                                    remoteHost = gethostbyaddr((char \*) &addr, 4, AF\_INET);

                        }

    }

    if (remoteHost == NULL)

    {

                        dwError = WSAGetLastError();

                        if (dwError != 0)

                        {

                                    if (dwError == WSAHOST\_NOT\_FOUND)

                                    {

                                                printf("Host not found!\n");

                                                WSACleanup();

                                                return 1;

                                    }

                                    else if (dwError == WSANO\_DATA)

                                    {

                                                printf("No data record found!\n");

                                                WSACleanup();

                                                return 1;

                                    }

                                    else

                                    {

                                            printf("Function failed with error code %ld\n", dwError);

                                            WSACleanup();

                                            return 1;

                                    }

                         }

    }

    else

    {

        printf("Function returned successfully!\n");

        printf("\tOfficial name: %s\n", remoteHost->h\_name);

        // Extract all the cname/aliases if any

        for (pAlias = remoteHost->h\_aliases; \*pAlias != 0; pAlias++)

        {

            printf("\tAlternate name (if any) #%d: %s\n", ++i, \*pAlias);

        }

                        // Check the address family type

                        printf("\tAddress type: ");

                        switch (remoteHost->h\_addrtype)

                        {

                        case AF\_INET:

                                    printf("AF\_INET family\n");

                                    break;

                        case AF\_INET6:

                                    printf("AF\_INET6 family\n");

                                    break;

                        case AF\_NETBIOS:

                                    printf("AF\_NETBIOS - Windows NetBIOS\n");

                                    break;

                        default:

                                    printf(" %d\n", remoteHost->h\_addrtype);

                                    break;

                        }

                        // Check the address family type

                        printf("\tAddress length: %d\n", remoteHost->h\_length);

                        // Traverse the hostent address list and print the IP address(s)

                        i = 0; // Reset counter

                        while (remoteHost->h\_addr\_list[i] != 0)

                        {

                                    addr.s\_addr = \*(u\_long \*)remoteHost->h\_addr\_list[i++];

                                    printf("\tIP Address #%d: %s\n", i, inet\_ntoa(addr));

                         }

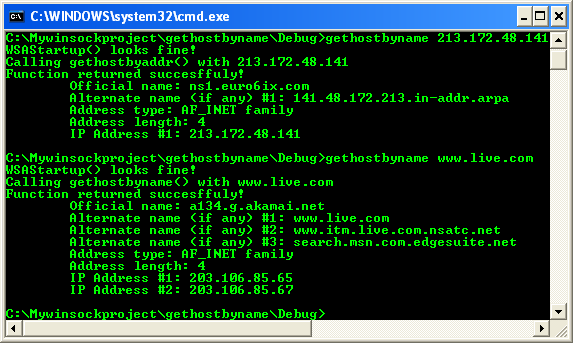
    }

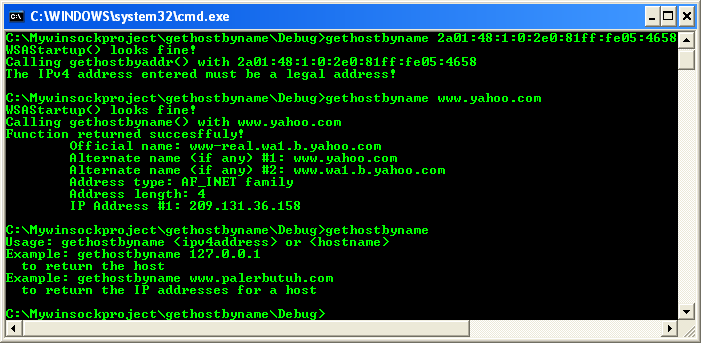
            WSACleanup();

    return 0;

}

The following screenshots show sample outputs using IPv4 and IPv6 addresses as the arguments.





 The next type of legacy resolution functions provide the capability to retrieve port numbers for well-known services and the reverse. The API functions getservbyname() and WSAAsyncGetServByName() take the name of a well-known service like “FTP” and return the port number that the service uses. The functions getservbyport() and WSAAsyncGetServByPort() perform the reverse operation by taking the port number and returning the service name that uses that port. These functions simply retrieve static information from a file named services. In Windows 95, Windows 98, and Windows Me, the services file is located under %WINDOWS%; in Windows NT, it is located under %WINDOWS%\System32\drivers\etc. These four functions return the service information in a SERVENT structure that is defined as:

 struct servent {

                        char FAR \*        s\_name;

                        char FAR \* FAR \*s\_aliases;

                        short                   s\_port;

                        char FAR \*        s\_proto

};

 The field s\_name is the name of the service and s\_aliases is a NULL terminated array of string pointers, each containing another name for the service. s\_port is the port number used by the service and s\_proto is the protocol used by the service, such as the strings “tcp” and “udp.” These functions are defined as follows:

|  |
| --- |
| struct servent FAR \* getservbyname(      const char FAR \* name,      const char FAR \* proto  ); |
| HANDLE WSAAsyncGetServByName(                          HWND hWnd,                          unsigned int wMsg,                          const char FAR \*name,                          const char FAR \*proto,                          char FAR \*buf,                          int buflen  ); |
| struct servent FAR \*getservbyport(                          int port,                          const char FAR \*proto  ); |
| HANDLE WSAAsyncGetServByPort(                          HWND hWnd,                          unsigned int wMsg,                          int port,                          const char FAR \*proto,                          char FAR \*buf,                          int buflen  ); |

 The name parameter represents the name of the service you are looking for. The proto parameter optionally points to a string that indicates the protocol that the service in name is registered under, such as “tcp” or “udp”. The second two functions simply take the port number to match to a service name. The synchronous API functions return a SERVENT structure, which is a system allocated buffer, and the asynchronous ones take an application supplied buffer, which should also be of the size MAXGETHOSTSTRUCT.

The last set of legacy name resolution API functions convert between a protocol string name, such as “tcp”, and its protocol number (“tcp” would resolve to IPPROTO\_TCP.). These functions are getprotobyname(), WSAAsyncGetProtoByName(), getprotobynumber(), and WSAAsyncGetProtoByNumber(). The first two convert from the string protocol to the protocol number and the latter two do the opposite, map the protocol number back to its string name. These functions return a PROTOENT structure defined as:

 struct protoent {

                        char FAR \*        p\_name;

                        char FAR \* FAR \*p\_aliases;

                        short                   p\_proto;

};

 The first field, p\_name, is the string name of the protocol, and p\_aliases is a NULL terminated array of string pointers that contain other names the protocol is known by. Finally, p\_proto is the protocol number (such as IPPROTO\_UDP or IPPROTO\_TCP). These function prototypes are:

|  |
| --- |
| struct protoent FAR \*getprotobyname(const char FAR \*name); |
| HANDLE WSAAsyncGetProtoByName(                          HWND hWnd,                          unsigned int wMsg,                          const char FAR \*name,                          char FAR \*buf,                          int buflen  ); |
| struct protoent FAR \*getprotobynumber(int number); |
| HANDLE WSAAsyncGetProtoByNumber(                          HWND hWnd,                          unsigned int wMsg,                          int number,                          char FAR \*buf,                          int buflen  ); |

 These functions behave the same way the legacy name resolution functions do described earlier in terms of synchronous and asynchronous functions. The following example demonstrates the use of getservbyname() function.

 #include <winsock2.h>

#include <stdio.h>

#include <string.h>

int main(int argc, char \*\*argv)

{

    // Declare and initialize variables

    WSADATA wsaData;

    char node\_name[100] = "";

    struct hostent \*hostptr;

    struct servent \*servptr;

    // http://www.iana.org/assignments/port-numbers

    char services[12][10] = {"http", "ftp", "ssh", "telnet", "imap4", "smtp", "gopher", "finger", "pop3", "tftp", "mysql"};

    int i;

    // Validate the parameters

    if (argc != 2)

    {

        printf("Usage: %s <host\_name>\n", argv[0]);

        printf("Example: %s www.yahoo.com\n", argv[0]);

        // Just return with error

        return 1;

    }

    // Initialize Winsock

    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)

    {

        printf("WSAStartup() failed with error code %ld\n", WSAGetLastError());

        return 1;

    }

    else

        printf("WSAStartup() looks fine!\n");

            strcpy\_s(node\_name, sizeof(node\_name), argv[1]);

            // Here, name of server computer is expected, better to add

            // the server IP address functionality...

            hostptr = gethostbyname(node\_name);

            if (hostptr == NULL)

            {

                        printf("gethostbyname() failed with error code %ld\n", WSAGetLastError());

             }

            else

                        printf("gethostbyname() is OK.\n Host name is %s\n", hostptr->h\_name);

            for(i=0; i <= 10;i++)

            {

                        // The proto pointer is NULL, the getservbyname() will returns

                        // the first service entry where name matches the s\_name member

                        // of the servent struct or the s\_aliases member of the servent struct.

                        servptr = getservbyname(services[i], NULL);

                        if (servptr == NULL)

                                    printf ("\ngetservbyname() - %s failed with error code %ld\n", services[i], WSAGetLastError());

                        else

                        {

                                    // Print the related info

                                    printf("\ngetservbyname() is OK - %s\n", services[i]);

                                    printf(" Service name is %s\n", servptr->s\_name);

                                    // Need to cast to host-byte order

                                    printf(" Port number is %u\n", ntohs(servptr->s\_port));

                                    printf(" Protocol is %s\n", servptr->s\_proto);

                        }

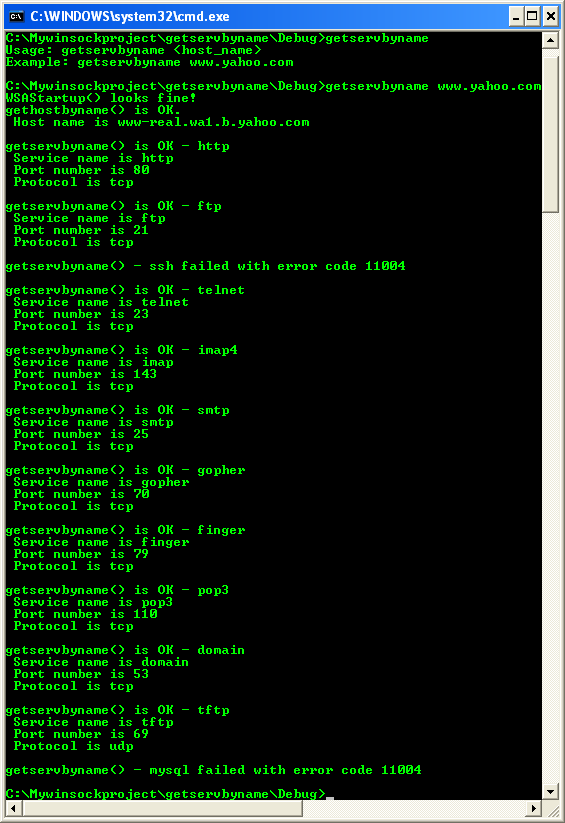
            }

            WSACleanup();

    return 0;

}

 Build and run the project. The following screenshot shows a sample output.



The 11004 error code is WSANO\_DATA which is defined as “Valid name, no data record of requested type.” This error possibly indicates the requested name is valid and was found in the database, but it does not have the correct associated data being resolved for. In this example case, it is normal. The following example demonstrates the use of getservbyport() function.

 #include <winsock2.h>

#include <stdio.h>

#include <string.h>

int main(int argc, char \*\*argv)

{

    // Declare and initialize variables

    WSADATA wsaData;

    char node\_name[100] = "";

    struct hostent \*hostptr;

    struct servent \*servptr;

    // http://www.iana.org/assignments/port-numbers

    int services[12] = {23, 80, 25, 21, 53, 69, 110, 143, 70, 7, 13};

    int i;

    // Validate the parameters

    if (argc != 2)

    {

        printf("Usage: %s <host\_name>\n", argv[0]);

        printf("Example: %s www.yahoo.com\n", argv[0]);

        // Just return with error

        return 1;

    }

    // Initialize Winsock

    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)

    {

        printf("WSAStartup() failed with error code %ld\n", WSAGetLastError());

        return 1;

    }

    else

        printf("WSAStartup() looks fine!\n");

            strcpy\_s(node\_name, sizeof(node\_name), argv[1]);

            // Here, name of server computer is expected, it is better

            // to add the server IP address functionality...

            hostptr = gethostbyname(node\_name);

            if (hostptr == NULL)

            {

                        printf("gethostbyname() failed with error code %ld\n", WSAGetLastError());

            }

            else

                        printf("gethostbyname() is OK.\n Host name is %s\n", hostptr->h\_name);

            for(i=0; i <= 10;i++)

            {

                        // If this is null, getservbyport returns the first service

                        // entry for which the port matches the s\_port of the servent structure.

                        servptr = getservbyport(ntohs(services[i]), NULL);

                        if (servptr == NULL)

                                    printf ("\ngetservbyport() - %d failed with error code %ld\n", services[i], WSAGetLastError());

                        else

                        {

                                    // Print the related info

                                    printf("\ngetservbyport() is OK - %d\n", services[i]);

                                    printf(" Service name is %s\n", servptr->s\_name);

                                    // Need to cast to host-byte order

                                    printf(" Port number is %u\n", ntohs(servptr->s\_port));

                                    printf(" Protocol is %s\n", servptr->s\_proto);

                        }

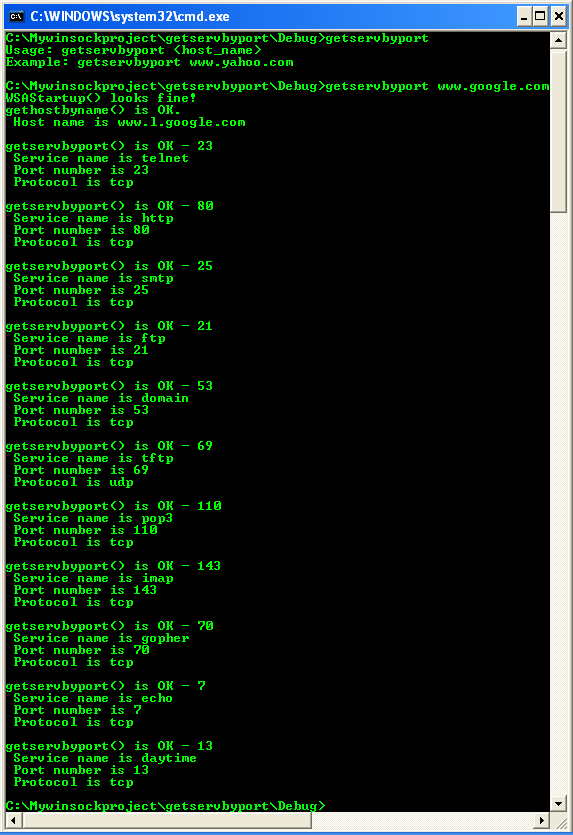
            }

            WSACleanup();

    return 0;

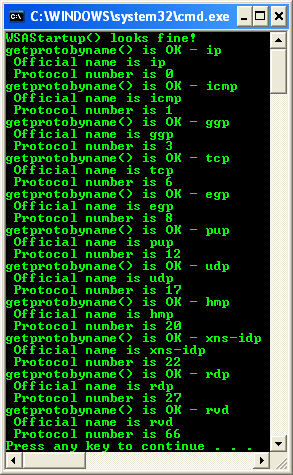
}

Build and run the project. The following screenshot shows a sample output.



|  |
| --- |
| The getprotobyname() Program Example    The following program example tries to demonstrate the getprotobyname() function. Create a new empty Win32 console mode application and add the project/solution name.    #include <winsock2.h>  #include <stdio.h>    int main(int argc, char \*\*argv)  {      // Declare and initialize variables      WSADATA wsaData;      struct protoent \*servptr;      // http://www.iana.org/assignments/port-numbers      char protocol[12][10] = {"ip","icmp","ggp","tcp", "egp","pup","udp","hmp","xns-idp", "rdp","rvd" };      int i;        // Initialize Winsock      if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)              {          printf("WSAStartup() failed with error code %ld\n", WSAGetLastError());          return 1;      }              else                          printf("WSAStartup() looks fine!\n");                // Traverse the array content...              // Be careful with the array index...              for(i=0; i <= 10;i++)              {                          servptr = getprotobyname(protocol[i]);                            if (servptr == NULL)                                      printf ("\ngetprotobyname() - %d failed with error code %ld\n", protocol[i], WSAGetLastError());                          else                          {                                      // Print the related info                                      printf("getprotobyname() is OK - %s\n", protocol[i]);                                      printf(" Official name is %s\n", servptr->p\_name);                                      printf(" Protocol number is %d\n", servptr->p\_proto);                          }              }                WSACleanup();      return 0;  } |

Build and run the project. The following screenshot shows a sample output.



The getprotobyport() Program Example

The following program example tries to demonstrate the getprotobyport() function.

#include <winsock2.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

    // Declare and initialize variables

    WSADATA wsaData;

    struct protoent \*servptr;

    // http://www.iana.org/assignments/protocol-numbers/

    // A sample of protocol numbers

    int protocol[12] = {1,2,3,11,30,20,22,27,8,17,12};

    int i;

    // Initialize Winsock

    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)

    {

        printf("WSAStartup() failed with error code %ld\n", WSAGetLastError());

        return 1;

    }

    else

        printf("WSAStartup() looks fine!\n");

            // Traverse the array content...

            // Be careful with the array index...

            for(i=0; i <= 10;i++)

            {

                        servptr = getprotobynumber(protocol[i]);

                        if (servptr == NULL)

                                    printf ("getprotobynumber() - %d failed with error code %ld\n", protocol[i], WSAGetLastError());

                        else

                        {

                                    // Print the related info

                                    printf("getprotobynumber() is OK - %d\n", protocol[i]);

                                    printf(" Official name is %s\n", servptr->p\_name);

                                    printf(" Protocol number is %d\n", servptr->p\_proto);

                        }

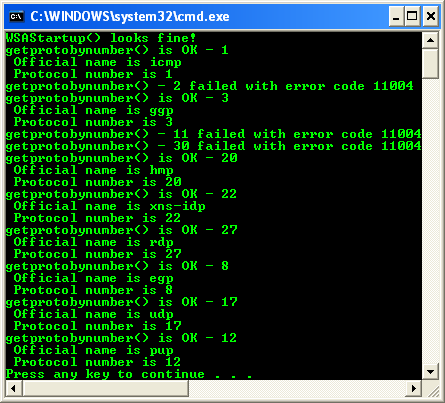
            }

            WSACleanup();

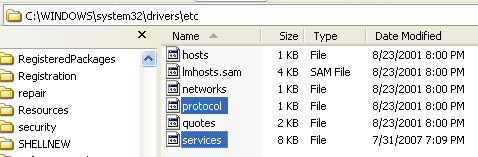
    return 0;

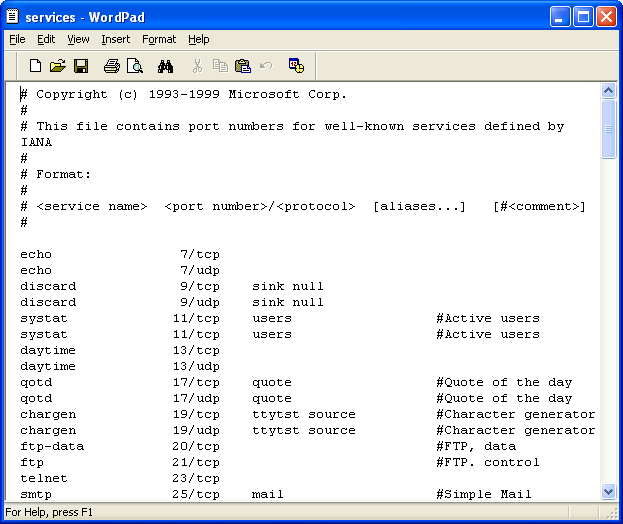
}

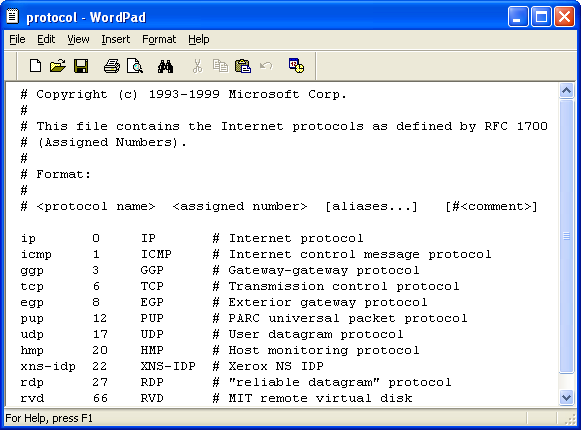
Build and run the project. The following screenshot shows a sample output.



As discussed previously, if you are curious where those protocol names and numbers are extracted in the local system, the files containing the information are under the %system32/driver/etc folder (Windows XP Pro SP2). You can open those files in text editor.







Writing IP Version–Independent Program Example

In this section, we'll cover how to develop applications that work seamlessly over IPv4 and IPv6. This method requires using the new name resolution APIs getaddrinfo() and getnameinfo() and requires a bit of rearranging Winsock calls from what you are probably used to.

Before we get into the specifics, let's cover some of the basic practices that you should follow. First, applications should not allocate the socket address structures specific to each protocol (such as SOCKADDR\_IN and SOCKADDR\_IN6 for IPv4 and IPv6, respectively) because they can be different sizes. Instead, a new socket address structure SOCKADDR\_STORAGE has been introduced that is as large as the largest possible protocol specific address structure and includes padding for 64-bit alignment issues. The following code uses a SOCKADDR\_STORAGE structure to store the destination IPv6 address.

SOCKADDR\_STORAGE                saDestination;

SOCKET                                            s;

int                                                        addrlen, rc;

s = socket(AF\_INET6, SOCK\_STREAM, IPPROTO\_TCP);

if (s == INVALID\_SOCKET)

{

            // socket failed

}

addrlen = sizeof(saDestination);

rc = WSAStringToAddress(

                                    "3ffe:2900:d005:f28d:250:8bff:fea0:92ed",

                                    AF\_INET6,

                                    NULL,

                                    (SOCKADDR \*)&saDestination,

                                    &addrlen

                                    );

if (rc == SOCKET\_ERROR)

{

            // conversion failed

}

rc = connect(s, (SOCKADDR \*)&saDestination, sizeof(saDestination));

if (rc == SOCKET\_ERROR)

{

            // connect failed

}

Second, functions that take an address as a parameter should pass the entire socket address structure and not the protocol specific types like struct in\_addr or struct in6\_addr. This is important for IPv6, which might require the scope ID information to successfully connect. The SOCKADDR\_STORAGE structure containing the address should be passed instead.

Third, avoid hardcode addresses regardless of whether they are IPv4 or IPv6. The Winsock header files define constants for all the address that are hard coded such as the loopback address and the wildcard address used for binding.

Now that some of the basic issues are out of the way, let's move to discussing how an application should be structured to be IP independent. We will divide our discussion into two sections: the client and the server.

The Client Example

For both TCP and UDP clients, the application typically possesses the server (or recipient's) IP address or hostname. Whether it resolves to an IPv4 address or IPv6 address doesn't matter. The client should follow these three steps:

1. Resolve the address using the getaddrinfo() function. The hints should contain AF\_UNSPEC as well as the socket type and protocol depending on whether the client uses TCP or UDP to communicate.
2. Create the socket using the ai\_family, ai\_socktype, and ai\_protocol fields from the addrinfo structure returned in step 1.
3. Call connect() or sendto() with the ai\_addr member of the addrinfo structure.

The following code sample illustrates these principles.

SOCKET                    s;

struct addrinfo            hints, \*res=NULL;

char                             \*szRemoteAddress=NULL, \*szRemotePort=NULL;

int                                rc;

// Parse the command line to obtain the remote server's

// hostname or address along with the port number, which are contained

// in szRemoteAddress and szRemotePort.

memset(&hints, 0, sizeof(hints));

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

// first resolve assuming string is a string literal address

rc = getaddrinfo(szRemoteAddress, szRemotePort, &hints, &res);

if (rc == WSANO\_DATA)

{

            // Unable to resolve name - bail out

}

s = socket(res->ai\_family, res->ai\_socktype, res->ai\_protocol);

if (s == INVALID\_SOCKET)

{

            // socket failed

}

rc = connect(s, res->ai\_addr, res->ai\_addrlen);

if (rc == SOCKET\_ERROR)

{

            // connect failed

}

freeaddrinfo(res);

First, you will notice that there are no explicit references to AF\_INET or AF\_INET6. Also, there's no need to manipulate the underlying SOCKADDR\_IN or SOCKADDR\_IN6 addresses. The getaddrinfo() call fully initializes the returned socket address structure with all the required information - address family, binary address, etc., that is necessary for connecting or sending datagrams.

If the client application needs to explicitly bind the socket to a local port after socket creation but before connect() or sendto(), then another getaddrinfo() call can be made. This call would specify the address family, socket type, and protocol returned from the first call along with the AI\_PASSIVE flag and desired local port, which will return another socket address structure initialized to the necessary bind address (such as 0.0.0.0 for IPv4 and :: for IPv6). The following program example shows the working version.

#include <winsock2.h>

#include <Ws2tcpip.h>

#include <stdio.h>

#include <string.h>

int main(int argc, char \*\*argv)

{

    // Declare and initialize variables

    WSADATA wsaData;

            SOCKET                    s;

            struct addrinfo            hints, \*res=NULL;

            char                             \*szRemoteAddress=NULL, \*szRemotePort=NULL;

            int                                rc;

            char                             sendbuf[1024] = "This is a test string from client!";

            int                                BytesSent;

    // Validate the parameters

    if (argc != 3)

    {

        printf("Usage: %s <remote\_host\_name> <port\_number>\n", argv[0]);

        printf("Example: %s www.bodopiang.com 7777\n", argv[0]);

        // Just return with error

        return 1;

    }

    // Initialize Winsock

    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)

    {

        printf("Client: WSAStartup() failed with error code %ld\n", WSAGetLastError());

        return 1;

    }

            else

                        printf("Client: WSAStartup() looks fine!\n");

            // Parse the command line to obtain the remote server's hostname

            // or IP address along with the port number, which are contained

            // in szRemoteAddress and szRemotePort.

            memset(&hints, 0, sizeof(hints));

            hints.ai\_family = AF\_UNSPEC;

            hints.ai\_socktype = SOCK\_STREAM;

            hints.ai\_protocol = IPPROTO\_TCP;

            szRemoteAddress = argv[1];

            szRemotePort = argv[2];

            // first resolve assuming string is a string literal address

            // The addrinfo structure is used by the ANSI getaddrinfo()

            // function to hold host address information. The addrinfoW()

            // structure is the version of this structure used by the

            // Unicode GetAddrInfoW() function

            // res struct contains info of the remote response

            rc = getaddrinfo(szRemoteAddress, szRemotePort, &hints, &res);

            if (rc != 0)

            {

                        printf("Client: getaddrinfo() failed miserably with error code %ld\n", WSAGetLastError());

                        WSACleanup();

                        return 1;

            }

            else

                        printf("Client: getaddrinfo() seems fine!\n");

            // The res struct info

            // http://msdn.microsoft.com/en-us/library/ms737530(VS.85).aspx

            printf("\n The getaddrinfo() options used: %d\n", res->ai\_flags);

            printf(" The address family: %d\n", res->ai\_family);

            printf(" The socket type: %d\n", res->ai\_socktype);

            printf(" The protocol type: %d\n", res->ai\_protocol);

            // The hints struct info

            printf("\n The hints.ai\_family: %d\n", hints.ai\_family);

            printf(" The hints.ai\_socktype: %d\n", hints.ai\_socktype);

            printf(" The hints.ai\_protocol: %d\n", hints.ai\_protocol);

            // Use the res info to create a socket

            s = socket(res->ai\_family, res->ai\_socktype, res->ai\_protocol);

            if (s == INVALID\_SOCKET)

            {

                        printf("\nClient: socket() failed miserably with error code %ld\n", WSAGetLastError());

                        WSACleanup();

                        return 1;

            }

            else

                        printf("\nClient: socket() is OK\n");

            // Then use the res to connect

            rc = connect(s, res->ai\_addr, res->ai\_addrlen);

            if (rc == SOCKET\_ERROR)

            {

                        printf("Client: connect() failed miserably with error code %ld\n", WSAGetLastError());

                        WSACleanup();

                        return 1;

            }

            else

                        printf("Client: connect() is OK\n");

            printf("Client: I\'m ready to send or receive data...\n");

            // Sends some data to server/receiver...

            printf(" Sending: \"%s\"\n", sendbuf);

            BytesSent = send(s, sendbuf, strlen(sendbuf), 0);

            if(BytesSent == SOCKET\_ERROR)

            {

                        printf("Client: send() failed with error code %ld.\n", WSAGetLastError());

                        closesocket(s);

                        WSACleanup();

                        return 1;

            }

            else

                        printf("Client: send() is OK - bytes sent: %ld\n", BytesSent);

            // If the socket type is TCP - IPPROTO\_TCP, do the shutdown

            if(res->ai\_protocol == 6)

            {

                        if(shutdown(s, SD\_SEND) != 0)

                        {

                                    printf("Client: shutdown() failed with error code: %ld\n", WSAGetLastError());

                                    WSACleanup();

                                    return 1;

                        }

                        else

                                    printf("Client: shutdown() the sending part...\n");

            }

            // Deallocate resource

            freeaddrinfo(res);

            // Close socket

            if(closesocket(s) != 0)

            {

                        printf("Client: Cannot close s socket. Error code: %ld\n", WSAGetLastError());

                        WSACleanup();

                        return 1;

            }

            else

                        printf("Client: Closing s socket...\n");

            // Clean up the WSA

            if(WSACleanup() != 0)

            {

                        printf("Client: WSACleanup() failed!...\n");

                        return 1;

            }

            else

                        printf("Client: WSACleanup() is OK...\n");

    return 0;

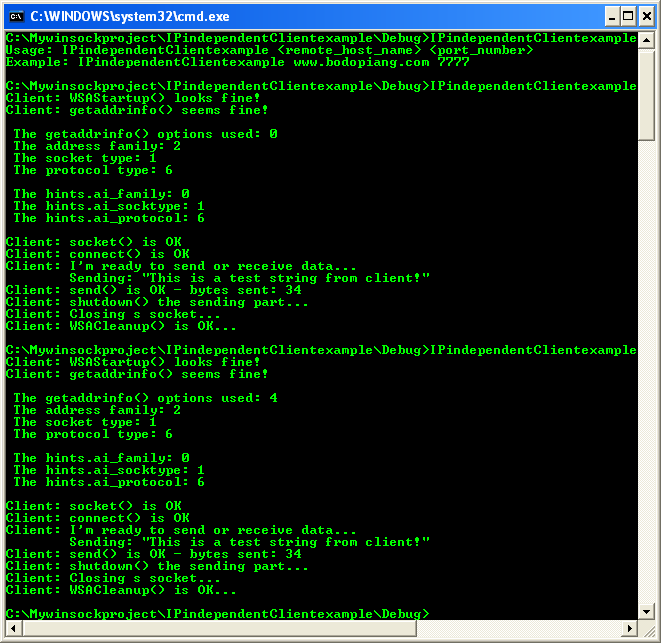
}

The following screenshot shows a sample output using the following arguments:

*IPindependentClientexample www.yahoo.com 80*

*IPindependentClientexample 216.239.61.104 80*

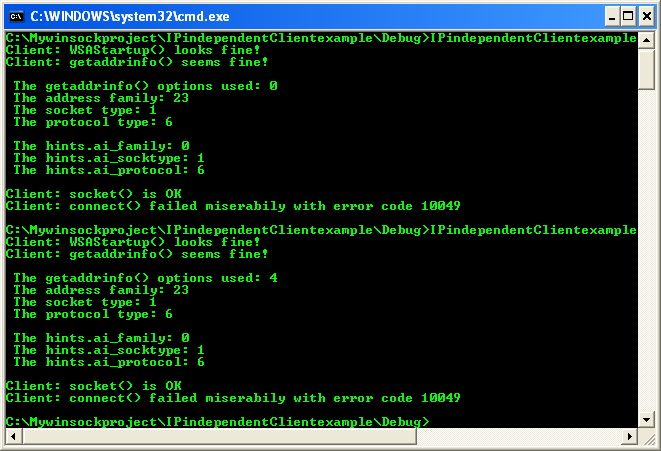
--------------------------------------------------------------



The following screenshot shows a sample output using the following arguments:

*IPindependentClientexample ipv6.internet2.edu 80*

*IPindependentClientexample 2001:1888:0:1:2d0:b7ff:fe7d:bed6 80*



|  |
| --- |
| The Server Example    The server side is a bit more involved than the client side. This is because the Windows IPv6 stack is a dual stack. That is, there is a separate stack for IPv4 and IPv6, so if a server wishes to accept both IPv4 and IPv6 connections, it must create a socket for each one. The two steps for creating an IP independent server are the following:     1. Call getaddrinfo() with hints containing AI\_PASSIVE, AF\_UNSPEC, and the desired socket type and protocol along with the desired local port to listen or receive data on. This will return two addrinfo structures: one containing the listening address for IPv4 and the other containing the listening address for IPv6. 2. For every addrinfo structure returned, create a socket with the ai\_family, ai\_socktype, and ai\_protocol fields followed by calling bind() with the ai\_addr and ai\_addrlen members.     The following code illustrates this principle.    SOCKET                                slisten[16];  char                                         \*szPort="5150";  struct addrinfo                        hints, \* res=NULL, \* ptr=NULL;  int                                            count=0, rc;    memset(&hints, 0, sizeof(hints));  hints.ai\_family = AF\_UNSPEC;  hints.ai\_socktype = SOCK\_STREAM;  hints.ai\_protocol = IPPROTO\_TCP;  hints.ai\_flags = AI\_PASSIVE;    rc = getaddrinfo(NULL, szPort, &hints, &res);    if (rc != 0)  {              // failed for some reason  }  ptr = res;  while (ptr)  {              slisten[count] = socket(ptr->ai\_family, ptr->ai\_socktype, ptr->ai\_protocol);                if (slisten[count] == INVALID\_SOCKET)              {                          // socket failed              }                rc = bind(slisten[count], ptr->ai\_addr, ptr->ai\_addrlen);              if (rc == SOCKET\_ERROR)              {                          // bind failed              }                rc = listen(slisten[count], 7);              if (rc == SOCKET\_ERROR)              {                          // listen failed              }              count++;              ptr = ptr->ai\_next;  } |

Once the sockets are created and bound, the application simply needs to wait for incoming connections on each. The following program example tries to demonstrate the server part.

#include <winsock2.h>

#include <Ws2tcpip.h>

#include <stdio.h>

#include <string.h>

int main(int argc, char \*\*argv)

{

    // Declare and initialize variables

    WSADATA            wsaData;

    SOCKET        slisten[16], NewConnection;

    char     \*szPort = "7777";

    struct addrinfo            hints, \* res=NULL, \* ptr=NULL;

    int        count=0, rc;

    char     recvbuff[1024];

    int                    ByteReceived, i;

    // Initialize Winsock

    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)

    {

        printf("Server: WSAStartup() failed with error code %ld\n", WSAGetLastError());

        return 1;

    }

    else

        printf("Server: WSAStartup() looks fine!\n");

            memset(&hints, 0, sizeof(hints));

            hints.ai\_family = AF\_UNSPEC;

            hints.ai\_socktype = SOCK\_STREAM;

            hints.ai\_protocol = IPPROTO\_TCP;

            hints.ai\_flags = AI\_PASSIVE;

            rc = getaddrinfo(NULL, szPort, &hints, &res);

            if (rc != 0)

            {

                        printf("Server: getaddrinfo() failed with error code %ld\n", WSAGetLastError());

                        WSACleanup();

                        return 1;

            }

            else

                        printf("Server: getaddrinfo() is OK...\n");

            ptr = res;

            while (ptr)

            {

                        printf("\nServer: count value = %d\n", count);

                        // Use the res struct info for listening...

                        slisten[count] = socket(ptr->ai\_family, ptr->ai\_socktype, ptr->ai\_protocol);

                        if (slisten[count] == INVALID\_SOCKET)

                        {

                                    printf("Server: socket() failed with error code %ld\n", WSAGetLastError());

                                    WSACleanup();

                                    return 1;

                        }

                        else

                                    printf("Server: socket() is OK...\n");

                        // The res struct info

                        printf("\n The address family: %d\n", res->ai\_family);

                        printf(" The socket type: %d\n", res->ai\_socktype);

                        printf(" The protocol: %d\n\n", res->ai\_protocol);

                        // Then bind

                        rc = bind(slisten[count], ptr->ai\_addr, ptr->ai\_addrlen);

                        if (rc == SOCKET\_ERROR)

                        {

                                    printf("Server: bind() failed with error code %ld\n", WSAGetLastError());

                                    WSACleanup();

                                    return 1;

                        }

                        else

                                    printf("Server: bind() is OK...\n");

                        // Next, listen

                        rc = listen(slisten[count], 10);

                        if (rc == SOCKET\_ERROR)

                        {

                                    printf("Server: listen() failed with error code %ld\n", WSAGetLastError());

                                    WSACleanup();

                                    return 1;

                        }

                        else

                        {

                                    printf("Server: listen() is OK...\n");

                                    NewConnection = SOCKET\_ERROR;

                                    // While the NewConnection socket equal to SOCKET\_ERROR

                                    // which is always true in this case...

                                    while(NewConnection == SOCKET\_ERROR)

                                    {

                                                // Accept connection on the slisten[count] socket and assign

                                                // it to the NewConnection socket, let the slisten[count]

                                                // do the listening for more connection

                                                NewConnection = accept(slisten[count], NULL, NULL);

                                                printf("Server: accept() is OK...\n");

                                                printf("Server: New client got connected, ready to receive and send data...\n");

                                                // Wait for more connections by calling accept again on ListeningSocket (loop)

                                                // or start sending or receiving data on NewConnection.

                                                ByteReceived = recv(NewConnection, recvbuff, sizeof(recvbuff), 0);

                                                // When there is problem

                                                if ( ByteReceived == SOCKET\_ERROR )

                                                {

                                                            printf("Server: recv() failed with error code %ld\n", WSAGetLastError());

                                                            WSACleanup();

                                                            break;

                                                }

                                                else

                                                {

                                                            printf("Server: recv() is OK....\n");

                                                            // Print the received bytes. Take note that this is the total

                                                            // byte received, it is not the size of the declared buffer

                                                            printf("Server: Bytes received: %d\n", ByteReceived);

                                                            // Print what those bytes represent

                                                            printf("Server: Those bytes are: \"");

                                                            // Print the string only, discard other

                                                            // remaining 'rubbish' in the 1024 buffer size

                                                            for(i=0;i < ByteReceived;i++)

                                                                        printf("%c", recvbuff[i]);

                                                            printf("\"\n");

                                                }

                                    }

                        }

                        if(res->ai\_protocol == 6)

                        {

                                    printf("Doing the TCP shutdown on the receiving part...\n");

                                    shutdown(slisten[count], SD\_RECEIVE);

                        }

                        closesocket(slisten[count]);

                        count++;

                        ptr = ptr->ai\_next;

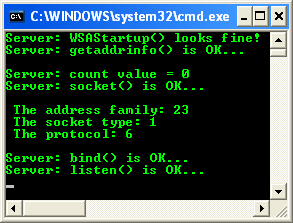
            }

            freeaddrinfo(res);

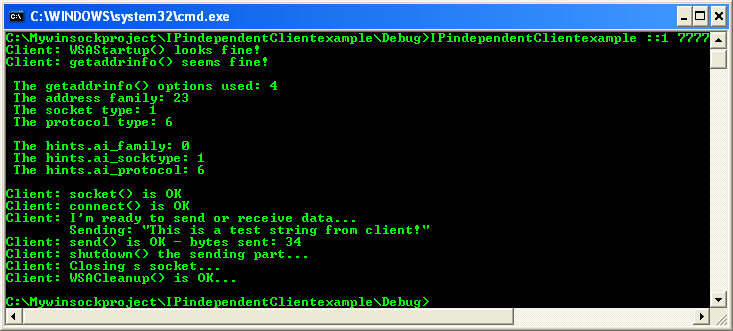
            WSACleanup();

            return 0;

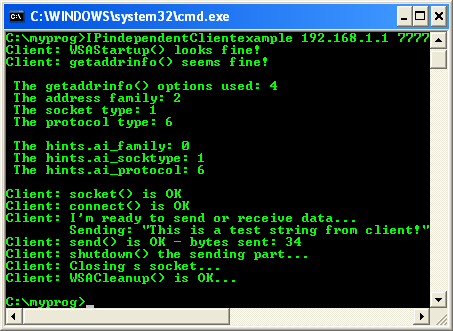
}



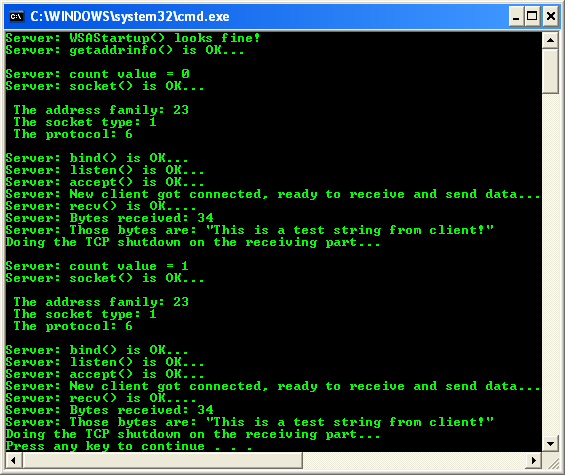
Then we run the previous client from the localhost and another peer computer in the same subnet. The ::1 is the IPv6 address for localhost (and make sure the computer is IPv6 enabled). Take note that more control should be implemented for the getaddrinfo() for example, if the address is numeric, getaddrinfo() should be disallowed from doing any name resolution. Otherwise a host name was provided that needs to be resolved when the getaddrinfo() call is issued



The following previous client example was run from another computer with IP address 192.168.1.2.



The following screenshot shows the server sample output when both client connections were completed.



Another Day, Another Example: The IPv6 Server Program

The server used to test this program example is IPv6 enabled and it will fail on IPv4 server. The setsockopt() and getpeername() functions are introduced in this example.

#include <winsock2.h>

#include <Ws2tcpip.h>

#include <stdio.h>

// Some constants used by this program

#define SERVER\_PORT     7171

#define BUFFER\_LENGTH    256

#define FALSE              0

int main(int argc, char \*\*argv)

{

            // Variable and structure declarations

            WSADATA    wsaData;

            int sd = -1, RetVal, sdconn = -1;

            int rc, on = 1, j = 0, rcdsize=BUFFER\_LENGTH;

            char buffer[BUFFER\_LENGTH];

            struct sockaddr\_in6 serveraddr, clientaddr;

            int addrlen = sizeof(struct sockaddr\_storage);

            WCHAR  MyAddrString[256] = L"Some dummy initializer";

            int str = sizeof(struct sockaddr\_storage);

            DWORD str2 = sizeof(struct sockaddr\_storage);

            // Initialize the use of the Winsock DLL by a process

            if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)

            {

                        printf("Server: WSAStartup() failed with error code %ld\n", WSAGetLastError());

                        return 1;

            }

            else

                        printf("Server: WSAStartup() looks fine!\n");

            // Using a loop for more 'efficient' code

            do

            {

                        // The socket() function returns a socket descriptor, which represents

                        // an endpoint.  Get a socket for address family AF\_INET6 to

                        // prepare to accept incoming connections on

                        if ((sd = socket(AF\_INET6, SOCK\_STREAM, 0)) == INVALID\_SOCKET)

                        {

                                    printf("Server: socket() failed with code error %ld\n", WSAGetLastError());

                                    break;

                        }

                        else

                                    printf("Server: socket() looks good!\n");

                        // The setsockopt() used to allow the local address to be reused when

                        // the server is restarted before the required wait time expires

                        if (setsockopt(sd, SOL\_SOCKET, SO\_REUSEADDR, (char \*)&on, sizeof(on)) == SOCKET\_ERROR)

                        {

                                    printf("Server: setsockopt(SO\_REUSEADDR) failed with error code %ld\n", WSAGetLastError());

                                    break;

                        }

                        else

                                    printf("Server: setsockopt(SO\_REUSEADDR) found OK!\n");

                        // After the socket descriptor is created, a bind() function gets a

                        // unique name for the socket.  In this example, the user sets the

                        // address to in6addr\_any, which (by default) allows connections to

                        // be established from any IPv4 or IPv6 client based on the hostname

                        // that specifies port 7171.

                        // That is, the bind is done to both the IPv4 and IPv6 TCP/IP

                        // stacks. However this sample program only accept the IPv4 hostname, then

                        // the client must prepare to convert the IPv4 address to the hostname

                        // before translating the IP string to network address before making a connection

                        // using various Winsock API such as getaddressinfo() etc.

                        memset(&serveraddr, 0, sizeof(serveraddr));

                        serveraddr.sin6\_family = AF\_INET6;

                        serveraddr.sin6\_port   = htons(SERVER\_PORT);

                        // Applications use in6addr\_any similarly to the way they use

                        // INADDR\_ANY in IPv4.

                        serveraddr.sin6\_addr   = in6addr\_any;

                        // The remaining fields in the sockaddr\_in6 are currently not

                        // supported and should be set to 0 to ensure upward compatibility

                        if (bind(sd, (struct sockaddr \*)&serveraddr, sizeof(serveraddr)) == SOCKET\_ERROR)

                        {

                                    printf("Server: bind() failed with error code %ld\n", WSAGetLastError());

                                    break;

                        }

                        else

                                    printf("Server: bind() also looks fine!\n");

                        // The listen() function allows the server to accept incoming

                        // client connections.  In this example, the backlog is set to 7

                        // This means that the system will queue 10 incoming connection

                        // requests before the system starts rejecting the incoming requests

                        if (listen(sd, 7) == SOCKET\_ERROR)

                        {

                                    printf("Server: listen() failed with error code %ld\n", WSAGetLastError());

                                    break;

                        }

                        printf("Server: listen() is OK and ready for client connect()...\n");

                        // The server uses the accept() function to accept an incoming

                        // connection request.  The accept() call will block indefinitely

                        // waiting for the incoming connection to arrive from an IPv4 or IPv6 client

                        if ((sdconn = accept(sd, NULL, NULL)) == INVALID\_SOCKET)

                        {

                                    printf("Server: accept() failed with error code %ld\n", WSAGetLastError());

                                    break;

                        }

                        else

                        {

                                    printf("Server: accept() return success!\n");

                                    // Display the client address.  Note that if the client is

                                    // an IPv4 client, the address will be shown as an IPv4 Mapped IPv6 address

                                    if(getpeername(sdconn, (struct sockaddr \*)&clientaddr, &addrlen) == SOCKET\_ERROR)

                                    {

                                                printf("Server: getpeername() failed miserably with error code %ld\n", WSAGetLastError());

                                    }

                                    else

                                                printf("Server: getpeername() found OK!\n");

                                    // converts an IPv4 or IPv6 Internet network address into a string in Internet standard

                                    // format. The ANSI version of this function is inet\_ntop() which is used in UNIX/BSD

                                    RetVal = WSAAddressToString((LPSOCKADDR)&clientaddr, addrlen, NULL, MyAddrString, &str);

                                    if(RetVal != 0)

                                                printf("Server: WSAAddressToString() failed with error code %ld\n", WSAGetLastError());

                                    else

                                    {

                                                printf("Server: Client address is %S\n", MyAddrString);

                                                printf("Server: Client port is %d\n", ntohs(clientaddr.sin6\_port));

                                    }

                        }

                        // In this example we know that the client will send 256 bytes of

                        // data over.  Knowing this, we can use the SO\_RCVTIMEO socket

                        // option and specify that we don't want our recv() to wake up

                        // until all 256 bytes of data have arrived

                        // http://msdn.microsoft.com/en-us/library/ms740476(VS.85).aspx

                        // Take note that some parameters are Microsoft specific

                        if (setsockopt(sdconn, SOL\_SOCKET, SO\_RCVTIMEO, (char \*)&rcdsize, sizeof(rcdsize)) == SOCKET\_ERROR)

                        {

                                    printf("Server: setsockopt(SO\_RCVTIMEO) failed with error code %ld\n", WSAGetLastError());

                                    break;

                        }

                        else

                                    printf("Server: setsockopt(SO\_RCVTIMEO) looks good!\n");

                        // Receive that 256 bytes of data from the client

                        rc = recv(sdconn, buffer, sizeof(buffer), 0);

                        if (rc == SOCKET\_ERROR)

                        {

                                    printf("Server: recv() failed with error code %ld\n", WSAGetLastError());

                                    break;

                        }

                        else

                        {

                                    printf("Server: recv() is OK...\n");

                        }

                        printf("Server: %d bytes of data were received\n", rc);

                        // Just for display!

                        printf("Server: Received money is - ");

                        for(j=0; buffer[j] == '$';j++)

                                    printf("%c",buffer[j]);

                        printf("\n");

                        if (rc == 0 || rc < sizeof(buffer))

                        {

                                    printf("Server: The client closed the connection before sending all the data!\n");

                                    break;

                        }

                        else

                        {

                                    printf("Server: All the data sent by client!\n");

                        }

                        // Echo the data back to the client

                        printf("Server: Echoing back the sent data to client...\n");

                        rc = send(sdconn, buffer, sizeof(buffer), 0);

                        if (rc == SOCKET\_ERROR)

                        {

                                    printf("Server: send() failed with error code %ld\n", WSAGetLastError());

                                    break;

                        }

                        else

                                    printf("Server: send() is OK...\n");

            } while (FALSE);

            // Program should complete here

            // Do the shutdown() for TCP so receive and send operation will no longer be allowed

            printf("Server: Ready to do the TCP shutdown()...\n");

            if(shutdown(sdconn, SD\_BOTH) == SOCKET\_ERROR)

                        printf("Server: shutdown sdconn socket and the other end failed with error %ld\n", WSAGetLastError());

            else

                        printf("Server: shutdown() on both end for sdconn socket looks OK...\n");

            // Close down any open socket descriptors,

            // BSD/UNIX uses close()

            if (closesocket(sd) == 0)

                        printf("Server: sd socket was closed...\n");

            if (closesocket(sdconn) == 0)

                        printf("Server: sdconn socket was closed...\n");

            // Terminates the use of the Ws2\_32.DLL for all threads (if any)

            printf("Server: Doing the WSACleanup()...\n");

            if( WSACleanup() == SOCKET\_ERROR)

                        printf("Server: WSACleanup() failed with error code %ld\n", WSAGetLastError());

            else

                        printf("Server: WSACleanup() confirmed OK!\n");

            return 0;

}

The following screenshot shows a sample output.

