<http://www.winsocketdotnetworkprogramming.com/winsock2programming/>

An Intro to Windows Socket Programming with C

Part 1

|  |
| --- |
| Winsock is a standard application programming interface (API) that allows two or more applications (or processes) to communicate either on the same machine or across a network and is primarily designed to foster data communication over a network. It is important to understand that Winsock is a network programming interface and not a protocol. Winsock provides the programming interface for applications to communicate using popular network protocols such as Transmission Control Protocol/Internet Protocol (TCP/IP) and Internetwork Packet Exchange (IPX). The Winsock interface inherits a great deal from the BSD Sockets implementation on UNIX platforms. In Windows environments, the interface has evolved into a truly protocol-independent interface, especially with the release of Winsock 2.  The examples presented in this chapter help to provide an understanding of the Winsock calls that are required for accepting connections, establishing connections, and sending and receiving data. Because the purpose of this chapter is to learn these fundamental Winsock calls, the examples presented use straight blocking Winsock calls.  You can differentiate the two functions with the WSA prefix. If Winsock 2 updated or added a new API function in its specification, the function name is prefixed with WSA. For example, the Winsock 1 function to create a socket is simply socket(). Winsock 2 introduces a newer version named WSASocket() that is capable of using some of the new features made available in Winsock 2. There are a few exceptions to this naming rule. WSAStartup(), WSACleanup(), WSARecvEx(), and WSAGetLastError() are in the Winsock 1.1 specification. If you fail to find the related Winsock headers in your machine or programming environment, you may want to read [Visual .NET doc 1](http://www.tenouk.com/Winsock/visualstudio2008ncnwinsock2.html) and [Visual .NET doc 2](http://www.tenouk.com/Winsock/visualstudio2008ncnwinsock2a.html) articles first. Those articles also show how to add the Winsock link library, ws2\_32.lib to the Visual C++ project.  **Winsock Headers and Libraries**   When developing new applications you should target the Winsock 2 specification by including **WINSOCK2.H** in your application. For compatibility with older Winsock applications and when developing on Windows CE platforms, **WINSOCK.H** is available. There is also an additional header file: **MSWSOCK.H**, which targets Microsoft-specific programming extensions that are normally used for developing high performance Winsock applications.  When compiling your application with WINSOCK2.H, you should link with **WS2\_32.LIB** library. When using WINSOCK.H (as on Windows CE) you should use **WSOCK32.LIB**. If you use extension APIs from MSWSOCK.H, you must also link with **MSWSOCK.LIB** (get the idea [here](http://www.tenouk.com/Winsock/visualstudio2008ncnwinsock2a.html)  - VC++ .NET or [here](http://www.tenouk.com/Winsock/Winsock2example.html) - VC++ 6). Once you have included the necessary header files and link environment, you are ready to begin coding your application, which requires initializing Winsock. |

**Initializing Winsock**

 Every Winsock application must load the appropriate version of the Winsock DLL. If you fail to load the Winsock library before calling a Winsock function, the function returns a SOCKET\_ERROR; the error will be WSANOTINITIALISED. Loading the Winsock library is accomplished by calling the WSAStartup() function, which is defined as:

 int WSAStartup(

    WORD wVersionRequested,

    LPWSADATA lpWSAData

);

 The wVersionRequested parameter is used to specify the version of the Winsock library you want to load. The high-order byte specifies the minor version of the requested Winsock library, while the low-order byte is the major version. You can use the handy macro MAKEWORD(x, y), in which x is the high byte and y is the low byte, to obtain the correct value for wVersionRequested. The lpWSAData parameter is a pointer to a LPWSADATA structure that WSAStartup() fills with information related to the version of the library it loads:

 typedef struct WSAData

{

    WORD           wVersion;

    WORD           wHighVersion;

    char           szDescription[WSADESCRIPTION\_LEN + 1];

    char           szSystemStatus[WSASYS\_STATUS\_LEN + 1];

    unsigned short iMaxSockets;

    unsigned short iMaxUdpDg;

    char FAR \*     lpVendorInfo;

} WSADATA, \* LPWSADATA;

 WSAStartup() sets the first field, wVersion, to the Winsock version you will be using. The wHighVersion parameter holds the highest version of the Winsock library available. Remember that in both of these fields, the high-order byte represents the Winsock minor version, and the low-order byte is the major version. The szDescription and szSystemStatus fields are set by the particular implementation of Winsock and aren't really useful. Do not use the next two fields, iMaxSockets and iMaxUdpDg. They are supposed to be the maximum number of concurrently open sockets and the maximum datagram size; however, to find the maximum datagram size you should query the protocol information through WSAEnumProtocols().

The maximum number of concurrent sockets isn't some magic number, it depends more on the physical resources available. Finally, the lpVendorInfo field is reserved for vendor-specific information regarding the implementation of Winsock. This field is not used on any Windows platforms.

For the most part, when writing new applications you will load the latest version of the Winsock library currently available. Remember that if, for example, Winsock 3 is released, your application that loads version 2.2 should run as expected. If you request a Winsock version later than that which the platform supports, WSAStartup() will fail. Upon return, the wHighVersion of the WSADATA structure will be the latest version supported by the library on the current system. When your application is completely finished using the Winsock interface, you should call WSACleanup(), which allows Winsock to free up any resources allocated by Winsock and cancel any pending Winsock calls that your application made. WSACleanup() is defined as:

 int WSACleanup(void);

 Failure to call WSACleanup when your application exits is not harmful because the operating system will free up resources automatically; however, your application will not be following the Winsock specification. Also, you should call WSACleanup for each call that is made to WSAStartup.

**Error Checking and Handling**

 We'll first cover error checking and handling, as they are vital to writing a successful Winsock application. It is actually common for Winsock functions to return an error; however, there are some cases in which the error is not critical and communication can still take place on that socket. The most common return value for an unsuccessful Winsock call is SOCKET\_ERROR, although this is certainly not always the case. When covering each API call in detail, we'll point out the return value corresponding to an error. The constant SOCKET\_ERROR actually is -1. If you make a call to a Winsock function and an error condition occurs, you can use the function WSAGetLastError() to obtain a code that indicates specifically what happened. This function is defined as:

 int WSAGetLastError (void);

 A call to the function after an error occurs will return an integer code for the particular error that occurred. These error codes returned from WSAGetLastError() all have predefined constant values that are declared in either WINSOCK.H or WINSOCK2.H, depending on the version of Winsock. The only difference between the two header files is that WINSOCK2.H contains more error codes for some of the newer API functions and capabilities introduced in Winsock 2. The constants defined for the various error codes (with #define directives) generally begin with WSAE. On the flip side of WSAGetLastError(), there is WSASetLastError(), which allows you to manually set error codes that WSAGetLastError() retrieves.

The following program demonstrates how to construct a skeleton Winsock application based on the discussion so far:

 #include <winsock2.h>

 void main(void)

{

   WSADATA wsaData;

   // Initialize Winsock version 2.2

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

   {

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

      return;

   }

   // Setup Winsock communication code here

   // When your application is finished call WSACleanup

   if (WSACleanup() == SOCKET\_ERROR)

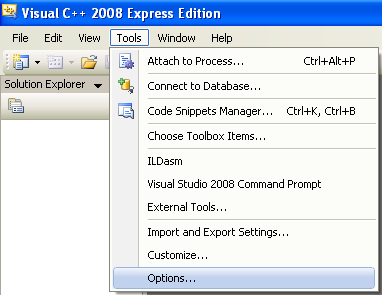
   {

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

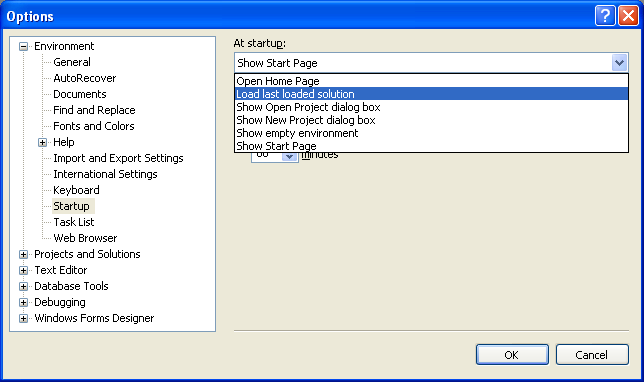
   }

}

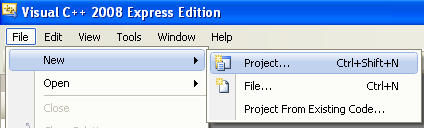
 Let try this program using Visual C++ 2008 Express Edition. First and foremost let change the newly installed VC++ startup page to last loaded solution. You can skip this ‘optional’ step. Click Tools menu > Options sub menu.



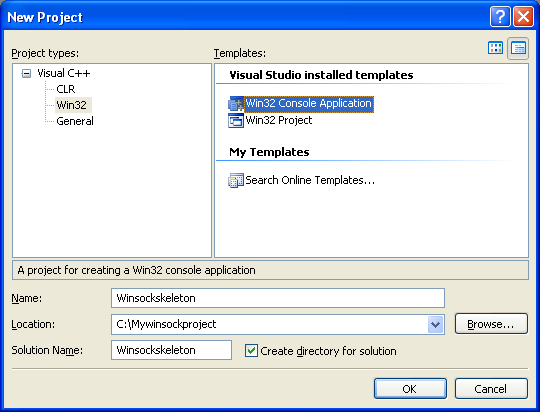
 Expand Environment folder > Select Startup link > Set the At Startup: to Load last loaded solution > Click OK.



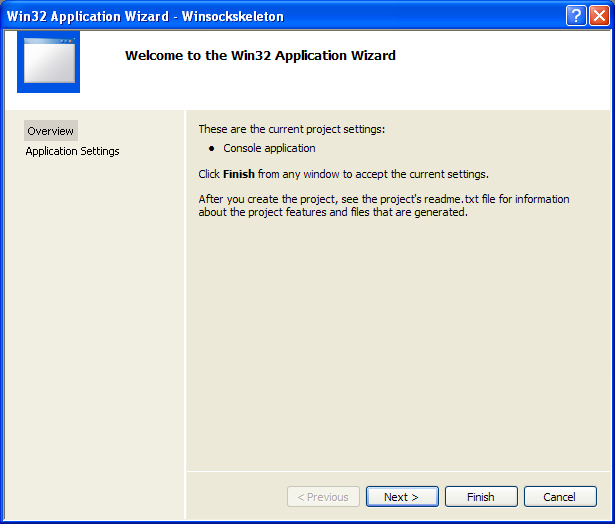
 1.      Then we can start creating the Win32 console application project. Click File menu > Project sub menu to create a new project.



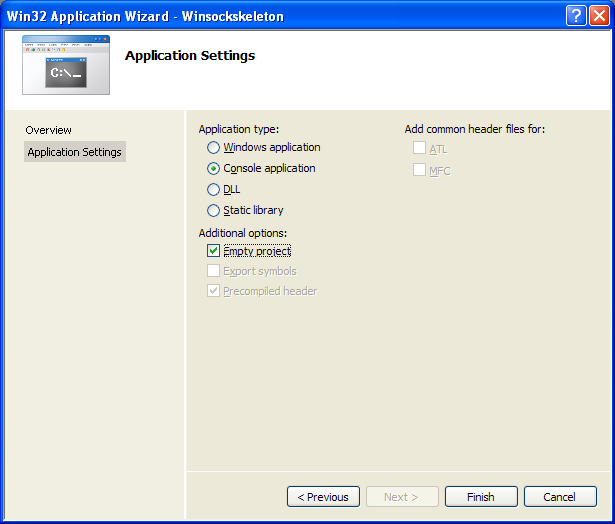
 2.      Select Win32 for the Project types: and Win32 Console Application for the Templates:. Put the project and solution name. Adjust the project location if needed and click OK.



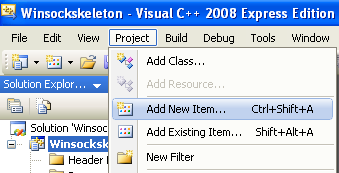
 3.      Click Next for the Win32 Application Wizard Overview page. We will remove all the unnecessary project items.

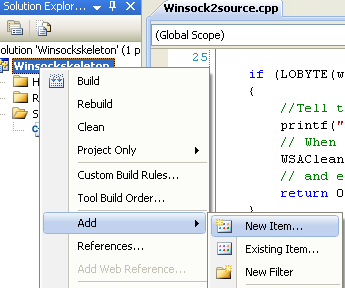


 4.      In the Application page, select Empty project for the Additional options:. Leave others as given and click Finish.

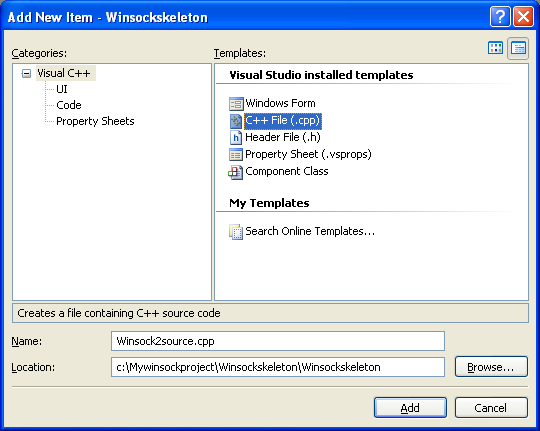


 5.      Next, we need to add new source file. Click Project menu > Add New Item sub menu or select the project folder in the Solution Explorer > Select Add menu > Select New Item sub menu.





 6.      Select C++ File (.cpp) for the Templates:. Put the source file name and click Add. Although the extension is .cpp, Visual C++ IDE will recognize that the source code used is C based on the Compile as C Code (/TC) option which will be set in the project property page later.



7.      Now, add the source code as given below.

#include <winsock2.h>

#include <stdio.h>

int main(void)

{

   WSADATA wsaData;

   int RetCode;

   // Initialize Winsock version 2.2

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

   {

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

      return 1;

   }

   else

   {

        printf("The Winsock dll found!\n");

        printf("The current status is: %s.\n", wsaData.szSystemStatus);

   }

   if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2 )

   {

        //Tell the user that we could not find a usable WinSock DLL

        printf("The dll do not support the Winsock version %u.%u!\n",

                    LOBYTE(wsaData.wVersion),HIBYTE(wsaData.wVersion));

        // When your application is finished call WSACleanup

        WSACleanup();

        // and exit

        return 0;

   }

   else

   {

        printf("The dll supports the Winsock version %u.%u!\n", LOBYTE(wsaData.wVersion),

                HIBYTE(wsaData.wVersion));

        printf("The highest version this dll can support: %u.%u\n", LOBYTE(wsaData.wHighVersion),

                HIBYTE(wsaData.wHighVersion));

        // Setup Winsock communication code here

        // When your application is finished call WSACleanup

        if (WSACleanup() == SOCKET\_ERROR)

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

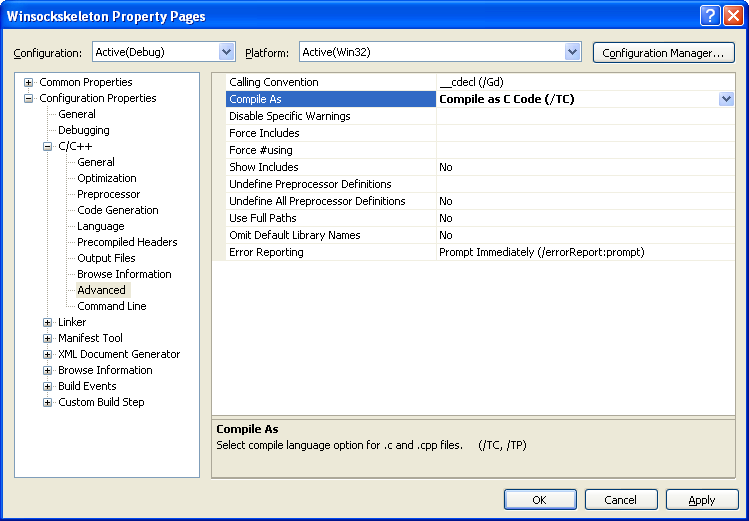
        // and exit

        return 1;

   }

}

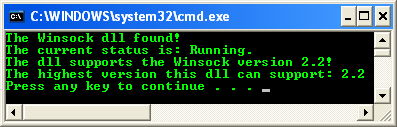
9.      Expand the Configuration folder > Expand the C/C++ sub folder. Select the Advanced link and for the Compile As option, select Compile as C Code (/TC).



10. Next, for the Additional Dependencies add ws2\_32.lib.

13. Build and run the project.

15. If there is no error, a sample of expected output is shown below.



Well, after completing this exercise you should be familiar with the steps to create an empty Win32 console application project. Those steps will be repeated for almost all the Winsock2 projects in this tutorial. Now we are ready to describe how to set up communication using a network protocol.

|  |
| --- |
| Addressing a Protocol   This chapter is limited to describing how to make fundamental Winsock calls to set up communication using the Internet Protocol (IP) because most Winsock applications developed today use it because it is widely available due to the popularity of the Internet. However, Winsock is a protocol-independent interface. Also, our discussion of IP in this chapter is limited to briefly describing IP version 4 (IPv4).  Throughout the remainder of this chapter, we will demonstrate the basics of how to set up Winsock communication using the IPv4 protocol. IP is widely available on most computer operating systems and can be used on most local area networks (LANs), such as a small network in your office, and on wide area networks (WANs), such as the Internet. By design, IP is a connectionless protocol and doesn't guarantee data delivery. Two higher-level protocols, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are used for connection-oriented and connectionless data communication over IP, which we will describe later. Both TCP and UDP use IP for data transmission and are normally referred to as TCP/IP and UDP/IP. To use IPv4 in Winsock, you need understand how to address IPv4.  **Addressing IPv4**   In IPv4, computers are assigned an address that is represented as a 32-bit quantity. When a client wants to communicate with a server through TCP or UDP, it must specify the server's IP address along with a service port number. Also, when servers want to listen for incoming client requests, they must specify an IP address and a port number. In Winsock, applications specify IP 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 we 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. Applications should be particularly careful in choosing a port because some of the available port numbers are reserved for well-known services, such as File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP).  The sin\_addr field of the SOCKADDR\_IN structure is used for storing an IPv4 address as a four-byte 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 (in decimal, octal, or hexadecimal format) 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.  A useful support function named inet\_addr() converts a dotted IP 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 IP address as a 32-bit unsigned long integer in network-byte order.

If no error occurs, inet\_addr() returns an unsigned long value containing a suitable binary representation of the Internet address given.  On Windows Server 2003 and later if the string in the cp parameter does not contain a legitimate Internet address, for example if a portion of an "a.b.c.d" address exceeds 255, then inet\_addr() returns the value INADDR\_NONE. This error is also returned if an empty string or NULL is passed in the cp parameter. On Windows XP and earlier if the string in the cp parameter does not contain a legitimate Internet address, for example if a portion of an "a.b.c.d" address exceeds 255, then inet\_addr() returns the value INADDR\_NONE. If the string in the cp parameter is an empty string or NULL, then inet\_addr() returns the value INADDR\_ANY.

The inet\_addr() function interprets the character string specified by the cp parameter. This string represents a numeric Internet address expressed in the Internet standard ".'' notation. The value returned is a number suitable for use as an Internet address. All Internet addresses are returned in IP's network order (bytes ordered from left to right). If you pass in " " (a space) to the inet\_addr() function, inet\_addr() returns zero.

On Windows Vista and later, the RtlIpv4StringToAddress() function can be used to convert a string representation of an IPv4 address to a binary IPv4 address represented as an IN\_ADDR structure. While the RtlIpv6StringToAddress() function can be used to convert a string representation of an IPv6 address to a binary IPv6 address represented as an IN6\_ADDR structure.

**Internet Addresses**

 Values specified using the ".'' notation (dot notation) takes one of the following forms:

 a.b.c.d

1. a.b.c
2. a.b
3. a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the 4 bytes of an Internet address. When an Internet address is viewed as a 32-bit integer quantity on the Intel architecture, the bytes referred to above appear as "d.c.b.a''. That is, the bytes on an Intel processor are ordered from right to left.

The parts that make up an address in "." notation can be decimal, octal or hexadecimal as specified in the C language. Numbers that start with "0x" or "0X" imply hexadecimal. Numbers that start with "0" imply octal. All other numbers are interpreted as decimal.

|  |  |
| --- | --- |
| **Internet address value** | **Meaning** |
| "4.3.2.16" | Decimal |
| "004.003.002.020" | Octal |
| "0x4.0x3.0x2.0x10" | Hexadecimal |
| "4.003.002.0x10" | Mix |

# Some Note

The inet\_addr() function supports the decimal, octal, hexadecimal, and mixed notations for the string passed in the cp parameter. The following para explains the notations which are only used by Berkeley software, and nowhere else on the Internet.

For compatibility with Berkeley software, the inet\_addr() function also supports the additional notations specified in the following paragraph. When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most 2 bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as "128.net.host''. When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the right-most 3 bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as "net.host''. When only one part is given, the value is stored directly in the network address without any byte rearrangement.

# inet\_ntoa() Function

The inet\_ntoa() function converts an (Ipv4) Internet network address into a string in Internet standard dotted-decimal format. The syntax is:

 char\* FAR inet\_ntoa(\_\_in  struct   in\_addr in);

 The in is an in\_addr() structure that represents an Internet host address. If no error occurs, inet\_ntoa() returns a character pointer to a static buffer containing the text address in standard ".'' notation. Otherwise, it returns NULL.

The inet\_ntoa() function takes an Internet address structure specified by the in parameter and returns a NULL-terminated ASCII string that represents the address in "." (dot) notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation. The string returned by inet\_ntoa() resides in memory that is allocated by Windows Sockets. The application should not make any assumptions about the way in which the memory is allocated. The string returned is guaranteed to be valid only until the next Windows Sockets function call is made within the same thread. Therefore, the data should be copied before another Windows Sockets call is made.

On Windows Vista and later, the RtlIpv4AddressToString() function 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. While, the RtlIpv6AddressToString() function can be used to convert an IPv6 address represented as an IN6\_ADDR structure to a string representation of an IPv6 address.

**InetNtop() Function**

 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(

  \_\_in   INT  Family,

  \_\_in   PVOID pAddr,

  \_\_out  PTSTR pStringBuf,

  \_\_in   size\_t StringBufSize

);

The Family is an 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 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. For an IPv6 address, this buffer should be large enough to hold at least 46 characters.

The final parameter, StringBufSize which is if on input, the length, in characters, of the buffer pointed to by the pStringBuf parameter. On the 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. The InetNtop() function provides a protocol-independent address-to-string translation. The InetNtop() function 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](http://tools.ietf.org/html/rfc2553). 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. While, 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 an IPv6 string in standard format.

**InetPton() Function**

 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(

  \_\_in   INT  Family,

  \_\_in   PCTSTR pszAddrString,

  \_\_out  PVOID pAddrBuf

);

 The Family is the address family. Possible values for the address family are defined in the Ws2def.h header file. Take 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 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. The InetPton() function 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](http://tools.ietf.org/html/rfc2553). 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. On Windows Vista and later, the RtlIpv4StringToAddress() and RtlIpv4StringToAddressEx() functions can be used to convert a text representation of an IPv4 address in Internet standard dotted-decimal notation to a numeric binary address represented as an IN\_ADDR structure. While, the RtlIpv6StringToAddress() and RtlIpv6StringToAddressEx() functions can be used to convert a string representation of an IPv6 address to a numeric binary IPv6 address represented as an IN6\_ADDR structure. The RtlIpv6StringToAddressEx() function is more flexible since it also converts a string representation of an IPv6 address that can include a scope ID and port in standard notation to a numeric binary form.

Other functions that are supported on Windows Vista and later include RtlIpv4AddressToString(), RtlIpv4StringToAddress(), RtlIpv4StringToAddressEx(), RtlIpv6AddressToString(), RtlIpv6AddressToStringEx(), RtlIpv6StringToAddress(), RtlIpv6StringToAddressEx().

The header file need to be included is Mstcpip.h and the library is **Ntdll.dll**. As the Windows Win32 legacy convention used by Microsoft, the RtlIpv4AddressToString**ExW**() is for Unicode) and RtlIpv4AddressToStringEx() is for ANSI.

**Byte Ordering**

Different computer processors represent numbers in big-endian and little-endian form, depending on how they are designed. For example, on Intel x86 processors, multibyte numbers are represented in little-endian form: the bytes are ordered from least significant to most significant. When an IP address and port number are specified as **multibyte quantities in a computer**, they are represented in **host-byte order**. However, when IP addresses and port numbers are **specified over a network**, Internet networking standards specify that **multibyte values must be represented in big-endian** form (most significant byte to least significant), normally referred to as **network-byte order**.

A series of functions can be used to convert a multibyte number from host-byte order to network-byte order and vice versa. The following four API functions convert a number from host-byte to network-byte order:

|  |  |
| --- | --- |
| **Host to Network-byte converter API Functions** | |
| 1 | u\_long htonl(u\_long hostlong); |
| 2 | int WSAHtonl(      SOCKET s,      u\_long hostlong,      u\_long FAR \* lpnetlong); |
| 3 | u\_short htons(u\_short hostshort); |
| 4 | int WSAHtons(      SOCKET s,      u\_short hostshort,      u\_short FAR \* lpnetshort); |

The hostlong parameter of htonl() and WSAHtonl() is a four-byte number in host-byte order. The htonl() function returns the number in network-byte order, whereas the WSAHtonl() function returns the number in network-byte order through the lpnetlong parameter. The hostshort parameter of htons() and WSAHtons() is a two-byte number in host-byte order. The htons() function returns the number as a two-byte value in network-byte order, whereas the WSAHtons() function returns the number through the lpnetshort parameter.

The next four functions are the opposite of the preceding four functions; they convert network-byte order to host-byte order.

|  |  |
| --- | --- |
| **Network to Host-byte Converter API Functions** | |
| 1 | u\_long ntohl(u\_long netlong); |
| 2 | int WSANtohl(      SOCKET s,      u\_long netlong,      u\_long FAR \* lphostlong); |
| 3 | u\_short ntohs(u\_short netshort); |
| 4 | int WSANtohs(      SOCKET s,      u\_short netshort,      u\_short FAR \* lphostshort); |

We will now demonstrate how to address IPv4 by creating a SOCKADDR\_IN structure using the inet\_addr() and htons() functions described previously.

SOCKADDR\_IN InternetAddr;

INT nPortId = 5150;

InternetAddr.sin\_family = AF\_INET;

// Convert the proposed dotted Internet address 136.149.3.29

// to a four-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);

As you can probably tell, IP addresses aren't easy to remember. Most people would much rather refer to a machine (or host) by using an easy-to-remember, user-friendly host name instead of an IP address. Hopefully, in other chapters we will describe useful address and name resolution functions that can help you resolve a host name, such as www.bodo.com, to an IP address and a service name, such as FTP, to a port number using functions such as getaddrinfo(), getnameinfo(), gethostbyaddr(), gethostbyname(), gethostname(), getprotobyname(), getprotobynumber(), get-servbyname(), and getservbyport(). There are also some asynchronous (non-blocking) versions of some of these functions:

1. WSAAsyncGetHostByAddr(),
2. WSAAsyncGetHostByName(),
3. WSAAsyncGetProtoByName(),
4. WSAAsyncGetProtoByNumber(),
5. WSAAsyncGetServByName(), and
6. WSAAsyncGetServByPort().

Keep in mind that some of the functions may already deprecated. Now that you have the basics of addressing a protocol such as IPv4, you can prepare to set up communication by creating a socket.

# Creating a Socket

If you're familiar with Winsock, you know that the API is based on the concept of a socket. A socket is a handle to a transport provider. In Windows, a socket is not the same thing as a file descriptor and therefore is a separate type: SOCKET in WINSOCK2.H. There are two functions that can be used to create a socket: socket and WSASocket. In the simplest form, we will briefly describe socket as:

 SOCKET socket (

    int af,

    int type,

    int protocol

);

The first parameter, af, is the protocol's address family. Since we describe Winsock in this chapter using only the IPv4 protocol, you should set this field to AF\_INET.

The second parameter, type, is the protocol's socket type. When you are creating a socket to use TCP/IP, set this field to SOCK\_STREAM, for UDP/IP use SOCK\_DGRAM.

The third parameter is protocol and is used to qualify a specific transport if there are multiple entries for the given address family and socket type. For TCP you should set this field to IPPROTO\_TCP; for UDP use IPPROTO\_UDP. Chapter 2 describes socket creation in greater detail for all protocols, including the WSASocket API.

Winsock features four useful functions to control various socket options and socket behaviors: setsockopt(), getsockopt(), ioctlsocket(), and WSAIoctl(). For simple Winsock programming, you will not need to use them specifically. Once you have successfully created a socket, you are ready to set up communication on the socket to prepare it for sending and receiving data. In Winsock there are two basic communication techniques: connection-oriented and connectionless communication.

**Connection-Oriented Communication**

 In this section, we'll cover the Winsock functions necessary for both receiving connections and establishing connections. We'll first discuss how to develop a server by listening for client connections and explore the process for accepting or rejecting a connection. Then we'll describe how to develop a client by initiating a connection to a server. Finally, we'll discuss how data is transferred in a connection-oriented session.

In IP, connection-oriented communication is accomplished through the TCP/IP protocol. TCP provides reliable error-free data transmission between two computers. When applications communicate using TCP, a virtual connection is established between the source computer and the destination computer. Once a connection is established, data can be exchanged between the computers as a two-way stream of bytes.

**Server API Functions**

 A server is a process that waits for any number of client connections with the purpose of servicing their requests. A server must listen for connections on a well-known name. In TCP/IP, this name is the IP address of the local interface and a port number. Every protocol has a different addressing scheme and therefore a different naming method. The first step in Winsock is to create a socket with either the socket() or WSASocket() call and bind the socket of the given protocol to its well-known name, which is accomplished with the bind() API call. The next step is to put the socket into listening mode, which is performed (appropriately enough) with the listen() API function. Finally, when a client attempts a connection, the server must accept the connection with either the accept() or WSAAccept() call. In the next few sections, we will discuss each API call that is required for binding, listening, and accepting a client connection.

**Binding, bind()**

Once the socket of a particular protocol is created, you must bind it to a well-known address. The bind() function associates the given socket with a well-known address. This function is declared as:

 int bind(

    SOCKET    s,

    const struct sockaddr FAR\* name,

    int    namelen

);

The first parameter, s, is the socket on which you want to wait for client connections.

The second parameter is of type struct sockaddr, which is simply a generic buffer. You must actually fill out an address buffer specific to the protocol you are using and cast that as a struct sockaddr when calling bind(). The Winsock header file defines the type SOCKADDR as struct sockaddr. We'll use this type throughout the chapter for brevity.

The third parameter is simply the size of the protocol-specific address structure being passed. For example, the following code illustrates how this is done on a TCP connection:

SOCKET    s;

SOCKADDR\_IN    tcpaddr;

int    port = 5150;

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

tcpaddr.sin\_family = AF\_INET;

tcpaddr.sin\_port = htons(port);

tcpaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

bind(s, (SOCKADDR \*)&tcpaddr, sizeof(tcpaddr));

From the example, you'll see a stream socket being created, followed by setting up the TCP/IP address structure on which client connections will be accepted. In this case, the socket is being bound to the default IP interface by using a special address, INADDR\_ANY, and occupies port number 5150. We could have specified an explicit IP address available on the system, but INADDR\_ANY allows us to bind to all available interfaces on the system so that any incoming client connection on any interface (but the correct port) will be accepted by our listening socket. The call to bind() formally establishes this association of the socket with the local IP interface and port.

On error, bind() returns SOCKET\_ERROR. The most common error encountered with bind is WSAEADDRINUSE. With TCP/IP, the WSAEADDRINUSE error indicates that another process is already bound to the local IP interface and port number or that the IP interface and port number are in the TIME\_WAIT state. If you call bind again on a socket that is already bound, WSAEFAULT will be returned.

# Listening, listen()

The next piece of the equation is to put the socket into listening mode. The bind() function merely associates the socket with a given address. The API function that tells a socket to wait for incoming connections is listen(), which is defined as:

 int listen(SOCKET s, int    backlog);

 Again, the first parameter is a bound socket.

The backlog parameter specifies the maximum queue length for pending connections. This is important when several simultaneous requests are made to the server. For example, let's say the backlog parameter is set to two. If three client requests are made at the same time, the first two will be placed in a “pending” queue so that the application can service their requests. The third connection request will fail with WSAECONNREFUSED. Note that once the server accepts a connection, the request is removed from the queue so that others can make a request. The backlog parameter is silently limited to a value that the underlying protocol provider determines. Illegal values are replaced with their nearest legal values. In addition, there is no standard provision for finding the actual backlog value.

The errors associated with listen() are fairly straightforward. By far the most common is WSAEINVAL, which usually indicates that you forgot to call bind() before listen(). Otherwise, it is possible to receive the WSAEADDRINUSE error on the listen() call as opposed to the bind() call. This error occurs most often on the bind() call.

**Accepting Connections, accept()**

 Now you're ready to accept client connections. This is accomplished with the accept(), WSAAccept(), or AcceptEx() function. (AcceptEx(), an extended version of accept similar to other Win32 Ex versions) The prototype for accept is:

 SOCKET accept(

    SOCKET s,

    struct sockaddr FAR\* addr,

    int FAR\* addrlen

);

 The parameter s is the bound socket that is in a listening state.

The second parameter should be the address of a valid SOCKADDR\_IN structure.

While addrlen should be a reference to the length of the SOCKADDR\_IN structure. For a socket of another protocol, substitute the SOCKADDR\_IN with the SOCKADDR structure corresponding to that protocol. A call to accept() services the first connection request in the queue of pending connections. When the accept() function returns, the addr structure contains the IPv4 address information of the client making the connection request, and the addrlen parameter indicates the size of the structure. In addition, accept() returns a new socket descriptor that corresponds to the accepted client connection. For all subsequent operations with this client, the new socket should be used. The original listening socket is still open to accept other client connections and is still in listening mode.

If an error occurs, INVALID\_SOCKET is returned. The most common error encountered is WSAEWOULDBLOCK if the listening socket is in asynchronous or non-blocking mode and there is no connection to be accepted. Block, non-blocking, and other socket modes are covered in another chapter. Winsock 2 introduced the function WSAAccept(), which has the capability to conditionally accept a connection based on the return value of a condition function.

At this point, we have described all the necessary elements to construct a simple Winsock TCP/IP server application. The following program example demonstrates how to write a simple server that can accept one TCP/IP connection. We did not perform full error checking on the calls to make reading the code less confusing.

#include <winsock2.h>

#include <stdio.h>

int main(void)

{

     WSADATA            wsaData;

     SOCKET             ListeningSocket;

     SOCKET             NewConnection;

     SOCKADDR\_IN        ServerAddr;

     int                Port = 5150;

     // Initialize Winsock version 2.2

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

     {

          // MSDN: An application can call the WSAGetLastError() function to determine the

          // extended error code for other Windows sockets functions as is normally

          // done in Windows Sockets even if the WSAStartup function fails or the WSAStartup

          // function was not called to properly initialize Windows Sockets before calling a

          // Windows Sockets function. The WSAGetLastError() function is one of the only functions

          // in the Winsock 2.2 DLL that can be called in the case of a WSAStartup failure

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

          // Exit with error

          return -1;

     }

     else

     {

          printf("Server: The Winsock dll found!\n");

          printf("Server: The current status is %s.\n", wsaData.szSystemStatus);

     }

     if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2 )

     {

          //Tell the user that we could not find a usable WinSock DLL

          printf("Server: The dll do not support the Winsock version %u.%u!\n",

                    LOBYTE(wsaData.wVersion),HIBYTE(wsaData.wVersion));

          // Do the clean up

          WSACleanup();

          // and exit with error

          return -1;

     }

     else

     {

          printf("Server: The dll supports the Winsock version %u.%u!\n", LOBYTE(wsaData.wVersion),

                    HIBYTE(wsaData.wVersion));

          printf("Server: The highest version this dll can support is %u.%u\n",

                    LOBYTE(wsaData.wHighVersion), HIBYTE(wsaData.wHighVersion));

     }

     // Create a new socket to listen for client connections.

     ListeningSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

     // Check for errors to ensure that the socket is a valid socket.

     if (ListeningSocket == INVALID\_SOCKET)

     {

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

          // Clean up

          WSACleanup();

          // and exit with error

          return -1;

     }

     else

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

     // Set up a SOCKADDR\_IN structure that will tell bind that we

     // want to listen for connections on all interfaces using port 5150.

     // The IPv4 family

     ServerAddr.sin\_family = AF\_INET;

     // host-to-network byte order

     ServerAddr.sin\_port = htons(Port);

     // Listen on all interface, host-to-network byte order

     ServerAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

     // Associate the address information with the socket using bind.

     // Call the bind function, passing the created socket and the sockaddr\_in

     // structure as parameters. Check for general errors.

     if (bind(ListeningSocket, (SOCKADDR \*)&ServerAddr, sizeof(ServerAddr)) == SOCKET\_ERROR)

     {

          printf("Server: bind() failed! Error code: %ld.\n", WSAGetLastError());

          // Close the socket

          closesocket(ListeningSocket);

          // Do the clean up

          WSACleanup();

          // and exit with error

          return -1;

     }

     else

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

     // Listen for client connections. We used a backlog of 5, which

     // is normal for many applications.

     if (listen(ListeningSocket, 5) == SOCKET\_ERROR)

     {

          printf("Server: listen(): Error listening on socket %ld.\n", WSAGetLastError());

          // Close the socket

          closesocket(ListeningSocket);

          // Do the clean up

          WSACleanup();

          // Exit with error

          return -1;

     }

     else

          printf("Server: listen() is OK, I'm waiting for connections...\n");

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

     // Accept a new connection when one arrives.

     while(1)

     {

          NewConnection = SOCKET\_ERROR;

          while(NewConnection == SOCKET\_ERROR)

          {

              NewConnection = accept(ListeningSocket, NULL, NULL);

          }

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

          printf("Server: Client connected, ready for receiving and sending data...\n");

          // Transfer the connected socket to the listening socket

          ListeningSocket = NewConnection;

          break;

     }

     // At this point you can do two things with these sockets. Wait

     // for more connections by calling accept again on ListeningSocket

     // and start sending or receiving data on NewConnection (need a loop). We will

     // describe how to send and receive data later in the chapter.

     // When you are finished sending and receiving data on the

     // NewConnection socket and are finished accepting new connections

     // on ListeningSocket, you should close the sockets using the closesocket API.

     if(closesocket(NewConnection) != 0)

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

     else

          printf("Server: Closing \"NewConnection\" socket...\n");

     // When your application is finished handling the connections,

     // call WSACleanup.

     if(WSACleanup() != 0)

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

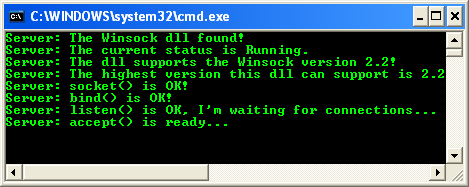
     else

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

     return 0;

}

12. If there is no error, a sample output is shown below.



Now that you understand how to construct a server that can receive a client connection, we will describe how to construct a client.

|  |
| --- |
| Client API Functions   The client is much simpler and involves fewer steps to set up a successful connection. There are only three steps for a client:     1. Create a socket. 2. Set up a SOCKADDR address structure with the name of server you are going to connect to (dependent on underlying protocol). For TCP/IP, this is the server's IP address and port number its application is listening on. 3. Initiate the connection with connect() or WSAConnect().     You already know how to create the socket and construct a SOCKADDR structure, so the only remaining step is establishing a connection.   TCP States   By knowing the actual TCP states you will gain a better understanding of how the Winsock API calls affect change in the underlying protocol. In addition, many programmers run into a common problem when closing sockets: the TCP states surrounding a socket closure are of the most interest.  The start state of every socket is the CLOSED state. When a client initiates a connection, it sends a SYN packet to the server and puts the client socket in the SYN\_SENT state. When the server receives the SYN packet, it sends a SYN-ACK packet, which the client responds to with an ACK packet. At this point, the client's socket is in the ESTABLISHED state. If the server never sends a SYN-ACK packet, the client time out and reverts to the CLOSED state. You can refer to the [TCP state diagram](http://tangentsoft.net/wskfaq/articles/debugging-tcp.html) to get a better description.  When a server's socket is bound and is listening on a local interface and port, the state of the socket is LISTEN. When a client attempts a connection, the server receives a SYN packet and responds with a SYN-ACK packet. The state of the server's socket changes to SYN\_RCVD. Finally, the client sends an ACK packet, which causes the state of the server's socket to change to ESTABLISHED.  Once the application is in the ESTABLISHED state, there are two paths for closure. If your application initiates the closure, it is known as an **active socket closure**; otherwise, the socket closure is **passive**. If you actively initiate a closure, your application sends a FIN packet. When your application calls closesocket or shutdown (with SD\_SEND as its second argument), your application sends a FIN packet to the peer, and the state of your socket changes to FIN\_WAIT\_1. Normally, the peer responds with an ACK packet, and your socket's state becomes FIN\_WAIT\_2. If the peer also closes the connection, it sends a FIN packet and your computer responds by sending an ACK packet and placing your socket in the TIME\_WAIT state. |

The TIME\_WAIT state is also called the 2MSL wait state. MSL stands for Maximum Segment Lifetime and represents the amount of time a packet can exist on the network before being discarded. Each IP packet has a time-to-live (TTL) field, which when decremented to 0 causes the packet to be discarded. Each router on the network that handles the packet decrements the TTL by 1 and passes the packet on. Once an application enters the TIME\_WAIT state, it remains there for twice the MSL time. This allows TCP to re-send the final ACK in case it's lost, causing the FIN to be retransmitted. After the 2MSL wait state completes, the socket goes to the CLOSED state.

On an active close, two other paths lead to the TIME\_WAIT state. In our previous discussion, only one side issues a FIN and receives an ACK response, but the peer is still free to send data until it too closes. This is where the other two paths come into play. In one path, the simultaneous close, a computer and its peer at the other side of a connection issue a close at the same time; the computer sends a FIN packet to the peer and receives a FIN packet from the peer. Then the computer sends an ACK packet in response to the peer's FIN packet and changes its socket to the CLOSING state. Once the computer receives the last ACK packet from the peer, the computer's socket state becomes TIME\_WAIT.

The other path for an active closure is just a variation on the simultaneous close: the socket transitions from the FIN\_WAIT\_1 state directly to the TIME\_WAIT state. This occurs when an application sends a FIN packet but shortly thereafter receives a FIN-ACK packet from the peer. In this case, the peer is acknowledging the application's FIN packet and sending its own, to which the application responds with an ACK packet.

The major effect of the TIME\_WAIT state is that while a TCP connection is in the 2MSL wait state, the socket pair defining that connection cannot be reused. A socket pair is the combination of local IP–local port and remote IP–remote port. Some TCP implementations do not allow the reuse of any port number in a socket pair in the TIME\_WAIT state. Microsoft's implementation does not suffer from this deficiency. However, if a connection is attempted in which the socket pair is already in the TIME\_WAIT state, the connection attempt will fail with error WSAEADDRINUSE. One way around this (besides waiting for the socket pair that is using that local port to leave the TIME\_WAIT state) is to use the socket option SO\_REUSEADDR.

The last point of discussion for socket states is the passive closure. In this scenario, an application receives a FIN packet from the peer and responds with an ACK packet. At this point, the application's socket changes to the CLOSE\_WAIT state. Because the peer has closed its end, it can't send any more data, but the application still can until it also closes its end of the connection. To close its end of the connection, the application sends its own FIN, causing the application's TCP socket state to become LAST\_ACK. After the application receives an ACK packet from the peer, the application's socket reverts to the CLOSED state. For more information regarding the TCP/IP protocol, consult [RFC 793](http://tools.ietf.org/html/rfc793).

**connect()**

Connecting a socket is accomplished by calling connect(), WSAConnect(), or ConnectEx(). We'll look at the Winsock 1 version of this function, which is defined as:

 int connect(

    SOCKET s,

    const struct sockaddr FAR\* name,

    int namelen

);

s is the valid TCP socket on which to establish the connection.

name is the socket address structure (SOCKADDR\_IN) for TCP that describes the server to connect to.

namelen is the length of the name variable.

If the computer you're attempting to connect to does not have a process listening on the given port, the connect() call fails with the WSAECONNREFUSED error. The other error you might encounter is WSAETIMEDOUT, which occurs if the destination you're trying to reach is unavailable (either because of a communication-hardware failure on the route to the host or because the host is not currently on the network).

The following program example demonstrates how to write a simple client that can connect to the server application demonstrated earlier.

 7.      Now, add the source code as given below.

#include <winsock2.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

     WSADATA              wsaData;

     SOCKET               SendingSocket;

     // Server/receiver address

     SOCKADDR\_IN          ServerAddr;

     // Server/receiver port to connect to

     unsigned int         Port = 80;

     int  RetCode;

     // Initialize Winsock version 2.2

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

     printf("Client: Winsock DLL status is %s.\n", wsaData.szSystemStatus);

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

     // AF\_INET = 2, The Internet Protocol version 4 (IPv4) address family, TCP protocol

     SendingSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

     if(SendingSocket == 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");

     // Set up a SOCKADDR\_IN structure that will be used to connect

     // to a listening server on port 5150. For demonstration

     // purposes, let's assume our server's IP address is 127.0.0.1 or localhost

     // IPv4

     ServerAddr.sin\_family = AF\_INET;

     // Port no.

     ServerAddr.sin\_port = htons(Port);

     // The IP address

     ServerAddr.sin\_addr.s\_addr = inet\_addr("209.131.36.158");

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

     RetCode = connect(SendingSocket, (SOCKADDR \*) &ServerAddr, sizeof(ServerAddr));

     if(RetCode != 0)

     {

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

          // Close the socket

          closesocket(SendingSocket);

          // 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 receiving data...\n");

     }

     // At this point you can start sending or receiving data on

     // the socket SendingSocket. We will describe sending and receiving data

     // later in the chapter.

     // When you are finished sending and receiving data on socket SendingSocket,

     // you should close the socket using the closesocket API. We will

     // describe socket closure later in the chapter.

     if(closesocket(SendingSocket) != 0)

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

     else

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

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

     if(WSACleanup() != 0)

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

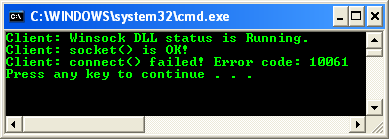
     else

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

     return 0;

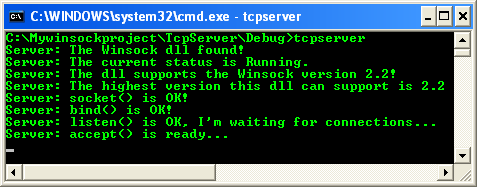
}

12. If there is no error, the following sample output should be expected.

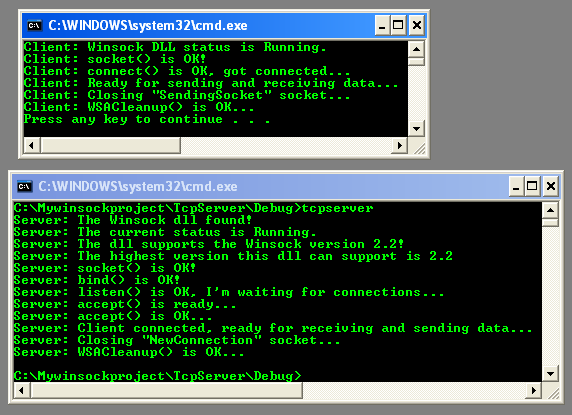


13. The error code 10061 is enumeration for WSAECONNREFUSED which is a Connection refused. It is because there is no listening server on the specified address and port number. The next steps will demonstrate the client and server programs running.

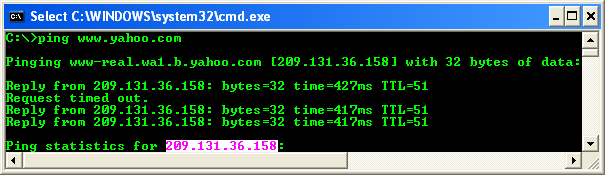
14. Run the previous server program from command prompt (or you can open another Visual C++ IDE instance and run it).



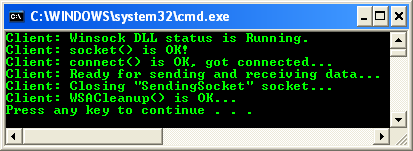
15. Then, run the client program.



16. Next, let test this client connection to the real server. In this case we change the port to 80 (standard http port) and the server is www.yahoo.com or www.google.com. Then we rebuild the project. You can use the ping tool to get the server address.



17. The following screenshot shows a sample output when connecting to one of the Yahoo.com web server.



Now that you can set up communication for a connection-oriented server and client, you are ready to begin handling data transmission.

|  |
| --- |
| Data Transmission   Sending and receiving data is what network programming is all about. For sending data on a connected socket, there are two API functions:     1. send() and 2. WSASend().     The second function is specific to Winsock 2. Likewise, two functions are for receiving data on a connected socket:     1. recv() and 2. WSARecv().     The latter is also a Winsock 2 call. An important thing to keep in mind is that all buffers associated with sending and receiving data are of the **simple char type** which is just simple **byte-oriented data**. In reality, it can be a buffer with any raw data in it, whether it's binary or string data doesn't matter.  In addition, the error code returned by all send and receive functions is SOCKET\_ERROR. Once an error is returned, call WSAGetLastError() to obtain extended error information. The two most common errors encountered are WSAECONNABORTED and WSAECONNRESET. Both of these deal with the connection being closed, either through a timeout or through the peer closing the connection. Another common error is WSAEWOULDBLOCK, which is normally encountered when either nonblocking or asynchronous sockets are used. This error basically means that the specified function cannot be completed at this time. In another chapter, we will describe various Winsock I/O methods that can help you avoid some of these errors.  **send() and WSASend()**   The first API function to send data on a connected socket is send(), which is prototyped as:   int send(      SOCKET s,      const char FAR \* buf,      int len,      int flags  ); |

The SOCKET parameter is the connected socket to send the data on.

The second parameter, buf, is a pointer to the character buffer that contains the data to be sent.

The third parameter, len, specifies the number of characters in the buffer to send.

Finally, the flags parameter can be either 0, MSG\_DONTROUTE, or MSG\_OOB. Alternatively, the flags parameter can be a bitwise OR any of those flags. The MSG\_DONTROUTE flag tells the transport not to route the packets it sends. It is up to the underlying transport to honor this request (for example, if the transport protocol doesn't support this option, it will be ignored). The MSG\_OOB flag signifies that the data should be sent out of band.

On a good return, send returns the number of bytes sent; otherwise, if an error occurs, SOCKET\_ERROR will be returned. A common error is WSAECO-NNABORTED, which occurs when the virtual circuit terminates because of a timeout failure or a protocol error. When this occurs, the socket should be closed, as it is no longer usable. The error WSAECONNRESET occurs when the application on the remote host resets the virtual circuit by executing a hard close or terminating unexpectedly, or when the remote host is rebooted. Again, the socket should be closed after this error occurs. The last common error is WSAETIMEDOUT, which occurs when the connection is dropped because of a network failure or the remote connected system going down without notice.

The Winsock 2 version of the send() API function, WSASend(), is defined as:

 int WSASend(

    SOCKET s,

    LPWSABUF lpBuffers,

    DWORD dwBufferCount,

    LPDWORD lpNumberOfBytesSent,

    DWORD dwFlags,

    LPWSAOVERLAPPED lpOverlapped,

    LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionRoutine

);

 The socket is a valid handle to a connection session.

The second parameter is a pointer to one or more WSABUF structures. This can be either a single structure or an array of such structures.

The third parameter indicates the number of WSABUF structures being passed. Remember that each WSABUF structure is a character buffer and the length of that buffer. You might wonder why you would want to send more than one buffer at a time. This is called **scatter-gather I/O** and will be discussed later in this chapter; however, in the case of data sent using multiple buffers on a connected socket, each buffer is sent from the first to the last WSABUF structure in the array.

The lpNumberOfBytesSent is a pointer to a DWORD that on return from the WSASend() call contains the total number of bytes sent.

The dwFlags parameter is equivalent to its counterpart in send.

The last two parameters, lpOverlapped and lpCompletionRoutine, are used for overlapped I/O. **Overlapped I/O is one of the asynchronous I/O models** that Winsock supports and is discussed in detail in other chapter.

The WSASend() function sets lpNumberOfBytesSent to the number of bytes written. The function returns 0 on success and SOCKET\_ERROR on any error, and generally encounters the same errors as the send function. There is one final send function you should be aware of: WSASendDisconnect().

**WSASendDisconnect()**

 This function is rather specialized and not generally used. The function prototype is:

 int WSASendDisconnect(SOCKET s, LPWSABUF lpOutboundDisconnectData);

# Out-of-Band Data

 When an application on a connected stream socket needs to send data that is **more important than regular data on the stream**, it can mark the important data as out-of-band (OOB) data. The application on the other end of a connection can receive and process OOB data through a separate logical channel that is conceptually independent of the data stream.

In TCP, OOB data is implemented via an urgent 1-bit marker (called **URG**) and a 16-bit pointer in the TCP segment header that identify a specific downstream byte as urgent data. Two specific ways of implementing urgent data currently exist for TCP. [RFC 793](http://rfc-ref.org/RFC-TEXTS/793/index.html), which describes TCP and introduces the concept of urgent data, indicates that the urgent pointer in the TCP header is a positive offset to the byte that follows the urgent data byte. However, [RFC 1122](http://rfc-ref.org/RFC-TEXTS/1122/index.html) describes the urgent offset as pointing to the urgent byte itself.

The Winsock specification uses the term OOB to refer to both protocol-independent OOB data and TCP's implementation of OOB data (urgent data). To check whether pending data contains urgent data, you must call the ioctlsocket function with the SIOCATMARK option.

Winsock provides several methods for obtaining the urgent data. Either the urgent data is inlined so that it appears in the normal data stream, or inlining can be turned off so that a discrete call to a receive function returns only the urgent data. The socket option SO\_OOBINLINE controls the behavior of OOB data.

Telnet and Rlogin use urgent data for several reasons. However, unless you plan to write your own Telnet or Rlogin, you should stay away from urgent data. It's not well defined and might be implemented differently on platforms other than Windows. If you require a method of signaling the peer for urgent reasons, implement a separate control socket for this urgent data and reserve the main socket connection for normal data transfers.

The function initiates a shutdown of the socket and sends disconnect data. Of course, this function is available only to those transport protocols that support **graceful close** and **disconnect data**. The WSASendDisconnect() function behaves like a call to the shutdown() function (which is described later) with an SD\_SEND argument, but it also sends the data contained in its lpOutboundDisconnectData parameter. Subsequent sends are not allowed on the socket. Upon failure, WSASendDisconnect() returns SOCKET\_ERROR. This function can encounter some of the same errors as the send function.

**recv() and WSARecv()**

 The recv() function is the most basic way to accept incoming data on a connected socket. This function is defined as:

 int recv(

    SOCKET s,

    char FAR\* buf,

    int len,

    int flags

);

 The first parameter, s, is the socket on which data will be received.

The second parameter, buf, is the character buffer that will receive the data,

The len is either the number of bytes you want to receive or the size of the buffer, buf.

Finally, the possible value of flags parameter is constructed by using the bitwise OR operator with any of the following values.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| MSG\_PEEK | Peeks at the incoming data. The data is copied into the buffer, but is not removed from the input queue. The function subsequently returns the amount of data that can be read in a single call to the recv (or recvfrom) function, which may not be the same as the total amount of data queued on the socket. The amount of data that can actually be read in a single call to the recv (or recvfrom) function is limited to the data size written in the send or sendto function call. |
| MSG\_OOB | Processes Out Of Band (OOB) data. |
| MSG\_WAITALL | The receive request will complete only when one of the following events occurs:   The buffer supplied by the caller is completely full.   1. The connection has been closed. 2. The request has been canceled or an error occurred.    Note that if the underlying transport does not support MSG\_WAITALL, or if the socket is in a non-blocking mode, then this call will fail with WSAEOPNOTSUPP. Also, if MSG\_WAITALL is specified along with MSG\_OOB, MSG\_PEEK, or MSG\_PARTIAL, then this call will fail with WSAEOPNOTSUPP. This flag is not supported on datagram sockets or message-oriented CO sockets. |

 Of course, 0 specifies no special actions. MSG\_PEEK causes the data that is available to be copied into the supplied **receive buffer**, but this data is not removed from the **system's buffer**. The number of bytes pending is also returned.

Message peeking is bad. Not only does it degrade performance, as you now need to make two system calls (one to peek and one without the MSG\_PEEK flag to actually remove the data), but it is also unreliable under certain circumstances. The data returned might not reflect the entire amount available. Also, by leaving data in the system buffers, the system has less space to contain incoming data. As a result, the system reduces the TCP window size for all senders. This prevents your application from achieving the maximum possible throughput. The best thing to do is to copy all the data you can into your own buffer and manipulate it there.

There are some considerations when using recv() on a message- or datagram-based socket such as UDP, which we will describe later. If the data pending is larger than the supplied buffer, the buffer is filled with as much data as it will contain. In this event, the recv() call generates the error WSAEMSGSIZE. Note that the message-size error occurs with **message-oriented protocols**. **Stream protocols** such as TCP buffer incoming data and will return as much data as the application requests, even if the amount of pending data is greater. Thus, for streaming protocols you will not encounter the WSAEMSGSIZE error.

The WSARecv() function adds some new capabilities over recv(), such as overlapped I/O and partial datagram notifications. The definition of WSARecv() is:

 int WSARecv(

    SOCKET s,

    LPWSABUF lpBuffers,

    DWORD dwBufferCount,

    LPDWORD lpNumberOfBytesRecvd,

    LPDWORD lpFlags,

    LPWSAOVERLAPPED lpOverlapped,

    LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionRoutine

);

Parameter s is the connected socket.

The second and third parameters are the buffers to receive the data. The lpBuffers parameter is an array of WSABUF structures, and dwBufferCount indicates the number of WSABUF structures in the array.

The lpNumberOfBytesReceived parameter points to the number of bytes received by this call if the receive operation completes immediately.

The lpFlags parameter can be one of the values MSG\_PEEK, MSG\_OOB, or MSG\_PARTIAL, or a bitwise OR combination of those values. The MSG\_PARTIAL flag has several different meanings depending on where it is used or encountered. For message-oriented protocols that support partial messaging (like AppleTalk), this flag is set upon return from WSARecv() (if the entire message could not be returned in this call because of insufficient buffer space). In this case, subsequent WSARecv() calls set this flag until the entire message is returned, when the MSG\_PARTIAL flag is cleared. If this flag is passed as an input parameter, the receive operation should complete as soon as data is available, even if it is only a portion of the entire message. The MSG\_PARTIAL flag is used only with message-oriented protocols, not with streaming ones. In addition, not all protocols support partial messages. The protocol entry for each protocol contains a flag indicating whether it supports this feature.

The lpOverlapped and lpCompletionRoutine parameters are used in overlapped I/O operations, discussed in other chapter. There is one other specialized receive function you should be aware of: WSARecvDisconnect().

**WSARecvDisconnect()**

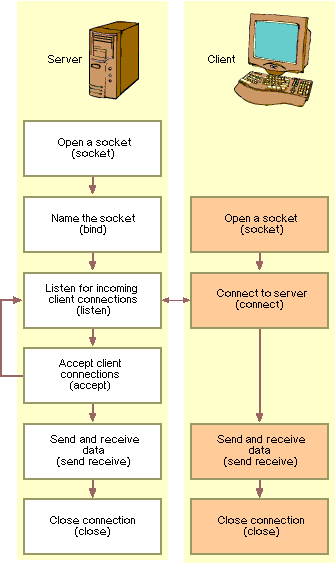
 This function is the opposite of WSASendDisconnect() and is defined as follows:

 int WSARecvDisconnect(SOCKET s, LPWSABUF lpInboundDisconnectData);

 Like its sending counterpart, the parameters of WSASendDisconnect() are the connected socket handle and a valid WSABUF structure with the data to be received. The data received can be only disconnecting data that is sent by a WSASendDisconnect() on the other side; it cannot be used to receive normal data. In addition, once the data is received, this function disables reception from the remote party, which is equivalent to calling the shutdown() function (which is described later) with SD\_RECEIVE.

**Stream Protocols**

 Because most connection-oriented communication, such as TCP, is streaming protocols, we'll briefly describe them here. A streaming protocol is one that the sender and receiver may break up or coalesce data into smaller or larger groups. The following Figure describes briefly the flow of TCP packet between client and server sides.



The main thing to be aware of with any function that sends or receives data on a stream socket is that you are not guaranteed to read or write the amount of data you request. Let's say you have a character buffer with 2048 bytes of data you want to send with the send function. The code to send this is:

 char sendbuff[2048];

int  nBytes = 2048;

// Fill sendbuff with 2048 bytes of data

 // Assume s is a valid, connected stream socket

ret = send(s, sendbuff, nBytes, 0);

It is possible for send to return having sent less than 2048 bytes. The ret variable will be set to the number of bytes sent because the **system allocates a certain amount of buffer space for each socket to send and receive data**. In the case of sending data, the internal buffers hold data to be sent until such time as the data can be placed on the wire. Several common situations can cause this. For example, simply transmitting a huge amount of data will cause these buffers to become filled quickly. Also, for TCP/IP, there is what is known as the **window size** ([sliding window demo](http://www2.rad.com/networks/2004/sliding_window/)). The receiving end will adjust this window size to indicate how much data it can receive. If the receiver is being flooded with data, it might set the window size to 0 to catch up with the pending data. This will force the sender to stop until it receives a new window size greater than 0. In the case of our send call, there might be buffer space to hold only 1024 bytes, in which case you would have to resubmit the remaining 1024 bytes. The following code ensures that all your bytes are sent:

 char sendbuff[2048];

int  nBytes = 2048, nLeft, idx;

// Fill sendbuff with 2048 bytes of data

nLeft = nBytes;

idx = 0;

 while (nLeft > 0)

{

    // Assume s is a valid, connected stream socket

    ret = send(s, &sendbuff[idx], nLeft, 0);

    if (ret == SOCKET\_ERROR)

    {

        // Error handler

    }

    nLeft -= ret;

    idx += ret;

}

The same principle holds true for receiving data on a stream socket but is less significant. Because stream sockets are a continuous stream of data, when an application reads, it isn't generally concerned with how much data it should read. If your application requires discrete messages over a stream protocol, you might have to do a little work. If all the messages are the same size, life is pretty simple, and the code for reading, say, 512-byte messages would look like this:

 char    recvbuff[1024];

int     ret, nLeft, idx;

nLeft = 512;

idx = 0;

while (nLeft > 0)

{

    ret = recv(s, &recvbuff[idx], nLeft, 0);

    if (ret == SOCKET\_ERROR)

    {

        // Error handler

    }

    idx += ret;

    nLeft -= ret;

}

Things get a little complicated if your message sizes vary. It is necessary to impose your own protocol to let the receiver know how big the forthcoming message will be. For example, the first four bytes written to the receiver will always be the integer size in bytes of the forthcoming message. The receiver will start every read by looking at the first four bytes, converting them to an integer, and determining how many additional bytes that message comprises.

**Scatter-Gather I/O**

 Scatter-gather support is a concept originally introduced in Berkeley Sockets with the functions recv and writev. This feature is available with the Winsock 2 functions WSARecv(), WSARecvFrom(), WSASend(), and WSASendTo(). It is most useful for applications that send and receive data that is formatted in a very specific way. For example, messages from a client to a server might always be composed of a fixed 32-byte header specifying some operation, followed by a 64-byte data block and terminated with a 16-byte trailer. In this example, WSASend() can be called with an array of three WSABUF structures, each corresponding to the three message types. On the receiving end, WSARecv() is called with three WSABUF structures, each containing data buffers of 32 bytes, 64 bytes, and 16 bytes.

When using stream-based sockets, scatter-gather operations simply treat the supplied data buffers in the WSABUF structures as one contiguous buffer. Also, the receive call might return before all buffers are full. On message-based sockets, each call to a receive operation receives a single message up to the buffer size supplied. If the buffer space is insufficient, the call fails with WSAEMSGSIZE and the data is truncated to fit the available space. Of course, with protocols that support partial messages, the MSG\_PARTIAL flag can be used to prevent data loss.

**Breaking the Connection**

 Once you are finished with a socket connection, you must close it and release any resources associated with that socket handle. To actually release the resources associated with an open socket handle, use the closesocket() call. Be aware, however, that closesocket() can have some adverse effects, depending on how it is called, that can lead to data loss. For this reason, a connection should be gracefully terminated with the shutdown() function before a call to the closesocket() function. These two API functions are discussed next.

# shutdown()

To ensure that all data an application sends is received by the peer, a well-written application should notify the receiver that no more data is to be sent. Likewise, the peer should do the same. This is known as a **graceful close** and is performed by the shutdown() function, defined as:

int shutdown(SOCKET s, int how);

The how parameter can be SD\_RECEIVE, SD\_SEND, or SD\_BOTH. For SD\_RECEIVE, subsequent calls to any receive function on the socket are disallowed. This has no effect on the lower protocol layers. And for TCP sockets, if data is queued for receive or if data subsequently arrives, the connection is reset. However, on UDP sockets incoming data is still accepted and queued (because shutdown() has no meaning for connectionless protocols). For SD\_SEND, subsequent calls to any send function are disallowed. For TCP sockets, this causes a FIN packet to be generated after all data is sent and acknowledged by the receiver. Finally, specifying SD\_BOTH disables both sends and receives.

Note that not all connection-oriented protocols support graceful closure, which is what the shutdown() API performs. For these protocols (such as ATM), only closesocket() needs to be called to terminate the session. A flag that describes what types of operation is summarized in the following Table. Possible values for this flag are listed in the Winsock2.h header file.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| SD\_SEND (0) | Shutdown send operations. |
| SD\_RECEIVE (1) | Shutdown receive operations. |
| SD\_BOTH (2) | Shutdown both send and receive operations. |

If no error occurs, shutdown returns zero. Otherwise, a value of SOCKET\_ERROR is returned, and a specific error code can be retrieved by calling WSAGetLastError().

|  |  |
| --- | --- |
| **Error code** | **Meaning** |
| WSANOTINITIALISED | A successful WSAStartup() call must occur before using this function. |
| WSAENETDOWN | The network subsystem has failed. |
| WSAEINVAL | The how parameter is not valid, or is not consistent with the socket type. For example, SD\_SEND is used with a UNI\_RECV socket type. |
| WSAEINPROGRESS | A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function. |
| WSAENOTCONN | The socket is not connected (connection-oriented sockets only). |
| WSAENOTSOCK | The descriptor is not a socket. |

Once the shutdown() function is called to disable send, receive, or both, there is no method to re-enable send or receive for the existing socket connection. An application should not rely on being able to reuse a socket after it has been shut down. In particular, a Windows Sockets provider is not required to support the use of connect() on a socket that has been shut down.

If an application wants to reuse a socket, then the DisconnectEx() function should be called with the dwFlags parameter set to TF\_REUSE\_SOCKET to close a connection on a socket and prepare the socket handle to be reused. When the DisconnectEx() request completes, the socket handle can be passed to the AcceptEx() or ConnectEx() function.

If an application wants to reuse a socket, the TransmitFile() or TransmitPackets() functions can be called with the dwFlags parameter set with TF\_DISCONNECT and TF\_REUSE\_SOCKET to disconnect after all the data has been queued for transmission and prepare the socket handle to be reused. When the TransmitFile() request completes, the socket handle can be passed to the function call previously used to establish the connection, such as AcceptEx() or ConnectEx(). When the TransmitPackets() function completes, the socket handle can be passed to the AcceptEx() function.

Take note that the socket level disconnect is subject to the behavior of the underlying transport. For example, a TCP socket may be subject to the TCP TIME\_WAIT state, causing the DisconnectEx(), TransmitFile(), or TransmitPackets() call to be delayed.

# closesocket()

The closesocket() function closes a socket and is defined as:

 int closesocket (SOCKET s);

 If no error occurs, closesocket() returns zero. Otherwise, a value of SOCKET\_ERROR is returned, and a specific error code can be retrieved by calling WSAGetLastError().

|  |  |
| --- | --- |
| **Error code** | **Meaning** |
| WSANOTINITIALISED | A successful WSAStartup() call must occur before using this function. |
| WSAENETDOWN | The network subsystem has failed. |
| WSAENOTSOCK | The descriptor is not a socket. |
| WSAEINPROGRESS | A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function. |
| WSAEINTR | The (blocking) Windows Socket 1.1 call was canceled through WSACancelBlockingCall(). |
| WSAEWOULDBLOCK | The socket is marked as nonblocking, but the l\_onoff member of the linger structure is set to non-zero and the l\_linger member of the linger structure is set to a nonzero timeout value. |

Calling closesocket() releases the socket descriptor and any further calls using the socket fail with WSAENOTSOCK. If there are no other references to this socket, all resources associated with the descriptor are released. This includes discarding any queued data.

Pending synchronous calls issued by any thread in this process are canceled without posting any notification messages. Pending overlapped operations are also canceled. Any event, completion routine, or completion port that is associated with the overlapped operation is performed but will fail with the error WSA\_OPERATION\_ABORTED. In addition, one other factor influences the behavior of closesocket(): whether the socket option SO\_LINGER has been set. An application should always have a matching call to closesocket() for each successful call to socket to return any socket resources to the system.

#include <winsock2.h>

#include <stdio.h>

// A sample of the select() return value

int recvTimeOutTCP(SOCKET socket, long sec, long usec)

{

    // Setup timeval variable

    struct [timeval](http://msdn.microsoft.com/en-us/library/ms740560.aspx" \t "_blank) timeout;

    struct [fd\_set](http://msdn.microsoft.com/en-us/library/ms737873(VS.85).aspx" \t "_blank) fds;

     // assign the second and microsecond variables

    timeout.tv\_sec = sec;

    timeout.tv\_usec = usec;

    // Setup fd\_set structure

    FD\_ZERO(&fds);

    FD\_SET(socket, &fds);

    // Possible return values:

    // -1: error occurred

    // 0: timed out

    // > 0: data ready to be read

    return [select](http://msdn.microsoft.com/en-us/library/ms740141(VS.85).aspx)(0, &fds, 0, 0, &timeout);

}

int main(int argc, char \*\*argv)

{

     WSADATA            wsaData;

     SOCKET             ListeningSocket, NewConnection;

     SOCKADDR\_IN        ServerAddr, SenderInfo;

     int                Port = 7171;

     // Receiving part

     char          recvbuff[1024];

     int                ByteReceived, i, nlen, SelectTiming;

     // Initialize Winsock version 2.2

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

     {

          // The WSAGetLastError() function is one of the only functions

          // in the Winsock 2.2 DLL that can be called in the case of a WSAStartup failure

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

          // Exit with error

          return 1;

     }

     else

     {

          printf("Server: The Winsock DLL found!\n");

          printf("Server: The current status is %s.\n", wsaData.szSystemStatus);

     }

     if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2 )

     {

          //Tell the user that we could not find a usable WinSock DLL

          printf("Server: The dll do not support the Winsock version %u.%u!\n",

                    LOBYTE(wsaData.wVersion),HIBYTE(wsaData.wVersion));

          // Do the clean up

          WSACleanup();

          // and exit with error

          return 1;

     }

     else

     {

          printf("Server: The dll supports the Winsock version %u.%u!\n", LOBYTE(wsaData.wVersion),

                    HIBYTE(wsaData.wVersion));

          printf("Server: The highest version this dll can support is %u.%u\n",

                    LOBYTE(wsaData.wHighVersion), HIBYTE(wsaData.wHighVersion));

     }

     // Create a new socket to listen for client connections.

     ListeningSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

     // Check for errors to ensure that the socket is a valid socket.

     if (ListeningSocket == INVALID\_SOCKET)

     {

          printf("Server: Error at socket(), error code: %ld.\n", WSAGetLastError());

          // Clean up

          WSACleanup();

          // and exit with error

          return 1;

     }

     else

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

     // Set up a SOCKADDR\_IN structure that will tell bind that we

     // want to listen for connections on all interfaces using port 7171.

     // The IPv4 family

     ServerAddr.sin\_family = AF\_INET;

     // host-to-network byte order

     ServerAddr.sin\_port = htons(Port);

     // Listen on all interface, host-to-network byte order

     ServerAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

     // Associate the address information with the socket using bind.

     // Call the bind function, passing the created socket and the sockaddr\_in

     // structure as parameters. Check for general errors.

     if (bind(ListeningSocket, (SOCKADDR \*)&ServerAddr, sizeof(ServerAddr)) == SOCKET\_ERROR)

     {

          printf("Server: bind() failed! Error code: %ld.\n", WSAGetLastError());

          // Close the socket

          closesocket(ListeningSocket);

          // Do the clean up

          WSACleanup();

          // and exit with error

          return 1;

     }

     else

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

     // Listen for client connections with a backlog of 5

     if (listen(ListeningSocket, 5) == SOCKET\_ERROR)

     {

          printf("Server: listen(): Error listening on socket %ld.\n", WSAGetLastError());

          // Close the socket

          closesocket(ListeningSocket);

          // Do the clean up

          WSACleanup();

          // Exit with error

          return 1;

     }

     else

          printf("Server: listen() is OK, I'm listening for connections...\n");

     // Set 10 seconds 10 useconds timeout

     SelectTiming = recvTimeOutTCP(ListeningSocket, 10, 10);

     switch (SelectTiming)

     {

          case 0:

              // Timed out, do whatever you want to handle this situation

               printf("\nServer: Timeout lor while waiting you retard client!...\n");

              break;

          case -1:

              // Error occurred, more tweaking here and the recvTimeOutTCP()...

              printf("\nServer: Some error encountered with code number: %ld\n", WSAGetLastError());

              break;

          default:

              {

                   // Accept a new connection when available. 'while' always true

                   while(1)

                   {

                        // Reset the NewConnection socket to SOCKET\_ERROR

                        // Take note that the NewConnection socket in not listening

                        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 ListeningSocket socket and assign

                             // it to the NewConnection socket, let the ListeningSocket

                             // do the listening for more connection

                             NewConnection = accept(ListeningSocket, NULL, NULL);

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

                             printf("Server: New client got connected, ready to

                                            receive and send data...\n");

                             // At this point you can do two things with these sockets

                             // 1. Wait for more connections by calling accept again

                             //    on ListeningSocket (loop)

                             // 2. Start sending or receiving data on NewConnection.

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

                             // When there is data

                             if ( ByteReceived > 0 )

                             {

                                  printf("Server: recv() looks fine....\n");

                                  // Some info on the receiver side...

                                  getsockname(ListeningSocket, (SOCKADDR \*)&ServerAddr,

                                                    (int \*)sizeof(ServerAddr));

                                  printf("Server: Receiving IP(s) used: %s\n",

                                                    inet\_ntoa(ServerAddr.sin\_addr));

                                  printf("Server: Receiving port used: %d\n", htons(ServerAddr.sin\_port));

                                  // Some info on the sender side

                                  // Allocate the required resources

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

                                  nlen = sizeof(SenderInfo);

                                  getpeername(NewConnection, (SOCKADDR \*)&SenderInfo, &nlen);

                                  printf("Server: Sending IP used: %s\n", inet\_ntoa(SenderInfo.sin\_addr));

                                  printf("Server: Sending port used: %d\n", htons(SenderInfo.sin\_port));

                                  // 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("\"");

                             }

                             // No data

                             else if ( ByteReceived == 0 )

                                  printf("Server: Connection closed!\n");

                             // Others

                             else

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

                        }

                        // Clean up all the send/recv communication, get ready for new one

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

                             printf("\nServer: Well, there is something wrong with the

                                        shutdown(). The error code: %ld\n", WSAGetLastError());

                        else

                             printf("\nServer: shutdown() looks OK...\n");

                        // Well, if there is no more connection in 15 seconds,

                        // just exit this listening loop...

                        if( recvTimeOutTCP(ListeningSocket, 15, 0) == 0)

                             break;

                   }

              }

     }

     printf("\nServer: The listening socket is timeout...\n");

     // When all the data communication and listening finished, close the socket

     if(closesocket(ListeningSocket) != 0)

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

     else

          printf("Server: Closing \"ListeningSocket\" socket...\n");

     // Finally and optionally, clean up all those WSA setup

     if(WSACleanup() != 0)

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

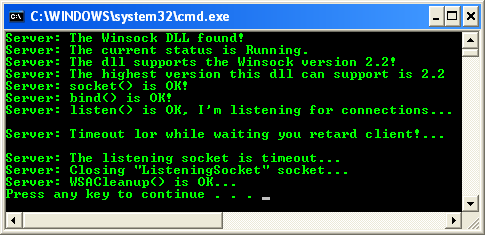
     else

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

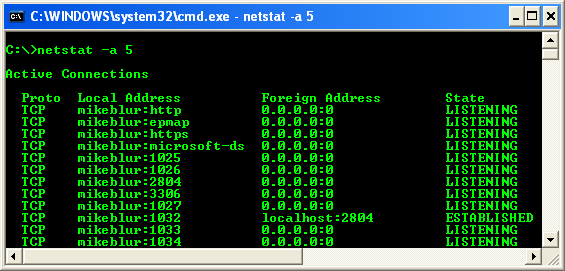
     return 0;

}

12. Build and run the project. If there is no error during the build and run processes, the following shows a sample output that should be expected. Without any connection in 10 seconds and 10 microseconds, the program terminates. We will test this receiver/server program with the sender/client which will be created in the next exercise.



 In the meantime, you may want to run the netstat tool, when you run the server program to view the TCP status. In this case we run the following netstat tool which displays all the traffic info for every 5 seconds. You may also redirect the output to a text file using: netstat –a 5 > text\_file\_name.txt.



 Then we run the server program. The status should be in the listening mode on the specified port. You can have more info on the TCP/IP states diagram at [Debugging the TCP](http://tangentsoft.net/wskfaq/articles/debugging-tcp.html).

