Some Protocol Characteristics Story 2

|  |
| --- |
| A Winsock provider implements a protocol that exhibits certain characteristics. A multitude of different transport protocols are available on Windows, such as TCP, UDP, IPX, and SPX. Each protocol behaves differently. Some require a connection to be established before sending or receiving data. Others don't guarantee the reliability or integrity of the data. In this section we'll look at the characteristics that apply to protocols.    Message-Oriented    A protocol is said to be message-oriented if for each discrete write command, it transmits only those bytes as a single message on the network. This also means that when the receiver requests data, the data returned is a discrete message written by the sender. The receiver will not get more than one message. In Figure 2-2, for example, the workstation on the left submits messages of 128, 64, and 32 bytes destined for the workstation on the right. The receiving workstation issues three recv() calls with a 256-byte buffer. Each call in succession returns 128, 64, and 32 bytes. The first call to recv() does not return all three packets even if all the packets have been received. This logic is known as “preserving message boundaries” and is often desired when structured data is exchanged. A network game is a good example of preserving message boundaries. Each player sends all other players a packet with positional information. The code behind such communication is simple: one player requests a packet of data, and that player gets exactly one packet of positional information from another player in the game.    Protocol Characteristics: Datagram services    Figure 2-2 Datagram services |

Stream-Oriented

A protocol that does not preserve message boundaries is often referred to as a stream-based protocol. Be aware that the term stream based is often loosely used to imply additional characteristics. Stream service is defined as transmitting data in a continual process; the receiver reads as much data as is available with no respect to message boundaries. For the sender, this means that the system is allowed to break up the original message into pieces or to lump several messages together to form a bigger packet of data. On the receiving end, the network stack reads data as it arrives and buffers it for the process. When the process requests an amount of data, the system returns as much data as possible without overflowing the buffer that the client call supplied. In Figure 2-3, the sender submits packets of 128, 64, and 32 bytes; however, the local system stack is free to gather the data into larger packets. In this case, the second two packets are transmitted together. The decision to lump discrete packets of data together is based on a number of factors, such as the maximum transmit unit or the Nagle algorithm. In TCP/IP, the Nagle algorithm consists of a host waiting for data to accumulate before sending it on the wire. The host will wait until it has a large enough chunk of data to send or until a predetermined amount of time elapses. When implementing the [Nagle algorithm](http://www.winsocketdotnetworkprogramming.com/xmlwebservicesaspnetworkprogramming11chap.html) the host's peer waits a predetermined amount of time for outgoing data before sending an acknowledgement to the host so the peer doesn't have to send a packet with only the acknowledgement. Sending many small packets is inefficient and adds substantial overhead for error checking and acknowledgments.

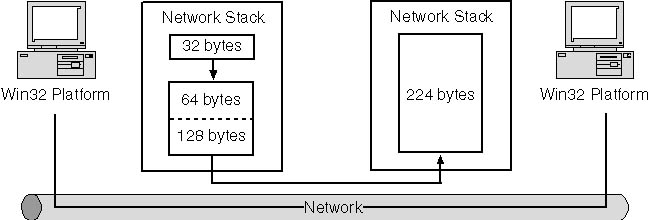


Figure 2-3 Stream services

On the receiving end, the network stack pools together all incoming data for the given process. Take a look at Figure 2-2. If the receiver performs a recv() with a 256-byte buffer, all 224 bytes are returned at once. If the receiver requests that only 20 bytes be read, the system returns only 20 bytes.

Pseudo Stream

Pseudo stream is a term often applied to a system with a message-based protocol that sends data in discrete packets, which the receiver reads and buffers in a pool so the receiving application reads data chunks of any size. Combining the sender in Figure 2-2 with the receiver in Figure 2-3 illustrates how pseudo streams work. The sender must send each individual packet separately, but the receiver is free to coalesce them in whatever sizes are available. For the most part, treat pseudo streaming as you would a stream-oriented protocol.

Connection-Oriented and Connectionless

A protocol provides either connection-oriented services or connectionless services. In connection-oriented services, a path is established between the two communicating parties before any data is exchanged. This ensures that there is a route between the two parties in addition to ensuring that both parties are alive and responding. This also means that establishing a communication channel between two participants requires substantial overhead. In addition, most connection-oriented protocols guarantee delivery, which increases overhead as additional computations are performed to verify correctness. On the other hand, a connectionless protocol makes no guarantees that the recipient is listening. A connectionless service is similar to the postal service: the sender addresses a letter to a particular person and puts it in the mail. The sender doesn't know if the recipient is expecting to receive a letter or if severe storms are preventing the post office from delivering the message.

Note that for some connectionless protocols, such as UDP, a Winsock application may call connect() with the destination's IP address but this does not imply that any physical connection is established. It is simply a convenient way to associate a destination address with the socket so that the send() and WSASend() APIs may be used instead of sendto() and WSASendTo().

Reliability and Ordering

Reliability and ordering are perhaps the most critical characteristics to be aware of when designing an application to use a particular protocol. Reliability, or guaranteed delivery, ensures that each byte of data from the sender will reach the intended recipient unaltered. An unreliable protocol does not ensure that each byte arrives, and it makes no guarantee of the data's integrity.

Ordering has to do with the order in which the data arrives at the recipient. A protocol that preserves ordering ensures that the recipient receives the data in the exact order that it was sent. Obviously, a protocol that does not preserve order makes no such guarantees.

In most cases, reliability and ordering are closely tied to whether a protocol is connectionless or connection-oriented. In the case of connection-oriented communications, if you are already making the extra effort to establish a clear communication channel between the two participants, you usually want to guarantee data integrity and data ordering. In most cases, connection-oriented protocols do guarantee reliability. Note that by ensuring packet ordering, you do not automatically guarantee data integrity. Of course, the great benefit of connectionless protocols is their speed; they don't bother to establish a virtual connection to the recipient. Why slow this down with error checking? This is why connectionless protocols generally don't guarantee data integrity or ordering and connection-oriented protocols do. Why would anyone use datagrams with all these faults? In general, connectionless protocols are much faster than connection-oriented communications. No checks need to be made for factors such as data integrity and acknowledgments of received data, factors that add a great deal of complexity to sending even small amounts of data. Datagrams are useful for noncritical data transfers. Datagrams are well suited for applications like the game example that we discussed earlier: each player can use data-grams to periodically send his or her positions within the game to every other player. If one client misses a packet, it quickly receives another, giving the player an appearance of seamless communication.

Graceful Close

A graceful close is associated with **connection-oriented protocols only**. In a graceful close, one side initiates the shutting down of a communication session and the other side still has the opportunity to read pending data on the wire or the network stack. A connection-oriented protocol that does not support graceful closes causes an immediate termination of the connection and loss of any data not read by the receiving end whenever either side closes the communication channel. In the case of TCP, each side of a connection has to perform a close to fully terminate the connection. [The initiating side sends a segment (datagram) with a FIN control flag to the peer. Upon receipt, the peer sends an ACK control flag back to the initiating side to acknowledge receipt of the FIN, but the peer is still able to send more data. The FIN control flag signifies that no more data will be sent from the side originating the close. Once the peer decides it no longer needs to send data, it too issues a FIN, which the initiator acknowledges with an ACK control flag](http://www.tenouk.com/Module42a.html). At this point, the connection has been closed completely.

Broadcast Data

To broadcast data is to send data from one workstation so that all other workstations on the LAN can receive it. This feature is available to connectionless protocols because all machines on the LAN can pick up and process a broadcast message. The drawback to using broadcast messages is that every machine has to process the message. For example, let's say the user broadcasts a message on the LAN, and the network card on each machine picks up the message and passes it up to the network stack. The stack then cycles through all network applications to see if they should receive this message. Usually, a majority of the machines on the network are not interested and simply discard the data. However, each machine still has to spend time processing the packet to see if any applications are interested in it. Consequently, high-broadcast traffic can bog down machines on a LAN as each workstation inspects the packet. In general, routers do not propagate broadcast packets between networks.

Multicast Data

Multicasting is the capability of one process to send data that one or more recipients will receive. The method by which a process joins a multicast session differs depending on the underlying protocol. For example, under the IP protocol, multicasting is a modified form of broadcasting. IP multicasting requires that all hosts interested in sending or receiving data join a special group. When a process wishes to join a multicast group, a filter is added on the network card so that data bound to that group address only will be picked up by the network hardware and propagated up the network stack to the appropriate process. Video conferencing applications often use multicasting.

Quality of Service (QOS)

QOS is an application's capability to request certain network bandwidth requirements to be dedicated for exclusive use. One good use for QOS is real-time video streaming. For the receiving end of a real-time video streaming application to display a smooth, clear picture, the data being sent must fall within certain restrictions. In the past, an application would buffer data and play back frames from the buffer to maintain a smooth video. If there was a period during which data was not being received fast enough, the playback routine had a certain number of buffered frames that it could play. QOS allows bandwidth to be reserved on the network, so frames can be read off the wire within a set of guaranteed constraints. Theoretically, this means that the same real-time video streaming application can use QOS and eliminate the need to buffer any frames.

Partial Messages

Partial messages apply to **message-oriented protocols only**. Let's say an application wants to receive a message but the local computer has received only part of the data. This can occur if the sending computer is transmitting large messages and the local machine does not have enough resources free to contain the whole message. In reality, most message-oriented protocols impose a reasonable limit on the maximum size of a datagram, so this particular event is not encountered often. However, most datagram protocols support messages large enough to require being broken into a number of smaller chunks for transmission on the physical medium. Thus the possibility exists that when a user's application requests to read a message, the user's system might have received only a portion of the message. If the protocol supports partial messages, the reader is notified that the data being returned is only a part of the whole message. If the protocol does not support partial messages, the underlying network stack holds onto the pieces until the whole message arrives. If for some reason the whole message does not arrive, most unreliable protocols that lack support for partial messages will simply discard the incomplete datagram.

Routing Considerations

One important consideration for application developers is whether a protocol is routable. If a protocol is routable, a successful communication path can be set up (either a virtual connection-oriented circuit or a data path for datagram communication) between two workstations, no matter what network hardware lies between them. For example, machine A is on a separate network from machine B. A router linking the two networks separates the two machines. A routable protocol realizes that machine B is not on the same network as machine A; therefore, the protocol directs the data to the router, which decides how to best forward it so that it reaches machine B. A nonroutable protocol is not able to make such provisions, the router drops any packets of nonroutable protocols that it receives. The router does not forward a packet from a nonroutable protocol even if the packet's intended destination is on the connected subnet. NetBEUI is the only protocol supported by Windows platforms that is not capable of being routed.

Other Characteristics

Each protocol that Windows supports has characteristics that are specialized or unique. Also, a myriad of other protocol characteristics, such as byte ordering and maximum transmission size, can be used to describe every protocol available on networks today. Not all of those characteristics are necessarily critical to writing a successful Winsock application. Winsock provides a facility to enumerate each available protocol provider and query its characteristics. The next section of this chapter explains this function and presents a code sample.

Winsock Catalog

The Winsock catalog is a database that contains the different protocols available on the system. Winsock provides a method for determining which protocols are installed on a given workstation and returning a variety of characteristics for each protocol. If a protocol is capable of multiple behaviors, each distinct behavior type has its own catalog entry within the system. For example, if you install TCP/IP on your system, there will be two IP entries: one for TCP, which is reliable and connection-oriented, and one for UDP, which is unreliable and connectionless. The function call to obtain information on installed network protocols is[WSAEnumProtocols()](http://msdn.microsoft.com/en-us/library/ms741574(VS.85).aspx) and is defined as:

int WSAEnumProtocols (

    LPINT lpiProtocols,

    LPWSAPROTOCOL\_INFO lpProtocolBuffer,

    LPDWORD lpdwBufferLength

);

This function supersedes the Winsock 1.1 function EnumProtocols(), the necessary function for older Windows CE version. The only difference is that WSAEnumProtocols() returns an array of WSAPROTOCOL\_INFO structures, whereas EnumProtocols() returns an array of PROTOCOL\_INFO structures that contain fewer fields than the WSAPROTOCOL\_INFO structure (but more or less the same information). The WSAPROTOCOL\_INFO structure is defined as:

typedef struct \_WSAPROTOCOL\_INFO {

    DWORD             dwServiceFlags1;

    DWORD             dwServiceFlags2;

    DWORD             dwServiceFlags3;

    DWORD             dwServiceFlags4;

    DWORD             dwProviderFlags;

    GUID              ProviderId;

    DWORD             dwCatalogEntryId;

    WSAPROTOCOLCHAIN  ProtocolChain;

    int               iVersion;

    int               iAddressFamily;

    int               iMaxSockAddr;

    int               iMinSockAddr;

    int               iSocketType;

    int               iProtocol;

    int               iProtocolMaxOffset;

    int               iNetworkByteOrder;

    int               iSecurityScheme;

    DWORD             dwMessageSize;

    DWORD             dwProviderReserved;

    TCHAR             szProtocol[WSAPROTOCOL\_LEN + 1];

} WSAPROTOCOL\_INFO, FAR \* LPWSAPROTOCOL\_INFO;

The easiest way to call WSAEnumProtocols() is to make the first call with lpProtocolBuffer equal to NULL and set lpdwBufferLength to 0. The call fails with WSAENOBUFS, but lpdwBufferLength then contains the correct size of the buffer required to return all the protocol information. Once you allocate the correct buffer size and make another call with the supplied buffer, the function returns the number of WSAPROTOCOL\_INFO structures returned. At this point, you can step through the structures to find the protocol entry with your required attributes.

The most commonly used field of the [WSAPROTOCOL\_INFO structure](http://msdn.microsoft.com/en-us/library/ms741675(VS.85).aspx) is dwServiceFlags1, which is a bit field for the various protocol attributes. Table 2-1 lists the various bit flags that can be set in the field and briefly describes the meaning of each property.

|  |  |
| --- | --- |
| Table 2-1 Protocol Flags | |
| **Property** | **Meaning** |
| XP1\_CONNECTIONLESS | This protocol provides connectionless service. If not set, the protocol supports connection-oriented data transfers. |
| XP1\_GUARANTEED\_DELIVERY | This protocol guarantees that all data sent will reach the intended recipient. |
| XP1\_GUARANTEED\_ORDER | This protocol guarantees that the data will arrive in the order in which it was sent and that it will not be duplicated. However, this does not guarantee delivery. |
| XP1\_MESSAGE\_ORIENTED | This protocol honors message boundaries. |
| XP1\_PSEUDO\_STREAM | This protocol is message-oriented, but the message boundaries are ignored on the receiver side. |
| XP1\_GRACEFUL\_CLOSE | This protocol supports two-phase closes: each party is notified of the other's intent to close the communication channel. If not set, only abortive closes are performed. |
| XP1\_EXPEDITED\_DATA | This protocol supports urgent data (out-of-band data). |
| XP1\_CONNECT\_DATA | This protocol supports transferring data with the connection request. |
| XP1\_DISCONNECT\_DATA | This protocol supports transferring data with the disconnect request. |
| XP1\_SUPPORT\_BROADCAST | This protocol supports the broadcast mechanism. |
| XP1\_SUPPORT\_MULTIPOINT | This protocol supports multipoint or multicast mechanisms. |
| XP1\_MULTIPOINT\_CONTROL\_ PLANE | If this flag is set, the control plane is rooted. Otherwise, it is nonrooted. |
| XP1\_MULTIPOINT\_DATA\_PLANE | If this flag is set, the data plane is rooted. Otherwise, it is nonrooted. |
| XP1\_QOS\_SUPPORTED | This protocol supports QOS requests. |
| XP1\_UNI\_SEND | This protocol is unidirectional in the send direction. |
| XP1\_UNI\_RECV | This protocol is unidirectional in the receive direction. |
| XP1\_IFS\_HANDLES | The socket descriptors returned by the provider are Installable File System (IFS) handles and can be used in API functions such as ReadFile() and WriteFile(). |
| XP1\_PARTIAL\_MESSAGE | The MSG\_PARTIAL flag is supported in WSASend() and WSASendTo(). |
| XP1\_INTERRUPT | Reserved flag. |

The other fields of importance are iProtocol, iSocketType, and iAddressFamily. The iProtocol field defines which protocol an entry belongs to. The iSocketType field is important if the protocol is capable of multiple behaviors, such as stream-oriented connections or datagram connections. Finally, iAddressFamily is used to distinguish the correct addressing structure to use for a given protocol.

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

// link with the Ws2\_32.lib

#include <winsock2.h>

// link with the Rpcrt4.lib

#include <rpc.h>

#include <stdio.h>

// Print out the contents of a WSAPROTOCOL\_INFO structure in

// a readable format.

static void PrintBufContent(WSAPROTOCOL\_INFO \*pProtocolBuf)

{

      unsigned char \*pszUuid;

      int i;

      printf("\nProtocol <%s>\n", pProtocolBuf->szProtocol);

      // A guid is the same as a uuid.

      UuidToString(&pProtocolBuf->ProviderId, (RPC\_WSTR \*)&pszUuid);

      printf("  ProviderId {%s}\n", pszUuid);

      RpcStringFree((RPC\_WSTR \*)&pszUuid);

      if (!pProtocolBuf->dwServiceFlags1)

            printf("  dwServiceFlags1: 0\n");

      else

            printf("  dwServiceFlags1: 0x%08X\n",pProtocolBuf->dwServiceFlags1);

      // Check which bit flags are set.

      printf("\nChecking the set bit flags...\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_CONNECTIONLESS)

            printf("    XP1\_CONNECTIONLESS\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_GUARANTEED\_DELIVERY)

            printf("    XP1\_GUARANTEED\_DELIVERY\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_GUARANTEED\_ORDER)

            printf("    XP1\_GUARANTEED\_ORDER\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_MESSAGE\_ORIENTED)

            printf("    XP1\_MESSAGE\_ORIENTED\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_PSEUDO\_STREAM)

            printf("    XP1\_PSEUDO\_STREAM\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_GRACEFUL\_CLOSE)

            printf("    XP1\_GRACEFUL\_CLOSE\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_EXPEDITED\_DATA)

            printf("    XP1\_EXPEDITED\_DATA\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_CONNECT\_DATA)

            printf("    XP1\_CONNECT\_DATA\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_DISCONNECT\_DATA)

            printf("    XP1\_DISCONNECT\_DATA\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_SUPPORT\_BROADCAST)

            printf("    XP1\_SUPPORT\_BROADCAST\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_SUPPORT\_MULTIPOINT)

            printf("    XP1\_SUPPORT\_MULTIPOINT\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_MULTIPOINT\_CONTROL\_PLANE)

            printf("    XP1\_MULTIPOINT\_CONTROL\_PLANE\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_MULTIPOINT\_DATA\_PLANE)

            printf("    XP1\_MULTIPOINT\_DATA\_PLANE\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_QOS\_SUPPORTED)

            printf("    XP1\_QOS\_SUPPORTED\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_INTERRUPT)

            printf("    XP1\_INTERRUPT\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_UNI\_SEND)

            printf("    XP1\_UNI\_SEND\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_UNI\_RECV)

            printf("    XP1\_UNI\_RECV\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_IFS\_HANDLES)

            printf("    XP1\_IFS\_HANDLES\n");

      if (pProtocolBuf->dwServiceFlags1 & XP1\_PARTIAL\_MESSAGE)

            printf("    XP1\_PARTIAL\_MESSAGE\n");

      printf("  dwServiceFlags2: reserved\n");

      printf("  dwServiceFlags3: reserved\n");

      printf("  dwServiceFlags4: reserved\n");

      printf("\nChecking the Provider flags...\n");

      printf("  dwProviderFlags:\n");

      if (pProtocolBuf->dwProviderFlags & PFL\_MULTIPLE\_PROTO\_ENTRIES)

            printf("    PFL\_MULTIPLE\_PROTO\_ENTRIES\n");

      if (pProtocolBuf->dwProviderFlags & PFL\_RECOMMENDED\_PROTO\_ENTRY)

            printf("    PFL\_RECOMMENDED\_PROTO\_ENTRY\n");

      if (pProtocolBuf->dwProviderFlags & PFL\_HIDDEN)

            printf("    PFL\_HIDDEN\n");

      if (pProtocolBuf->dwProviderFlags & PFL\_MATCHES\_PROTOCOL\_ZERO)

            printf("    PFL\_MATCHES\_PROTOCOL\_ZERO\n");

      printf("\nMore digging...\n");

      printf("  dwCatalogEntryId = %u\n", pProtocolBuf->dwCatalogEntryId);

      printf("  ProtocolChain.ChainLen = %d ",  pProtocolBuf->ProtocolChain.ChainLen);

      if (pProtocolBuf->ProtocolChain.ChainLen == 1)

            printf("  (This is a base service provider)\n");

      else if (pProtocolBuf->ProtocolChain.ChainLen > 1)

      {

            printf("  (ProtocolChain layered to base protocol)\n");

            for (i=0; i<pProtocolBuf->ProtocolChain.ChainLen; i++)

            {

                  printf("    Chain Catalog Entry Id = %u\n", pProtocolBuf->ProtocolChain.ChainEntries[i]);

            }

      }

      else if (pProtocolBuf->ProtocolChain.ChainLen == 0)

            printf("  (This is a layered service provider)\n");

      else

            printf("  Invalid\n");

      printf("  iVersion = %d\n", pProtocolBuf->iVersion);

      printf("  iAddressFamily = %d\n", pProtocolBuf->iAddressFamily);

      printf("  iMaxSockAddr = %d\n", pProtocolBuf->iMaxSockAddr);

      printf("  iMinSockAddr = %d\n", pProtocolBuf->iMinSockAddr);

      // iProtocols returns a negative number for Microsoft NetBIOS

      // service providers corresponding to the lana number \* -1 (for

      // example, -2 implies lana 2), except for lana 0 which is equal to

      // 0x80000000 because protocol 0 is reserved for special use.

      printf("  iProtocol = %d\n", pProtocolBuf->iProtocol);

      printf("  iProtocolMaxOffset = %d\n", pProtocolBuf->iProtocolMaxOffset);

      printf("  iNetworkByteOrder = %s\n", ((pProtocolBuf->iNetworkByteOrder == LITTLEENDIAN) ?

            "LittleEndian" : "BigEndian"));

      printf("  iSecurityScheme = %d\n", pProtocolBuf->iSecurityScheme);

      printf("  dwMessageSize = %u\n", pProtocolBuf->dwMessageSize);

      printf("  dwProviderReserved = reserved\n");

      return;

}

int main(int argc, char \*argv[])

{

      WSADATA WSAData;

      int i, nRet;

      DWORD dwErr;

      WSAPROTOCOL\_INFO \*lpProtocolBuf = NULL;

      DWORD dwBufLen = 0;

      // The WSAStartup() function directly returns the extended error code

      // in the return value for this function. A call to the WSAGetLastError

      // function is not needed and should not be used

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

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

      else

      {

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

            // First, have WSAEnumProtocols tell you how big a buffer you need.

            nRet = WSAEnumProtocols(NULL, lpProtocolBuf, &dwBufLen);

            if (nRet != SOCKET\_ERROR)

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

            else if ((dwErr = WSAGetLastError()) != WSAENOBUFS)

                  // WSAEnumProtocols() failed for some reason not relating to buffer size

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

            else

            {

                  // WSAEnumProtocols failed for the "expected" reason.

                  // Now you need to allocate a buffer that is the right size.

                  lpProtocolBuf = (WSAPROTOCOL\_INFO \*)malloc(dwBufLen);

                  if (lpProtocolBuf)

                  {

                        // Now you can call WSAEnumProtocols again with the

                        // expectation that it will succeed

                        // because you have allocated a big enough buffer.

                        nRet = WSAEnumProtocols(NULL, lpProtocolBuf, &dwBufLen);

                        if (nRet == SOCKET\_ERROR)

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

                        else

                        {

                              // Enumerate the protocols.

                              printf("Enumerating the protocols...");

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

                              {

                                    PrintBufContent(&lpProtocolBuf[i]);

                                    printf("\nPress <Enter> for more info, if any...\n");

                                    \_fgetchar();

                              }

                        }

                        // Free up the allocated resource

                        free(lpProtocolBuf);

                  }

            }

      }

      if(WSACleanup() != 0)

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

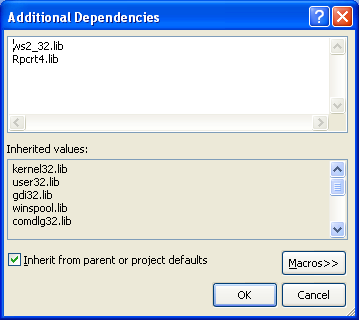
      else

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

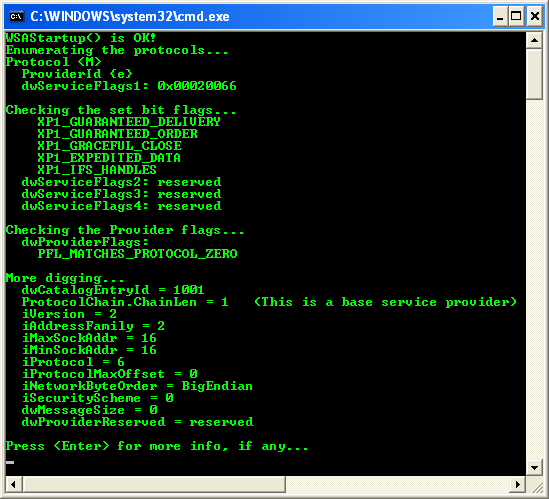
      return 0;

}

10.  Additional Dependencies



11.  Run the project.



Winsock Catalog and Win64

On 64-bit Windows, it is possible to run 32-bit applications under the [WOW64 (Windows on Windows) subsystem](http://msdn.microsoft.com/en-us/library/aa384249(VS.85).aspx). Because both 32-bit and 64-bit applications may need to access the Winsock catalog, the system maintains two separate catalogs. When a 64-bit Winsock application runs and calls WSAEnumProtocols, the 64-bit catalog is used. Likewise, when a 32-bit Winsock application calls WSAEnumProtocols, the 32-bit catalog is used. This will become more important when dealing with the Winsock Service Provider Interface (WSP\*).

Creating Sockets with WSASocket()

In Chapter 1 we saw simple examples of how to create a socket using the socket() function. This function takes three parameters: address family, socket type, and protocol. When an application creates a socket, the Winsock catalog is consulted and an attempt is made to match the supplied parameters with those contained in each WSAPROTOCOL\_INFO structure. If the parameters match, then a socket is created from that provider. Note that in some instances the protocol parameter can be 0. If the dwProviderFlags field of the WSAPROTOCOL\_INFO structure is PFL\_MATCHES\_PROTOCOL\_ZERO and if the requested address family and socket type match its entries, then that provider is used to create a socket. For example, consider the following call:

SOCKET    s;

s = socket(AF\_INET, SOCK\_STREAM, 0);

Winsock will enumerate the catalog and first match the address family followed by the socket type. Since the protocol value is 0 and the MSAFD TCP provider contains the PFL\_MATCHES\_PROTOCOL\_ZERO flag, this call will create a TCP socket from the MSAFD TCP provider. The system will not attempt to match the request to any further providers.

In some instances, multiple providers may share the same address family, socket type, and protocol. This is the case with the RSVP provider. [The RSVP provider offers QOS over TCP/IP and UDP/IP](http://msdn.microsoft.com/en-us/library/aa373396(VS.85).aspx) (Take note that MSDN said RSVP signaling is not supported on Windows XP, Windows Server 2003, or later versions of Windows.) Because multiple providers share the same address family, socket type, and protocol, there is no way to use the socket() API to create a socket from the RSVP provider. To do so, you must use the Winsock 2 function WSASocket(), which is defined as:

SOCKET WSASocket(

            int af,

            int type,

            int protocol,

            LPWSAPROTOCOL\_INFO lpProtocolInfo,

            GROUP g,

            DWORD dwFlags);

The first three parameters are the same as those of socket but the fourth parameter takes a WSAPROTOCOL\_INFO structure. If lpProtocolInfo references a provider entry and each of the first three parameters is the predefined value FROM\_PROTOCOL\_INFO, then a socket is created from the given provider. An application can create an RSVP socket using this method.

The fifth parameter deals with socket groups that are discussed in the Winsock specification but are not implemented in Windows. The last parameter is optional flags that may be passed. For now, the only flag of importance is WSA\_FLAG\_OVERLAPPED. If you plan on using overlapped IO, then this flag needs to be present when creating the socket. Note that when using the socket API, the overlapped flag is always implied. The other socket flags pertain to multicasting.

WSASocket() with WSAPROTOCOL\_INFO Program Example

#include <winsock2.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

      WSADATA WSAData;

      int i, nRet;

      WSAPROTOCOL\_INFO \*lpProtocolBuf = NULL;

      DWORD dwBufLen = 0;

      SOCKET SocDescriptor;

      DWORD dwErr;

      BOOL bProtocolFound;

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

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

      else

      {

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

            // First, have WSAEnumProtocols() tell you how big a buffer you need

            nRet = WSAEnumProtocols(NULL, lpProtocolBuf, &dwBufLen);

            printf("The buffer size is %d bytes...\n", dwBufLen);

            // If failed, returns SOCKET\_ERROR

            if (nRet != SOCKET\_ERROR)

                  printf("WSAEnumProtocols() does not return SOCKET\_ERROR!\n");

            // WSAENOBUFS - The buffer length was too small

            else if ((dwErr = WSAGetLastError()) != WSAENOBUFS)

                  // WSAEnumProtocols() failed for some reason not relating to buffer size

                  printf("WSAEnumProtocols() does not return WSAENOBUFS,\n"

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

            else

            {

                  // WSAEnumProtocols() failed for the "expected" reason. Therefore,

                  // you need to allocate a buffer of the appropriate size.

                  printf("Allocating the buffer of the appropriate size...\n");

                  lpProtocolBuf = (WSAPROTOCOL\_INFO \*)malloc(dwBufLen);

                  printf("Recheck, the buffer size is %d bytes...\n", dwBufLen);

                  if (lpProtocolBuf)

                  {

                        // Now you can call WSAEnumProtocols() again with the expectation

                        // that it will succeed because you have allocated a big enough buffer.

                        nRet = WSAEnumProtocols(NULL, lpProtocolBuf, &dwBufLen);

                        if (nRet == SOCKET\_ERROR)

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

                        else

                        {

                              printf("WSAEnumProtocols() looks pretty fine!\n");

                              // Loop through protocols, looking for the first service

                              // provider that meets the matching criteria.

                              // Originally FALSE, if found change to TRUE

                              printf("Matching the specified service provider...\n");

                              printf("    IPPROTO\_TCP and XP1\_QOS\_SUPPORTED...\n");

                              bProtocolFound = FALSE;

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

                              {

                                    if ((lpProtocolBuf[i].iProtocol == IPPROTO\_TCP) &&

                                          ((XP1\_QOS\_SUPPORTED & lpProtocolBuf[i].dwServiceFlags1) ==

                                                    XP1\_QOS\_SUPPORTED))

                                    {

                                          bProtocolFound = TRUE;

                                          // first service provider matched, exit the loop

                                          break;

                                    }

                              }

                              printf("The specified service provider found/matched! buffer index, i is

                                        %d...\n", i);

                              // Once you have chosen a service provider, the following sample code

                              // demonstrates how to actually create a socket using a supplied

                              // WSAPROTOCOL\_INFO structure

                              // from the service provider you have chosen

                              SocDescriptor = WSASocket(

                                    FROM\_PROTOCOL\_INFO, // manifest constant

                                    FROM\_PROTOCOL\_INFO, // manifest constant

                                    FROM\_PROTOCOL\_INFO, // manifest constant

                                    &lpProtocolBuf[i], // from WSAPROTOCOL\_INFO

                                    0, // reserved

                                    WSA\_FLAG\_OVERLAPPED);

                              printf("Reconfirm the buffer index, i is %d\n", i);

                              if (SocDescriptor == INVALID\_SOCKET)

                              {

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

                                    WSACleanup();

                              }

                              else

                                    printf("WSASocket() looks fine, continue on next tasks!\n");

                        }

                  }

            }

      }

      // If not needed anymore, deallocate memory to the system's heap

      free(lpProtocolBuf);

      // WSA cleanup

      if(WSACleanup() != 0)

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

      else

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

      return 0;

}

8.    Link to ws2\_32.lib

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

