# Winsock 2 I/O Methods

|  |
| --- |
| This chapter focuses on managing I/O in a Windows sockets application. Winsock features socket modes and socket I/O models to control how I/O is processed on a socket. A socket mode simply determines how Winsock functions behave when called with a socket. A socket model, on the other hand, describes how an application manages and processes I/O on a socket.  Winsock features two socket modes: **blocking** and **non-blocking**. The first part of this chapter describes these modes in detail and demonstrates how an application can use them to manage I/O. As you'll see later in the chapter, Winsock offers some interesting I/O models that help applications manage communication on one or more sockets at a time in an **asynchronous fashion**: **blocking**, **select()**, **WSAAsyncSelect()**, **WSAEventSelect()**, **overlapped I/O**, and **completion port**. By the chapter's end, we'll review the pros and cons of the various socket modes and I/O models and help you decide which one best meets your application's needs.  All Windows platforms offer blocking and non-blocking socket operating modes. However, not every I/O model is available for every platform. Every I/O model is available on Windows NT and later versions.    **Socket Modes**    As we mentioned, Windows sockets perform I/O operations in two socket operating modes: blocking and non-blocking. In blocking mode, Winsock calls that perform I/O, such as send() and recv() wait until the operation is complete before they return to the program. In non-blocking mode, the Winsock functions return immediately. Applications running on the Windows CE and Windows 95 (with Winsock 1) platforms, which support few of the I/O models, require you to take certain steps with blocking and non-blocking sockets to handle a variety of situations.    **Blocking Mode**    Blocking sockets cause concern because any Winsock API call on a blocking socket can do just that, block for some period of time. Most Winsock applications follow a producer-consumer model in which the application reads (or writes) a specified number of bytes and performs some computation on that data. The following code snippet illustrates this model: |

SOCKET  sock;

char    buff[256];

int     done = 0, nBytes;

...

while(!done)

{

    nBytes = recv(sock, buff, 65);

    if (nBytes == SOCKET\_ERROR)

     {

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

        return;

    }

    DoComputationOnData(buff);

}

...

The problem with this code is that the recv() function might never return if no data is pending because the statement says to return only after reading some bytes from the system's input buffer. Some programmers might be tempted to peek for the necessary number of bytes in the system's buffer by using the MSG\_PEEK flag in recv() or by calling ioctlsocket() with the FIONREAD option. Peeking for data without actually reading it is considered bad programming practice and should be avoided at all costs (reading the data actually removes it from the system's buffer). The overhead associated with peeking is great because one or more system calls are necessary just to check the number of bytes available. Then, of course, there is the overhead of making the recv() call that removes the data from the system buffer. To avoid this, you need to prevent the application from totally freezing because of lack of data (either from network problems or from client problems) without continually peeking at the system network buffers. One method is to **separate the application into a reading thread and a computation thread**. Both threads share a common data buffer. Access to this buffer is protected with a synchronization object, such as an **event** or a **mutex**. The purpose of the reading thread is to continually read data from the network and place it in the shared buffer. When the reading thread has read the minimum amount of data necessary for the computation thread to do its work, it can signal an event that notifies the computation thread to begin. The computation thread then removes a chunk of data from the buffer and performs the necessary calculations.

The following section of code illustrates this approach by providing two functions: one responsible for reading network data (ReadThread()) and one for performing the computations on the data (ReadThread()).

#define MAX\_BUFFER\_SIZE    4096

// Initialize critical section (data) and create

// an auto-reset event (hEvent) before creating the two threads

CRITICAL\_SECTION data;

HANDLE           hEvent;

SOCKET           sock;

TCHAR            buff[MAX\_BUFFER\_SIZE];

int              done=0;

// Create and connect sock

...

// Reader thread

void ReadThread(void)

{

            int nTotal = 0,

        nRead = 0,

        nLeft = 0,

        nBytes = 0;

    while (!done)

    {

        nTotal = 0;

        nLeft = NUM\_BYTES\_REQUIRED;

            // However many bytes constitutes enough data for processing (i.e. non-zero)

            while (nTotal != NUM\_BYTES\_REQUIRED)

            {

            EnterCriticalSection(&data);

            nRead = recv(sock, &(buff[MAX\_BUFFER\_SIZE - nBytes]), nLeft, 0);

            if (nRead == -1)

            {

                printf("error\n");

                ExitThread();

            }

            nTotal += nRead;

            nLeft -= nRead;

            nBytes += nRead;

            LeaveCriticalSection(&data);

        }

        SetEvent(hEvent);

    }

}

// Computation thread

void ProcessThread(void)

{

    WaitForSingleObject(hEvent);

    EnterCriticalSection(&data);

    DoSomeComputationOnData(buff);

    // Remove the processed data from the input

    // buffer, and shift the remaining data to the start of the array

    nBytes -= NUM\_BYTES\_REQUIRED;

    LeaveCriticalSection(&data);

}

One drawback of blocking sockets is that communicating via more than one connected socket at a time becomes difficult for the application. Using the foregoing scheme, the application could be modified to have a reading thread and a data processing thread per connected socket. This adds quite a bit of housekeeping overhead, but it is a feasible solution. The only drawback is that the solution does not scale well once you start dealing with a large number of sockets.

**Non-blocking Mode**

The alternative to blocking sockets is non-blocking sockets. Non-blocking sockets are a bit more challenging to use, but they are every bit as powerful as blocking sockets, with a few advantages. The following example illustrates how to create a socket and put it into non-blocking mode.

SOCKET        s;

unsigned long ul = 1;

int           nRet;

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

nRet = ioctlsocket(s, FIONBIO, (unsigned long \*) &ul);

if (nRet == SOCKET\_ERROR)

{

    // Failed to put the socket into non-blocking mode

}

Once a socket is placed in non-blocking mode, Winsock API calls that deal with sending and receiving data or connection management return immediately. In most cases, these calls fail with the error WSAEWOULDBLOCK, which means that the requested operation did not have time to complete during the call. For example, a call to recv() returns WSAEWOULDBLOCK if no data is pending in the system's input buffer. Often additional calls to the same function are required until it encounters a successful return code. Table 5-2 describes the meaning of WSAEWOULDBLOCK when returned by commonly used Winsock calls.

|  |  |
| --- | --- |
| Table 5-2 WSAEWOULDBLOCK Errors on Non-blocking Sockets | |
| **Function Name** | **Description** |
| WSAAccept() and accept() | The application has not received a connection request. Call again to check for a connection. |
| closesocket() | In most cases, this means that setsockopt was called with the SO\_LINGER option and a nonzero timeout was set. |
| WSAConnect() and connect() | The connection is initiated. Call again to check for completion. |
| WSARecv(), recv(), WSARecvFrom(), and recvfrom() | No data has been received. Check again later. |
| WSASend(), send(), WSASendTo(), and sendto() | No buffer space available for outgoing data. Try again later. |

Because non-blocking calls frequently fail with the WSAEWOULDBLOCK error, you should check all return codes and be prepared for failure at any time. The trap many programmers fall into is that of continually calling a function until it returns a success. For example, placing a call to recv() in a tight loop to read 200 bytes of data is no better than polling a blocking socket with the MSG\_PEEK flag mentioned previously. Winsock's socket I/O models can help an application determine when a socket is available for reading and writing.

Each socket mode, blocking and non-blocking has advantages and disadvantages. Blocking sockets are easier to use from a conceptual standpoint but become difficult to manage when dealing with multiple connected sockets or when data is sent and received in varying amounts and at arbitrary times. On the other hand, non-blocking sockets are more difficult because more code needs to be written to handle the possibility of receiving a WSAEWOULDBLOCK error on every Winsock call. Socket I/O models help applications manage communications on one or more sockets at a time in an asynchronous fashion.

**Socket I/O Models**

Essentially, six types of socket I/O models are available that allow Winsock applications to manage I/O:

1. blocking
2. select()
3. WSAAsyncSelect()
4. WSAEventSelect()
5. overlapped, and
6. completion port

This section explains the features of each I/O model and outlines how to use it to develop an application that can manage one or more socket requests.

Note that technically speaking, there could be a straight non-blocking I/O model, that is, an application that places all sockets into non-blocking mode with ioctlsocket(). However, this soon becomes unmanageable because the application will spend most of its time cycling through socket handles and I/O operations until they succeed.

**The blocking Model**

Most Winsock programmers begin with the blocking model because it is the easiest and most straightforward model. The Winsock samples in Chapter 1 use this model. As we have mentioned, applications following this model typically use one or two threads per socket connection for handling I/O. Each thread will then issue blocking operations, such as send() and recv().

The advantage to the blocking model is its simplicity. For very simple applications and rapid prototyping, this model is very useful. The disadvantage is that it does not scale up to many connections as the creation of more threads consumes valuable system resources.

**The select Model**

The select model is another I/O model widely available in Winsock. We call it the select model because it centers on using the select() function to manage I/O. The design of this model originated on UNIX-based computers featuring Berkeley socket implementations. The select model was incorporated into Winsock 1.1 to allow applications that want to avoid blocking on socket calls the capability to manage multiple sockets in an organized manner. Because Winsock 1.1 is backward-compatible with Berkeley socket implementations, a Berkeley socket application that uses the select() function should technically be able to run without modification.

The select() function can be used to determine if there is data on a socket and if a socket can be written to. The reason for having this function is to prevent your application from blocking on an I/O bound call such as send() or recv() when a socket is in a blocking mode and to prevent the WSAEWOULDBLOCK error when a socket is in a non-blocking mode. The select() function blocks for I/O operations until the conditions specified as parameters are met. The function prototype for select is as follows:

int select(

    int nfds,

    fd\_set FAR \* readfds,

    fd\_set FAR \* writefds,

    fd\_set FAR \* exceptfds,

    const struct timeval FAR \* timeout

);

The first parameter, nfds, is ignored and is included only for compatibility with Berkeley socket applications. You'll notice that there are three fd\_set parameters:

1. One for checking readability (readfds),
2. One for writeability (writefds), and
3. One for out-of-band data (exceptfds).

Essentially, the fd\_set data type represents a collection of sockets. The readfds set identifies sockets that meet one of the following conditions:

1. Data is available for reading.
2. Connection has been closed, reset, or terminated.
3. If listen() has been called and a connection is pending, the accept() function will succeed.

The writefds set identifies sockets in which one of the following is true:

1. Data can be sent.
2. If a non-blocking connect() call is being processed, the connection has succeeded.

Finally, the exceptfds set identifies sockets in which one of the following is true:

1. If a non-blocking connect() call is being processed, the connection attempt failed.
2. OOB data is available for reading.

For example, when you want to test a socket for readability, you must add it to the readfds set and wait for the select() function to complete. When the select() call completes, you have to determine if your socket is still part of the readfds set. If so, the socket is readable, you can begin to retrieve data from it. Any two of the three parameters (readfds, writefds, exceptfds) can be null values (at least one must not be null), and any non-null set must contain at least one socket handle; otherwise, the select() function won't have anything to wait for. The final parameter, timeout, is a pointer to a timeval structure that determines how long the select() function will wait for I/O to complete. If timeout is a null pointer, select() will block indefinitely until at least one descriptor meets the specified criteria. The timeval structure is defined as:

struct timeval

{

    long tv\_sec;

    long tv\_usec;

};

The tv\_sec field indicates how long to wait in seconds; the tv\_usec field indicates how long to wait in milliseconds. The timeout value {0, 0} indicates select() will return immediately, allowing an application to poll on the select() operation. This should be avoided for performance reasons. When select() completes successfully, it returns the total number of socket handles that have I/O operations pending in the fd\_set structures. If the timeval limit expires, it returns 0. If select() fails for any reason, it returns SOCKET\_ERROR.

Before you can begin to use select() to monitor sockets, your application has to set up either one or all of the read, write, and exception fd\_set structures by assigning socket handles to a set. When you assign a socket to one of the sets, you are asking select() to let you know if the I/O activities just described have occurred on a socket. Winsock provides the following set of macros to manipulate and check the fd\_set sets for I/O activity.

1. FD\_ZERO(\*set) - Initializes set to the empty set. A set should always be cleared before using.
2. FD\_CLR(s, \*set) - Removes socket s from set.
3. FD\_ISSET(s, \*set) - Checks to see if s is a member of set and returns TRUE if so.
4. FD\_SET(s, \*set) - Adds socket s to set.

For example, if you want to find out when it is safe to read data from a socket without blocking, simply assign your socket to the fd\_read set using the FD\_SET macro and then call select(). To test whether your socket is still part of the fd\_read set, use the FD\_ISSET macro. The following five steps describe the basic flow of an application that uses select with one or more socket handles:

1. Initialize each fd\_set of interest by using the FD\_ZERO macro.
2. Assign socket handles to each of the fd\_set sets of interest by using the FD\_SET macro.
3. Call the select() function and wait until I/O activity sets one or more of the socket handles in each fd\_set set provided. When select() completes, it returns the total number of socket handles that are set in all of the fd\_set sets and updates each set accordingly.
4. Using the return value of select(), your application can determine which application sockets have I/O pending by checking each fd\_set set using the FD\_ISSET macro.
5. After determining which sockets have I/O pending in each of the sets, process the I/O and go to step 1 to continue the select process.

When select() returns, it modifies each of the fd\_set structures by removing the socket handles that do not have pending I/O operations. This is why you should use the FD\_ISSET macro as in step 4 to determine if a particular socket is part of a set. The following code sample outlines the basic steps needed to set up the select() model for a single socket. Adding more sockets to this application simply involves maintaining a list or an array of additional sockets.

SOCKET  s;

fd\_set  fdread;

int     ret;

// Create a socket, and accept a connection

// Manage I/O on the socket

while(TRUE)

{

    // Always clear the read set before calling select()

    FD\_ZERO(&fdread);

    // Add socket s to the read set

    FD\_SET(s, &fdread);

    if ((ret = select(0, &fdread, NULL, NULL, NULL)) == SOCKET\_ERROR)

    {

        // Error condition

    }

    if (ret > 0)

    {

        // For this simple case, select() should return

        // the value 1. An application dealing with

        // more than one socket could get a value

        // greater than 1. At this point, your

        // application should check to see whether the socket is part of a set.

        if (FD\_ISSET(s, &fdread))

        {

            // A read event has occurred on socket s

        }

    }

}

The advantage of using select() is the capability to multiplex connections and I/O on many sockets from a single thread. This prevents the explosion of threads associated with blocking sockets and multiple connections. The disadvantage is the **maximum number of sockets** that may be added to the fd\_set structures. By default, the maximum is defined as FD\_SETSIZE, which is defined in WINSOCK2.H as 64. To increase this limit, an application might define FD\_SETSIZE to something large. This define must appear before including WINSOCK2.H. Also, the underlying provider imposes an arbitrary maximum fd\_set size, which typically is 1024 but is not guaranteed to be. Finally, for a large FD\_SETSIZE, consider the performance hit of setting 1000 sockets before calling select() followed by checking whether each of those 1000 sockets is set after the call returns.

|  |
| --- |
| The select() Program Example   The following program example demonstrates the use of select() for server/receiver program.  Winsock 2 socket I/O Methods: A program example demonstrates the use of select() for server/receiver program - Create a new empty Win32 console mode application and add the project/solution name.   Add the following source code. |

 // Description:

//

//    This sample illustrates how to develop a simple echo server Winsock

//    application using the select() API I/O model. This sample is

//    implemented as a console-style application and simply prints

//    messages when connections are established and removed from the server.

//    The application listens for TCP connections on port 5150 and accepts

//    them as they arrive. When this application receives data from a client,

//    it simply echos (this is why we call it an echo server) the data back in

//    it's original form until the client closes the connection.

//

//    Note: There are no command line options for this sample.

//

// Link to ws2\_32.lib

#include <winsock2.h>

#include <windows.h>

#include <stdio.h>

#define PORT 5150

#define DATA\_BUFSIZE 8192

typedef struct \_SOCKET\_INFORMATION {

   CHAR Buffer[DATA\_BUFSIZE];

   WSABUF DataBuf;

   SOCKET Socket;

   OVERLAPPED Overlapped;

   DWORD BytesSEND;

   DWORD BytesRECV;

} SOCKET\_INFORMATION, \* LPSOCKET\_INFORMATION;

// Prototypes

BOOL CreateSocketInformation(SOCKET s);

void FreeSocketInformation(DWORD Index);

// Global var

DWORD TotalSockets = 0;

LPSOCKET\_INFORMATION SocketArray[FD\_SETSIZE];

int main(int argc, char \*\*argv)

{

   SOCKET ListenSocket;

   SOCKET AcceptSocket;

   SOCKADDR\_IN InternetAddr;

   WSADATA wsaData;

   INT Ret;

   FD\_SET WriteSet;

   FD\_SET ReadSet;

   DWORD i;

   DWORD Total;

   ULONG NonBlock;

   DWORD Flags;

   DWORD SendBytes;

   DWORD RecvBytes;

   if ((Ret = WSAStartup(0x0202,&wsaData)) != 0)

   {

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

      WSACleanup();

      return 1;

   }

   else

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

   // Prepare a socket to listen for connections

   if ((ListenSocket = WSASocket(AF\_INET, SOCK\_STREAM, 0, NULL, 0, WSA\_FLAG\_OVERLAPPED)) == INVALID\_SOCKET)

   {

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

      return 1;

   }

   else

      printf("WSASocket() is OK!\n");

   InternetAddr.sin\_family = AF\_INET;

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

   InternetAddr.sin\_port = htons(PORT);

   if (bind(ListenSocket, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr)) == SOCKET\_ERROR)

   {

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

      return 1;

   }

   else

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

   if (listen(ListenSocket, 5))

   {

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

      return 1;

   }

   else

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

   // Change the socket mode on the listening socket from blocking to

   // non-block so the application will not block waiting for requests

   NonBlock = 1;

   if (ioctlsocket(ListenSocket, FIONBIO, &NonBlock) == SOCKET\_ERROR)

   {

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

      return 1;

   }

   else

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

   while(TRUE)

   {

      // Prepare the Read and Write socket sets for network I/O notification

      FD\_ZERO(&ReadSet);

      FD\_ZERO(&WriteSet);

      // Always look for connection attempts

      FD\_SET(ListenSocket, &ReadSet);

      // Set Read and Write notification for each socket based on the

      // current state the buffer.  If there is data remaining in the

      // buffer then set the Write set otherwise the Read set

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

         if (SocketArray[i]->BytesRECV > SocketArray[i]->BytesSEND)

            FD\_SET(SocketArray[i]->Socket, &WriteSet);

         else

            FD\_SET(SocketArray[i]->Socket, &ReadSet);

      if ((Total = select(0, &ReadSet, &WriteSet, NULL, NULL)) == SOCKET\_ERROR)

      {

         printf("select() returned with error %d\n", WSAGetLastError());

         return 1;

      }

      else

         printf("select() is OK!\n");

      // Check for arriving connections on the listening socket.

      if (FD\_ISSET(ListenSocket, &ReadSet))

      {

         Total--;

         if ((AcceptSocket = accept(ListenSocket, NULL, NULL)) != INVALID\_SOCKET)

         {

            // Set the accepted socket to non-blocking mode so the server will

            // not get caught in a blocked condition on WSASends

            NonBlock = 1;

            if (ioctlsocket(AcceptSocket, FIONBIO, &NonBlock) == SOCKET\_ERROR)

            {

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

               return 1;

            }

            else

               printf("ioctlsocket(FIONBIO) is OK!\n");

            if (CreateSocketInformation(AcceptSocket) == FALSE)

            {

                 printf("CreateSocketInformation(AcceptSocket) failed!\n");

                 return 1;

            }

            else

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

         }

         else

         {

            if (WSAGetLastError() != WSAEWOULDBLOCK)

            {

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

               return 1;

            }

            else

               printf("accept() is fine!\n");

         }

      }

      // Check each socket for Read and Write notification until the number

      // of sockets in Total is satisfied

      for (i = 0; Total > 0 && i < TotalSockets; i++)

      {

         LPSOCKET\_INFORMATION SocketInfo = SocketArray[i];

         // If the ReadSet is marked for this socket then this means data

         // is available to be read on the socket

         if (FD\_ISSET(SocketInfo->Socket, &ReadSet))

         {

            Total--;

            SocketInfo->DataBuf.buf = SocketInfo->Buffer;

            SocketInfo->DataBuf.len = DATA\_BUFSIZE;

            Flags = 0;

            if (WSARecv(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &RecvBytes, &Flags, NULL, NULL) == SOCKET\_ERROR)

            {

               if (WSAGetLastError() != WSAEWOULDBLOCK)

               {

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

                  FreeSocketInformation(i);

               }

               else

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

               continue;

            }

            else

            {

               SocketInfo->BytesRECV = RecvBytes;

               // If zero bytes are received, this indicates the peer closed the connection.

               if (RecvBytes == 0)

               {

                  FreeSocketInformation(i);

                  continue;

               }

            }

         }

         // If the WriteSet is marked on this socket then this means the internal

         // data buffers are available for more data

         if (FD\_ISSET(SocketInfo->Socket, &WriteSet))

         {

            Total--;

            SocketInfo->DataBuf.buf = SocketInfo->Buffer + SocketInfo->BytesSEND;

            SocketInfo->DataBuf.len = SocketInfo->BytesRECV - SocketInfo->BytesSEND;

            if (WSASend(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &SendBytes, 0, NULL, NULL) == SOCKET\_ERROR)

            {

               if (WSAGetLastError() != WSAEWOULDBLOCK)

               {

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

                  FreeSocketInformation(i);

               }

               else

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

               continue;

            }

            else

            {

               SocketInfo->BytesSEND += SendBytes;

               if (SocketInfo->BytesSEND == SocketInfo->BytesRECV)

               {

                  SocketInfo->BytesSEND = 0;

                  SocketInfo->BytesRECV = 0;

               }

            }

         }

      }

   }

}

BOOL CreateSocketInformation(SOCKET s)

{

   LPSOCKET\_INFORMATION SI;

   printf("Accepted socket number %d\n", s);

   if ((SI = (LPSOCKET\_INFORMATION) GlobalAlloc(GPTR, sizeof(SOCKET\_INFORMATION))) == NULL)

   {

      printf("GlobalAlloc() failed with error %d\n", GetLastError());

      return FALSE;

   }

   else

      printf("GlobalAlloc() for SOCKET\_INFORMATION is OK!\n");

   // Prepare SocketInfo structure for use

   SI->Socket = s;

   SI->BytesSEND = 0;

   SI->BytesRECV = 0;

   SocketArray[TotalSockets] = SI;

   TotalSockets++;

   return(TRUE);

}

void FreeSocketInformation(DWORD Index)

{

   LPSOCKET\_INFORMATION SI = SocketArray[Index];

   DWORD i;

   closesocket(SI->Socket);

   printf("Closing socket number %d\n", SI->Socket);

   GlobalFree(SI);

   // Squash the socket array

   for (i = Index; i < TotalSockets; i++)

   {

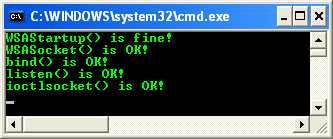
      SocketArray[i] = SocketArray[i + 1];

   }

   TotalSockets--;

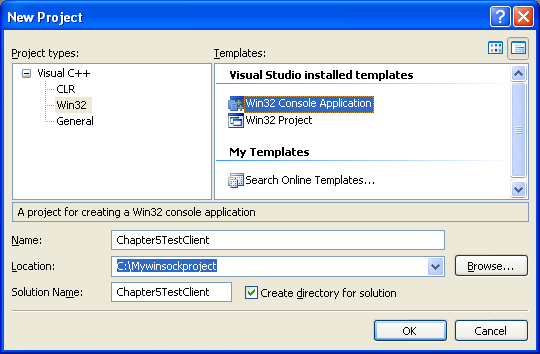
}

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



# The Client Test Program Example

The following program example is a client that will be used to test all the server/receiver program created in this chapter. This program uses TCP protocol.



Add the following source code.

 // Description:

//    This sample is the echo client. It connects to the TCP server,

//    sends data, and reads data back from the server.

//

// Command Line Options:

//    client [-p:x] [-s:IP] [-n:x] [-o]

//           -p:x      Remote port to send to

//           -s:IP     Server's IP address or hostname

//           -n:x      Number of times to send message

//           -o        Send messages only; don't receive

//

// Link to ws2\_32.lib

#include <winsock2.h>

#include <stdio.h>

#include <stdlib.h>

#define DEFAULT\_COUNT       20

#define DEFAULT\_PORT        5150

#define DEFAULT\_BUFFER      2048

#define DEFAULT\_MESSAGE     "\'A test message from client\'"

char  szServer[128],                          // Server to connect to

      szMessage[1024];                      // Message to send to sever

int   iPort     = DEFAULT\_PORT;    // Port on server to connect to

DWORD dwCount   = DEFAULT\_COUNT; // Number of times to send message

BOOL  bSendOnly = FALSE;                        // Send data only; don't receive

// Function: usage:

// Description: Print usage information and exit

void usage()

{

    printf("Chapter5TestClient: client [-p:x] [-s:IP] [-n:x] [-o]\n\n");

    printf("       -p:x      Remote port to send to\n");

    printf("       -s:IP     Server's IP address or hostname\n");

    printf("       -n:x      Number of times to send message\n");

    printf("       -o        Send messages only; don't receive\n");

    printf("\n");

}

// Function: ValidateArgs

// Description:

//    Parse the command line arguments, and set some global flags

//    to indicate what actions to perform

void ValidateArgs(int argc, char \*\*argv)

{

    int    i;

    for(i = 1; i < argc; i++)

    {

        if ((argv[i][0] == '-') || (argv[i][0] == '/'))

        {

            switch (tolower(argv[i][1]))

            {

                case 'p':        // Remote port

                    if (strlen(argv[i]) > 3)

                        iPort = atoi(&argv[i][3]);

                    break;

                case 's':       // Server

                    if (strlen(argv[i]) > 3)

                        strcpy\_s(szServer, sizeof(szServer),&argv[i][3]);

                    break;

                case 'n':       // Number of times to send message

                    if (strlen(argv[i]) > 3)

                        dwCount = atol(&argv[i][3]);

                    break;

                case 'o':       // Only send message; don't receive

                    bSendOnly = TRUE;

                    break;

                default:

                    usage();

                    break;

            }

        }

    }

}

// Function: main

// Description:

//    Main thread of execution. Initialize Winsock, parse the

//    command line arguments, create a socket, connect to the

//    server, and then send and receive data.

int main(int argc, char \*\*argv)

{

    WSADATA       wsd;

    SOCKET        sClient;

    char          szBuffer[DEFAULT\_BUFFER];

    int           ret, i;

    struct sockaddr\_in server;

    struct hostent    \*host = NULL;

            if(argc < 2)

            {

                        usage();

                        exit(1);

            }

    // Parse the command line and load Winsock

    ValidateArgs(argc, argv);

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

    {

        printf("Failed to load Winsock library! Error %d\n", WSAGetLastError());

        return 1;

    }

    else

        printf("Winsock library loaded successfully!\n");

    strcpy\_s(szMessage, sizeof(szMessage),DEFAULT\_MESSAGE);

    // Create the socket, and attempt to connect to the server

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

    if (sClient == INVALID\_SOCKET)

    {

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

        return 1;

    }

    else

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

    server.sin\_family = AF\_INET;

    server.sin\_port = htons(iPort);

    server.sin\_addr.s\_addr = inet\_addr(szServer);

    // If the supplied server address wasn't in the form

    // "aaa.bbb.ccc.ddd" it's a hostname, so try to resolve it

    if (server.sin\_addr.s\_addr == INADDR\_NONE)

    {

        host = gethostbyname(szServer);

        if (host == NULL)

        {

            printf("Unable to resolve server %s\n", szServer);

            return 1;

        }

        else

            printf("The hostname resolved successfully!\n");

        CopyMemory(&server.sin\_addr, host->h\_addr\_list[0], host->h\_length);

    }

    if (connect(sClient, (struct sockaddr \*)&server, sizeof(server)) == SOCKET\_ERROR)

    {

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

        return 1;

    }

    else

        printf("connect() is pretty damn fine!\n");

    // Send and receive data

    printf("Sending and receiving data if any...\n");

    for(i = 0; i < (int)dwCount; i++)

    {

        ret = send(sClient, szMessage, strlen(szMessage), 0);

        if (ret == 0)

            break;

        else if (ret == SOCKET\_ERROR)

        {

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

            break;

        }

        printf("send() should be fine. Send %d bytes\n", ret);

        if (!bSendOnly)

        {

            ret = recv(sClient, szBuffer, DEFAULT\_BUFFER, 0);

            if (ret == 0)        // Graceful close

            {

                   printf("It is a graceful close!\n");

                   break;

            }

            else if (ret == SOCKET\_ERROR)

            {

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

                break;

            }

            szBuffer[ret] = '\0';

            printf("recv() is OK. Received %d bytes: %s\n", ret, szBuffer);

           }

    }

    if(closesocket(sClient) == 0)

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

    else

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

    if (WSACleanup() == 0)

            printf("WSACleanup() is fine!\n");

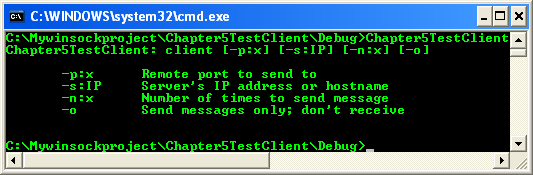
    else

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

    return 0;

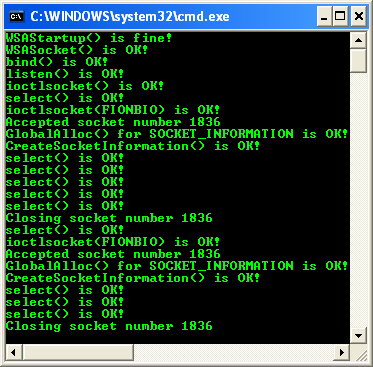
}

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



|  |
| --- |
| **The Client-server Testing**    We will test the client and server communication. Firstly we run the server.    Winsock 2 socket I/O Methods: Running the select() server program    Then we run the client form local computer in the same subnet.    Winsock 2 socket I/O Methods: Running the select() client program |

The previous server screenshot sample output is shown below.



**The WSAAsyncSelect() Model**

Winsock provides a useful asynchronous I/O model that allows an application to receive Windows message–based notification of network events on a socket. This is accomplished by calling the WSAAsyncSelect() function after creating a socket. Before we continue, however, we need to make one subtle distinction. The WSAAsyncSelect() and WSAEventSelect() models provide asynchronous notification of the capability to read or write data. It does not provide asynchronous data transfer like the overlapped and completion port models.

This model originally existed in Winsock 1.1 implementations to help application programmers cope with the cooperative multitasking message-based environment of 16-bit Windows platforms, such as Windows for Workgroups. Applications can still benefit from this model, especially if they manage window messages in a standard Windows procedure, usually referred to as a winproc. This model is also used by the Microsoft Foundation Class (MFC) CSocket object.

**Message Notification**

To use the WSAAsyncSelect() model, your application must first create a window using the CreateWindow() function and supply a window procedure (winproc) support function for it. You can also use a dialog box with a dialog procedure instead of a window because dialog boxes are windows. For our purposes, we will demonstrate this model using a simple window with a supporting window procedure. Once you have set up the window infrastructure, you can begin creating sockets and turning on window message notification by calling the WSAAsyncSelect() function, which is defined as:

int WSAAsyncSelect(

    SOCKET s,

    HWND hWnd,

    unsigned int wMsg,

    long lEvent

);

The s parameter represents the socket we are interested in. The hWnd parameter is a window handle identifying the window or the dialog box that receives a message when a network event occurs. The wMsg parameter identifies the message to be received when a network event occurs. This message is posted to the window that is identified by the hWnd window handle. Applications usually set this message to a value greater than the Windows WM\_USER value to avoid confusing a network window message with a predefined standard window message. The last parameter, lEvent, represents a bitmask that specifies a combination of network events, listed in Table 5-3, that the application is interested in. Most applications are typically interested in the FD\_READ, FD\_WRITE, FD\_ACCEPT, FD\_CONNECT, and FD\_CLOSE network event types. Of course, the use of the FD\_ACCEPT or the FD\_CONNECT type depends on whether your application is a client or a server. If your application is interested in more than one network event, simply set this field by performing a bitwise OR on the types and assigning them to lEvent. For example:

WSAAsyncSelect(s, hwnd, WM\_SOCKET, FD\_CONNECT │ FD\_READ │ FD\_WRITE │ FD\_CLOSE);

This allows our application to get connect, send, receive, and socket-closure network event notifications on socket s. It is impossible to register multiple events one at a time on the socket. Also note that once you turn on event notification on a socket, it remains on unless the socket is closed by a call to closesocket or the application changes the registered network event types by calling WSAAsyncSelect() (again, on the socket). Setting the lEvent parameter to 0 effectively stops all network event notification on the socket.

When your application calls WSAAsyncSelect() on a socket, the socket mode is automatically changed from blocking to the non-blocking mode that we described previously. As a result, if a Winsock I/O call such as WSARecv() is called and has to wait for data, it will fail with error WSAEWOULDBLOCK. To avoid this error, applications should rely on the user-defined window message specified in the wMsg parameter of WSAAsyncSelect() to indicate when network event types occur on the socket.

|  |  |
| --- | --- |
| Table 5-3 Network Event Types for the WSAAsyncSelect() Function | |
| **Event Type** | **Meaning** |
| FD\_READ | The application wants to receive notification of readiness for reading. |
| FD\_WRITE | The application wants to receive notification of readiness for writing. |
| FD\_OOB | The application wants to receive notification of the arrival of OOB data. |
| FD\_ACCEPT | The application wants to receive notification of incoming connections. |
| FD\_CONNECT | The application wants to receive notification of a completed connection or a multipoint join operation. |
| FD\_CLOSE | The application wants to receive notification of socket closure. |
| FD\_QOS | The application wants to receive notification of socket QOS changes. |
| FD\_GROUP\_QOS | The application wants to receive notification of socket group QOS changes (reserved for future use with socket groups). |
| FD\_ROUTING\_INTERFACE\_CHANGE | The application wants to receive notification of routing interface changes for the specified destination(s). |
| FD\_ADDRESS\_LIST\_CHANGE | The application wants to receive notification of local address list changes for the socket's protocol family. |

After your application successfully calls WSAAsyncSelect() on a socket, the application begins to receive network event notification as Windows messages in the window procedure associated with the hWnd parameter window handle. A window procedure is normally defined as:

LRESULT CALLBACK WindowProc(

    HWND hWnd,

    UINT uMsg,

    WPARAM wParam,

    LPARAM lParam

);

The hWnd parameter is a handle to the window that invoked the window procedure. The uMsg parameter indicates which message needs to be processed. In your case, you will be looking for the message defined in the WSAAsyncSelect() call. The wParam parameter identifies the socket on which a network event has occurred. This is important if you have more than one socket assigned to this window procedure. The lParam parameter contains two important pieces of information, the low word of lParam specifies the network event that has occurred, and the high word of lParam contains any error code.

When network event messages arrive at a window procedure, the application should first check the lParam high-word bits to determine whether a network error has occurred on the socket. There is a special macro, WSAGETSELECTERROR, that returns the value of the high-word bits error information. After the application has verified that no error occurred on the socket, the application should determine which network event type caused the Windows message to fire by reading the low-word bits of lParam. Another special macro, WSAGETSELECTEVENT, returns the value of the low-word portion of lParam.

The following example demonstrates how to manage window messages when using the WSAAsyncSelect() I/O model. The code highlights the steps needed to develop a basic server application and removes the programming details of developing a fully featured Windows application.

#define WM\_SOCKET WM\_USER + 1

#include <winsock2.h>

#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

{

    WSADATA wsd;

    SOCKET Listen;

    SOCKADDR\_IN InternetAddr;

    HWND Window;

    // Create a window and assign the ServerWinProc below to it

    Window = CreateWindow();

    // Start Winsock and create a socket

    WSAStartup(MAKEWORD(2, 2), &wsd);

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

    // Bind the socket to port 5150 and begin listening for connections

    InternetAddr.sin\_family = AF\_INET;

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

    InternetAddr.sin\_port = htons(5150);

    bind(Listen, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr));

    // Set up window message notification on

    // the new socket using the WM\_SOCKET define above

    WSAAsyncSelect(Listen, Window, WM\_SOCKET, FD\_ACCEPT │ FD\_CLOSE);

    listen(Listen, 5);

    // Translate and dispatch window messages until the application terminates

    while (1)

{

     // ...

}

}

BOOL CALLBACK ServerWinProc(HWND hDlg, UINT wMsg, WPARAM wParam, LPARAM lParam)

{

    SOCKET Accept;

    switch(wMsg)

    {

        case WM\_PAINT:

            // Process window paint messages

            break;

        case WM\_SOCKET:

            // Determine whether an error occurred on the

            // socket by using the WSAGETSELECTERROR() macro

            if (WSAGETSELECTERROR(lParam))

            {

                 // Display the error and close the socket

                closesocket( (SOCKET) wParam);

                break;

            }

            // Determine what event occurred on the socket

            switch(WSAGETSELECTEVENT(lParam))

            {

                case FD\_ACCEPT:

                    // Accept an incoming connection

                    Accept = accept(wParam, NULL, NULL);

                    // Prepare accepted socket for read, write, and close notification

                    WSAAsyncSelect(Accept, hDlg, WM\_SOCKET, FD\_READ │ FD\_WRITE │ FD\_CLOSE);

                    break;

                case FD\_READ:

                    // Receive data from the socket in wParam

                    break;

                case FD\_WRITE:

                    // The socket in wParam is ready for sending data

                    break;

                case FD\_CLOSE:

                    // The connection is now closed

                    closesocket((SOCKET)wParam);

                    break;

            }

            break;

    }

    return TRUE;

}

One final detail worth noting is how applications should process FD\_WRITE event notifications. FD\_WRITE notifications are sent under only three conditions:

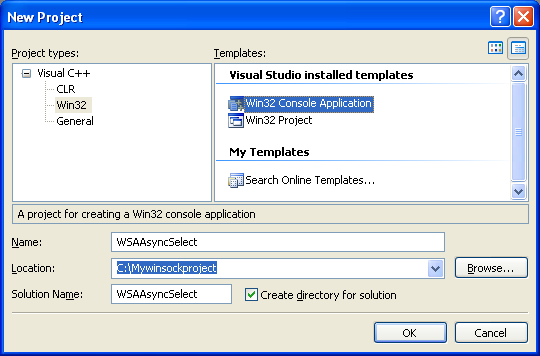
1. After a socket is first connected with connect or WSAConnect().
2. After a socket is accepted with accept or WSAAccept().
3. When a send(), WSASend(), sendto(), or WSASendTo() operation fails with WSAEWOULDBLOCK and buffer space becomes available.

Therefore, an application should assume that sends are always possible on a socket starting from the first FD\_WRITE message and lasting until a send(), WSASend(), sendto, or WSASendTo() returns the socket error WSAEWOULDBLOCK. After such failure, another FD\_WRITE message notifies the application that sends are once again possible.

The WSAAsyncSelect() model offers many advantages; foremost is the capability to handle many connections simultaneously without much overhead, unlike the select model's requirement of setting up the fd\_set structures. The disadvantages are having to use a window if your application requires no windows (such as a service or console application). Also, having a single window procedure to service all the events on thousands of socket handles can become a performance bottleneck (meaning this model doesn't scale very well).

**The WSAAsyncSelect() Model Program Example**

The following example shows a server which implementing the WSAAsyncSelect() model.



Add the following source code.

// Description:

//

//    This sample illustrates how to develop a simple echo server Winsock

//    application using the WSAAsyncSelect() I/O model. This sample is

//    implemented as a console-style application (to reduce the programming

//    complexity of writing a real Windows application) and simply prints

//    messages when connections are established and removed from the server.

//    The application listens for TCP connections on port 5150 and accepts them

//    as they arrive. When this application receives data from a client, it

//    simply echoes the data back in it's original form until the client

//            closes the connection.

//

//    Since the WSAAsyncSelect I/O model requires an application to manage

//    window messages when network event occur, this application creates

//    a window for the I/O model only. The window stays hidden during the

//    entire execution of this application.

//

//    Note: There are no command line options for this sample.

//

// Discard unnecessary/unused headers

#define WIN32\_LEAN\_AND\_MEAN

// Take note that windows.h already contained winsock.h! And by

// putting winsock2.h first, it will block the winsock.h re-inclusion

// Link to ws2\_32.lib

#include <winsock2.h>

#include <windows.h>

#include <stdio.h>

#include <conio.h>

#define PORT 5150

#define DATA\_BUFSIZE 8192

// typedef definition

typedef struct \_SOCKET\_INFORMATION {

   BOOL RecvPosted;

   CHAR Buffer[DATA\_BUFSIZE];

   WSABUF DataBuf;

   SOCKET Socket;

   DWORD BytesSEND;

   DWORD BytesRECV;

   struct \_SOCKET\_INFORMATION \*Next;

} SOCKET\_INFORMATION, \*LPSOCKET\_INFORMATION;

#define WM\_SOCKET (WM\_USER + 1)

// Prototypes

void CreateSocketInformation(SOCKET s);

LPSOCKET\_INFORMATION GetSocketInformation(SOCKET s);

void FreeSocketInformation(SOCKET s);

HWND MakeWorkerWindow(void);

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

// Global var

LPSOCKET\_INFORMATION SocketInfoList;

int main(int argc, char \*\*argv)

{

   MSG msg;

   DWORD Ret;

   SOCKET Listen;

   SOCKADDR\_IN InternetAddr;

   HWND Window;

   WSADATA wsaData;

   if ((Window = MakeWorkerWindow()) == NULL)

   {

      printf("MakeWorkerWindow() failed!\n");

      return 1;

   }

   else

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

   // Prepare echo server

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

   {

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

      return 1;

   }

   else

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

   if ((Listen = socket(PF\_INET, SOCK\_STREAM, 0)) == INVALID\_SOCKET)

   {

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

      return 1;

   }

   else

      printf("socket() is pretty fine!\n");

   if(WSAAsyncSelect(Listen, Window, WM\_SOCKET, FD\_ACCEPT|FD\_CLOSE) == 0)

      printf("WSAAsyncSelect() is OK lol!\n");

   else

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

   InternetAddr.sin\_family = AF\_INET;

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

   InternetAddr.sin\_port = htons(PORT);

   if (bind(Listen, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr)) == SOCKET\_ERROR)

   {

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

      return 1;

   }

   else

      printf("bind() is OK maaa!\n");

   if (listen(Listen, 5))

   {

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

      return 1;

   }

   else

      printf("listen() is also OK! I am listening now...\n");

   // Translate and dispatch window messages for the application thread

   while(Ret = GetMessage(&msg, NULL, 0, 0))

   {

      if (Ret == -1)

      {

         printf("\nGetMessage() failed with error %d\n", GetLastError());

         return 1;

      }

      else

        printf("\nGetMessage() is pretty fine!\n");

      printf("Translating a message...\n");

      TranslateMessage(&msg);

      printf("Dispatching a message...\n");

      DispatchMessage(&msg);

   }

}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

   SOCKET Accept;

   LPSOCKET\_INFORMATION SocketInfo;

   DWORD RecvBytes;

   DWORD SendBytes;

   DWORD Flags;

   if (uMsg == WM\_SOCKET)

   {

      if (WSAGETSELECTERROR(lParam))

      {

         printf("Socket failed with error %d\n", WSAGETSELECTERROR(lParam));

         FreeSocketInformation(wParam);

      }

      else

      {

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

         switch(WSAGETSELECTEVENT(lParam))

         {

            case FD\_ACCEPT:

               if ((Accept = accept(wParam, NULL, NULL)) == INVALID\_SOCKET)

               {

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

                  break;

               }

               else

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

               // Create a socket information structure to associate with the socket for processing I/O

               CreateSocketInformation(Accept);

               printf("Socket number %d connected\n", Accept);

               WSAAsyncSelect(Accept, hwnd, WM\_SOCKET, FD\_READ|FD\_WRITE|FD\_CLOSE);

               break;

            case FD\_READ:

               SocketInfo = GetSocketInformation(wParam);

               // Read data only if the receive buffer is empty

               if (SocketInfo->BytesRECV != 0)

               {

                  SocketInfo->RecvPosted = TRUE;

                  return 0;

               }

               else

               {

                  SocketInfo->DataBuf.buf = SocketInfo->Buffer;

                  SocketInfo->DataBuf.len = DATA\_BUFSIZE;

                  Flags = 0;

                  if (WSARecv(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &RecvBytes,

                     &Flags, NULL, NULL) == SOCKET\_ERROR)

                  {

                     if (WSAGetLastError() != WSAEWOULDBLOCK)

                     {

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

                        FreeSocketInformation(wParam);

                        return 0;

                     }

                  }

                  else // No error so update the byte count

                  {

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

                     SocketInfo->BytesRECV = RecvBytes;

                  }

               }

               // DO NOT BREAK HERE SINCE WE GOT A SUCCESSFUL RECV. Go ahead

               // and begin writing data to the client

            case FD\_WRITE:

               SocketInfo = GetSocketInformation(wParam);

               if (SocketInfo->BytesRECV > SocketInfo->BytesSEND)

               {

                  SocketInfo->DataBuf.buf = SocketInfo->Buffer + SocketInfo->BytesSEND;

                  SocketInfo->DataBuf.len = SocketInfo->BytesRECV - SocketInfo->BytesSEND;

                  if (WSASend(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &SendBytes, 0,

                     NULL, NULL) == SOCKET\_ERROR)

                  {

                     if (WSAGetLastError() != WSAEWOULDBLOCK)

                     {

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

                        FreeSocketInformation(wParam);

                        return 0;

                     }

                  }

                  else // No error so update the byte count

                  {

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

                     SocketInfo->BytesSEND += SendBytes;

                  }

               }

               if (SocketInfo->BytesSEND == SocketInfo->BytesRECV)

               {

                  SocketInfo->BytesSEND = 0;

                  SocketInfo->BytesRECV = 0;

                  // If a RECV occurred during our SENDs then we need to post an FD\_READ notification on the socket

                  if (SocketInfo->RecvPosted == TRUE)

                  {

                     SocketInfo->RecvPosted = FALSE;

                     PostMessage(hwnd, WM\_SOCKET, wParam, FD\_READ);

                  }

               }

               break;

            case FD\_CLOSE:

               printf("Closing socket %d\n", wParam);

               FreeSocketInformation(wParam);

               break;

         }

      }

      return 0;

   }

   return DefWindowProc(hwnd, uMsg, wParam, lParam);

}

void CreateSocketInformation(SOCKET s)

{

   LPSOCKET\_INFORMATION SI;

   if ((SI = (LPSOCKET\_INFORMATION) GlobalAlloc(GPTR, sizeof(SOCKET\_INFORMATION))) == NULL)

   {

      printf("GlobalAlloc() failed with error %d\n", GetLastError());

      return;

   }

   else

     printf("GlobalAlloc() for SOCKET\_INFORMATION is OK!\n");

   // Prepare SocketInfo structure for use

   SI->Socket = s;

   SI->RecvPosted = FALSE;

   SI->BytesSEND = 0;

   SI->BytesRECV = 0;

   SI->Next = SocketInfoList;

   SocketInfoList = SI;

}

LPSOCKET\_INFORMATION GetSocketInformation(SOCKET s)

{

   SOCKET\_INFORMATION \*SI = SocketInfoList;

   while(SI)

   {

      if (SI->Socket == s)

         return SI;

      SI = SI->Next;

   }

   return NULL;

}

void FreeSocketInformation(SOCKET s)

{

   SOCKET\_INFORMATION \*SI = SocketInfoList;

   SOCKET\_INFORMATION \*PrevSI = NULL;

   while(SI)

   {

      if (SI->Socket == s)

      {

         if (PrevSI)

            PrevSI->Next = SI->Next;

         else

            SocketInfoList = SI->Next;

         closesocket(SI->Socket);

         GlobalFree(SI);

         return;

      }

      PrevSI = SI;

      SI = SI->Next;

   }

}

HWND MakeWorkerWindow(void)

{

   WNDCLASS wndclass;

   CHAR \*ProviderClass = "AsyncSelect";

   HWND Window;

   wndclass.style = CS\_HREDRAW | CS\_VREDRAW;

   wndclass.lpfnWndProc = (WNDPROC)WindowProc;

   wndclass.cbClsExtra = 0;

   wndclass.cbWndExtra = 0;

   wndclass.hInstance = NULL;

   wndclass.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

   wndclass.hCursor = LoadCursor(NULL, IDC\_ARROW);

   wndclass.hbrBackground = (HBRUSH) GetStockObject(WHITE\_BRUSH);

   wndclass.lpszMenuName = NULL;

   wndclass.lpszClassName = (LPCWSTR)ProviderClass;

   if (RegisterClass(&wndclass) == 0)

   {

      printf("RegisterClass() failed with error %d\n", GetLastError());

      return NULL;

   }

   else

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

   // Create a window

   if ((Window = CreateWindow(

      (LPCWSTR)ProviderClass,

      L"",

      WS\_OVERLAPPEDWINDOW,

      CW\_USEDEFAULT,

      CW\_USEDEFAULT,

      CW\_USEDEFAULT,

      CW\_USEDEFAULT,

      NULL,

      NULL,

      NULL,

      NULL)) == NULL)

   {

      printf("CreateWindow() failed with error %d\n", GetLastError());

      return NULL;

   }

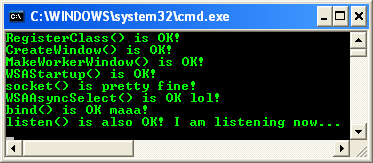
   else

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

   return Window;

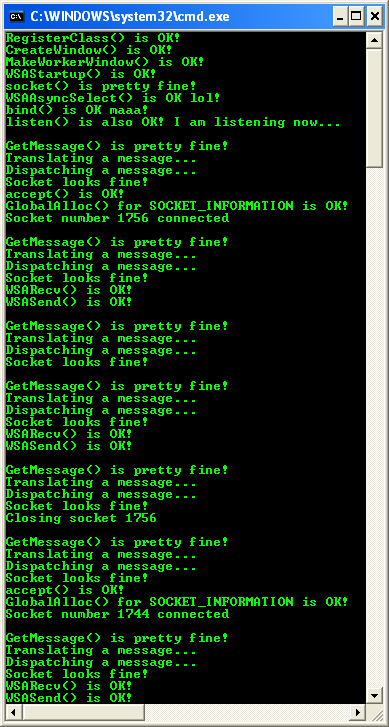
}

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



|  |
| --- |
| Running the Client-server Program   Firstly we run the server program.    The WSAAsyncSelect() IO Model: Program example shows a server which implementing the WSAAsyncSelect() model - running the program as server, waiting for connection    Then we run the previously created client.    The WSAAsyncSelect() IO Model: Program example shows a server which implementing the WSAAsyncSelect() model - running the client, sending some data |

The previous server program sample output is shown below.



# The WSAEventSelect Model

Winsock provides another useful asynchronous event notification I/O model that is similar to the WSAAsyncSelect model that allows an application to receive event-based notification of network events on one or more sockets. This model is similar to the WSAAsyncSelect model because your application receives and processes the same network events listed in Table 5-3 that the WSAAsyncSelect model uses. The major difference with this model is that network events are notified via an event object handle instead of a window procedure.

# Event Notification

The event notification model requires your application to create an event object for each socket used by calling the WSACreateEvent() function, which is defined as:

 WSAEVENT WSACreateEvent(void);

The WSACreateEvent() function simply returns a manual reset event object handle. Once you have an event object handle, you have to associate it with a socket and register the network event types of interest, as shown in Table 5-3. This is accomplished by calling the WSAEventSelect() function, which is defined as:

int WSAEventSelect(

    SOCKET s,

    WSAEVENT hEventObject,

    long lNetworkEvents

);

The s parameter represents the socket of interest. The hEventObject parameter represents the event object, obtained with WSACreateEvent(), to associate with the socket. The last parameter, lNetworkEvents, represents a bitmask that specifies a combination of network event types (listed in Table 5-3) that the application is interested in. For a detailed discussion of these event types, see the WSAAsyncSelect I/O model discussed previously.

The event created for WSAEventSelect() has two operating states and two operating modes. The operating states are known as signaled and non-signaled. The operating modes are known as manual reset and auto reset. WSACreateEvent() initially creates event handles in a non-signaled operating state with a manual reset operating mode. As network events trigger an event object associated with a socket, the operating state changes from non-signaled to signaled. Because the event object is created in a manual reset mode, your application is responsible for changing the operating state from signaled to non-signaled after processing an I/O request. This can be accomplished by calling the WSAResetEvent() function, which is defined as:

BOOL WSAResetEvent(WSAEVENT hEvent);

The function takes an event handle as its only parameter and returns TRUE or FALSE based on the success or failure of the call. When an application is finished with an event object, it should call the WSACloseEvent() function to free the system resources used by an event handle. The WSACloseEvent() function is defined as:

BOOL WSACloseEvent(WSAEVENT hEvent);

This function also takes an event handle as its only parameter and returns TRUE if successful or FALSE if the call fails.

Once a socket is associated with an event object handle, the application can begin processing I/O by waiting for network events to trigger the operating state of the event object handle. The WSAWaitForMultipleEvents() function is designed to wait on one or more event object handles and returns either when one or all of the specified handles are in the signaled state or when a specified timeout interval expires. WSAWaitForMultipleEvents() is defined as:

DWORD WSAWaitForMultipleEvents(

    DWORD cEvents,

    const WSAEVENT FAR \* lphEvents,

    BOOL fWaitAll,

    DWORD dwTimeout,

    BOOL fAlertable

);

The cEvents and lphEvents parameters define an array of WSAEVENT objects in which cEvents is the number of event objects in the array and lphEvents is a pointer to the array. WSAWaitForMultipleEvents() can support only a maximum of WSA\_MAXIMUM\_WAIT\_EVENTS objects, which is defined as 64. Therefore, this I/O model is capable of supporting only a maximum of 64 sockets at a time for each thread that makes the WSAWaitForMultipleEvents() call. If you need to have this model manage more than 64 sockets, you should create additional worker threads to wait on more event objects. The fWaitAll parameter specifies how WSAWaitForMultipleEvents() waits for objects in the event array. If TRUE, the function returns when all event objects in the lphEvents array are signaled. If FALSE, the function returns when any one of the event objects is signaled. In the latter case, the return value indicates which event object caused the function to return. Typically, applications set this parameter to FALSE and service one socket event at a time. The dwTimeout parameter specifies how long (in milliseconds) WSAWaitForMultipleEvents() will wait for a network event to occur. The function returns if the interval expires, even if conditions specified by the fWaitAll parameter are not satisfied. If the timeout value is 0, the function tests the state of the specified event objects and returns immediately, which effectively allows an application to poll on the event objects. If no events are ready for processing, WSAWaitForMultipleEvents() returns WSA\_WAIT\_TIMEOUT. If dwsTimeout is set to WSA\_INFINITE, the function returns only when a network event signals an event object. The final parameter, fAlertable, can be ignored when you're using the WSAEventSelect() model and should be set to FALSE. It is intended for use in processing completion routines in the overlapped I/O model, which will be described later in this chapter.

Note that by servicing signaled events one at a time (by setting the fWaitAll parameter to FALSE), it is possible to starve sockets toward the end of the event array. Consider the following code:

 WSAEVENT              HandleArray[WSA\_MAXIMUM\_WAIT\_EVENTS];

int        WaitCount=0, ret, index;

// Assign event handles into HandleArray

while (1)

{

            ret = WSAWaitForMultipleEvents(

                        WaitCount,

                        HandleArray,

                        FALSE,

                        WSA\_INFINITE,

                        TRUE);

            if ((ret != WSA\_WAIT\_FAILED) && (ret != WSA\_WAIT\_TIMEOUT))

            {

                        index = ret - WSA\_WAIT\_OBJECT\_0;

                        // Service event signaled on HandleArray[index]WSAResetEvent(HandleArray[index]);

            }

}

If the socket connection associated in index 0 of the event array is continually receiving data such that after the event is reset additional data arrives causing the event to be signaled again, the rest of the events in the array are starved. This is clearly undesirable. Once an event within the loop is signaled and handled, all events in the array should be checked to see if they are signaled as well. This can be accomplished by using WSAWaitForMultipleEvents() with each individual event handle after the first signaled event and specifying a dwTimeOut of zero.

When WSAWaitForMultipleEvents() receives network event notification of an event object, it returns a value indicating the event object that caused the function to return. As a result, your application can determine which network event type is available on a particular socket by referencing the signaled event in the event array and matching it with the socket associated with the event. When you reference the events in the event array, you should reference them using the return value of WSAWaitForMultipleEvents() minus the predefined value WSA\_WAIT\_EVENT\_0. For example:

 Index = WSAWaitForMultipleEvents(...);

MyEvent = EventArray[Index - WSA\_WAIT\_EVENT\_0];

 Once you have the socket that caused the network event, you can determine which network events are available by calling the WSAEnumNetworkEvents() function, which is defined as:

int WSAEnumNetworkEvents(

    SOCKET s,

    WSAEVENT hEventObject,

    LPWSANETWORKEVENTS lpNetworkEvents

);

The s parameter represents the socket that caused the network event, and the hEventObject parameter is an optional parameter representing an event handle identifying an associated event object to be reset. Because our event object is in a signaled state, we can pass it in and it will be set to a non-signaled state. The hEventObject parameter is optional in case you wish to reset the event manually via the WSAResetEvent() function. The final parameter, lpNetworkEvents, takes a pointer to a WSANETWORKEVENTS structure, which is used to retrieve network event types that occurred on the socket and any associated error codes. The WSANETWORKEVENTS structure is defined as:

 typedef struct \_WSANETWORKEVENTS

{

            long lNetworkEvents;

    int  iErrorCode[FD\_MAX\_EVENTS];

} WSANETWORKEVENTS, FAR \* LPWSANETWORKEVENTS;

 The lNetworkEvents parameter is a value that indicates all the network event types (see Table 5-3) that have occurred on the socket.

More than one network event type can occur whenever an event is signaled. For example, a busy server application might receive FD\_READ and FD\_WRITE notification at the same time.

The iErrorCode parameter is an array of error codes associated with the events in lNetworkEvents. For each network event type, there is a special event index similar to the event type names, except for an additional “\_BIT” string appended to the event name. For example, for the FD\_READ event type, the index identifier for the iErrorCode array is named FD\_READ\_BIT. The following code fragment demonstrates this for an FD\_READ event:

// Process FD\_READ notification

if (NetworkEvents.lNetworkEvents & FD\_READ)

{

    if (NetworkEvents.iErrorCode[FD\_READ\_BIT] != 0)

    {

       printf("FD\_READ failed with error %d\n", NetworkEvents.iErrorCode[FD\_READ\_BIT]);

    }

}

After you process the events in the WSANETWORKEVENTS structure, your application should continue waiting for more network events on all of the available sockets. The following example demonstrates how to develop a server and manage event objects when using the WSAEventSelect() I/O model. The code highlights the steps needed to develop a basic server application capable of managing one or more sockets at a time.

 SOCKET SocketArray [WSA\_MAXIMUM\_WAIT\_EVENTS];

WSAEVENT EventArray [WSA\_MAXIMUM\_WAIT\_EVENTS], NewEvent;

SOCKADDR\_IN InternetAddr;

SOCKET Accept, Listen;

DWORD EventTotal = 0;

DWORD Index, i;

// Set up a TCP socket for listening on port 5150

Listen = socket (PF\_INET, SOCK\_STREAM, 0);

InternetAddr.sin\_family = AF\_INET;

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

InternetAddr.sin\_port = htons(5150);

bind(Listen, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr));

NewEvent = WSACreateEvent();

WSAEventSelect(Listen, NewEvent, FD\_ACCEPT │ FD\_CLOSE);

listen(Listen, 5);

SocketArray[EventTotal] = Listen;

EventArray[EventTotal] = NewEvent;

EventTotal++;

while(TRUE)

{

    // Wait for network events on all sockets

    Index = WSAWaitForMultipleEvents(EventTotal, EventArray, FALSE, WSA\_INFINITE, FALSE);

    Index = Index - WSA\_WAIT\_EVENT\_0;

    // Iterate through all events to see if more than one is signaled

    for(i=Index; i < EventTotal ;i++)

    {

     Index = WSAWaitForMultipleEvents(1, &EventArray[i], TRUE, 1000, FALSE);

     if ((Index == WSA\_WAIT\_FAILED) ││ (Index == WSA\_WAIT\_TIMEOUT))

         continue;

     else

     {

         Index = i;

         WSAEnumNetworkEvents(SocketArray[Index], EventArray[Index], &NetworkEvents);

         // Check for FD\_ACCEPT messages

         if (NetworkEvents.lNetworkEvents & FD\_ACCEPT)

         {

             if (NetworkEvents.iErrorCode[FD\_ACCEPT\_BIT] != 0)

             {

                 printf("FD\_ACCEPT failed with error %d\n", NetworkEvents.iErrorCode[FD\_ACCEPT\_BIT]);

                 break;

             }

             // Accept a new connection, and add it to the socket and event lists

             Accept = accept(SocketArray[Index], NULL, NULL);

             // We cannot process more than

             // WSA\_MAXIMUM\_WAIT\_EVENTS sockets, so close the accepted socket

             if (EventTotal > WSA\_MAXIMUM\_WAIT\_EVENTS)

             {

                 printf("Too many connections");

                 closesocket(Accept);

                 break;

             }

             NewEvent = WSACreateEvent();

             WSAEventSelect(Accept, NewEvent, FD\_READ │ FD\_WRITE │ FD\_CLOSE);

             EventArray[EventTotal] = NewEvent;

             SocketArray[EventTotal] = Accept;

             EventTotal++;

             printf("Socket %d connected\n", Accept);

         }

         // Process FD\_READ notification

         if (NetworkEvents.lNetworkEvents & FD\_READ)

         {

             if (NetworkEvents.iErrorCode[FD\_READ\_BIT] != 0)

             {

                 printf("FD\_READ failed with error %d\n",NetworkEvents.iErrorCode[FD\_READ\_BIT]);

                 break;

             }

             // Read data from the socket

             recv(SocketArray[Index - WSA\_WAIT\_EVENT\_0], buffer, sizeof(buffer), 0);

         }

         // Process FD\_WRITE notification

         if (NetworkEvents.lNetworkEvents & FD\_WRITE)

         {

             if (NetworkEvents.iErrorCode[FD\_WRITE\_BIT] != 0)

             {

                 printf("FD\_WRITE failed with error %d\n",  NetworkEvents.iErrorCode[FD\_WRITE\_BIT]);

                 break;

             }

             send(SocketArray[Index - WSA\_WAIT\_EVENT\_0], buffer, sizeof(buffer), 0);

            }

            if (NetworkEvents.lNetworkEvents & FD\_CLOSE)

            {

                if (NetworkEvents.iErrorCode[FD\_CLOSE\_BIT] != 0)

                {

                    printf("FD\_CLOSE failed with error %d\n",  NetworkEvents.iErrorCode[FD\_CLOSE\_BIT]);

                    break;

                }

                closesocket(SocketArray[Index]);

                // Remove socket and associated event from

                // the Socket and Event arrays and decrement EventTotal

                CompressArrays(EventArray, SocketArray, &EventTotal);

            }

       }

   }

}

The WSAEventSelect model offers several advantages. It is conceptually simple and it does not require a windowed environment. The only drawback is its limitation of waiting on only 64 events at a time, which necessitates managing a thread pool when dealing with many sockets. Also, because many threads are required to handle a large number of socket connections, this model does not scale as well as the overlapped models discussed next.

|  |
| --- |
| The WSAEventSelect server/receiver model Program Example   The following program example tries to demonstrate the WSAEventSelect server/receiver model.  The WSAEventSelect Model IO Model: Program example shows a server which implementing the WSAEventSelect model - creating a new empty Win32 console mode application and add the project/solution name.    Add the following source code. |

//    This sample illustrates how to develop a simple echo server Winsock

//    application using the WSAEventSelect() I/O model. This sample is

//    implemented as a console-style application and simply prints

//    messages when connections are established and removed from the server.

//    The application listens for TCP connections on port 5150 and accepts them

//    as they arrive. When this application receives data from a client, it

//    simply echoes (this is why we call it an echo server) the data back in

//    it's original form until the client closes the connection.

//

//    Note: There are no command line options for this sample.

// Link to ws2\_32.lib

#include <winsock2.h>

#include <windows.h>

#include <stdio.h>

#define PORT 5150

#define DATA\_BUFSIZE 8192

typedef struct \_SOCKET\_INFORMATION {

   CHAR Buffer[DATA\_BUFSIZE];

   WSABUF DataBuf;

   SOCKET Socket;

   DWORD BytesSEND;

   DWORD BytesRECV;

} SOCKET\_INFORMATION, \* LPSOCKET\_INFORMATION;

BOOL CreateSocketInformation(SOCKET s);

void FreeSocketInformation(DWORD Event);

DWORD EventTotal = 0;

WSAEVENT EventArray[WSA\_MAXIMUM\_WAIT\_EVENTS];

LPSOCKET\_INFORMATION SocketArray[WSA\_MAXIMUM\_WAIT\_EVENTS];

int main(int argc, char \*\*argv)

{

            SOCKET Listen;

            SOCKET Accept;

            SOCKADDR\_IN InternetAddr;

            LPSOCKET\_INFORMATION SocketInfo;

            DWORD Event;

            WSANETWORKEVENTS NetworkEvents;

            WSADATA wsaData;

            DWORD Flags;

            DWORD RecvBytes;

            DWORD SendBytes;

            if (WSAStartup(0x0202, &wsaData) != 0)

            {

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

                        return 1;

            }

            else

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

            if ((Listen = socket(AF\_INET, SOCK\_STREAM, 0)) == INVALID\_SOCKET)

            {

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

                        return 1;

            }

            else

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

            if(CreateSocketInformation(Listen) == FALSE)

                        printf("CreateSocketInformation() failed!\n");

            else

                        printf("CreateSocketInformation() is OK lol!\n");

            if (WSAEventSelect(Listen, EventArray[EventTotal - 1], FD\_ACCEPT|FD\_CLOSE) == SOCKET\_ERROR)

            {

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

                        return 1;

            }

            else

                        printf("WSAEventSelect() is pretty fine!\n");

            InternetAddr.sin\_family = AF\_INET;

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

            InternetAddr.sin\_port = htons(PORT);

            if (bind(Listen, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr)) == SOCKET\_ERROR)

            {

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

                        return 1;

            }

            else

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

            if (listen(Listen, 5))

            {

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

                        return 1;

            }

            else

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

            while(TRUE)

            {

                        // Wait for one of the sockets to receive I/O notification and

                        if ((Event = WSAWaitForMultipleEvents(EventTotal, EventArray, FALSE,WSA\_INFINITE, FALSE)) == WSA\_WAIT\_FAILED)

                        {

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

                                    return 1;

                        }

                        else

                                    printf("WSAWaitForMultipleEvents() is pretty damn OK!\n");

                        if (WSAEnumNetworkEvents(SocketArray[Event - WSA\_WAIT\_EVENT\_0]->Socket,

                                    EventArray[Event - WSA\_WAIT\_EVENT\_0], &NetworkEvents) == SOCKET\_ERROR)

                        {

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

                                    return 1;

                        }

                        else

                                    printf("WSAEnumNetworkEvents() should be fine!\n");

                        if (NetworkEvents.lNetworkEvents & FD\_ACCEPT)

                        {

                                    if (NetworkEvents.iErrorCode[FD\_ACCEPT\_BIT] != 0)

                                    {

                                                printf("FD\_ACCEPT failed with error %d\n", NetworkEvents.iErrorCode[FD\_ACCEPT\_BIT]);

                                                break;

                                    }

                                    if ((Accept = accept(SocketArray[Event - WSA\_WAIT\_EVENT\_0]->Socket, NULL, NULL)) == INVALID\_SOCKET)

                                    {

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

                                                break;

                                    }

                                    else

                                                printf("accept() should be OK!\n");

                                    if (EventTotal > WSA\_MAXIMUM\_WAIT\_EVENTS)

                                    {

                                                printf("Too many connections - closing socket...\n");

                                                closesocket(Accept);

                                                break;

                                    }

                                    CreateSocketInformation(Accept);

                                    if (WSAEventSelect(Accept, EventArray[EventTotal - 1], FD\_READ|FD\_WRITE|FD\_CLOSE) == SOCKET\_ERROR)

                                    {

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

                                                return 1;

                                    }

                                    else

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

                                    printf("Socket %d got connected...\n", Accept);

                        }

                        // Try to read and write data to and from the data buffer if read and write events occur

                        if (NetworkEvents.lNetworkEvents & FD\_READ || NetworkEvents.lNetworkEvents & FD\_WRITE)

                        {

                                    if (NetworkEvents.lNetworkEvents & FD\_READ && NetworkEvents.iErrorCode[FD\_READ\_BIT] != 0)

                                    {

                                                printf("FD\_READ failed with error %d\n", NetworkEvents.iErrorCode[FD\_READ\_BIT]);

                                                break;

                                    }

                                    else

                                                printf("FD\_READ is OK!\n");

                                    if (NetworkEvents.lNetworkEvents & FD\_WRITE && NetworkEvents.iErrorCode[FD\_WRITE\_BIT] != 0)

                                    {

                                                printf("FD\_WRITE failed with error %d\n", NetworkEvents.iErrorCode[FD\_WRITE\_BIT]);

                                                break;

                                    }

                                    else

                                                printf("FD\_WRITE is OK!\n");

                                    SocketInfo = SocketArray[Event - WSA\_WAIT\_EVENT\_0];

                                    // Read data only if the receive buffer is empty

                                    if (SocketInfo->BytesRECV == 0)

                                    {

                                                SocketInfo->DataBuf.buf = SocketInfo->Buffer;

                                                SocketInfo->DataBuf.len = DATA\_BUFSIZE;

                                                Flags = 0;

                                                if (WSARecv(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &RecvBytes, &Flags, NULL, NULL) == SOCKET\_ERROR)

                                                {

                                                            if (WSAGetLastError() != WSAEWOULDBLOCK)

                                                            {

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

                                                                        FreeSocketInformation(Event - WSA\_WAIT\_EVENT\_0);

                                                                        return 1;

                                                            }

                                                }

                                                else

                                                {

                                                            printf("WSARecv() is working!\n");

                                                            SocketInfo->BytesRECV = RecvBytes;

                                                }

                                    }

                                    // Write buffer data if it is available

                                    if (SocketInfo->BytesRECV > SocketInfo->BytesSEND)

                                    {

                                                SocketInfo->DataBuf.buf = SocketInfo->Buffer + SocketInfo->BytesSEND;

                                                SocketInfo->DataBuf.len = SocketInfo->BytesRECV - SocketInfo->BytesSEND;

                                                if (WSASend(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &SendBytes, 0, NULL, NULL) == SOCKET\_ERROR)

                                                {

                                                            if (WSAGetLastError() != WSAEWOULDBLOCK)

                                                            {

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

                                                                        FreeSocketInformation(Event - WSA\_WAIT\_EVENT\_0);

                                                                        return 1;

                                                            }

                                                            // A WSAEWOULDBLOCK error has occurred. An FD\_WRITE event will be posted

                                                            // when more buffer space becomes available

                                                }

                                                else

                                                {

                                                            printf("WSASend() is fine! Thank you...\n");

                                                            SocketInfo->BytesSEND += SendBytes;

                                                            if (SocketInfo->BytesSEND == SocketInfo->BytesRECV)

                                                            {

                                                                        SocketInfo->BytesSEND = 0;

                                                                        SocketInfo->BytesRECV = 0;

                                                            }

                                                }

                                    }

                        }

                        if (NetworkEvents.lNetworkEvents & FD\_CLOSE)

                        {

                                    if (NetworkEvents.iErrorCode[FD\_CLOSE\_BIT] != 0)

                                    {

                                                printf("FD\_CLOSE failed with error %d\n", NetworkEvents.iErrorCode[FD\_CLOSE\_BIT]);

                                                break;

                                    }

                                    else

                                                printf("FD\_CLOSE is OK!\n");

                                    printf("Closing socket information %d\n", SocketArray[Event - WSA\_WAIT\_EVENT\_0]->Socket);

                                    FreeSocketInformation(Event - WSA\_WAIT\_EVENT\_0);

                        }

  }

  return 0;

}

BOOL CreateSocketInformation(SOCKET s)

{

            LPSOCKET\_INFORMATION SI;

            if ((EventArray[EventTotal] = WSACreateEvent()) == WSA\_INVALID\_EVENT)

            {

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

                        return FALSE;

            }

            else

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

            if ((SI = (LPSOCKET\_INFORMATION) GlobalAlloc(GPTR, sizeof(SOCKET\_INFORMATION))) == NULL)

            {

                        printf("GlobalAlloc() failed with error %d\n", GetLastError());

                        return FALSE;

            }

            else

                        printf("GlobalAlloc() for LPSOCKET\_INFORMATION is OK!\n");

            // Prepare SocketInfo structure for use

            SI->Socket = s;

            SI->BytesSEND = 0;

            SI->BytesRECV = 0;

            SocketArray[EventTotal] = SI;

            EventTotal++;

            return(TRUE);

}

void FreeSocketInformation(DWORD Event)

{

            LPSOCKET\_INFORMATION SI = SocketArray[Event];

            DWORD i;

            closesocket(SI->Socket);

            GlobalFree(SI);

            if(WSACloseEvent(EventArray[Event]) == TRUE)

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

            else

                        printf("WSACloseEvent() failed miserably!\n\n");

            // Squash the socket and event arrays

            for (i = Event; i < EventTotal; i++)

            {

                        EventArray[i] = EventArray[i + 1];

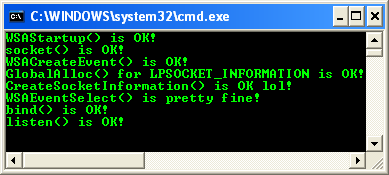
                        SocketArray[i] = SocketArray[i + 1];

            }

            EventTotal--;

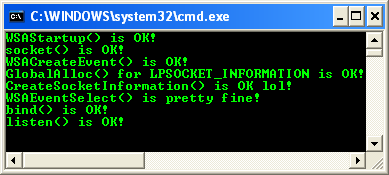
}

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

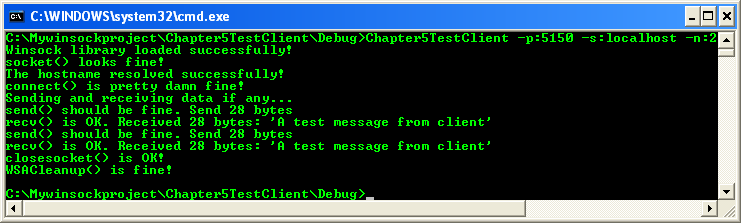


 Running the Client and Server Programs.

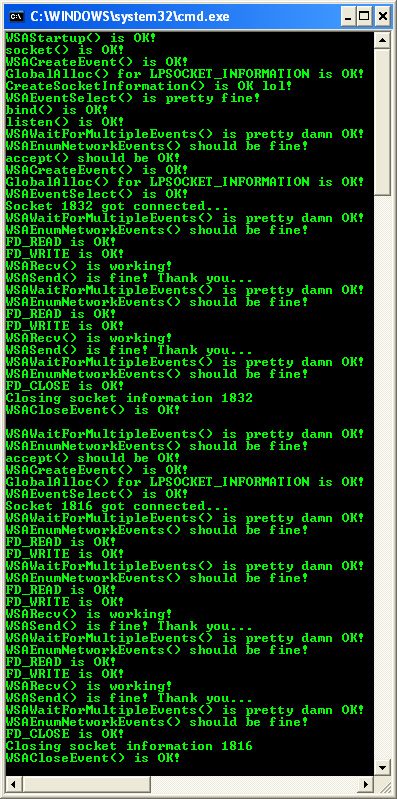
 Firstly we run the server program.



 Then we run the client program.



The previous server sample output is shown below.



# Next : [The Overlapped Model](http://www.winsocketdotnetworkprogramming.com/winsock2programming/winsock2advancediomethod5e.html)