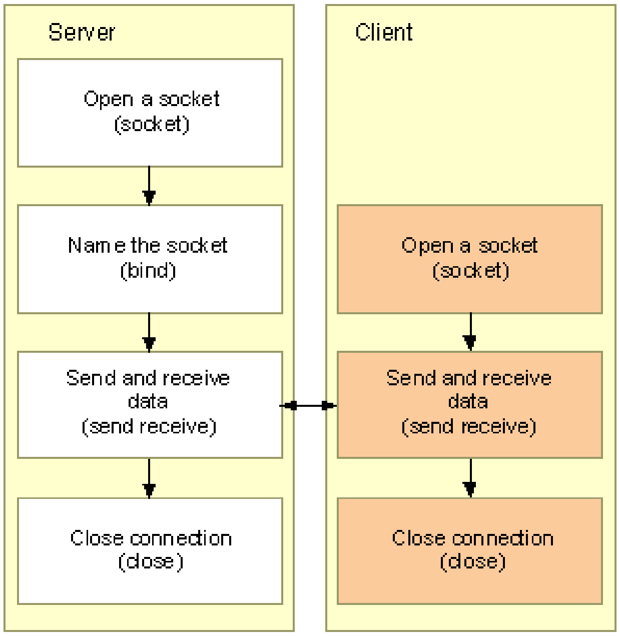
< TCP Client-server Examples | Winsock2 Main | UDP Example & Other Winsock2 APIs >

# An Intro to Windows Socket Programming with C Part 6

What do we have in this chapter 1 part 6?

## Connectionless Communication

Connectionless communication behaves differently than connection-oriented communication, so the method for sending and receiving data is substantially different. First we'll discuss the receiver (or server, if you prefer) because the connectionless receiver requires little change when compared with the connection-oriented servers. After that we'll look at the sender.
In IP, connectionless communication is accomplished through UDP/IP. UDP doesn't guarantee reliable data transmission and is capable of sending data to multiple destinations and receiving it from multiple sources. For example, if a client sends data to a server, the data is transmitted immediately regardless of whether the server is ready to receive it. If the server receives data from the client, it doesn't acknowledge the receipt. Data is transmitted using datagrams, which are discrete message packets. The following Figure shows a simplified UDP communication flow between server and client.



## Receiver

The steps in the process of receiving data on a connectionless socket are simple. First, create the socket with either socket() or WSASocket(). Next, bind the socket to the interface on which you wish to receive data. This is done with the bind() function (exactly like the session-oriented example). The difference with connectionless sockets is that you do not call listen() or accept(). Instead, you simply wait to receive the incoming data. Because there is no connection, the receiving socket can receive datagrams originating from any machine on the network. The simplest of the receive functions is recvfrom(), which is defined as:

```
int recvfrom(
    SOCKET s,
    char FAR* buf,
    int len,
    int flags,
    struct sockaddr FAR* from,
    int FAR* fromlen
);
```

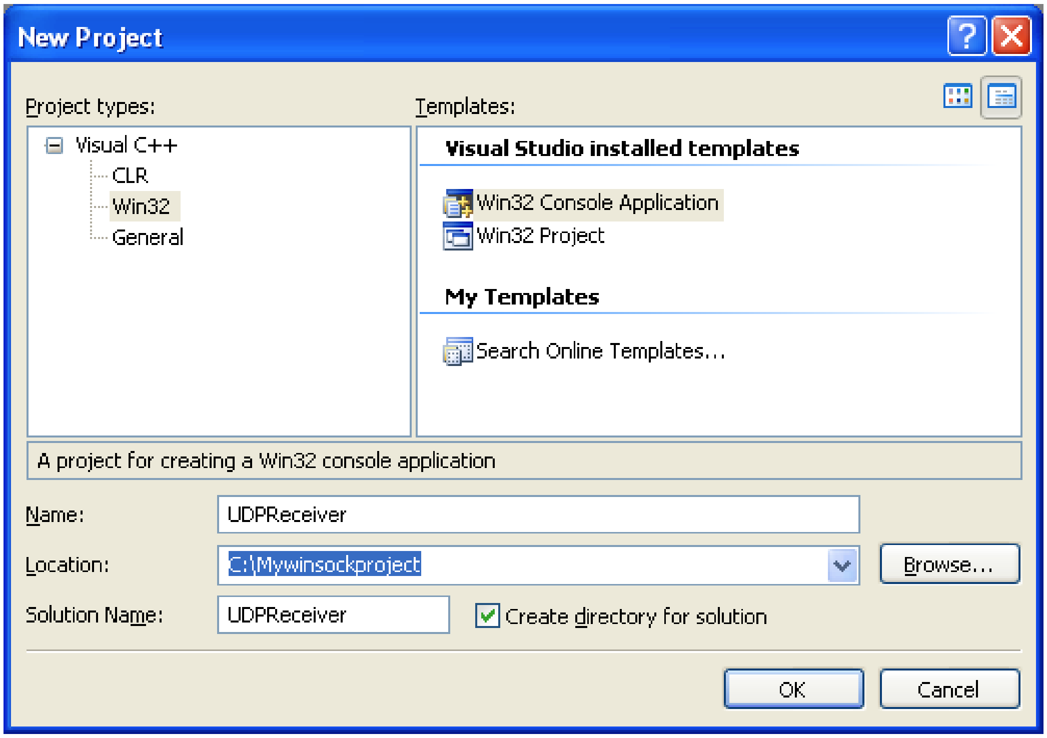
The first four parameters are the same as recv(), including the possible values for flags: MSG_OOB and MSG_PEEK. The same warnings for using the MSG_PEEK flag also apply to connectionless sockets. The from parameter is a SOCKADDR structure for the given protocol of the listening socket, with fromlen pointing to the size of the address structure. When the API call returns with data, the SOCKADDR structure is filled with the address of the workstation that sent the data. The Winsock 2 version of the recvfrom() function is WSARecvFrom(). The prototype for this function is:

```
int WSARecvFrom(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    struct sockaddr FAR * lpFrom,
    LPINT lpFromlen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

The difference is the use of WSABUF structures for receiving the data. You can supply one or more WSABUF buffers to WSARecvFrom() with dwBufferCount indicating this. By supplying multiple buffers, scatter-gather I/O is possible. The total number of bytes read is returned in lpNumberOfBytesRecvd. When you call WSARecvFrom(), the lpFlags parameter can be 0 for no options, MSG_OOB, MSG_PEEK, or MSG_PARTIAL. These flags can be bitwise OR together. If MSG_PARTIAL is specified when the function is called, the provider knows to return data even if only a partial message has been received. Upon return, the flag MSG_PARTIAL is set if only a partial message was received. Upon return, WSARecvFrom() will store the address of the sending machine in the lpFrom parameter (a pointer to a SOCKADDR structure). Again, lpFromLen points to the size of the SOCKADDR structure, except that in this function it is a pointer to a DWORD. The last two parameters, lpOverlapped and lpCompletionRoutine, are used for overlapped I/O. Another method of receiving (and sending) data on a connectionless socket is to establish a connection. This might seem strange, but it's not quite what it sounds like. Once a connectionless socket is created, you can call connect() or WSAConnect() with the SOCKADDR parameter set to the address of the remote machine to communicate with. No actual connection is made, however. The socket address passed into a connect() function is associated with the socket so recv() and WSARecv() can be used instead of recvfrom() or WSARecvFrom() because the data's origin is known. The capability to connect a datagram socket is handy if you intend to communicate with only one endpoint at a time in your application. The following code sample demonstrates how to construct a simple UDP receiver application.

1. While in the Visual C++ IDE, click File menu > Project sub menu to create a new project.

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



New Project

Project types:

- Visual C++
  - CLR
  - Win32
  - General

Templates:

Visual Studio installed templates

- Win32 Console Application
- Win32 Project

My Templates

- Search Online Templates...

A project for creating a Win32 console application

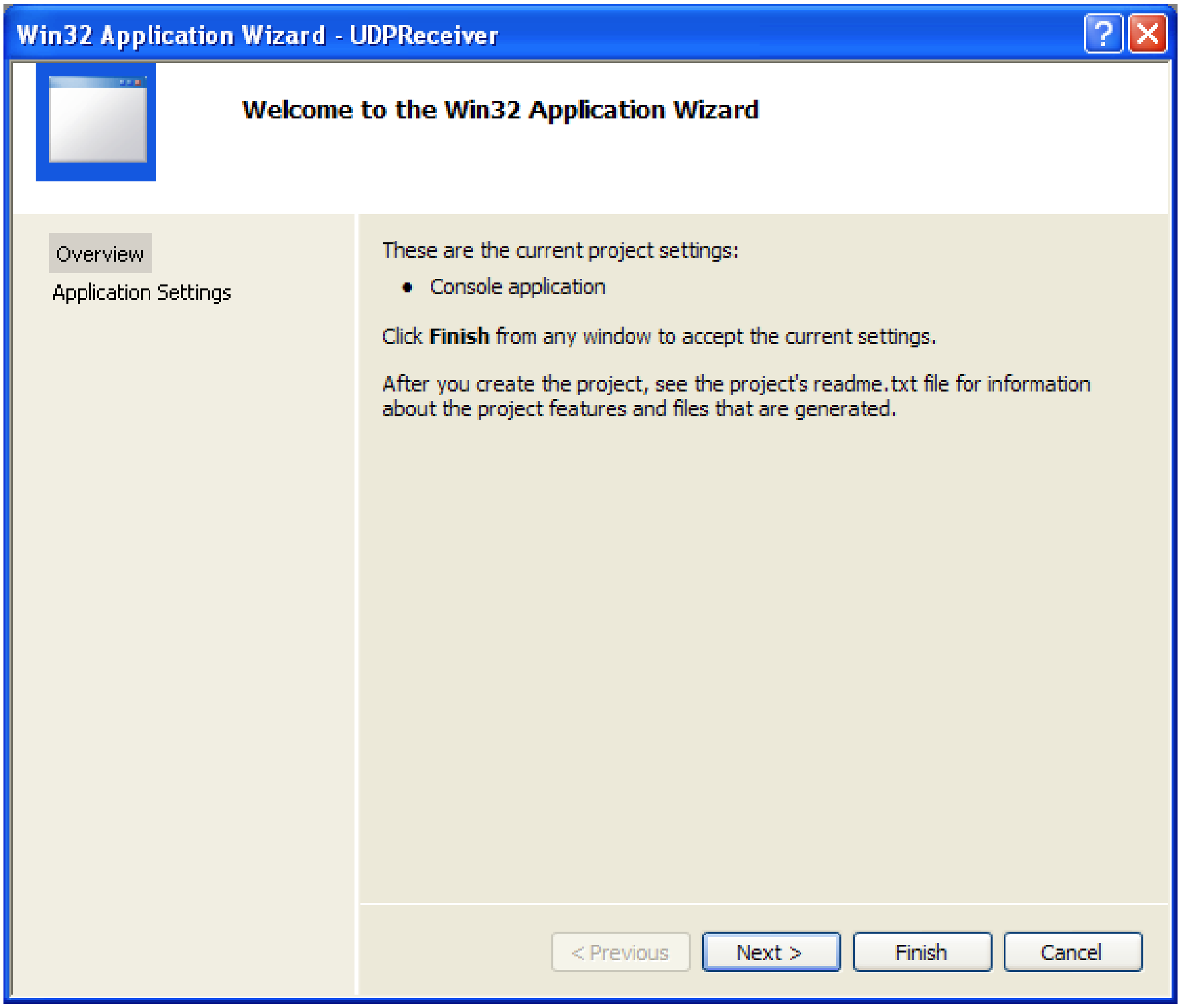
Name: UDPReceiver

Location: C:\Mywinsockproject

Solution Name: UDPReceiver

☑ Create directory for solution

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



Win32 Application Wizard - UDPReceiver

Welcome to the Win32 Application Wizard

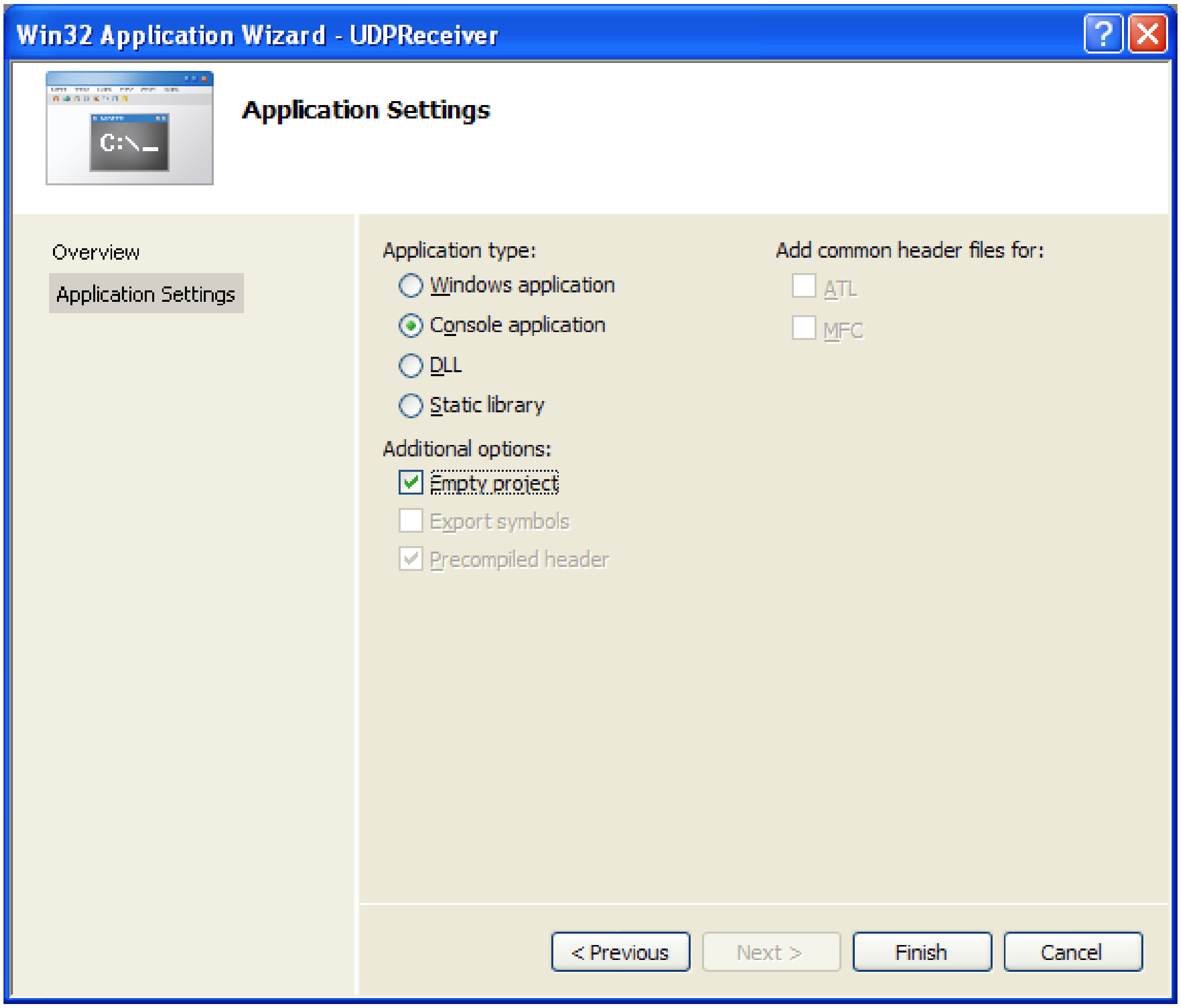
These are the current project settings:

- Console application

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's readme.txt file for information about the project features and files that are generated.

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



Win32 Application Wizard - UDPReceiver

Application Settings

Application type:

○ Windows application

● Console application

○ DLL

○ Static library

Additional options:

☑ Empty project

□ Export symbols

☑ Precompiled header

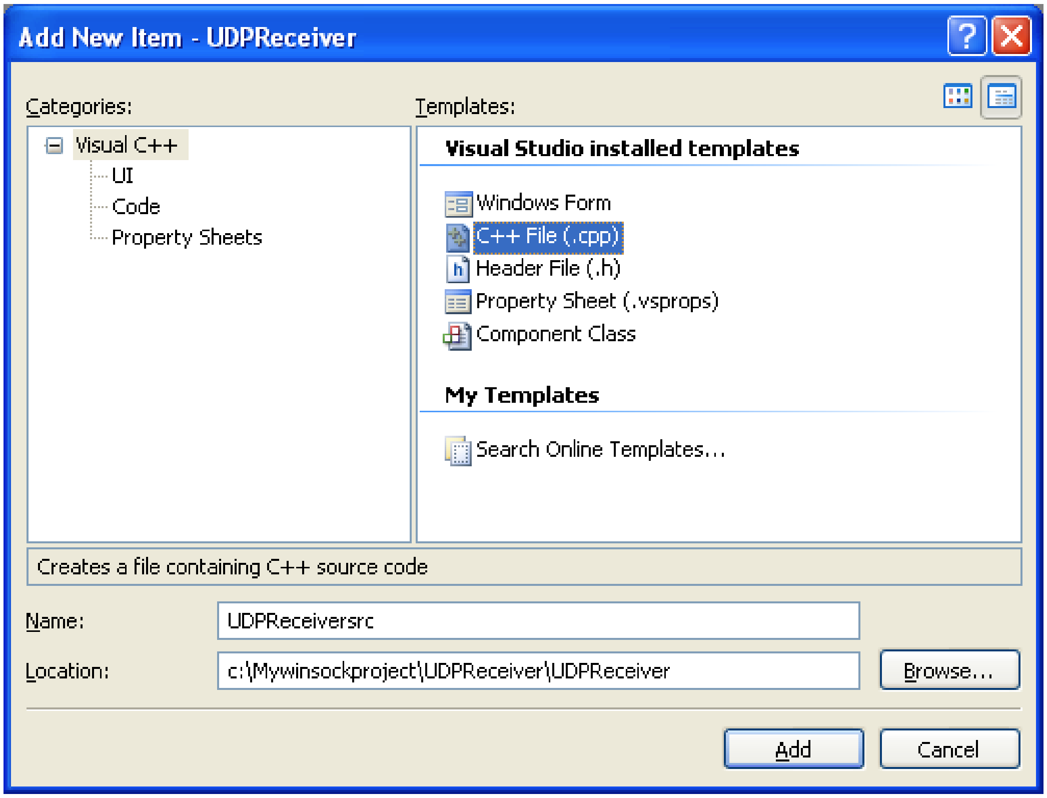
Add common header files for:

□ ATL

□ MFC

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

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

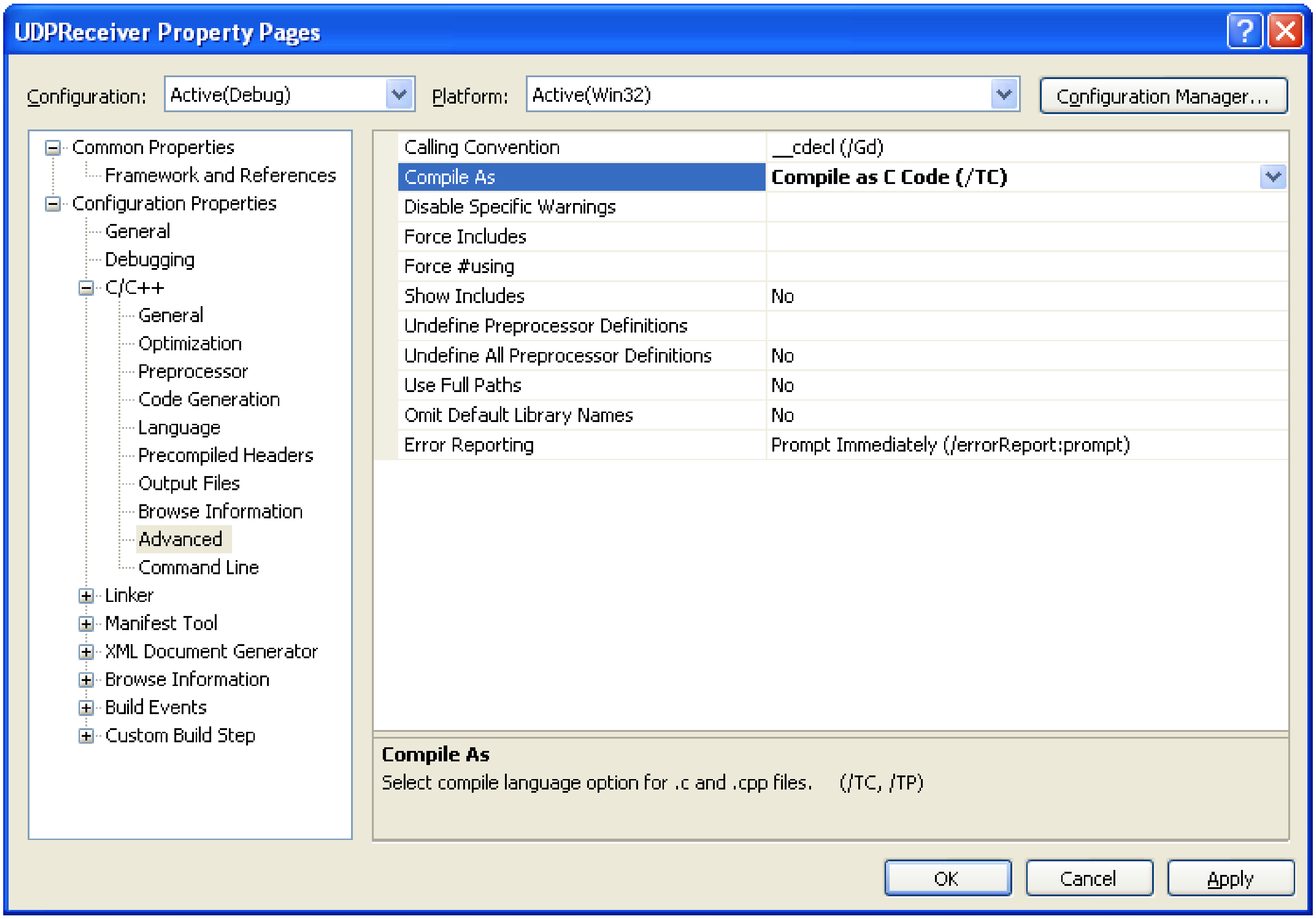


Creates a file containing C++ source code

Name: UDPReceiversrc

Location: c:\Mywinsockproject\UDPReceiver\UDPReceiver

7. Before we can build this Winsock C Win32 console application project, we need to set the project to be compiled as C code and link to ws2_32.lib, the Winsock2 library. Invoke the project property page.
8. Expand the Configuration folder > Expand the C/C++ sub folder. Select the Advanced link and for the Compile As option, select Compile as C Code (/TC).



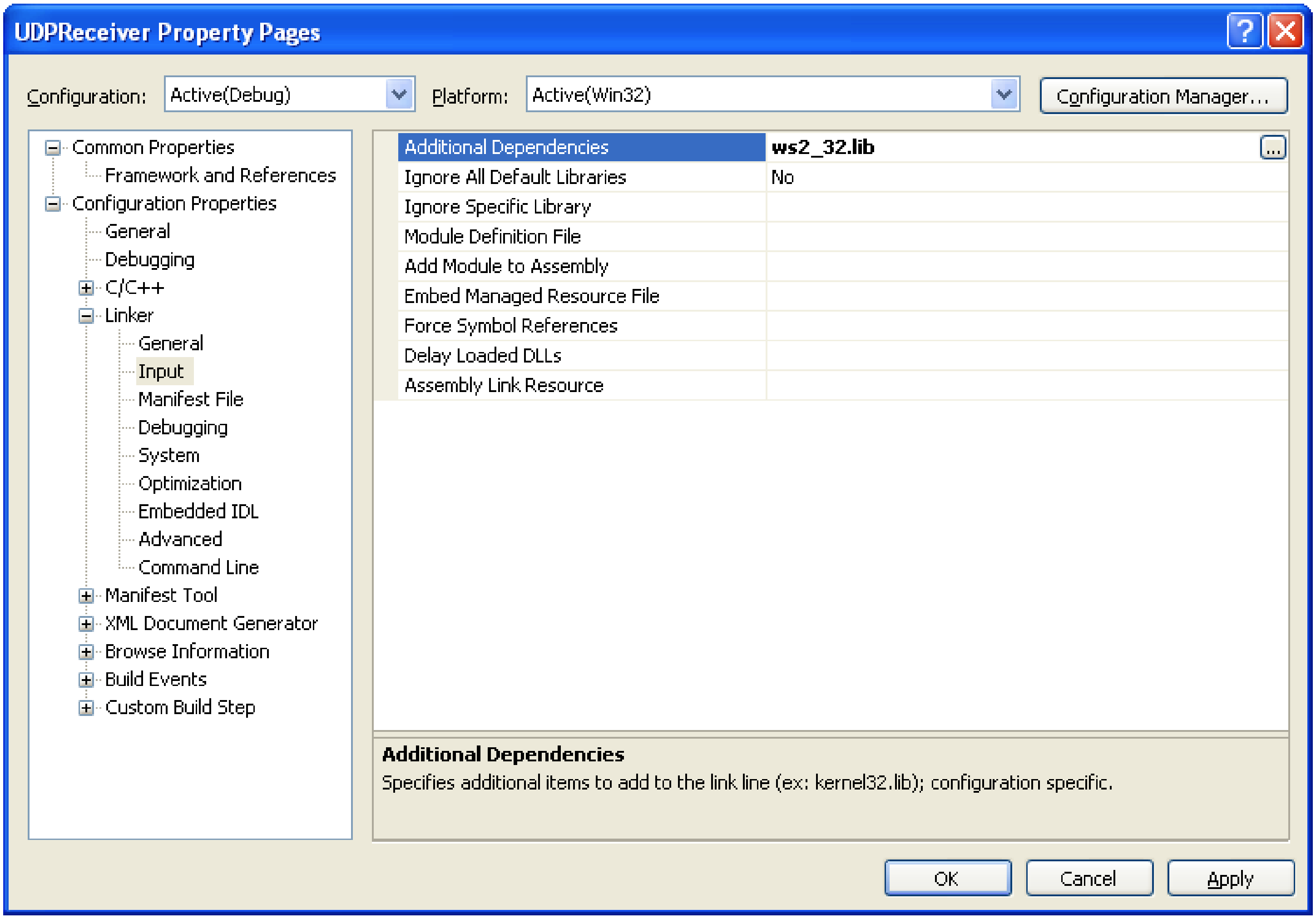
Configuration: Active(Debug)

Platform: Active(Win32)

| | |
|---|---|
| Calling Convention | __cdecl (/Gd) |
| Compile As | **Compile as C Code (/TC)** |
| Disable Specific Warnings | |
| Force Includes | |
| Force #using | |
| Show Includes | No |
| Undefine Preprocessor Definitions | |
| Undefine All Preprocessor Definitions | No |
| Use Full Paths | No |
| Omit Default Library Names | No |
| Error Reporting | Prompt Immediately (/errorReport:prompt) |

**Compile As**
Select compile language option for .c and .cpp files. (/TC, /TP)

9. Next, expand the Linker folder and select the Input link. For the Additional Dependencies option, click the ellipses at the end of the empty field on the right side.
10. Manually, type the library name and click OK or you can just directly type the library name in the empty field on the right of the Additional Dependencies. Click OK.



Configuration: Active(Debug) Platform: Active(Win32)

| | |
|---|---|
| Additional Dependencies | ws2_32.lib |
| Ignore All Default Libraries | No |
| Ignore Specific Library | |
| Module Definition File | |
| Add Module to Assembly | |
| Embed Managed Resource File | |
| Force Symbol References | |
| Delay Loaded DLLs | |
| Assembly Link Resource | |

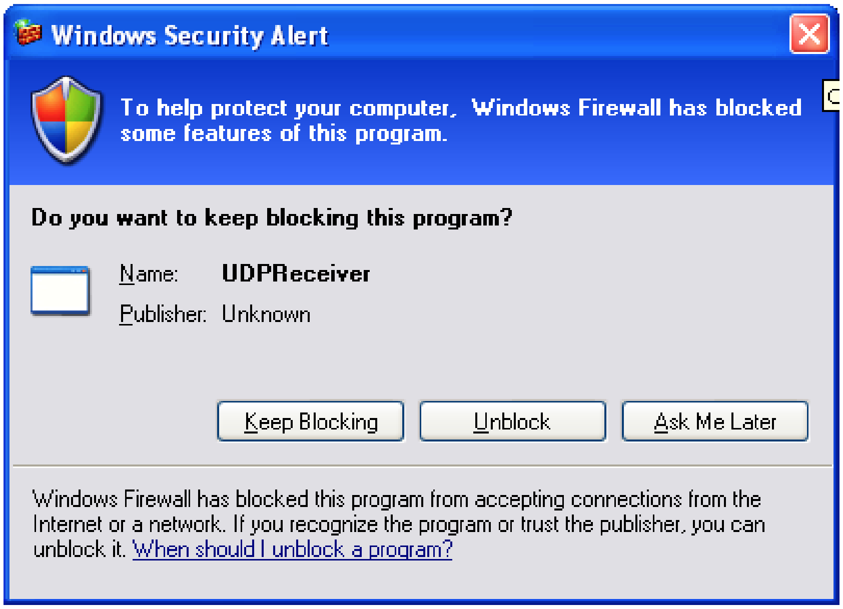
**Additional Dependencies**
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.

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

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

int main(int argc, char **argv)
{
     WSADATA            wsaData;
     SOCKET             ReceivingSocket;
     SOCKADDR_IN        ReceiverAddr;
     int                Port = 5150;
     char          ReceiveBuf[1024];
     int                BufLength = 1024;
     SOCKADDR_IN        SenderAddr;
     int                SenderAddrSize = sizeof(SenderAddr);
     int                ByteReceived = 5;

   // Initialize Winsock version 2.2
   if( WSAStartup(MAKEWORD(2,2), &wsaData) != 0)
   {
        printf("Server: WSAStartup failed with error %ld\n",
WSAGetLastError());
        return -1;
   }
   else
        printf("Server: The Winsock DLL status is %s.\n",
wsaData.szSystemStatus);

     // Create a new socket to receive datagrams on.
     ReceivingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

     if (ReceivingSocket == INVALID_SOCKET)
     {
          printf("Server: Error at socket(): %ld\n", WSAGetLastError());
          // Clean up
          WSACleanup();
          // Exit with error
          return -1;
     }
     else
          printf("Server: socket() is OK!\n");

     // Set up a SOCKADDR_IN structure that will tell bind that we
     // want to receive datagrams from all interfaces using port 5150.

     // The IPv4 family
     ReceiverAddr.sin_family = AF_INET;
     // Port no. 5150
     ReceiverAddr.sin_port = htons(Port);
     // From all interface (0.0.0.0)
     ReceiverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

   // Associate the address information with the socket using bind.
   // At this point you can receive datagrams on your bound socket.
   if (bind(ReceivingSocket, (SOCKADDR *)&ReceiverAddr,
sizeof(ReceiverAddr)) == SOCKET_ERROR)
   {
        printf("Server: bind() failed! Error: %ld.\n", WSAGetLastError());
        // Close the socket
        closesocket(ReceivingSocket);
        // Do the clean up
        WSACleanup();
        // and exit with error
        return -1;
     }
     else
        printf("Server: bind() is OK!\n");

   // Some info on the receiver side...
   getsockname(ReceivingSocket, (SOCKADDR *)&ReceiverAddr, (int
*)sizeof(ReceiverAddr));

   printf("Server: Receiving IP(s) used: %s\n",
inet_ntoa(ReceiverAddr.sin_addr));
   printf("Server: Receiving port used: %d\n",
htons(ReceiverAddr.sin_port));

   printf("Server: I\'m ready to receive a datagram...\n");

   // At this point you can receive datagrams on your bound socket.
   ByteReceived = recvfrom(ReceivingSocket, ReceiveBuf, BufLength, 0,
                            (SOCKADDR *)&SenderAddr, &SenderAddrSize);
   if ( ByteReceived > 0 )
   {
        printf("Server: Total Bytes received: %d\n", ByteReceived);
        printf("Server: The data is \"%s\"\n", ReceiveBuf);
   }
   else if ( ByteReceived <= 0 )
        printf("Server: Connection closed with error code: %ld\n",
WSAGetLastError());
   else
        printf("Server: recvfrom() failed with error code: %d\n",
WSAGetLastError());

   // Some info on the sender side
   getpeername(ReceivingSocket, (SOCKADDR *)&SenderAddr, &SenderAddrSize);
   printf("Server: Sending IP used: %s\n", inet_ntoa(SenderAddr.sin_addr));
   printf("Server: Sending port used: %d\n", htons(SenderAddr.sin_port));

   // When your application is finished receiving datagrams close the
socket.
   printf("Server: Finished receiving. Closing the listening socket...\n");
   if (closesocket(ReceivingSocket) != 0)
        printf("Server: closesocket() failed! Error code: %ld\n",
WSAGetLastError());
   else
        printf("Server: closesocket() is OK\n");

   // When your application is finished call WSACleanup.
   printf("Server: Cleaning up...\n");
   if(WSACleanup() != 0)
        printf("Server: WSACleanup() failed! Error code: %ld\n",
WSAGetLastError());
   else
        printf("Server: WSACleanup() is OK\n");
   // Back to the system
   return 0;
}
```

12. Build the project and make sure there is no error which can be seen (if any) in the Output window normally docked at the bottom of the IDE by default. Run the project and unblock the Windows firewall if any.

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



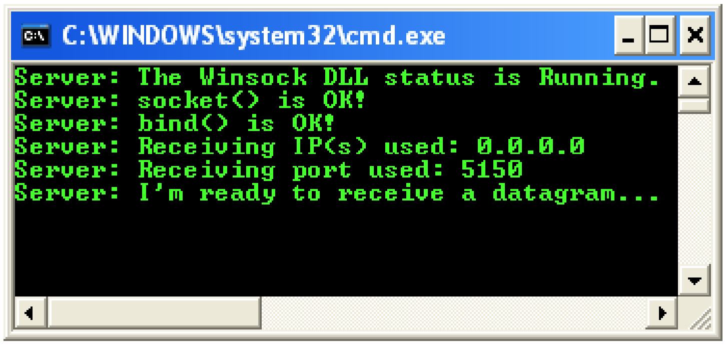
To help protect your computer, Windows Firewall has blocked some features of this program.

Do you want to keep blocking this program?

Name: UDPReceiver
Publisher: Unknown

Windows Firewall has blocked this program from accepting connections from the Internet or a network. If you recognize the program or trust the publisher, you can unblock it. When should I unblock a program?

13. If there is no error, a sample output is shown below. The UDP receiver is ready to receive data, unreliably.



```
Server: The Winsock DLL status is Running.
Server: socket() is OK!
Server: bind() is OK!
Server: Receiving IP(s) used: 0.0.0.0
Server: Receiving port used: 5150
Server: I'm ready to receive a datagram...
```

Now that you understand how to construct a receiver that can receive a datagram, we will describe how to construct a sender.

## Sender

There are two options to send data on a connectionless socket. The first, and simplest, is to create a socket and call either sendto() or WSASendTo(). We'll cover sendto() first, which is defined as:

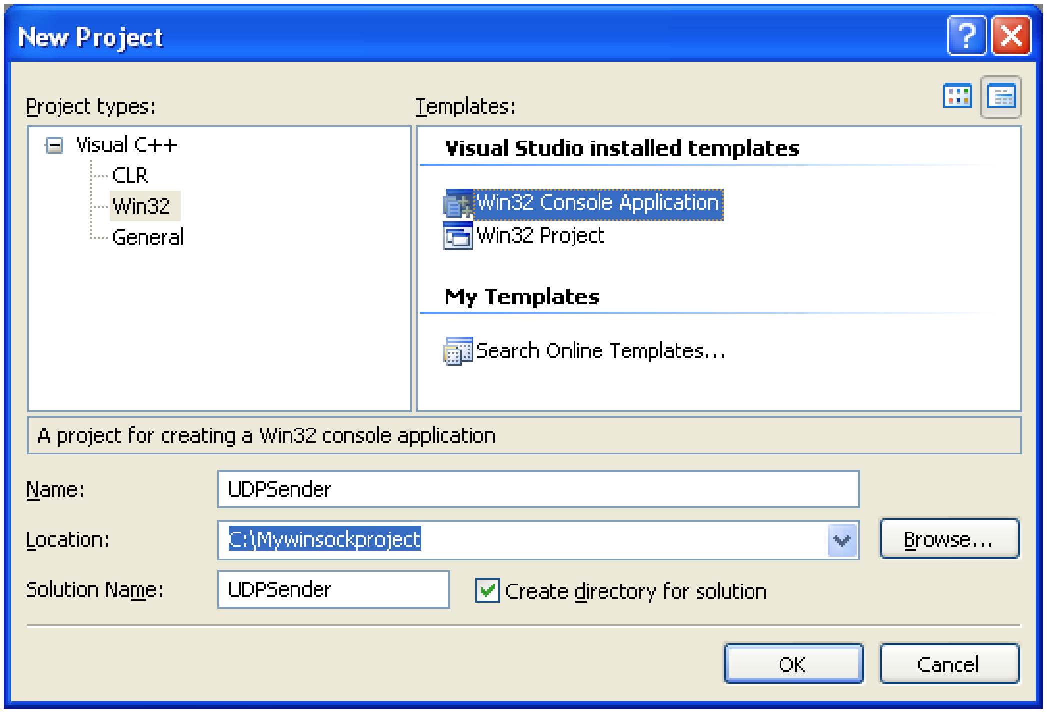
```
int sendto(
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags,
    const struct sockaddr FAR * to,
    int tolen
);
```

The parameters are the same as recvfrom() except that buf is the buffer of data to send and len indicates how many bytes to send. Also, the to parameter is a pointer to a SOCKADDR structure with the destination address of the workstation to receive the data. The Winsock 2 function WSASendTo() can also be used. This function is defined as:

```
int WSASendTo(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    const struct sockaddr FAR * lpTo,
    int iToLen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Again, WSASendTo() is similar to its ancestor. This function takes a pointer to one or more WSABUF structures with data to send to the recipient as the lpBuffers parameter, with dwBufferCount indicating how many structures are present. You can send multiple WSABUF structures to enable scatter-gather I/O. Before returning, WSASendTo() sets the fourth parameter, lpNumberOfBytesSent, to the number of bytes actually sent to the receiver. The lpTo parameter is a SOCKADDR structure for the given protocol, with the recipient's address. The iToLen parameter is the length of the SOCKADDR structure. The last two parameters, lpOverlapped and lpCompletionRoutine, are used for overlapped I/O.
As with receiving data, a connectionless socket can be connected to an endpoint address and data can be sent with send and WSASend(). Once this association is established, you cannot go back to using sendto or WSASendTo() with an address other than the address passed to one of the connect functions. If you attempt to send data to a different address, the call will fail with WSAEISCONN. The only way to disassociate the socket handle from that destination is to call connect() with the destination address of INADDR_ANY.
The following program example demonstrates how to construct a simple UDP sender application which can be used together with the previous UDP receiver program example.

1. While in the Visual C++ IDE, click File menu > Project sub menu to create a new project.
2. Select Win32 for the Project types: and Win32 Console Application for the Templates:. Put the project and solution name. Adjust the project location if needed and click OK.
3. Click Next for the Win32 Application Wizard Overview page. We will remove all the unnecessary project items.
4. In the Application page, select Empty project for the Additional options:. Leave others as given and click Finish.



New Project

Project types: Visual C++ > Win32

Templates: Win32 Console Application

A project for creating a Win32 console application

Name: UDPSender

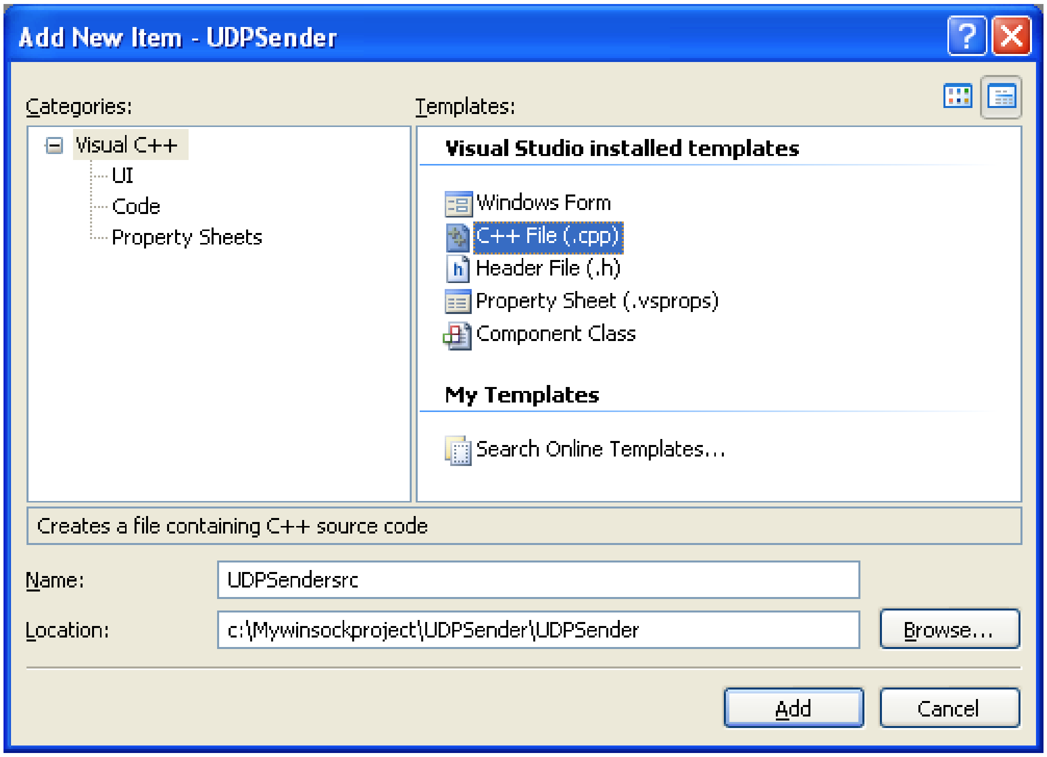
Location: C:\Mywinsockproject

Solution Name: UDPSender

☑ Create directory for solution

Visual C++
CLR
Win32
General
Visual Studio installed templates
Win32 Console Application
Win32 Project
My Templates
Search Online Templates...
Browse...
OK
Cancel

5. Next, we need to add new source file. Click Project menu > Add New Item sub menu or select the project folder in the Solution Explorer > Select Add menu > Select New Item sub menu.
6. Select C++ File (.cpp) for the Templates:. Put the source file name and click Add. Although the extension is .cpp, Visual C++ IDE will recognize that the source code used is C based on the Compile as C Code (/TC) option which will be set in the project property page later.



Creates a file containing C++ source code

Name: UDPSendersrc

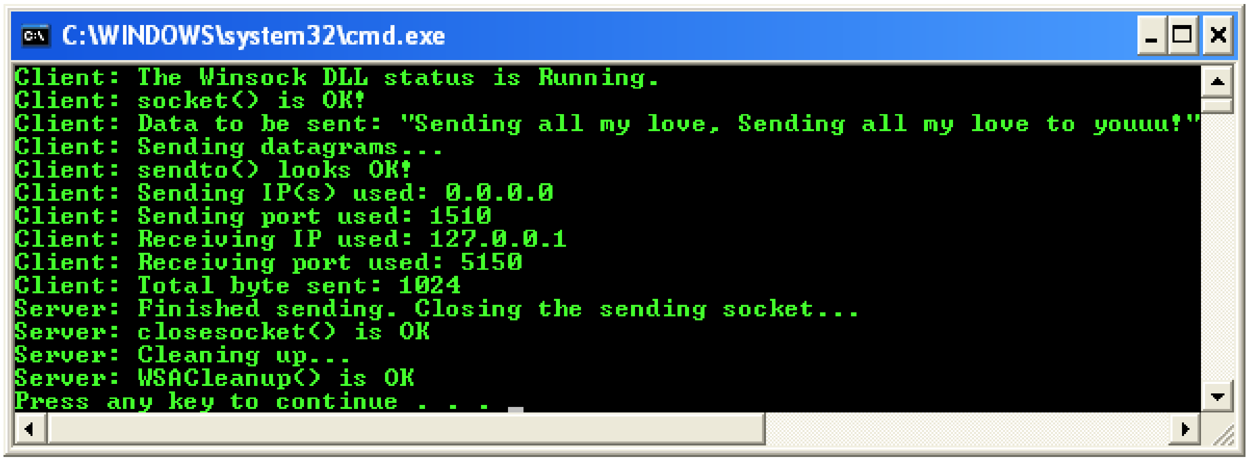
Location: c:\Mywinsockproject\UDPSender\UDPSender

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

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

int main(int argc, char **argv)
{
     WSADATA              wsaData;
     SOCKET               SendingSocket;
     SOCKADDR_IN          ReceiverAddr, SrcInfo;
     int                  Port = 5150;
     char                 *SendBuf = "Sending all my love, Sending all my
love to youuu!";
     int                  BufLength = 1024;
     int len;
     int TotalByteSent;

     // Initialize Winsock version 2.2
     if( WSAStartup(MAKEWORD(2,2), &wsaData) != 0)
     {
          printf("Client: WSAStartup failed with error %ld\n",
WSAGetLastError());
          // Clean up
          WSACleanup();
          // Exit with error
          return -1;
     }
     else
          printf("Client: The Winsock DLL status is %s.\n",
wsaData.szSystemStatus);

     // Create a new socket to receive datagrams on.
     SendingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
     if (SendingSocket == INVALID_SOCKET)
     {
          printf("Client: Error at socket(): %ld\n", WSAGetLastError());
          // Clean up
          WSACleanup();
          // Exit with error
          return -1;
     }
     else
          printf("Client: socket() is OK!\n");

     // Set up a SOCKADDR_IN structure that will identify who we
     // will send datagrams to. For demonstration purposes, let's
     // assume our receiver's IP address is 127.0.0.1 and waiting
     // for datagrams on port 5150.
     ReceiverAddr.sin_family = AF_INET;
     ReceiverAddr.sin_port = htons(Port);
     ReceiverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

     // Send a datagram to the receiver.
     printf("Client: Data to be sent: \"%s\"\n", SendBuf);
     printf("Client: Sending datagrams...\n");
     TotalByteSent = sendto(SendingSocket, SendBuf, BufLength, 0,
                             (SOCKADDR *)&ReceiverAddr,
sizeof(ReceiverAddr));

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

     // Some info on the receiver side...
     // Allocate the required resources
     memset(&SrcInfo, 0, sizeof(SrcInfo));
     len = sizeof(SrcInfo);

     getsockname(SendingSocket, (SOCKADDR *)&SrcInfo, &len);
     printf("Client: Sending IP(s) used: %s\n",
inet_ntoa(SrcInfo.sin_addr));
     printf("Client: Sending port used: %d\n", htons(SrcInfo.sin_port));

     // Some info on the sender side
     getpeername(SendingSocket, (SOCKADDR *)&ReceiverAddr, (int
*)sizeof(ReceiverAddr));
     printf("Client: Receiving IP used: %s\n",
inet_ntoa(ReceiverAddr.sin_addr));
     printf("Client: Receiving port used: %d\n",
htons(ReceiverAddr.sin_port));
     printf("Client: Total byte sent: %d\n", TotalByteSent);

   // When your application is finished receiving datagrams close the
socket.
```

```
   printf("Client: Finished sending. Closing the sending socket...\n");
   if (closesocket(SendingSocket) != 0)
        printf("Client: closesocket() failed! Error code: %ld\n",
WSAGetLastError());
   else
        printf("Server: closesocket() is OK\n");

   // When your application is finished call WSACleanup.
   printf("Client: Cleaning up...\n");
   if(WSACleanup() != 0)
        printf("Client: WSACleanup() failed! Error code: %ld\n",
WSAGetLastError());
   else
        printf("Client: WSACleanup() is OK\n");
   // Back to the system
   return 0;
}
```

8. Before we can build this Winsock C Win32 console application project, we need to set the project to be compiled as C code and link to ws2_32.lib, the Winsock2 library. Invoke the project property page.
9. Expand the Configuration folder > Expand the C/C++ sub folder. Select the Advanced link and for the Compile As option, select Compile as C Code (/TC).
10. Next, expand the Linker folder and select the Input link. For the Additional Dependencies option, click the ellipses at the end of the empty field on the right side.
11. Manually, type the library name and click OK or you can just directly type the library name in the empty field on the right of the Additional Dependencies. Click OK.
12. Build the project and make sure there is no error which can be seen (if any) in the Output window normally docked at the bottom of the IDE by default. Run the project and if there is no error during the build and run stages, a sample output is shown below should be expected.



```
Client: The Winsock DLL status is Running.
Client: socket() is OK!
Client: Data to be sent: "Sending all my love, Sending all my love to youuu!"
Client: Sending datagrams...
Client: sendto() looks OK!
Client: Sending IP(s) used: 0.0.0.0
Client: Sending port used: 1510
Client: Receiving IP used: 127.0.0.1
Client: Receiving port used: 5150
Client: Total byte sent: 1024
Server: Finished sending. Closing the sending socket...
Server: closesocket() is OK
Server: Cleaning up...
Server: WSACleanup() is OK
Press any key to continue . . .
```

< TCP Client-server Examples | Winsock2 Main | UDP Example & Other Winsock2 APIs >