

TCP/IP, Winsock, and WinInet

Program examples compiled using Visual C++ 6.0 compiler on Windows XP Pro machine with Service Pack 2 and some screen snapshot Figures have been taken on Windows 2000 server. Topics and sub topics for this tutorial are listed below. Don't forget to read Tenouk's small [disclaimer](#). Supplementary item is [WEBSITE](#).

TCP/IP, Winsock, and WinInet

To COM or Not to COM

Internet Primer

Network Protocols: Layering

The Internet Protocol

The User Datagram Protocol - UDP

IP Address Format: Network Byte Order

The Transmission Control Protocol - TCP

The Domain Name System

Servers and Domain Names

Clients and Domain Names

HTTP Basics

FTP Basics

Internet vs. Intranet

Winsock

Synchronous vs. Asynchronous Winsock Programming

The MFC Winsock Classes

The Blocking Socket Classes

The CSockAddr Helper Class

The CBlockingSocketException Class

The CBlockingSocket Class

The CHttpBlockingSocket Class

A Simplified HTTP Server Program

Initializing Winsock

Starting the Server

The Server Thread

Cleaning Up

A Simplified HTTP Client Program

Building a Web Server with CHttpBlockingSocket

MYEX33A Server Limitations

MYEX33A Server Architecture

Using the Win32 TransmitFile() Function

Building MYEX33A From Scratch

Back to the Story

Building and Testing MYEX33A

Using Telnet

Building a Web Client with CHttpBlockingSocket

The MYEX33A Winsock Client

MYEX33A Support for Proxy Servers

Testing the MYEX33A Winsock Client

WinInet

WinInet's Advantages over Winsock

The MFC WinInet Classes

CInternetSession

CHttpConnection

CFTPConnection, CGopherConnection

CInternetFile
CHttpFile
CFtpFileFind, CGopherFileFind
CInternetException
Internet Session Status Callbacks
A Simplified WinInet Client Program
Building a Web Client with the MFC WinInet Classes
The MYEX33A WinInet Client #1: Using CHttpConnection
Testing the WinInet Client #1
The MYEX33A WinInet Client #2: Using OpenURL()
Testing the WinInet Client #2
Asynchronous Moniker Files
Monikers
The MFC CAsyncMonikerFile Class
Using the CAsyncMonikerFile Class in a Program
Asynchronous Moniker Files vs. WinInet Programming

TCP/IP, Winsock, and WinInet

As a C++ programmer, you're going to be asked to do more than create Web pages. You'll be the one who makes the Internet reach its true potential and who creates distributed applications that haven't even been imagined yet. To be successful, you'll have to understand how the Internet works and how to write programs that can access other computers on the Internet.

In this section, you'll start with a primer on the Transmission Control Protocol/Internet Protocol (TCP/IP) that's used throughout the Internet, and then you'll move up one level to see the workings of HyperText Transport Protocol (HTTP). Then it's time to get something running. You'll assemble your own intranet (a local version of the Internet) and study an HTTP client-server program based on **Winsock**, the **fundamental API for TCP/IP** in Windows. Finally you'll move on to **WinInet**, which is a **higher level API** than Winsock and part of Microsoft's ActiveX technology.

To COM or Not to COM

Surely you've read about ActiveX Controls for the Internet. You've probably encountered concepts such as **composite monikers** and **anti-monikers**, which are part of the Microsoft Component Object Model (COM). If you were overwhelmed, don't worry, it's possible to program for the Internet without COM, and that's a good place to start. This module and the [next module](#) are mostly COM-free. In [Module 34](#), you'll be writing a COM-based ActiveX document server, but MFC effectively hides the COM details so you can concentrate on **Winsock** and **WinInet programming**. It's not that ActiveX controls aren't important, but we can't do them justice in this book. We'll defer to Adam Denning's book on this subject, *ActiveX Controls Inside Out* (Microsoft Press, 1997). Your study of this book's COM material and Internet material will prepare you well for Adam's book.

Internet Primer

You can't write a good Winsock program without understanding the concept of a [socket](#), which is used to send and receive packets of data across the network. To fully understand sockets, you need a thorough knowledge of the underlying Internet protocols. This section contains a concentrated dose of Internet theory. It should be enough to get you going, but you might want to refer to one of the TCP/IP textbooks if you want more theory or you can dig the [Linux Socket](#), a complete story of the TCP/IP and OSI, down to the packet level, RAW socket and working program example, available at [Tenouk.com](#).

Network Protocols: Layering

All networks use layering for their transmission protocols, and the collection of layers is often called a **stack**. The application program talks to the top layer and the bottom layer talks to the network. Figure 1 shows you the stack for a local area network (LAN) running TCP/IP. Each layer is logically connected to the corresponding layer at the other end of the communications channel. The server program, as shown at the right in Figure 1, continuously listens on one end of the channel, while the client program, as shown on the left, periodically connects with the server to exchange data.

Think of the server as an HTTP-based World Wide Web server, and think of the client as a browser program running on your computer.

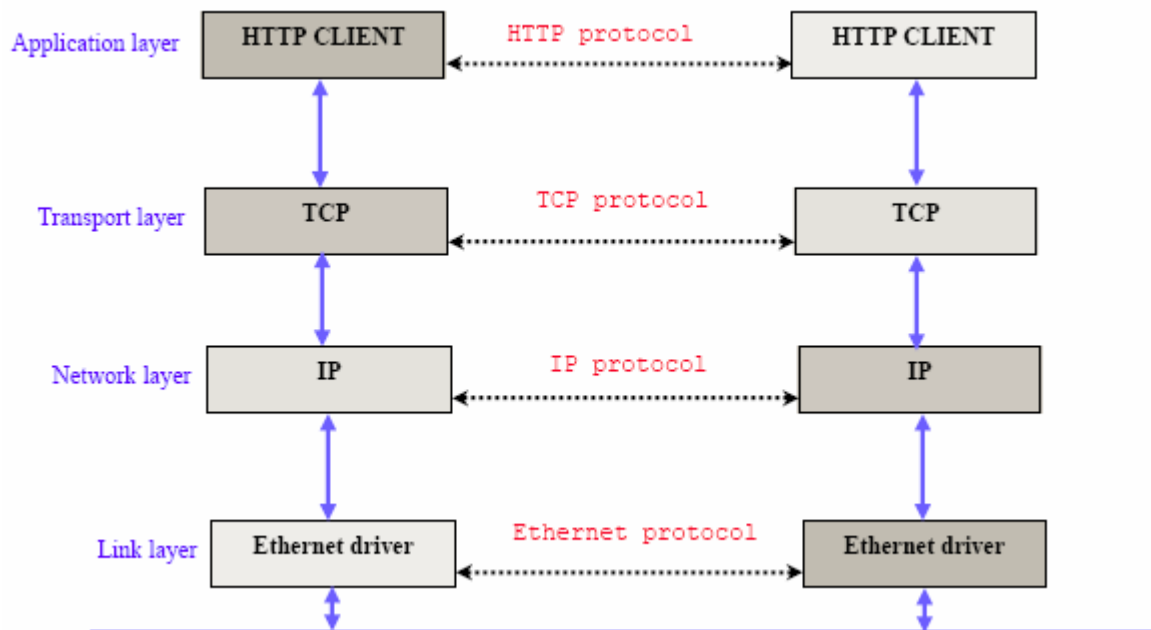


Figure 1: The stack for a LAN running TCP/IP.

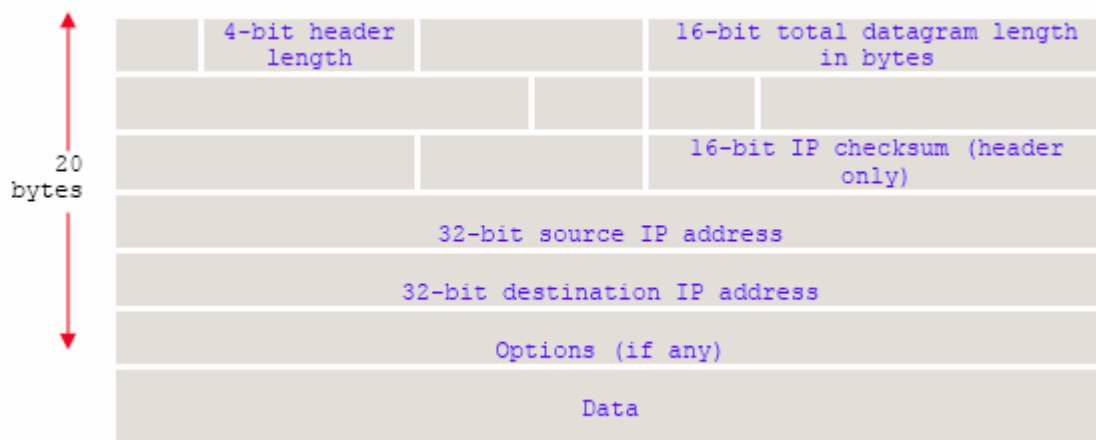
You can get more information of the Internet Protocol suite and other standards discussed in this module at [RFC Editor](#).

The Internet Protocol

The Internet Protocol (IP) layer is the best place to start in your quest to understand TCP/IP. The IP protocol defines packets called **datagrams** that are fundamental units of Internet communication. These packets, typically less than 1000 bytes in length, go bouncing all over the world when you open a Web page, download a file, or send e-mail. Figure 2 shows a simplified layout of an IP datagram.

Notice that the IP datagram contains 32-bit addresses for both the source and destination computers. These IP addresses uniquely identify computers on the Internet and are used by **routers** (specialized computers that act like telephone switches, Layer 3 of the TCP/IP stack) to direct the individual datagrams to their destinations. The routers don't care about what's inside the datagrams, they're only interested in that datagram's **destination address** and **total length**. Their job is to resend the datagram as quickly as possible.

The IP layer doesn't tell the sending program whether a datagram has successfully reached its destination. That's a job for the next layer up the stack, Transmission Control Protocol, TCP. The receiving program can look only at the checksum to determine whether the IP datagram header was corrupted or not.



The User Datagram Protocol - UDP

16-bit source port number	16-bit destination port number
16-bit length (UDP header + data)	16-bit checksum (UDP header + data)
Data (if any)	

A complete UDP/IP datagram is shown in Figure 4.

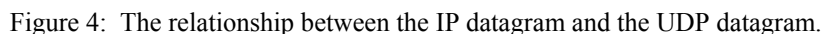


Figure 3 shows that the UDP header does convey some additional information, namely **the source and destination port numbers**. The application programs on each end use these 16-bit numbers. For example, a client program might send a datagram addressed to port 1700 on the server. The server program is listening for any datagram that includes 1700 in its destination port number, and when the server finds one, it can respond by sending another datagram back to the client, which is listening for a datagram that includes 1701 in its destination port number.

You know that IP addresses are 32-bits long. You might think that 2³² (more than 4 billion) uniquely addressed computers could exist on the Internet, but that's not true. Part of the address identifies the LAN on which the host computer is located, and part of it identifies the host computer within the network. Most IP addresses are Class C addresses, version 4 (**IPv4**) which are formatted as shown in Figure 5.



Figure 5: The layout of a Class C IP address.

This means that slightly more than 2 million networks can exist, and each of those networks can have 28 (256) addressable host computers. The Class A and Class B IP addresses, which allow more host computers on a network, are all used up.

The Internet "powers-that-be" have recognized the shortage of IP addresses, so they have proposed a new standard, the IP Next Generation (**IPng**) protocol or **IPv6**. IPng defines a new IP datagram format that uses **128-bit addresses** instead of **32-bit addresses**. With IPng, you'll be able, for example, to assign a unique Internet address to each light switch in your house, so you can switch off your bedroom light from your portable computer from anywhere in the world. IPng already implemented in new computer, network and electronics devices.

By convention, IP addresses are written in dotted-decimal format. The four parts of the address refer to the individual byte values. An example of a Class C IP address is 194.128.198.201. In a computer with an Intel CPU, the address bytes are stored **low-order-to-the-left**, in so-called **little-endian order**. In most other computers, including the UNIX machines that first supported the Internet, bytes are stored **high-order-to-the-left**, in **big-endian order**. Because the Internet imposes a machine-independent standard for data interchange, all multibyte numbers **must be transmitted in big-endian order**. This means that programs running on Intel-based machines must convert between network byte order (big-endian) and host byte order (little-endian). This rule applies to **2-byte port numbers** as well as to **4-byte IP addresses**.

The Transmission Control Protocol - TCP

You've learned about the limitations of UDP. What you really need is a protocol that supports error-free transmission of large blocks of data. Obviously, you want the receiving program to be able to reassemble the bytes in the exact sequence in which they are transmitted, even though the individual datagrams might arrive in the wrong sequence. TCP is that protocol, and it's the principal transport protocol for all Internet applications, including HTTP and File Transfer Protocol (FTP). Figure 6 shows the layout of a TCP segment. (It's not called a datagram.) The TCP segment fits inside an IP datagram, as shown in Figure 7.

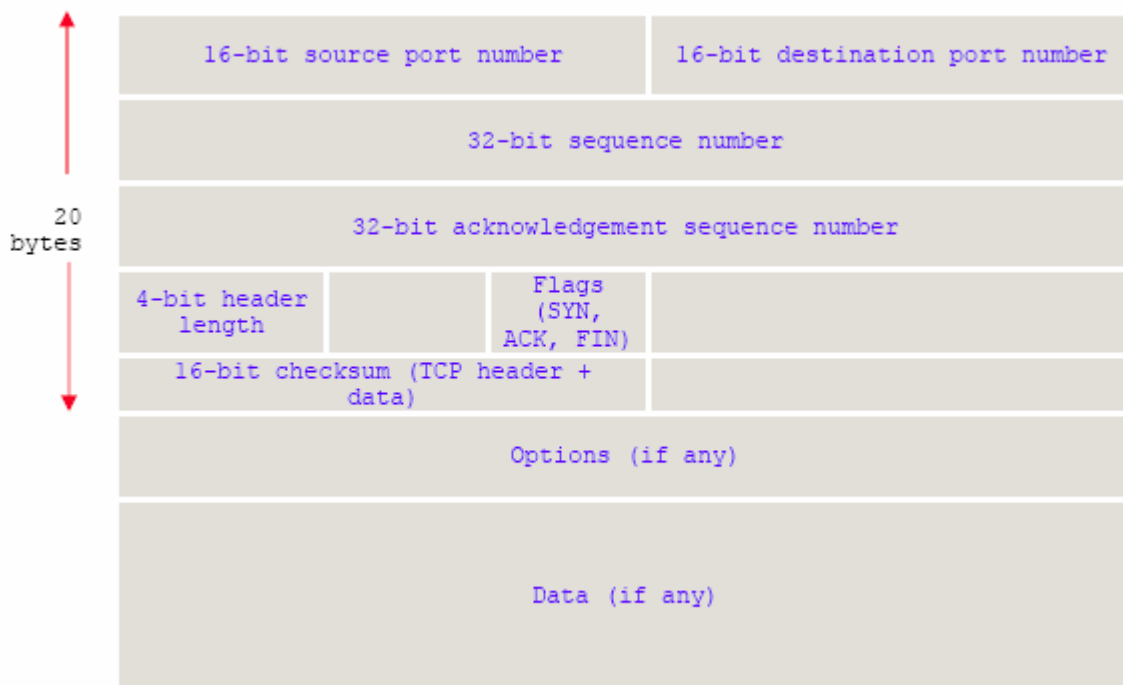


Figure 6: A simple layout of a [TCP](#) segment.

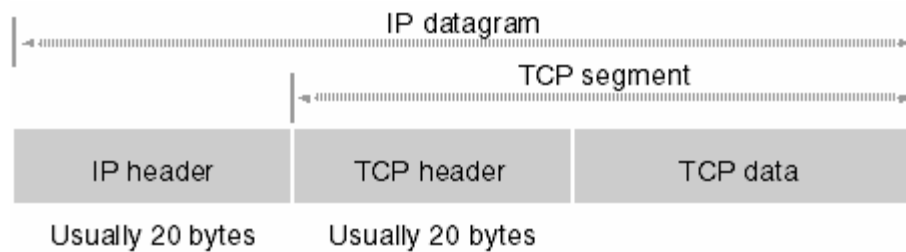


Figure 7: The relationship between an IP datagram and a TCP segment.

The TCP protocol establishes a **full-duplex, point-to-point connection** between two computers, and a program at each end of this connection uses its own port. **The combination of an IP address and a port number is called a socket.** The connection is first established with a three-way handshake. The initiating program sends a segment with the SYN flag set, the responding program sends a segment with both the SYN and ACK flags set, and then the initiating program sends a segment with the ACK flag set.

After the connection is established, each program can send a stream of bytes to the other program. TCP uses the sequence number fields together with ACK flags to control this flow of bytes. The sending program doesn't wait for each segment to be acknowledged but instead sends a number of segments together and then waits for the first acknowledgment. If the receiving program has data to send back to the sending program, it can piggyback its acknowledgment and outbound data together in the same segments.

The sending program's sequence numbers are not segment indexes but rather indexes into the byte stream. The receiving program sends back the sequence numbers (in the acknowledgment number field) to the sending program, thereby ensuring that all bytes are received and assembled in sequence. The sending program resends unacknowledged segments.

Each program closes its end of the TCP connection by sending a segment with the FIN flag set, which must be acknowledged by the program on the other end. A program can no longer receive bytes on a connection that has been closed by the program on the other end.

Don't worry about the complexity of the TCP protocol. The Winsock and WinInet APIs hide most of the details, so you don't have to worry about ACK flags and sequence numbers. Your program calls a function to transmit a block of data, and Windows takes care of splitting the block into segments and stuffing them inside IP datagrams. Windows also takes care of delivering the bytes on the receiving end, but that gets tricky, as you'll see later in this module.

The Domain Name System

When you surf the Web, you don't use IP addresses. Instead, you use human-friendly names like microsoft.com or www.cnn.com. A significant portion of Internet resources is consumed when host names (such as microsoft.com) are translated into IP addresses that TCP/IP can use. A distributed network of name server (domain server) computers performs this translation by processing DNS queries. The entire Internet namespace is organized into domains, starting with an unnamed **root domain**. Under the root is a series of top-level domains (TLDs) such as com, edu, gov, and org. Do not confuse Internet domains with Microsoft Windows NT domains. The latter are logical groups of networked computers that share a common security database.

Servers and Domain Names

Let's look at the server end first. Suppose a company named SlowSoft has two host computers connected to the Internet, one for World Wide Web (WWW) service and the other for FTP service. By convention, these **host computers** are named **www.slowsoft.com** and **ftp.slowsoft.com**, respectively, and both are members of the **second-level domain** slowsoft, which SlowSoft has registered with an organization called InterNIC or other delegated domain name registrars. ([INTERNIC](#).)

Now SlowSoft must designate two (or more) host computers as its name servers. The name servers for the **com domain** each have a database entry (zone record) for the slowsoft domain, and that entry contains the names and IP addresses of SlowSoft's two name servers. Each of the two slowsoft name servers has database entries for both of SlowSoft's host computers. These servers might also have database entries for hosts in other domains, and they might have entries for

name servers in **third-level domains**. Thus, if a name server can't provide a host's IP address directly, it can redirect the query to a lower-level name server. Figure 34-8 illustrates SlowSoft's domain configuration.

A top-level name server runs on its own host computer. InterNIC manages (at last count) nine computers that serve the root domain and top-level domains (root-servers.org). Lower-level name servers could be programs running on host computers anywhere on the Net. SlowSoft's Internet service provider (ISP), ExpensiveNet, can furnish one of SlowSoft's name servers. If the ISP is running Windows NT Server, the name server is usually the DNS program that comes bundled with the operating system. That name server might be designated **ns1.expensivenet.com**. Unix/Linux system will normally use BIND program for the name server.

Clients and Domain Names

Now for the client side. A user types `http://www.slowsoft.com` in the browser. The **http://** prefix tells the browser to use the HTTP protocol when it eventually finds the host computer. The browser must then resolve `www.slowsoft.com` into an IP address, so it uses TCP/IP to send a DNS query to the default gateway IP address for which TCP/IP is configured at the client machine as shown below.

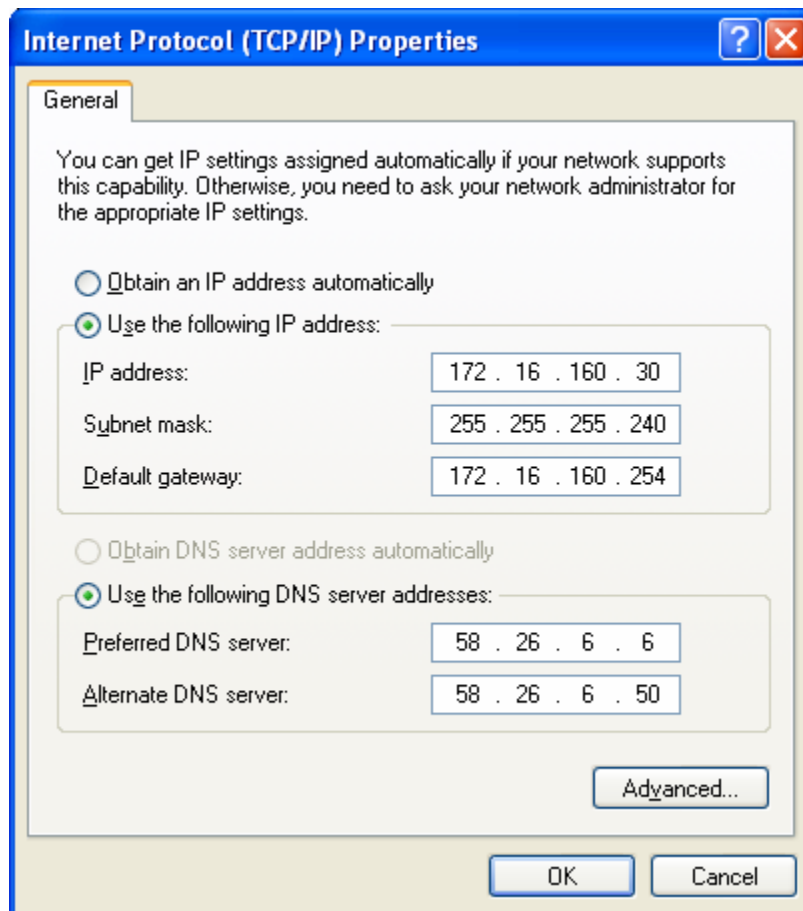


Figure 8: The TCP/IP settings of the network card.

This default gateway address identifies a local name server, which might have the needed host IP address in its cache. If not, the local name server relays the DNS query up to one of the root name servers. The root server looks up `slowsoft` in its database and sends the query back down to one of SlowSoft's designated name servers. In the process, the IP address for `www.slowsoft.com` will be cached for later use if it was not cached already. If you want to go the other way, name servers are also capable of converting an IP address to a name.

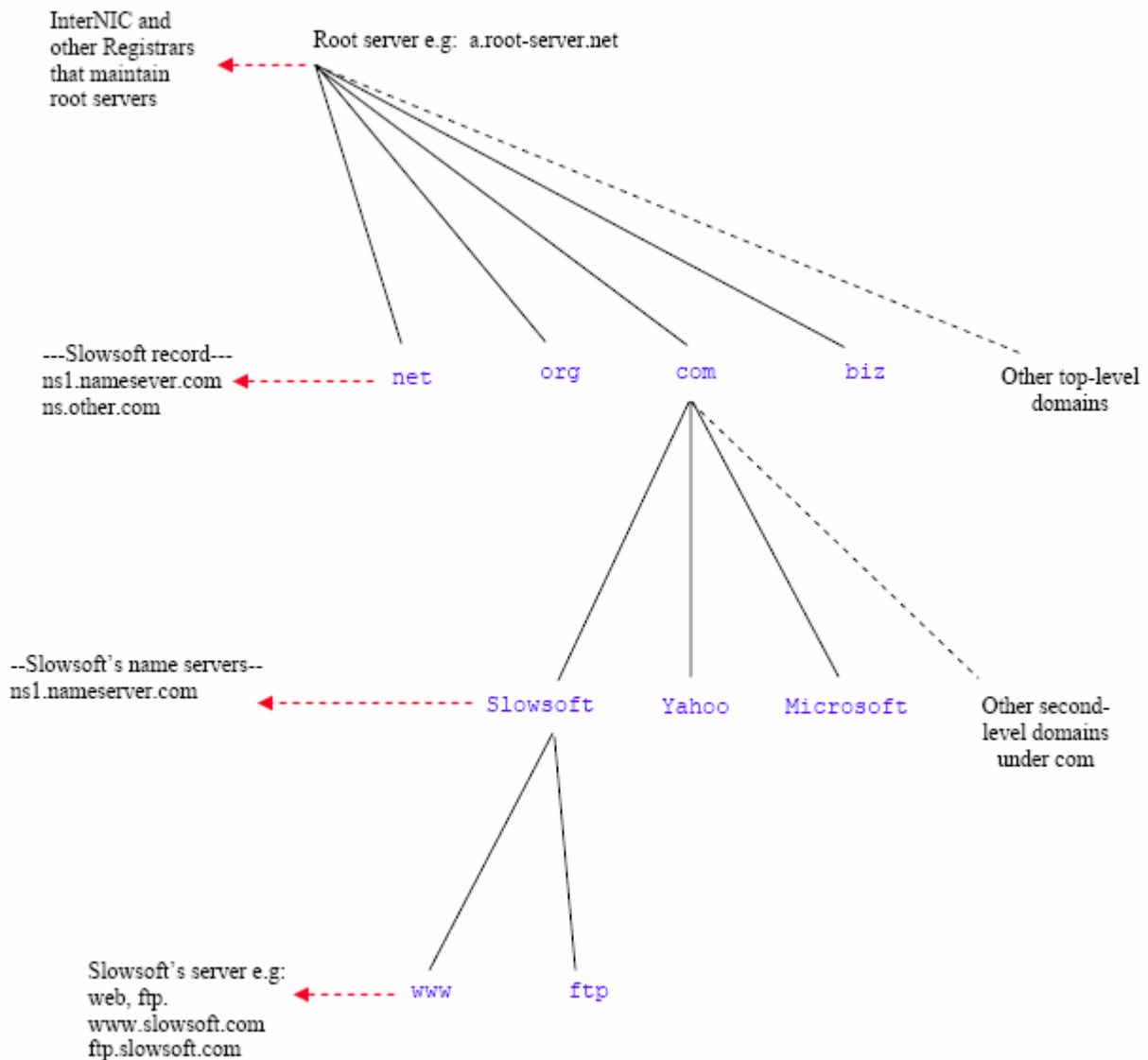


Figure 9: SlowSoft's domain configuration.

HTTP Basics

You're going to be doing some Winsock programming soon, but just sending raw byte streams back and forth isn't very interesting. You need to use a **higher-level protocol** in order to be compatible with existing Internet servers and browsers. HTTP is a good place to start because it's the protocol of the popular World Wide Web and it's relatively simple.

HTTP is built on TCP, and this is the way it works: First a server program listens on the default port 80. Then some client program (typically a browser) connects to the server (www.slowsoft.com, in this case) after receiving the server's IP address from a name server. Using its own port number, the client sets up a two-way TCP connection to the server. As soon as the connection is established, the client sends a request to the server, which might look something like this:

```
GET /customers/newproducts.html HTTP/1.0
```

The server identifies the request as a GET, the most common type, and it concludes that the client wants a file named **newproducts.html** that's located in a server directory known as **/customers** (which might or might not be **customers** on the server's hard disk). Immediately following are request headers, which mostly describe the client's capabilities.

```
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/x
```



```
-jg, */*
Accept-Language: en
UA-pixels: 1024x768
UA-color: color8
UA-OS: Windows NT
UA-CPU: x86
User-Agent: Mozilla/2.0 (compatible; MSIE 3.0; AK; Windows NT)
Host: www.slowsoft.com
Connection: Keep-Alive
If-Modified-Since: Wed, 26 Mar 2005 20:23:04 GMT
(blank line)
```

The If-Modified-Since header tells the server not to bother to transmit newproducts.html unless the file has been modified since March 26, 2005. This implies that the browser already has a dated copy of this file stored in its cache. The blank line at the end of the request is crucial; it provides the only way for the server to tell that it is time to stop receiving and start transmitting, and that's because the TCP connection stays open. Now the server springs into action. It sends newproducts.html, but first it sends an OK response:

```
HTTP/1.0 200 OK
```

Immediately followed by some response header lines:

```
Server: Microsoft-IIS/2.0
Date: Thu, 03 Mar 2005 17:33:12 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Wed, Mar 26 2005 20:23:04 GMT
Content-Length: 407
(blank line)
```

The contents of newproducts.html immediately follow the blank line:

```
<html>
<head><title>SlowSoft's New Products</title></head>
<body><body background="/images/clouds.jpg">
<h1><center>Welcome to SlowSoft's New Products List
</center></h1><p>
Unfortunately, budget constraints have prevented SlowSoft from
  introducing any new products this year. We suggest you keep
  enjoying the old products.<p>
<a href="default.htm">SlowSoft's Home Page</a><p>
</body>
</html>
```

You're looking at elementary **HyperText Markup Language** (HTML) code here, and the resulting Web page won't win any prizes. We won't go into details because dozens of HTML books are already available. From these books, you'll learn that HTML tags are contained in angle brackets and that there's often an "end" tag (with a / character) for every "start" tag. Some tags, such as <a> (hypertext anchor), have attributes. In the example above, the line:

```
<a href="default.htm">SlowSoft's Home Page</a><p>
```

creates a link to another HTML file. The user clicks on "SlowSoft's Home Page," and the browser requests default.htm from the original server.

Actually, newproducts.html references two server files, **default.htm** and **/images/clouds.jpg**. The clouds.jpg file is a JPEG file that contains a background picture for the page. The browser downloads each of these files as a separate transaction, establishing and closing a separate TCP connection each time. The server just dishes out files to any client that asks for them. In this case, the server doesn't know or care whether the same client requested newproducts.html and clouds.jpg. To the server, clients are simply IP addresses and port numbers. In fact, the port number is different for each

request from a client. For example, if ten of your company's programmers are surfing the Web via your company's proxy server (more on proxy servers later), the server sees the same IP address for each client.

Web pages use two dominant graphics formats, GIF and JPEG. GIF files are compressed images that retain all the detail of the original uncompressed image but are usually limited to 256 colors. They support transparent regions and animation. JPEG files are smaller, but they don't carry all the detail of the original file. GIF files are often used for small images such as buttons, and JPEG files are often used for photographic images for which detail is not critical. Visual C++ can read, write, and convert both GIF and JPEG files, but the Win32 API cannot handle these formats unless you supply a special compression/decompression module. There are other formats as well such as PNG.

The HTTP standard includes a `PUT` request type that enables a client program to upload a file to the server. Client programs and server programs seldom implement `PUT`.

FTP Basics

The [File Transfer Protocol](#) handles the uploading and downloading of server files plus directory navigation and browsing. A Windows command-line program called **ftp** (it doesn't work through a Web proxy server) lets you connect to an FTP server using UNIX-like keyboard commands. Browser programs also usually support the FTP protocol in a more user-friendly manner. Normally, you can protect an FTP server's directories with a user-name/password combination, but both strings are passed over the Internet as clear text. Nowadays we have a dedicated ftp client programs such as gFtp, cuteFtp etc. and the connection can be secured one. FTP is based on TCP. Two separate connections are established between the client and server, one for control and one for data.

Internet vs. Intranet

Up to now, we've been assuming that client and server computers were connected to the worldwide Internet. The fact is you can run exactly the same client and server software on a local intranet. An intranet is often implemented on a company's LAN and is used for distributed applications. Users see the familiar browser interface at their client computers, and server computers supply simple Web-like pages or do complex data processing in response to user input. An intranet offers a lot of flexibility. If, for example, you know that all your computers are Intel-based, you can use ActiveX controls and ActiveX document servers that provide ActiveX document support. If necessary, your server and client computers can run custom TCP/IP software that allows communication beyond HTTP and FTP. To secure your company's data, you can separate your intranet completely from the Internet or you can connect it through a **firewall**, which is another name for a **proxy server**.

Winsock

[Winsock](#) is the **lowest level Windows API for TCP/IP programming**. Part of the code is located in **wsock32.dll** (the exported functions that your program calls), and part is inside the **Windows kernel**. You can write both internet server programs and internet client programs using the Winsock API. This API is based on the original Berkeley Sockets API for UNIX. A new and much more complex version, [Winsock 2](#), is included for the first time with Windows NT 4.0, but we'll stick with the old version because of its simplicity. The Winsock 2 has been discussed in module [Winsock2](#).

Synchronous vs. Asynchronous Winsock Programming

Winsock was introduced first for Win16, which did not support multithreading. Consequently, most developers used Winsock in the asynchronous mode. In that mode, all sorts of hidden windows and `PeekMessage()` calls enabled single-threaded programs to make Winsock send and receive calls without blocking, thus keeping the user interface (UI) alive. Asynchronous Winsock programs were complex, often implementing "state machines" that processed callback functions, trying to figure out what to do next based on what had just happened. Well, we're not in 16-bit land anymore, so we can do modern multithreaded programming. If this scares you, go back and review [Module 22](#). Once you get used to multithreaded programming, you'll love it.

In this module, we will make the most of our Winsock calls from worker threads so that the program's main thread is able to carry on with the UI. The worker threads contain nice, sequential logic consisting of blocking Winsock calls.

The MFC Winsock Classes

We try to use MFC classes where it makes sense to use them, but the MFC developers informed us that the `CAsyncSocket` and `CSocket` classes were not appropriate for 32-bit synchronous programming. The Visual C++

online help says you can use CSocket for synchronous programming, but if you look at the source code you'll see some ugly message-based code left over from Win16.

The Blocking Socket Classes

Since we couldn't use MFC, we had to write our own Winsock classes. CBlockingSocket is a thin wrapping of the Winsock API, designed only for synchronous use in a worker thread. The only fancy features are exception-throwing on errors and time-outs for sending and receiving data. The exceptions help you write cleaner code because you don't need to have error tests after every Winsock call. The time-outs (implemented with the Winsock select function) prevent a communication fault from blocking a thread indefinitely.

CHttpBlockingSocket is derived from CBlockingSocket and provides functions for reading HTTP data.

CSockAddr and CBlockingSocketException are helper classes.

The CSockAddr Helper Class

Many Winsock functions take socket address parameters. As you might remember, a socket address consists of a 32-bit IP address plus a 16-bit port number. The actual Winsock type is a 16-byte sockaddr_in structure, which looks like this:

```
struct sockaddr_in {
    short    sin_family;
    u_short  sin_port;
    struct   in_addr sin_addr;
    char     sin_zero[8];
};
```

The IP address is stored as type in_addr, which looks like this:

```
struct in_addr {
    union {
        struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
        struct { u_short s_w1,s_w2; } S_un_w;
        u_long S_addr;
    } S_un;
}
```

These are ugly structures, so we'll derive a programmer-friendly C++ class from sockaddr_in. The file **Blocksock.h** contains the following code for doing this, with inline functions included:

```
class CSockAddr : public sockaddr_in {
public:
    // constructors
    CSockAddr()
    {
        sin_family = AF_INET;
        sin_port = 0;
        sin_addr.s_addr = 0;
    } // Default
    CSockAddr(const SOCKADDR& sa) { memcpy(this, &sa,
        sizeof(SOCKADDR)); }
    CSockAddr(const SOCKADDR_IN& sin) { memcpy(this, &sin,
        sizeof(SOCKADDR_IN)); }
    CSockAddr(const ULONG ulAddr, const USHORT ushPort = 0)
    // parms are host byte ordered
    {
        sin_family = AF_INET;
        sin_port = htons(ushPort);
        sin_addr.s_addr = htonl(ulAddr);
    }
};
```

```

    }
    CSockAddr(const char* pchIP, const USHORT ushPort = 0)
    // dotted IP addr string
    {
        sin_family = AF_INET;
        sin_port = htons(ushPort);
        sin_addr.s_addr = inet_addr(pchIP);
    } // already network byte ordered
    // Return the address in dotted-decimal format
    CString DottedDecimal()
    { return inet_ntoa(sin_addr); }
    // constructs a new CString object
    // Get port and address (even though they're public)
    USHORT Port() const
    { return ntohs(sin_port); }
    ULONG IPAddr() const
    { return ntohl(sin_addr.s_addr); }
    // operators added for efficiency
    const CSockAddr& operator=(const SOCKADDR& sa)
    {
        memcpy(this, &sa, sizeof(SOCKADDR));
        return *this;
    }
    const CSockAddr& operator=(const SOCKADDR_IN& sin)
    {
        memcpy(this, &sin, sizeof(SOCKADDR_IN));
        return *this;
    }
    operator SOCKADDR()
    { return *((LPSOCKADDR) this); }
    operator LPSOCKADDR()
    { return (LPSOCKADDR) this; }
    operator LPSOCKADDR_IN()
    { return (LPSOCKADDR_IN) this; }
};

```

As you can see, this class has some useful constructors and conversion operators, which make the `CSockAddr` object interchangeable with the type `sockaddr_in` and the equivalent types `SOCKADDR_IN`, `sockaddr`, and `SOCKADDR`. There's a constructor and a member function for IP addresses in dotted-decimal format. The internal socket address is in network byte order, but the member functions all use host byte order parameters and return values. The Winsock functions `htonl`, `htons`, `ntohs`, and `ntohl` take care of the conversions between network and host byte order.

The CBlockingSocketException Class

All the `CBlockingSocket` functions throw a `CBlockingSocketException` object when Winsock returns an error. This class is derived from the MFC `CException` class and thus overrides the `GetErrorMessage()` function. This function gives the Winsock error number and a character string that `CBlockingSocket` provided when it threw the exception.

The CBlockingSocket Class

Listing 1 shows an excerpt from the header file for the `CBlockingSocket` class.

BLOCKSOCK.H

```

class CBlockingSocket : public CObject
{
    DECLARE_DYNAMIC(CBlockingSocket)
public:

```

```

SOCKET m_hSocket;
CBlockingSocket(); { m_hSocket = NULL; }
void Cleanup();
void Create(int nType = SOCK_STREAM);
void Close();
void Bind(LPCSOCKADDR psa);
void Listen();
void Connect(LPCSOCKADDR psa);
BOOL Accept(CBlockingSocket& s, LPCSOCKADDR psa);
int Send(const char* pch, const int nSize, const int nSecs);
int Write(const char* pch, const int nSize, const int nSecs);
int Receive(char* pch, const int nSize, const int nSecs);
int SendDatagram(const char* pch, const int nSize, LPCSOCKADDR psa,
    const int nSecs);
int ReceiveDatagram(char* pch, const int nSize, LPCSOCKADDR psa,
    const int nSecs);
void GetPeerAddr(LPCSOCKADDR psa);
void GetSockAddr(LPCSOCKADDR psa);
static CSockAddr GetHostByName(const char* pchName,
    const USHORT ushPort = 0);
static const char* GetHostByAddr(LPCSOCKADDR psa);
operator SOCKET();
    { return m_hSocket; }
};

```

Listing 1: Excerpt from the header file for the CBlockingSocket class.

Following is a list of the CBlockingSocket member functions, starting with the constructor:

- **Constructor()**: The CBlockingSocket constructor makes an uninitialized object. You must call the **Create()** member function to create a Windows socket and connect it to the C++ object.
- **Create()**: This function calls the Winsock socket function and then sets the **m_hSocket** data member to the returned 32-bit SOCKET handle.

Parameter	Description
nType	Type of socket; should be SOCK_STREAM (the default value) or SOCK_DGRAM .

Table 1.

- **Close()**: This function closes an open socket by calling the Winsock closesocket function. The **Create()** function must have been called previously. The destructor does not call this function because it would be impossible to catch an exception for a global object. Your server program can call **Close()** anytime for a socket that is listening.
- **Bind()**: This function calls the Winsock bind function to bind a previously created socket to a specified socket address. Prior to calling **Listen()**, your server program calls **Bind()** with a socket address containing the listening port number and server's IP address. If you supply **INADDR_ANY** as the IP address, Winsock deciphers your computer's IP address.

Parameter	Description
psa	A CSockAddr object or a pointer to a variable of type sockaddr.

Table 2.

- **Listen()**: This TCP function calls the Winsock listen function. Your server program calls **Listen()** to begin listening on the port specified by the previous **Bind()** call. The function returns immediately.
- **Accept()**: This TCP function calls the Winsock accept function. Your server program calls **Accept()** immediately after calling **Listen()**. **Accept** returns when a client connects to the socket, sending back a new socket (in a CBlockingSocket object that you provide) that corresponds to the new connection.

Parameter	Description
-----------	-------------

<code>s</code>	A reference to an existing <code>CBlockingSocket</code> object for which <code>Create()</code> has not been called.
<code>psa</code>	A <code>CSocketAddr</code> object or a pointer to a variable of type <code>sockaddr</code> for the connecting socket's address.
Return value	TRUE if successful.

Table 3.

- `Connect()`: This TCP function calls the Winsock connect function. Your client program calls `Connect()` after calling `Create()`. `Connect` returns when the connection has been made.

Parameter	Description
<code>psa</code>	A <code>CSocketAddr</code> object or a pointer to a variable of type <code>sockaddr</code> .

Table 4.

- `Send()`: This TCP function calls the Winsock send function after calling select to activate the time-out. The number of bytes actually transmitted by each `Send()` call depends on how quickly the program at the other end of the connection can receive the bytes. `Send()` throws an exception if the program at the other end closes the socket before it reads all the bytes.

Parameter	Description
<code>pch</code>	A pointer to a buffer that contains the bytes to send.
<code>nSize</code>	The size (in bytes) of the block to send.
<code>nSecs</code>	Time-out value in seconds.
Return value	The actual number of bytes sent.

Table 5.

- `Write()`: This TCP function calls `Send()` repeatedly until all the bytes are sent or until the receiver closes the socket.

Parameter	Description
<code>pch</code>	A pointer to a buffer that contains the bytes to send.
<code>nSize</code>	The size (in bytes) of the block to send.
<code>nSecs</code>	Time-out value in seconds.
Return value	The actual number of bytes sent.

Table 6.

- `Receive()`: This TCP function calls the Winsock recv function after calling select to activate the time-out. This function returns only the bytes that have been received. For more information, see the description of the `CHttpBlockingSocket` class in the next section.

Parameter	Description
<code>pch</code>	A pointer to an existing buffer that will receive the incoming bytes.
<code>nSize</code>	The maximum number of bytes to receive.
<code>nSecs</code>	Time-out value in seconds.
Return value	The actual number of bytes received.

Table 7.

- `SendDatagram()`: This UDP function calls the Winsock `sendto()` function. The program on the other end needs to call `ReceiveDatagram()`. There is no need to call `Listen()`, `Accept()`, or `Connect()` for datagrams. You must have previously called `Create()` with the parameter set to `SOCK_DGRAM`.

Parameter	Description
-----------	-------------

<code>pch</code>	A pointer to a buffer that contains the bytes to send.
<code>nSize</code>	The size (in bytes) of the block to send.
<code>psa</code>	The datagram's destination address; a <code>CsockAddr</code> object or a pointer to a variable of type <code>sockaddr</code> .
<code>nSecs</code>	Time-out value in seconds.
Return value	The actual number of bytes sent.

Table 8.

- `ReceiveDatagram()`: This UDP function calls the Winsock `recvfrom()` function. The function returns when the program at the other end of the connection calls `SendDatagram()`. You must have previously called `Create()` with the parameter set to `SOCK_DGRAM`.

Parameter	Description
<code>pch</code>	A pointer to an existing buffer that will receive the incoming bytes.
<code>nSize</code>	The size (in bytes) of the block to send.
<code>psa</code>	The datagram's destination address; a <code>CsockAddr</code> object or a pointer to a variable of type <code>sockaddr</code> .
<code>nSecs</code>	Time-out value in seconds.
Return value	The actual number of bytes received.

Table 9.

- `GetPeerAddr()`: This function calls the Winsock `getpeername()` function. It returns the port and IP address of the socket on the other end of the connection. If you are connected to the Internet through a Web proxy server, the IP address is the proxy server's IP address.

Parameter	Description
<code>psa</code>	A <code>CsockAddr</code> object or a pointer to a variable of type <code>sockaddr</code> .

Table 10.

- `GetSockAddr()`: This function calls the Winsock `getsockname()` function. It returns the socket address that Winsock assigns to this end of the connection. If the other program is a server on a LAN, the IP address is the address assigned to this computer's network board. If the other program is a server on the Internet, your service provider assigns the IP address when you dial in. In both cases, Winsock assigns the port number, which is different for each connection.

Parameter	Description
<code>psa</code>	A <code>CsockAddr</code> object or a pointer to a variable of type <code>sockaddr</code> .

Table 11.

- `GetHostByName()`: This static function calls the Winsock function `gethostbyname()`. It queries a name server and then returns the socket address corresponding to the host name. The function times out by itself.

Parameter	Description
<code>pchName</code>	A pointer to a character array containing the host name to resolve.
<code>ushPort</code>	The port number (default value 0) that will become part of the returned socket address.
Return value	The socket address containing the IP address from the DNS plus the port number <code>ushPort</code> .

Table 12.

- `GetHostByAddr()`: This static function calls the Winsock `gethostbyaddr()` function. It queries a name server and then returns the host name corresponding to the socket address. The function times out by itself.

Parameter	Description
<code>psa</code>	A <code>CsockAddr</code> object or a pointer to a variable of type <code>sockaddr</code> .
Return value	A pointer to a character array containing the host name; the caller should not delete this memory.

Table 13.

- `Cleanup()`: This function closes the socket if it is open. It doesn't throw an exception, so you can call it inside an exception catch block.
- Operator `SOCKET`: This overloaded operator lets you use a `CBlockingSocket` object in place of a `SOCKET` parameter.

The `CHttpBlockingSocket` Class

If you call `CBlockingSocket::Receive`, you'll have a difficult time knowing when to stop receiving bytes. Each call returns the bytes that are stacked up at your end of the connection at that instant. If there are no bytes, the call blocks, but if the sender closed the socket, the call returns zero bytes. In the HTTP section, you learned that the client sends a request terminated by a blank line. The server is supposed to send the response headers and data as soon as it detects the blank line, but the client needs to analyze the response headers before it reads the data. This means that as long as a TCP connection remains open, the receiving program must process the received data as it comes in. A simple but inefficient technique would be to call `Receive()` for 1 byte at a time. A better way is to use a buffer. The `CHttpBlockingSocket` class adds buffering to `CBlockingSocket`, and it provides two new member functions. Here is part of the **Blocksock.h** file:

```
class CHttpBlockingSocket : public CBlockingSocket
{
public:
    DECLARE_DYNAMIC(CHttpBlockingSocket)
    enum {nSizeRecv = 1000}; // max receive buffer size (> hdr line
                            // length)
    CHttpBlockingSocket();
    ~CHttpBlockingSocket();
    int ReadHttpRequestLine(char* pch, const int nSize, const int nSecs);
    int ReadHttpResponse(char* pch, const int nSize, const int nSecs);
private:
    char* m_pReadBuf; // read buffer
    int m_nReadBuf; // number of bytes in the read buffer
};
```

The constructor and destructor take care of allocating and freeing a 1000-character buffer. The two new member functions are as follows:

- `ReadHttpRequestLine()`: This function returns a single header line, terminated with a **<cr><lf>** pair. `ReadHttpRequestLine()` inserts a terminating zero at the end of the line. If the line buffer is full, the terminating zero is stored in the last position.

Parameter	Description
<code>pch</code>	A pointer to an existing buffer that will receive the incoming line (zero-terminated).
<code>nSize</code>	The size of the <code>pch</code> buffer.
<code>nSecs</code>	Time-out value in seconds.
Return value	The actual number of bytes received, excluding the terminating zero.

Table 14.

- `ReadHttpResponse()`: This function returns the remainder of the server's response received when the socket is closed or when the buffer is full. Don't assume that the buffer contains a terminating zero.

Parameter	Description
<code>pch</code>	A pointer to an existing buffer that will receive the incoming data.
<code>nSize</code>	The maximum number of bytes to receive.
<code>nSecs</code>	Time-out value in seconds.
Return value	The actual number of bytes received.

Table 15.

A Simplified HTTP Server Program

Now it's time to use the blocking socket classes to write an HTTP server program. All the frills have been eliminated, but the code actually works with a browser. This server doesn't do much except return some hard-coded headers and HTML statements in response to any GET request. (See the MYEX33A program later in this module for a more complete HTTP server.)

Initializing Winsock

Before making any Winsock calls, the program must initialize the Winsock library. The following statements in the application's `InitInstance()` member function do the job:

```
WSADATA wsd;
WSAStartup(0x0101, &wsd);
```

Starting the Server

The server starts in response to some user action, such as a menu choice. Here's the command handler:

```
CBlockingSocket g_sListen; // one-and-only global socket for listening
void CSocketView::OnInternetStartServer()
{
    try {
        CSockAddr saServer(INADDR_ANY, 80);
        g_sListen.Create();
        g_sListen.Bind(saServer);
        g_sListen.Listen();
        AfxBeginThread(ServerThreadProc, GetSafeHwnd());
    }
    catch(CBlockingSocketException* e) {
        g_sListen.Cleanup();
        // Do something about the exception
        e->Delete();
    }
}
```

Pretty simple, really. The handler creates a socket, starts listening on it, and then starts a worker thread that waits for some client to connect to port 80. If something goes wrong, an exception is thrown. The global `g_sListen` object lasts for the life of the program and is capable of accepting multiple simultaneous connections, each managed by a separate thread.

The Server Thread

Now let's look at the `ServerThreadProc()` function:

```
UINT ServerThreadProc(LPVOID pParam)
{
```

```

CSockAddr saClient;
CHttpBlockingSocket sConnect;
char request[100];
char headers[] = "HTTP/1.0 200 OK\r\n"
    "Server: Inside Visual C++ SOCK01\r\n"
    "Date: Thu, 05 Sep 2005 17:33:12 GMT\r\n"
    "Content-Type: text/html\r\n"
    "Accept-Ranges: bytes\r\n"
    "Content-Length: 187\r\n"
    "\r\n"; // the important blank line
char html[] =
    "<html><head><title>Inside Visual C++ Server</title></head>\r\n"
    "<body><body background=\"../samples/images/usa1.jpg\">\r\n"
    "<h1><center>This is a custom home page</center></h1><p>\r\n"
    "</body></html>\r\n\r\n";
try {
    if(!g_sListen.Accept(sConnect, saClient)) {
        // Handler in view class closed the listening socket
        return 0;
    }
    AfxBeginThread(ServerThreadProc, pParam);
    // read request from client
    sConnect.ReadHttpHeaderLine(request, 100, 10);
    TRACE("SERVER: %s", request); // Print the first header
    if(strncmp(request, "GET", 3) == 0) {
        do { // Process the remaining request headers
            sConnect.ReadHttpHeaderLine(request, 100, 10);
            TRACE("SERVER: %s", request); // Print the other headers
        } while(strcmp(request, "\r\n"));
        sConnect.Write(headers, strlen(headers), 10); // response hdrs
        sConnect.Write(html, strlen(html), 10); // HTML code
    }
    else {
        TRACE("SERVER: not a GET\n");
        // don't know what to do
    }
    sConnect.Close(); // Destructor doesn't close it
}
catch(CBlockingSocketException* e) {
    // Do something about the exception
    e->Delete();
}
return 0;
}

```

The most important function call is the `Accept()` call. The thread blocks until a client connects to the server's port 80, and then `Accept()` returns with a new socket, `sConnect`. The current thread immediately starts another thread. In the meantime, the current thread must process the client's request that just came in on `sConnect`. It first reads all the request headers by calling `ReadHttpHeaderLine()` until it detects a blank line. Then it calls `Write()` to send the response headers and the HTML statements. Finally, the current thread calls `Close()` to close the connection socket. End of story for this connection. The next thread is sitting, blocked at the `Accept()` call, waiting for the next connection.

Cleaning Up

To avoid a memory leak on exit, the program must ensure that all worker threads have been terminated. The simplest way to do this is to close the listening socket. This forces any thread's pending `Accept()` to return `FALSE`, causing the thread to exit.

```

try {
    g_sListen.Close();
    Sleep(340); // Wait for thread to exit
    WSACleanup(); // Terminate Winsock
}
catch(CUserException* e) {
    e->Delete();
}

```

A problem might arise if a thread were in the process of fulfilling a client request. In that case, the main thread should positively ensure that all threads have terminated before exiting.

A Simplified HTTP Client Program

Now for the client side of the story, a simple working program that does a blind GET request. When a server receives a GET request with a slash, as shown below, it's supposed to deliver its default HTML file:

```
GET / HTTP/1.0
```

If you typed **http://www.slowsoft.com** in a browser, the browser sends the blind GET request. This client program can use the same `CHttpBlockingSocket` class you've already seen, and it must initialize Winsock the same way the server did. A command handler simply starts a client thread with a call like this:

```
AfxBeginThread(ClientSocketThreadProc, GetSafeHwnd());
```

Here's the client thread code:

```

CString g_strServerName = "localhost"; // or some other host name
UINT ClientSocketThreadProc(LPVOID pParam)
{
    CHttpBlockingSocket sClient;
    char* buffer = new char[MAXBUF];
    int nBytesReceived = 0;
    char request[] = "GET / HTTP/1.0\r\n";
    char headers[] = // Request headers
        "User-Agent: Mozilla/1.22 (Windows; U; 32bit)\r\n"
        "Accept: */*\r\n"
        "Accept: image/gif\r\n"
        "Accept: image/x-xbitmap\r\n"
        "Accept: image/jpeg\r\n"
        "\r\n"; // need this
    CSockAddr saServer, saClient;
    try {
        sClient.Create();
        saServer = CBlockingSocket::GetHostByName(g_strServerName, 80);
        sClient.Connect(saServer);
        sClient.Write(request, strlen(request), 10);
        sClient.Write(headers, strlen(headers), 10);
        do { // Read all the server's response headers
            nBytesReceived = sClient.ReadHttpHeaderLine(buffer, 100, 10);
        } while(strcmp(buffer, "\r\n")); // through the first blank line
        nBytesReceived = sClient.ReadHttpResponse(buffer, 100, 10);
        if(nBytesReceived == 0) {
            AfxMessageBox("No response received");
        }
        else {
            buffer[nBytesReceived] = '\0';
            AfxMessageBox(buffer);
        }
    }
}

```

```

    }
    catch(CBlockingSocketException* e) {
        // Log the exception
        e->Delete();
    }
    sClient.Close();
    delete [] buffer;
    return 0; // The thread exits
}

```

This thread first calls `CBlockingSocket::GetHostByName` to get the server computer's IP address. Then it creates a socket and calls `Connect()` on that socket. Now there's a two-way communication channel to the server. The thread sends its `GET` request followed by some request headers, reads the server's response headers, and then reads the response file itself, which it assumes is a text file. After the thread displays the text in a message box, it exits.

Building a Web Server with `CHttpBlockingSocket`

If you need a Web server, your best bet is to buy one or to use the Microsoft Internet Information Server (IIS) that comes bundled with Windows NT/2000/2003 Server. Of course, you'll learn more if you build your own server and you'll also have a useful diagnostic tool. And what if you need features that IIS can't deliver? Suppose you want to add Web server capability to an existing Windows application, or suppose you have a custom ActiveX control that sets up its own non-HTTP TCP connection with the server. Take a good look at the server code in `MYEX33A`, which works under Windows NT, Windows 95, and Windows 98. It might work as a foundation for your next custom server application.

MYEX33A Server Limitations

The server part of the `MYEX33A` program honors `GET` requests for files, and it has logic for processing `POST` requests. (`POST` requests are described in [Module 33](#).) These are the two most common HTTP request types. `MYEX33A` will not, however, launch **Common Gateway Interface** (CGI) scripts or load **Internet Server Application Programming Interface** (ISAPI) DLLs. You'll learn more about ISAPI in [Module 33](#).) `MYEX33A` makes no provision for security, and it doesn't have FTP capabilities. Other than that, it's a great server! If you want the missing features, just write the code for them yourself.

MYEX33A Server Architecture

You'll soon see that `MYEX33A` combines an **HTTP server**, a **Winsock HTTP client**, and two **WinInet HTTP clients**. All three clients can talk to the built-in server or to any other server on the Internet. Any client program, including the Telnet utility and standard browsers such as Microsoft Internet Explorer, can communicate with the `MYEX33A` server. You'll examine the client sections a little later in this module.

`MYEX33A` is a standard **MFC SDI document-view application** with a view class derived from `CEditView`. The main menu includes **Start Server** and **Stop Server** menu choices as well as a **Configuration** command that brings up a tabbed dialog for setting the home directory, the default file for blind `GET`s, and the listening port number (default is 80). The **Start Server** command handler starts a global socket listening and then launches a thread, as in the simplified HTTP server described previously. Look at the `ServerThreadProc()` function included in the file **ServerThread.cpp** of the `MYEX33A` project. Each time a server thread processes a request, it logs the request by sending a message to the `CEditView` window. It also sends messages for exceptions, such as bind errors.

The primary job of the server is to deliver files. It first opens a file, storing a `CFile` pointer in `pFile`, and then it reads 10 KB (`SERVERMAXBUF`) blocks and writes them to the socket `sConnect`, as shown in the code below:

```

char* buffer = new char[SERVERMAXBUF];
DWORD dwLength = pFile->GetLength();
nBytesSent = 0;
DWORD dwBytesRead = 0;
UINT uBytesToRead;
while(dwBytesRead < dwLength) {
    uBytesToRead = min(SERVERMAXBUF, dwLength - dwBytesRead);
    VERIFY(pFile->Read(buffer, uBytesToRead) == uBytesToRead);
    nBytesSent += sConnect.Write(buffer, uBytesToRead, 10);
}

```

```
        dwBytesRead += uBytesToRead;
    }
```

The server is programmed to respond to a GET request for a phony file named `Custom`. It generates some HTML code that displays the client's IP address, port number, and a sequential connection number. This is one possibility for server customization.

The server normally listens on a socket bound to address `INADDR_ANY`. This is the server's default IP address determined by the Ethernet board or assigned during your connection to your ISP. If your server computer has several IP addresses, you can force the server to listen to one of them by filling in the **Server IP Address** in the **Advanced Configuration** page. You can also change the server's listening port number on the Server page. If you choose port 90, for example, browser users would connect to `http://localhost:90` but make sure the port that you select does not conflict with the [well known ports](#). The leftmost status bar indicator pane displays "Listening" when the server is running.

Using the Win32 `TransmitFile()` Function

If you have Windows NT 4.0 and above, you can make your server more efficient by using the Win32 `TransmitFile()` function in place of the `CFile::Read` loop in the code excerpt shown. `TransmitFile()` sends bytes from an open file directly to a socket and is highly optimized. The MYEX33A `ServerThreadProc()` function contains the following line:

```
if (::TransmitFile(sConnect, (HANDLE) pFile > m_hFile, dwLength, 0,
    NULL, NULL, TF_DISCONNECT))
```

If you have Windows NT, uncomment the line:

```
#define USE_TRANSMITFILE
```

at the top of **ServerThread.cpp** to activate the `TransmitFile()` logic.

Building MYEX33A From Scratch

Well, before we dig any deeper, let build MYMFC33A project from scratch. As usual, select **New Project** in Visual C++ and follow the shown steps.

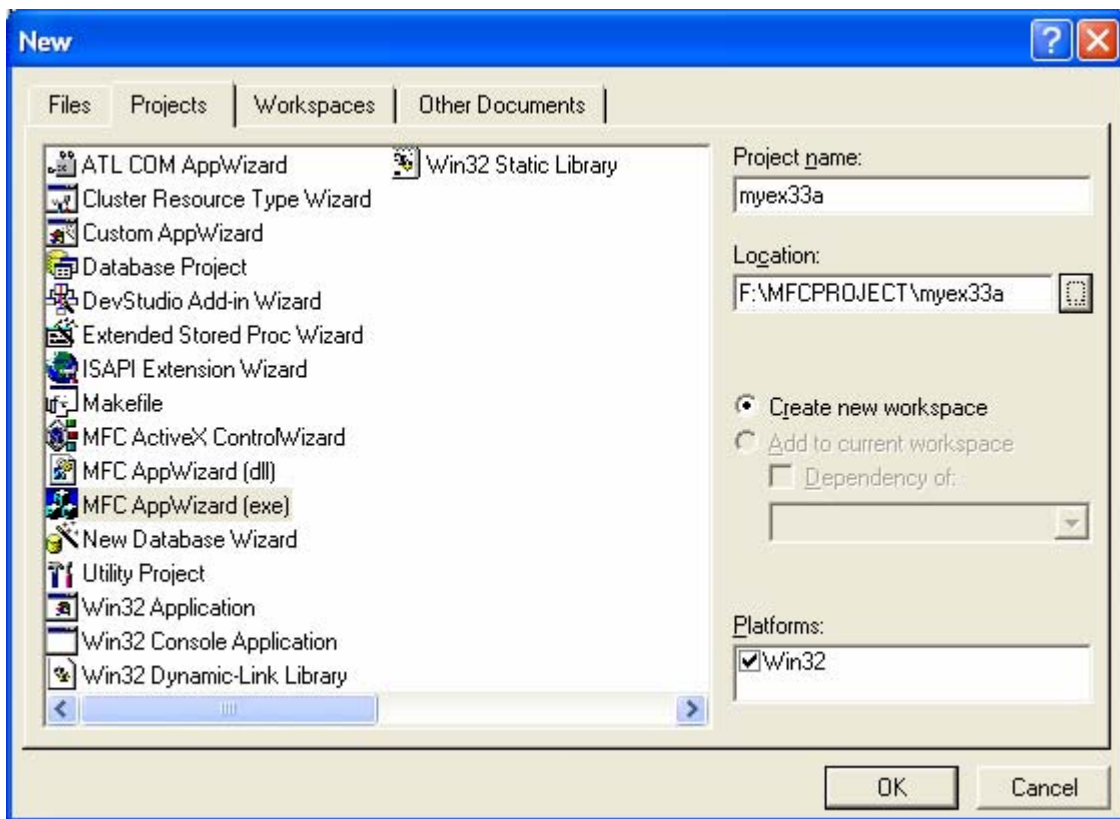


Figure 10: MYEX33A – Visual C++ New project dialog.

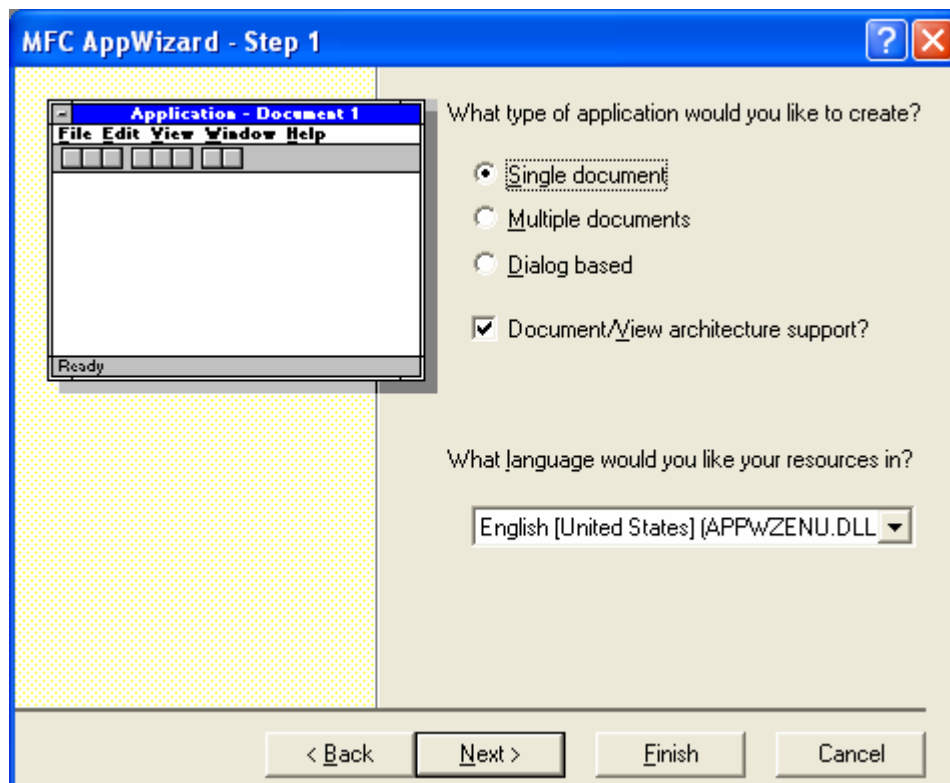


Figure 11: MYEX33A – AppWizard step 1 of 6, SDI project.

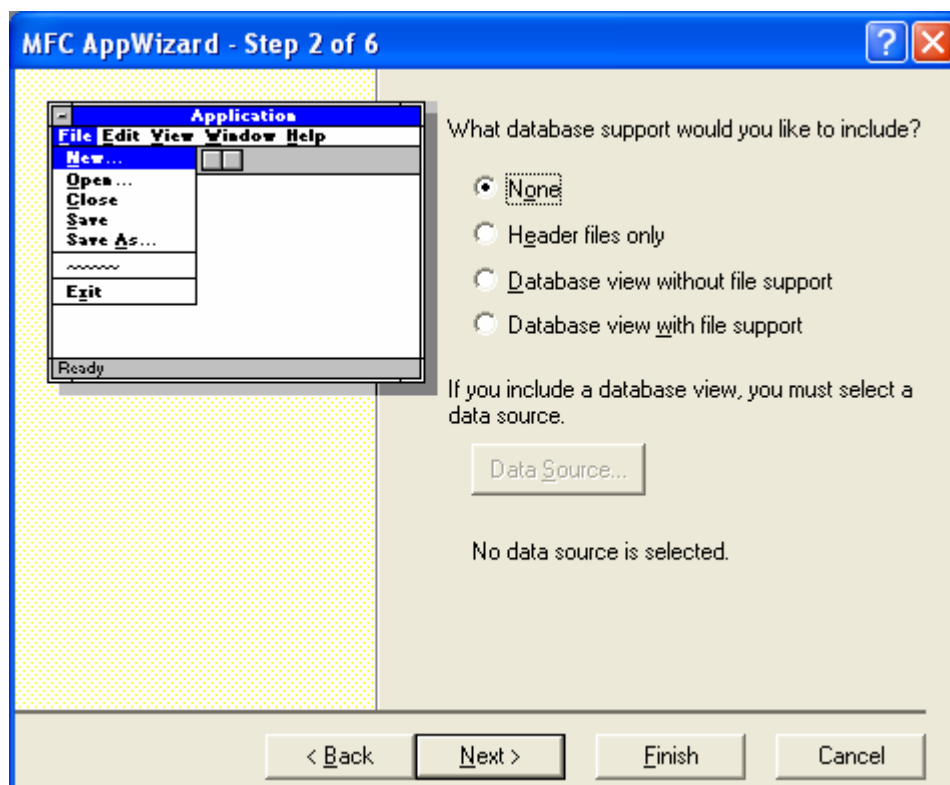


Figure 12: MYEX33A – AppWizard step 2 of 6.

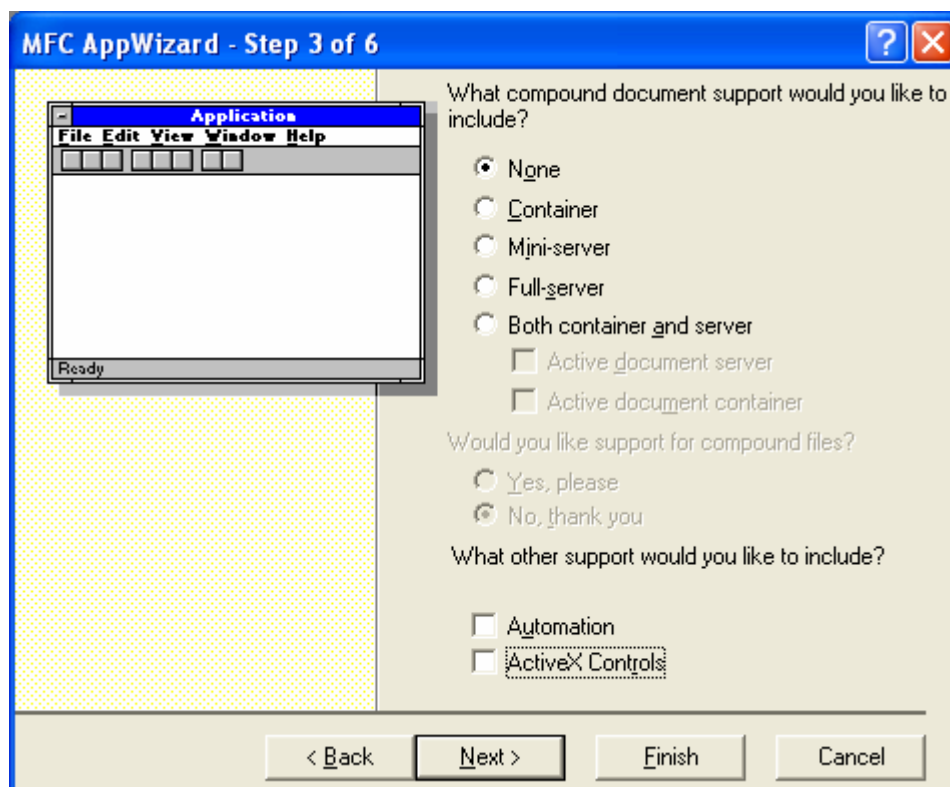


Figure 13: MYEX33A – AppWizard step 3 of 6, deselect **Automation** and **ActiveX Controls**.

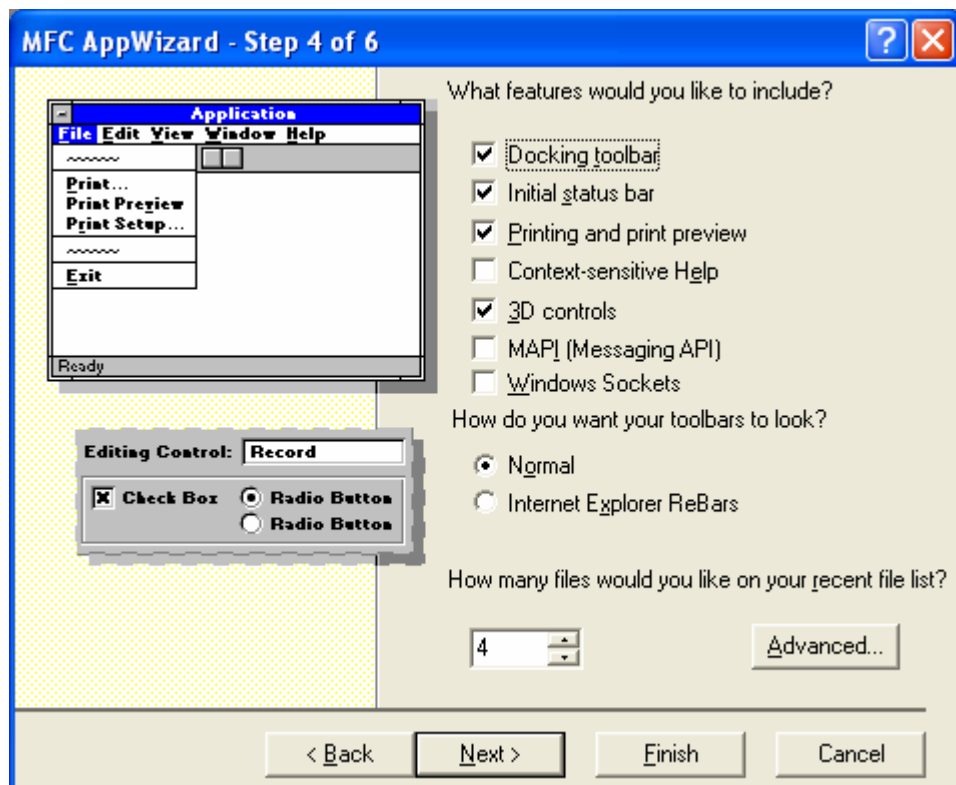


Figure 14: MYEX33A – AppWizard step 4 of 6, we will use a normal dialog to create an address bar.

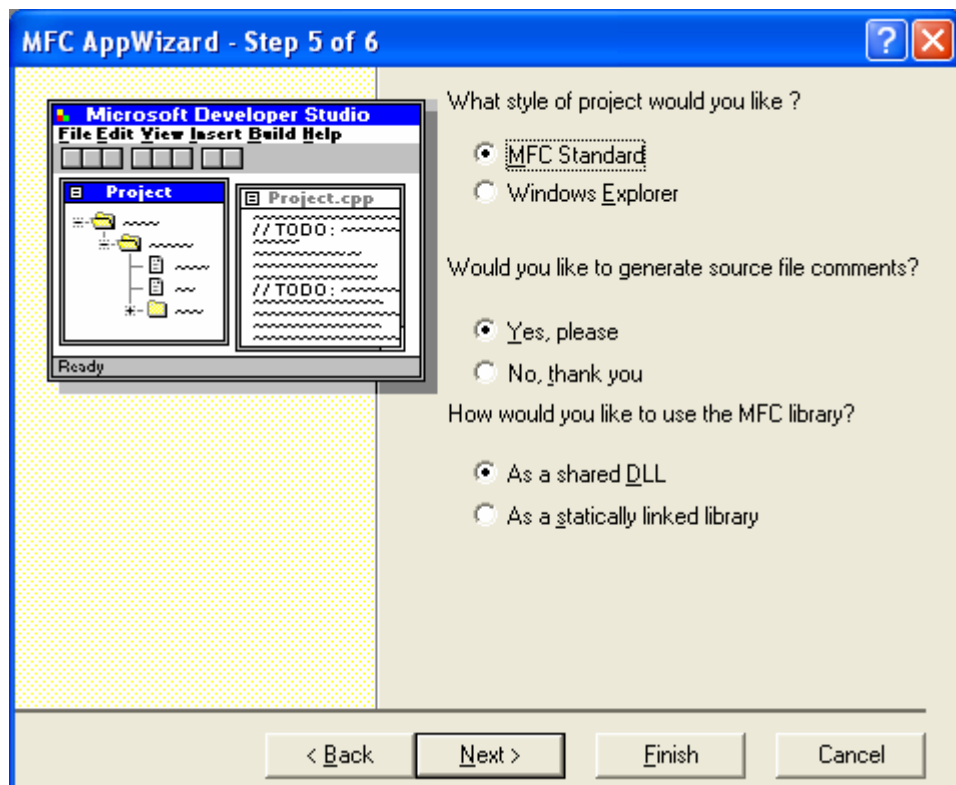


Figure 15: MYEX33A – AppWizard step 5 of 6.

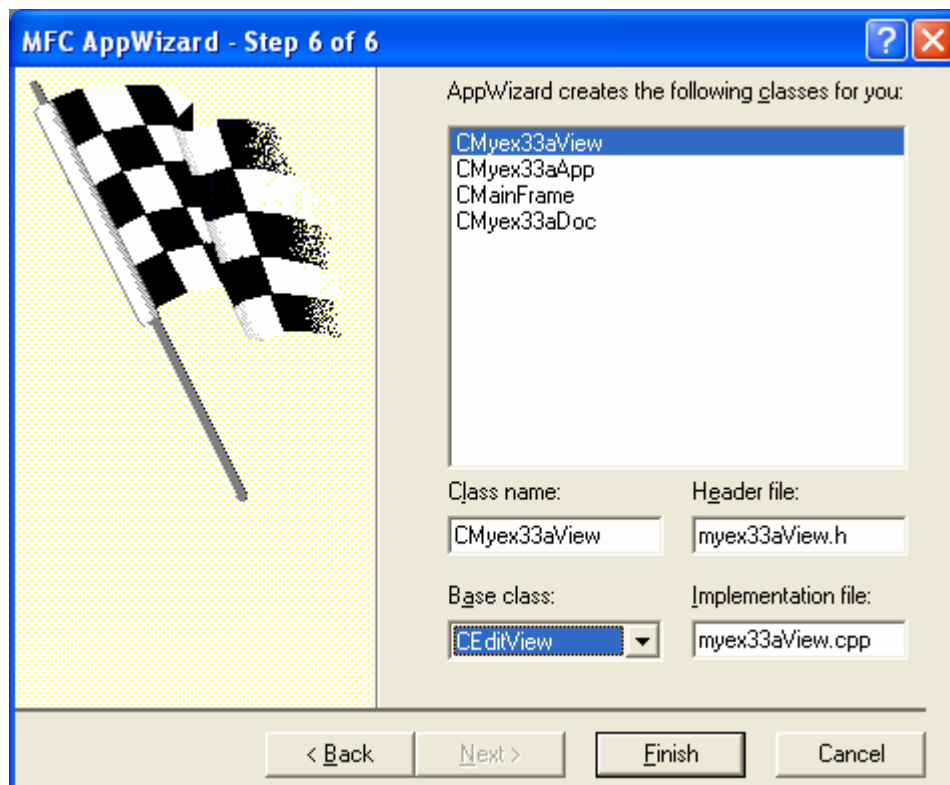


Figure 16: MYEX33A – AppWizard step 6 of 6, selecting CEditView as the view base class.

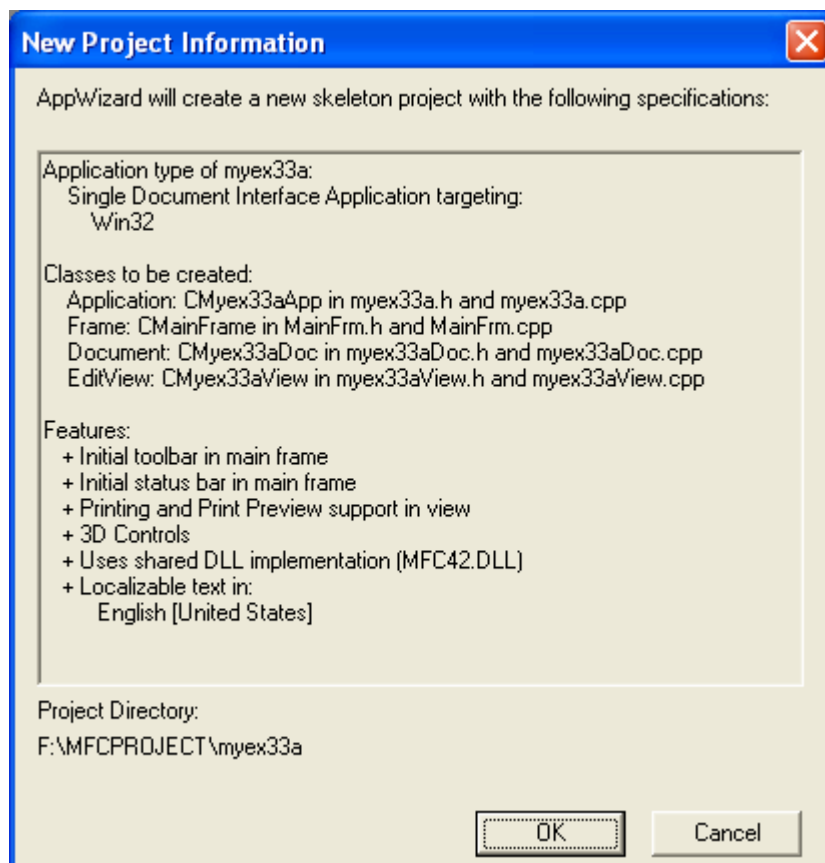


Figure 17: MYEX33A project summary.

First of all let add the project resources. In ResourceView, add new dialog, rename the ID to IDD_DIALOGBAR. In **Styles** tab, deselect the **Title bar** option, select **Child** for **Style** and **None** for **Border**. Add Static text, Edit and Button controls as shown below. Follow the shown steps.

Control/resource	ID	Caption/Text
Dialog	IDD_DIALOGBAR	-
Static text	Default	Address:
Edit control	IDC_URL	-
Button	IDC_REQUEST	URL Request

Table 16.

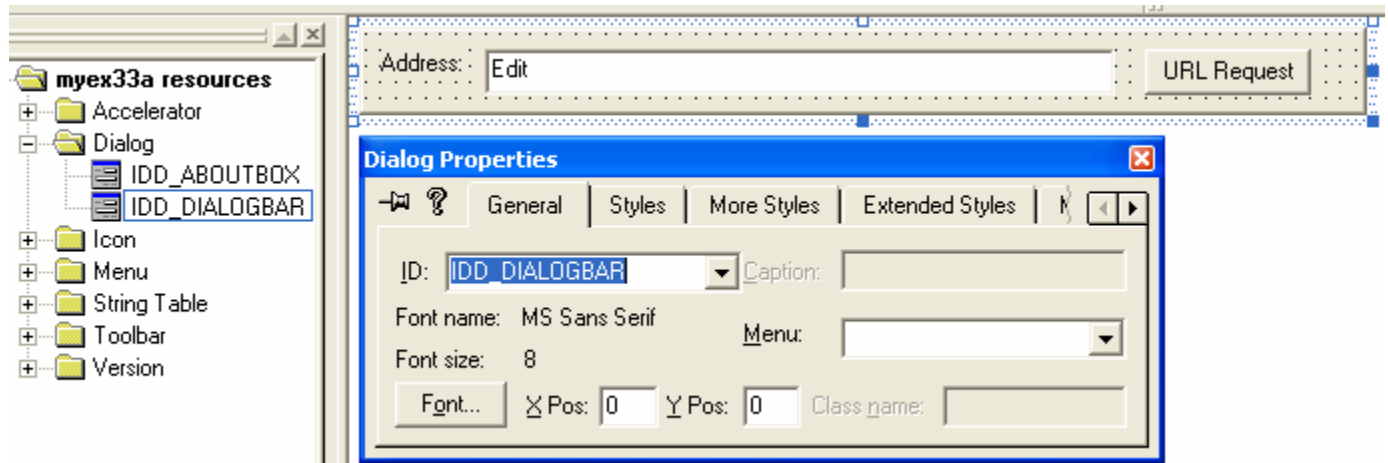


Figure 18: IDD_DIALOGBAR property page.

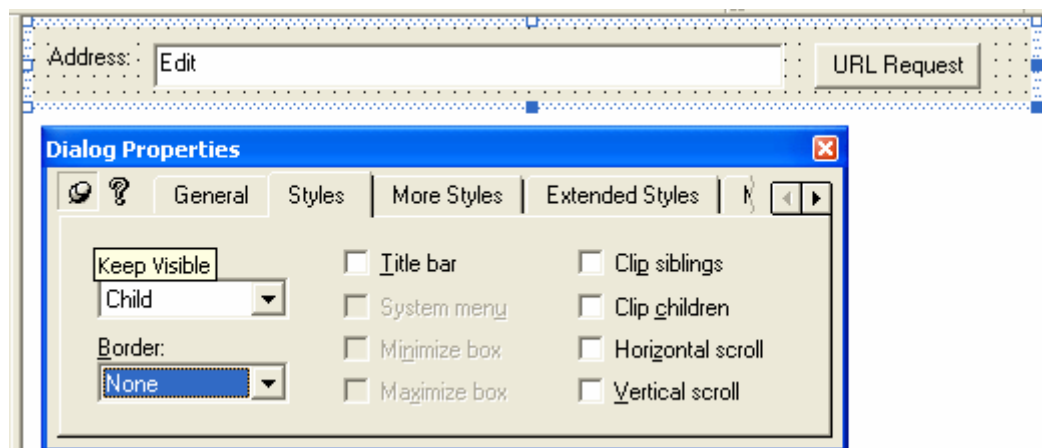


Figure 19: Setting dialog's **Styles** and **Border** properties.

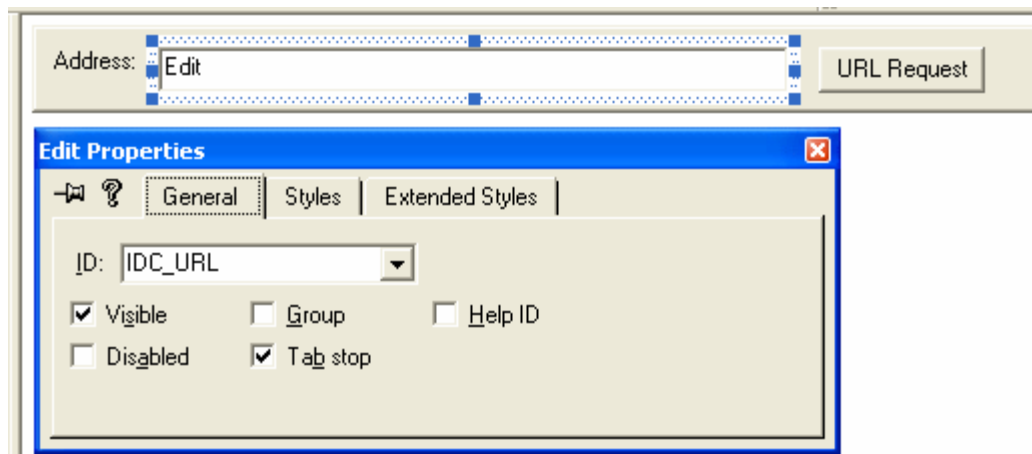


Figure 20: The IDC_URL property.

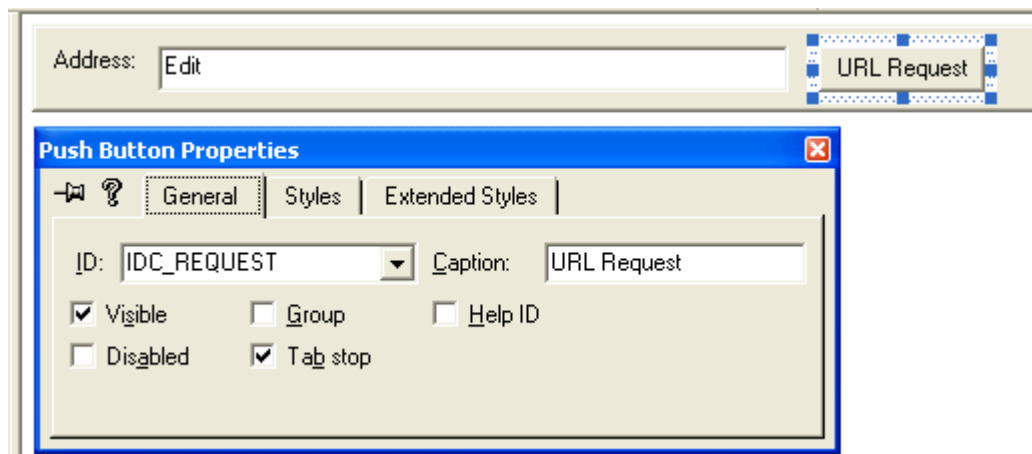


Figure 21: The IDC_REQUEST property.

Add another three dialogs for the property page of the **Configuration** menu. Firstly add IDD_PROPPAGE_ADV, for the advanced page settings and the following controls. Follow the shown steps.

Control/resource	ID	Caption/Text
Dialog	IDD_PROPPAGE_ADV	Advanced
Static text	Default	Server IP Address:
Edit control	IDC_IPSERVER	-
Static text	Default	Client IP Address:
Edit control	IDC_IPCLIENT	-
Static text	Default	Static text - Use these fields ONLY if your computer supports multiple IP addresses. The client IP address applies to WinSock client ONLY.

Table 17.

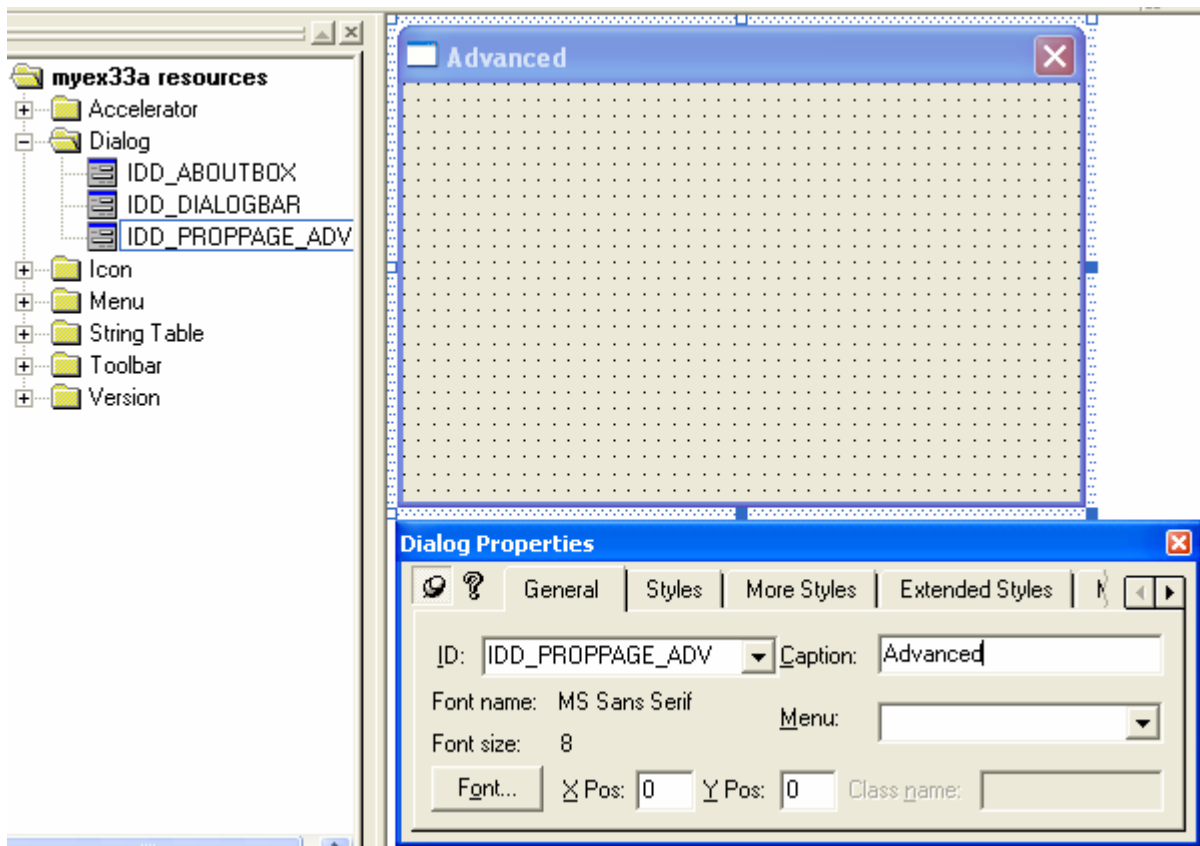


Figure 22: The IDD_PROPPAGE_ADV property.

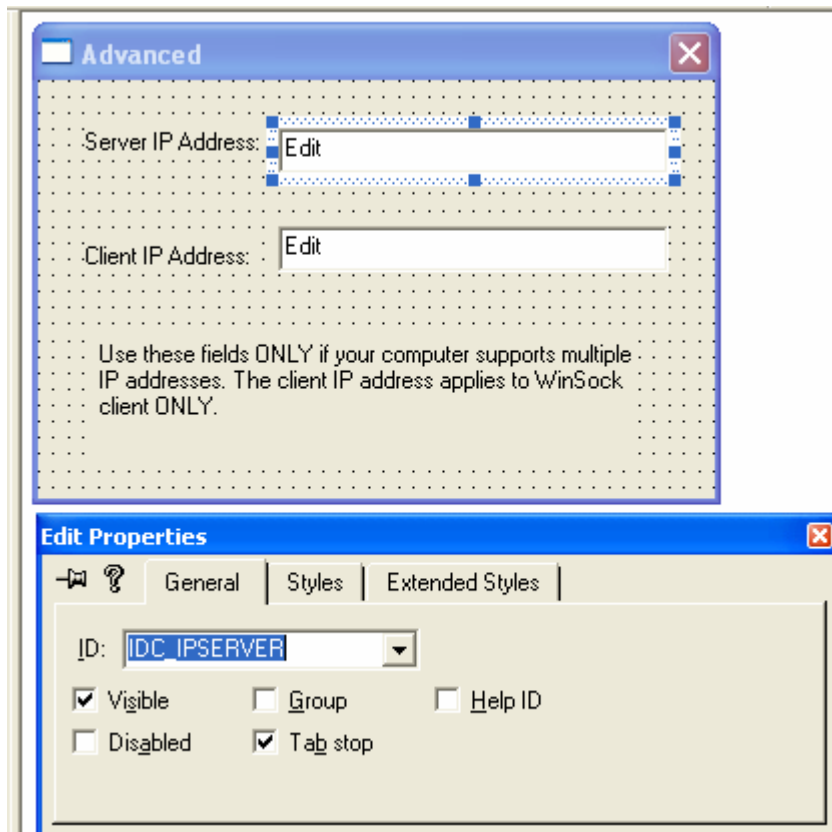


Figure 23: The IDC_IPSERVER property.

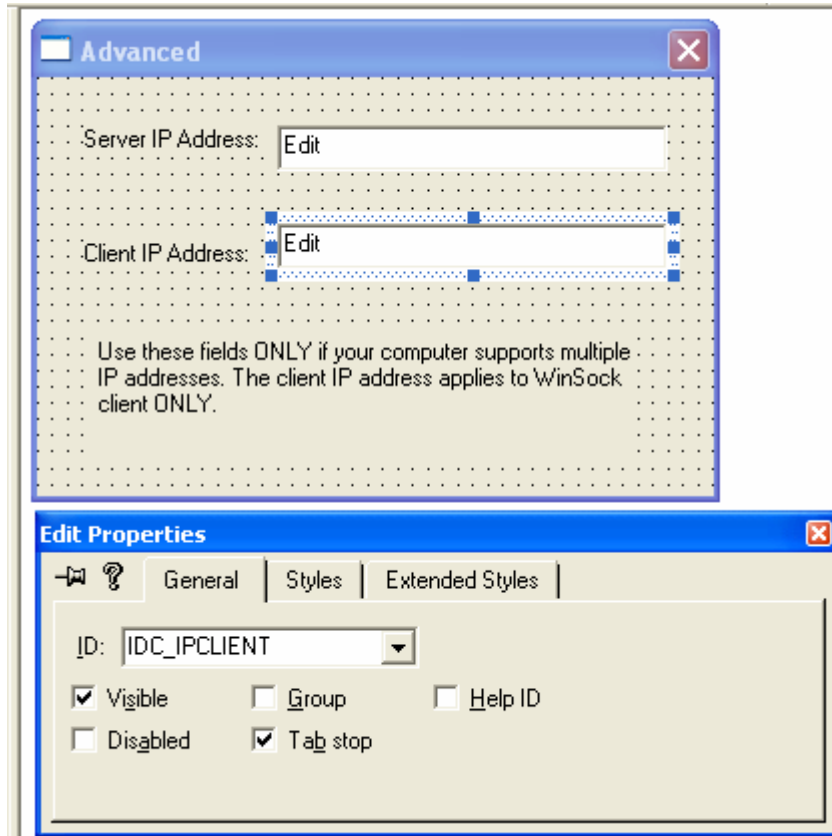


Figure 24: The IDC_IPCLIENT property.

Add new dialog, IDD_PROPPAGE_CLIENT, the client settings property page and the following controls. Follow the shown steps.

Control/resource	ID	Caption/Text
Dialog	IDD_PROPPAGE_CLIENT	Client
Static text	Default	Server Name:
Edit control	IDC_SERVER	-
Static text	Default	Server IP Addr:
Edit control	IDC_IPADDR	-
Static text	Default	Port:
Edit control	IDC_PORT	-
Static text	Default	Server File:
Edit control	IDC_FILE	-
Tick box	IDC_USEPROXY	Use Web Proxy
Static text	Default	Proxy Server:
Edit control	IDC_PROXY	-
Static text	Default	Specify either a name or IP address but not both. Server name and IP address apply to both the WinSock and WinInet clients. The Proxy fields apply only to the WinSock client. WinInet client reads registry for proxy information.

Table 18.

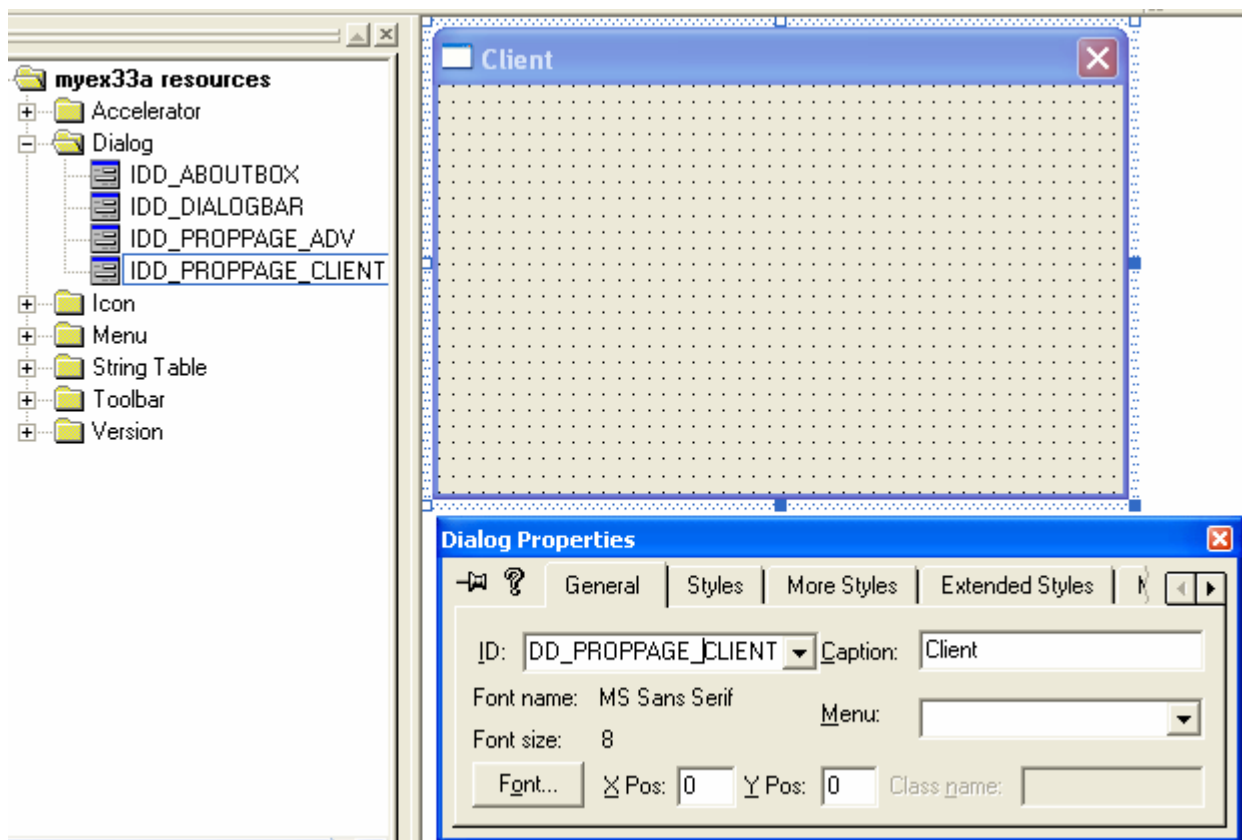


Figure 25: The IDD_PROPPAGE_CLIENT property.

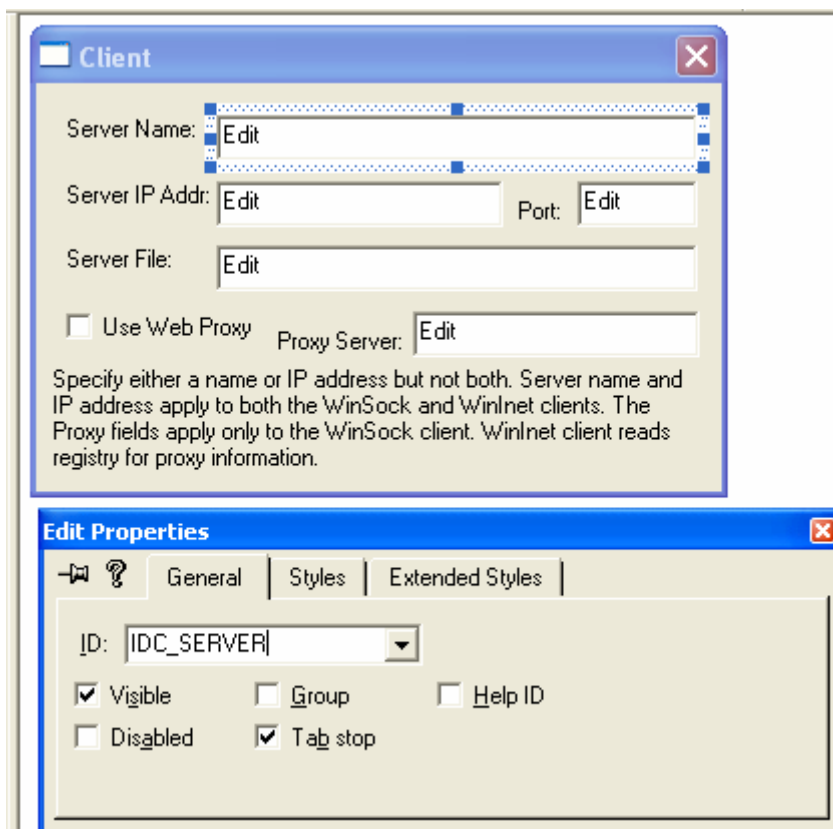


Figure 26: The IDC_SERVER property.

The image displays two overlapping Windows-style dialog boxes. The top dialog, titled 'Client', contains fields for 'Server Name', 'Server IP Addr', 'Port', and 'Server File', each with an 'Edit' button. It also has a checkbox for 'Use Web Proxy' and a 'Proxy Server' field. A paragraph of text at the bottom explains the fields. The bottom dialog, titled 'Edit Properties', has tabs for 'General', 'Styles', and 'Extended Styles'. The 'General' tab is active, showing a dropdown for 'ID' set to 'IDC_IPADDR' and several checkboxes: 'Visible' (checked), 'Disabled' (unchecked), 'Group' (unchecked), 'Tab stop' (checked), and 'Help ID' (unchecked).

Client

Server Name: Edit

Server IP Addr: Edit Port: Edit

Server File: Edit

☐ Use Web Proxy Proxy Server: Edit

Specify either a name or IP address but not both. Server name and IP address apply to both the WinSock and Wininet clients. The Proxy fields apply only to the WinSock client. Wininet client reads registry for proxy information.

Edit Properties

General Styles Extended Styles

ID: IDC_IPADDR

☒ Visible ☐ Group ☐ Help ID

☐ Disabled ☒ Tab stop

Figure 27: The IDC_IPADDR property.

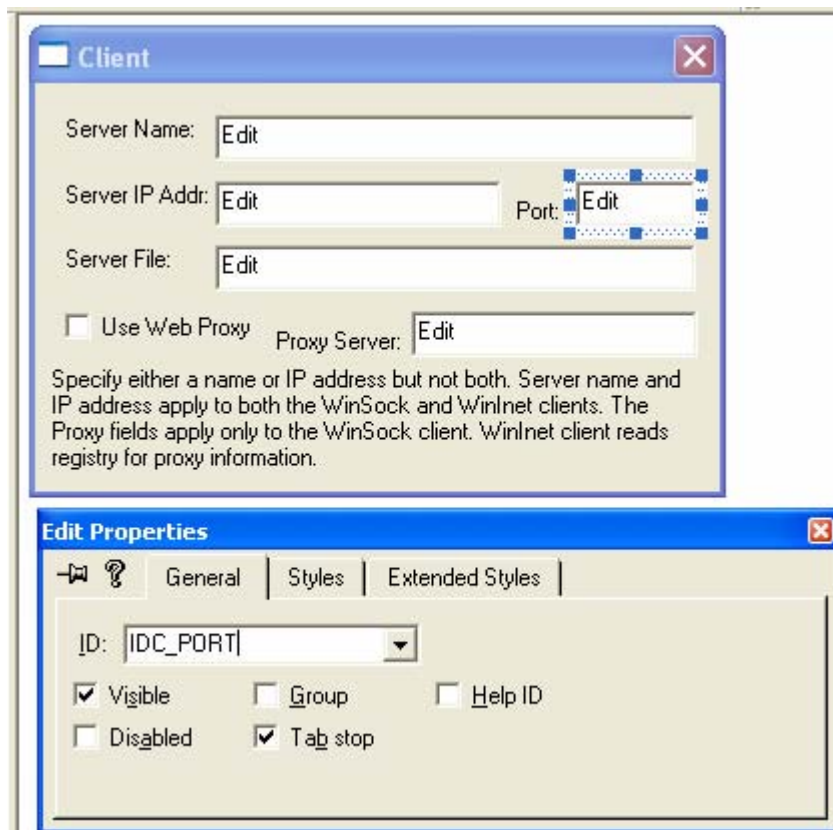


Figure 28: The IDC_PORT property.

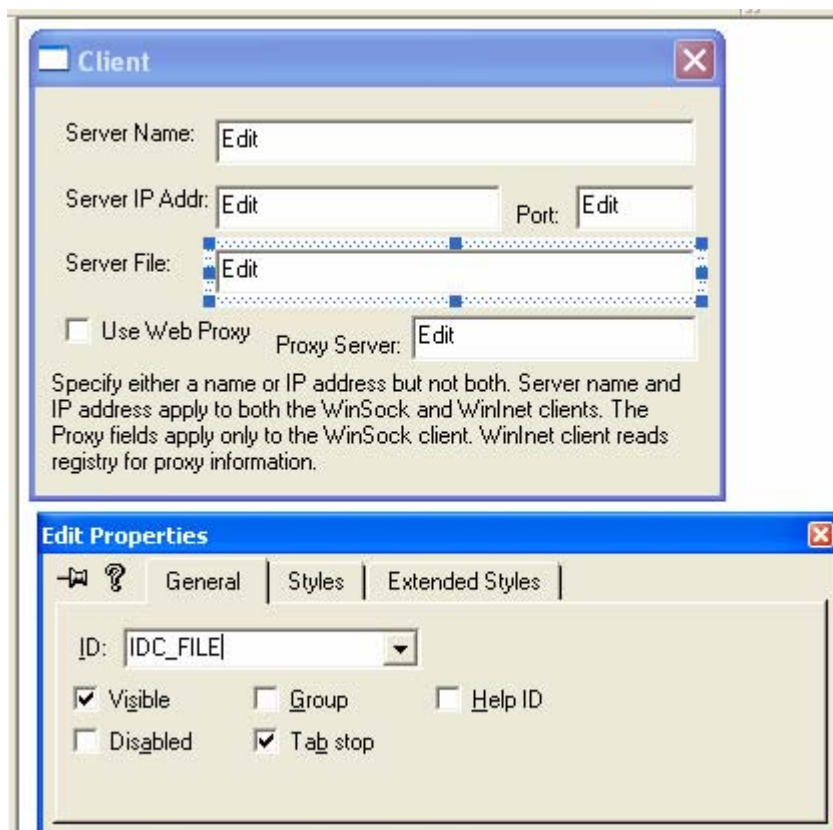


Figure 29: The IDC_FILE property.

The image shows two overlapping Windows-style dialog boxes. The top dialog, titled 'Client', has a blue header bar with a close button. It contains several text input fields: 'Server Name: Edit', 'Server IP Addr: Edit' followed by 'Port: Edit', and 'Server File: Edit'. Below these is a checkbox labeled 'Use Web Proxy' which is currently unchecked, followed by a 'Proxy Server: Edit' field. A paragraph of text at the bottom explains that server name and IP address apply to both WinSock and WinInet clients, while proxy fields apply only to WinSock. The bottom dialog, titled 'Check Box Properties', has a blue header bar with a close button and three tabs: 'General' (selected), 'Styles', and 'Extended Styles'. It features a dropdown menu for 'ID:' set to 'IDC_USEPROXY' and a 'Caption:' field with the text 'Use Web Proxy'. There are six checkboxes: 'Visible' (checked), 'Group' (unchecked), 'Help ID' (unchecked), 'Disabled' (unchecked), 'Tab stop' (checked), and 'Help ID' (unchecked).

Client

Server Name: Edit

Server IP Addr: Edit Port: Edit

Server File: Edit

☐ Use Web Proxy Proxy Server: Edit

Specify either a name or IP address but not both. Server name and IP address apply to both the WinSock and WinInet clients. The Proxy fields apply only to the WinSock client. WinInet client reads registry for proxy information.

Check Box Properties

General Styles Extended Styles

ID: IDC_USEPROXY Caption: Use Web Proxy

☒ Visible ☐ Group ☐ Help ID

☐ Disabled ☒ Tab stop

Figure 30: The IDC_USEPROXY property.

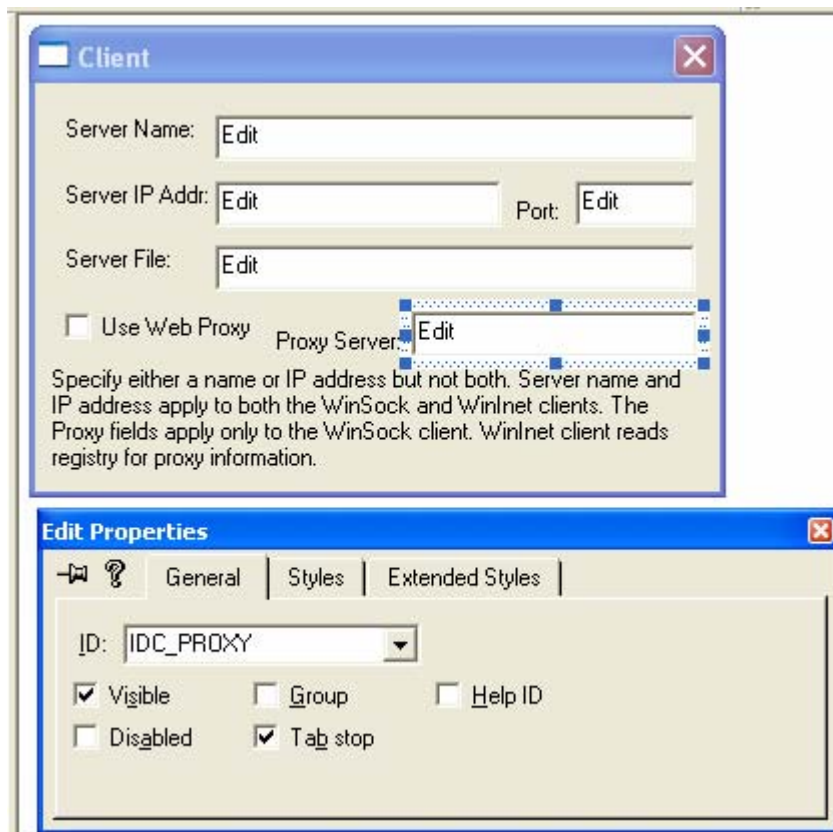


Figure 31: The IDC_PROXY property.

Add new dialog and the following controls for server settings property page. Follow the shown steps.

Control/resource	ID	Caption/Text
Dialog	IDD_PROPPAGE_SERVER	Server
Static text	Default	Home Directory:
Edit control	IDC_DIRECT	-
Static text	Default	Default File:
Edit control	IDC_DEFAULT	-
Static text	Default	Listening Port:
Edit control	IDC_PORTSERVER	-

Table 19.

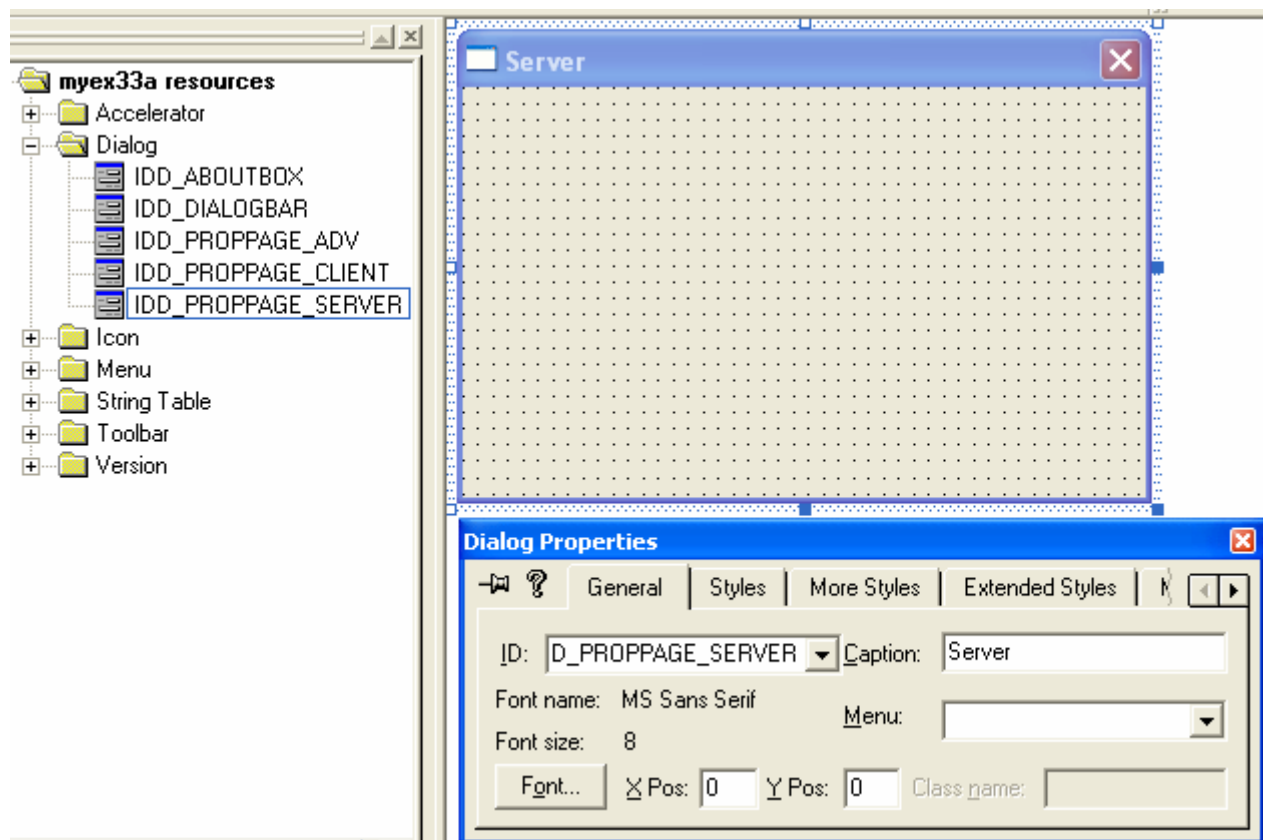


Figure 32: The IDD_PROPPAGE_SERVER property.

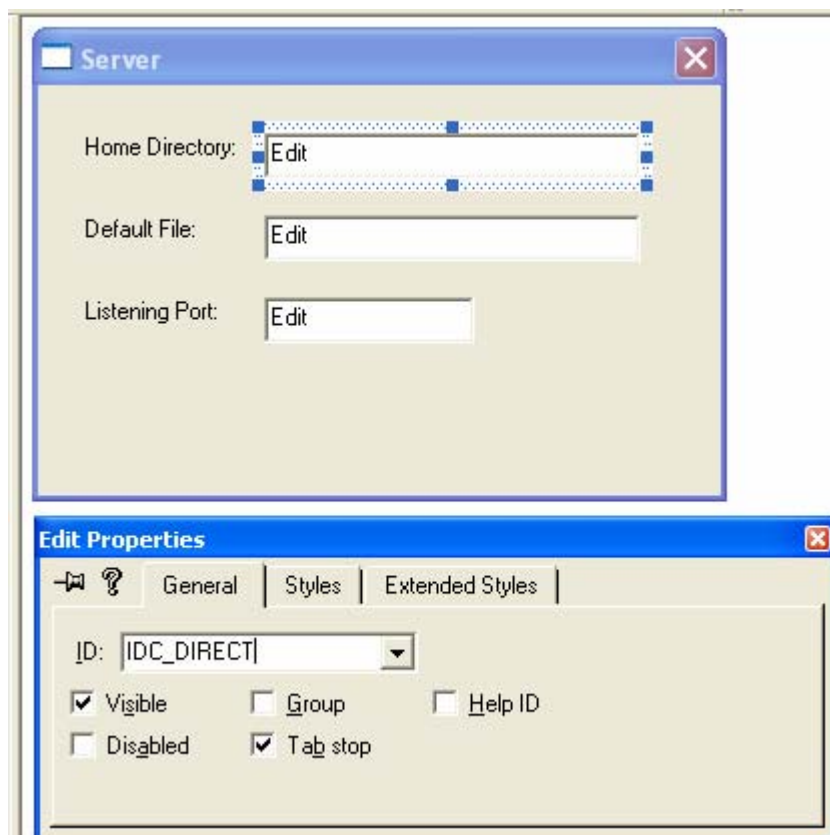


Figure 33: The IDC_DIRECT property.

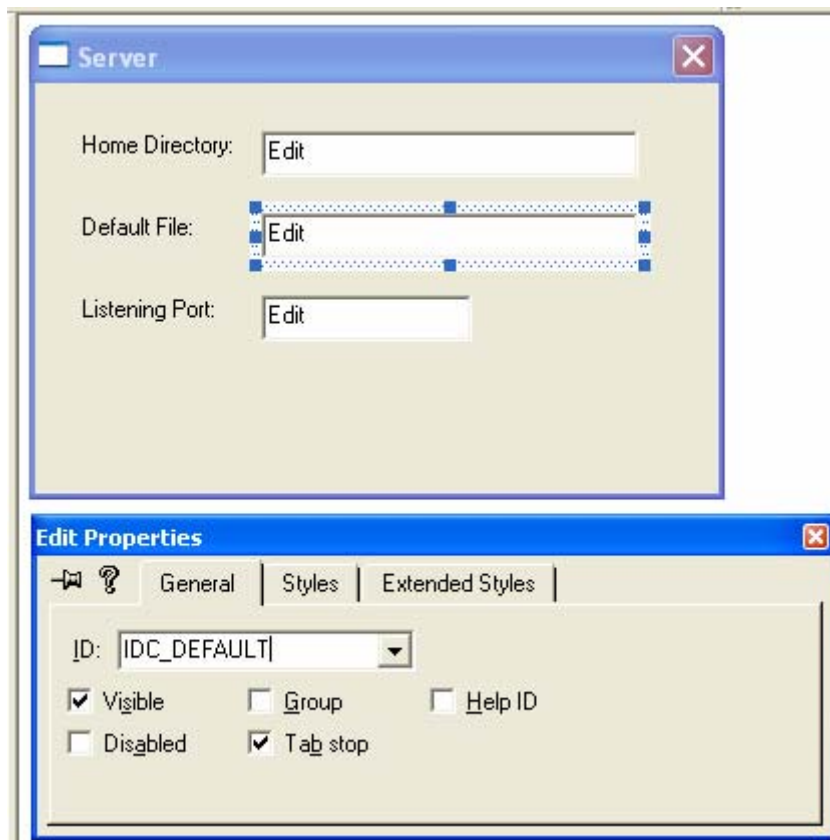


Figure 34: The IDC_DEFAULT property.

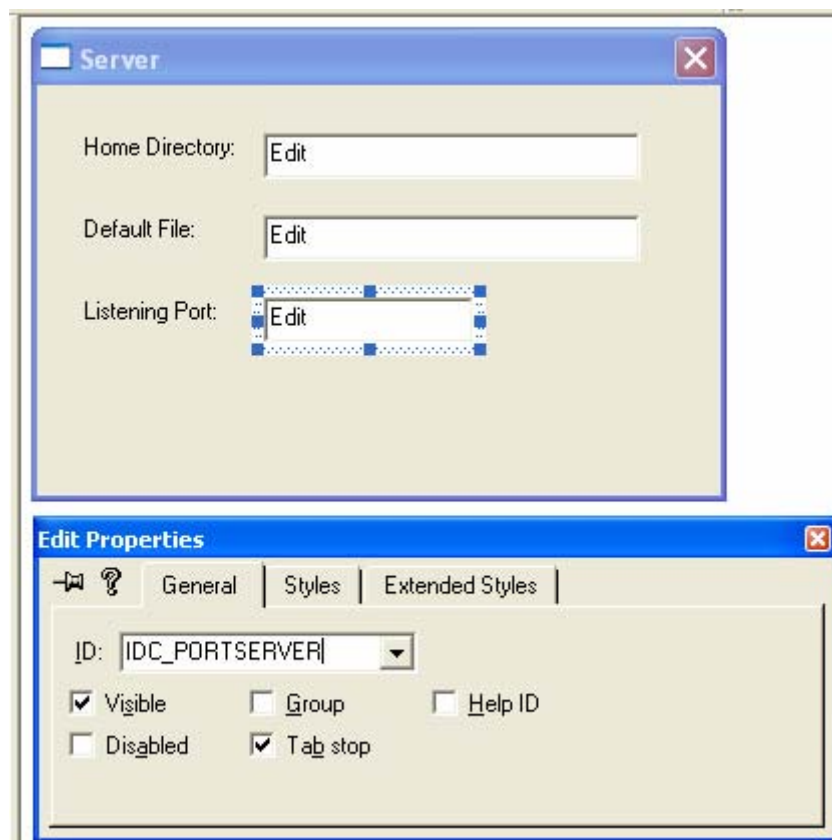


Figure 35: The IDC_PORTSERVER property.

Add the following context menu. Follow the shown steps.

ID	Caption
IDR_CONTEXT_MENU	X
ID_EDIT_CLEAR_ALL	Clear All

Table 20.

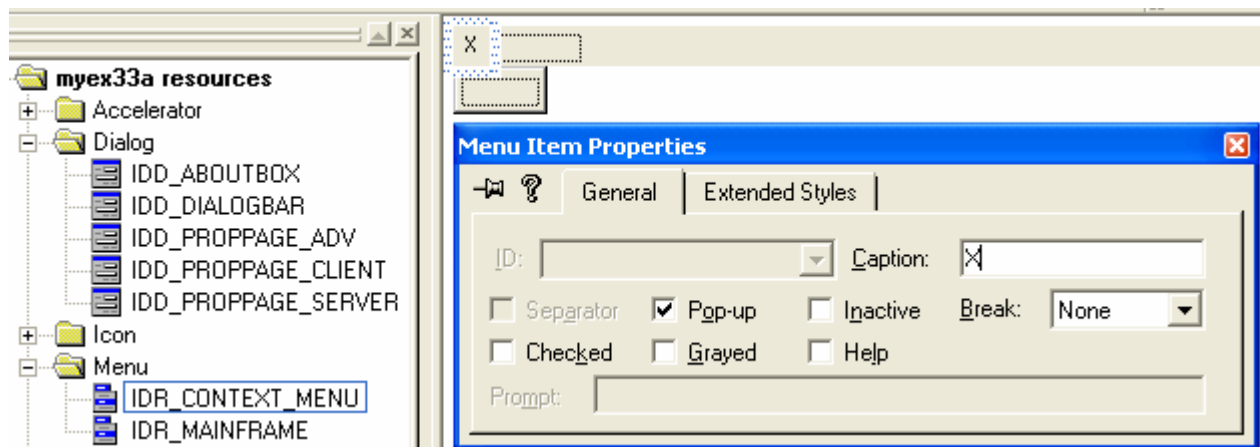


Figure 36: The X main menu property page.

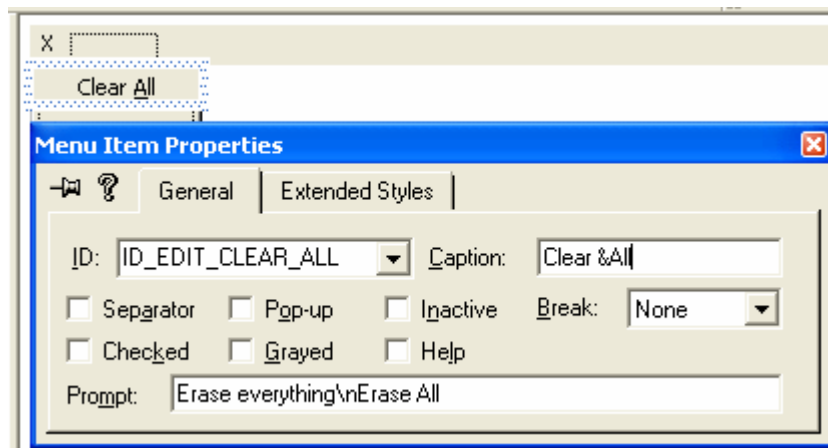


Figure 37: The ID_EDIT_CLEAR_ALL property.

Add the following menu and their items. Follow the shown steps.

ID	Caption	Prompt
-	Internet	-
ID_INTERNET_START_SERVER	Start Server	Start the server thread
ID_INTERNET_STOP_SERVER	Stop Server	Stop the server thread
ID_INTERNET_REQUEST SOCK	Request (Winsock)	Client request using Winsock functions
ID_INTERNET_REQUEST_INET	Request (WinInet)	Client request using WinInet functions
ID_INTERNET_CONFIGURATION	Configuration	Set home directory, server etc. for client

Table 21.

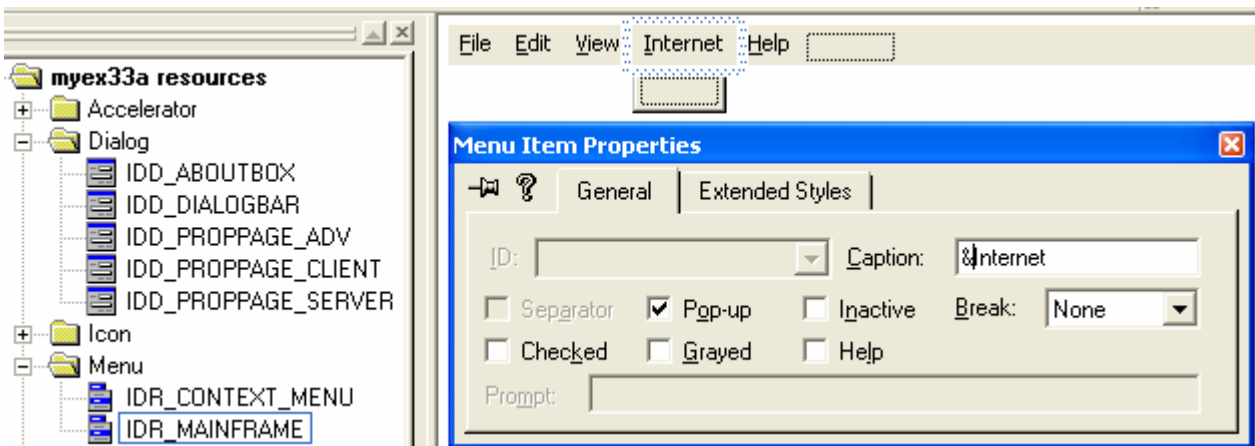


Figure 38: The **Internet** menu property.

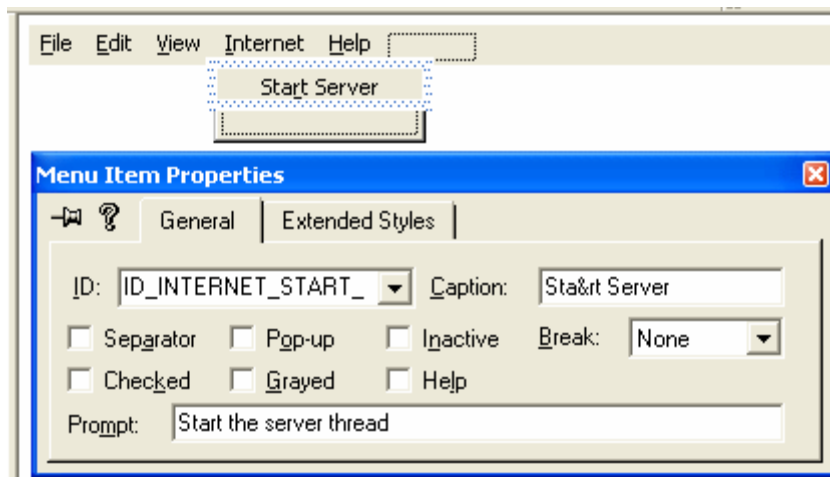


Figure 39: The **Start Server** menu property.

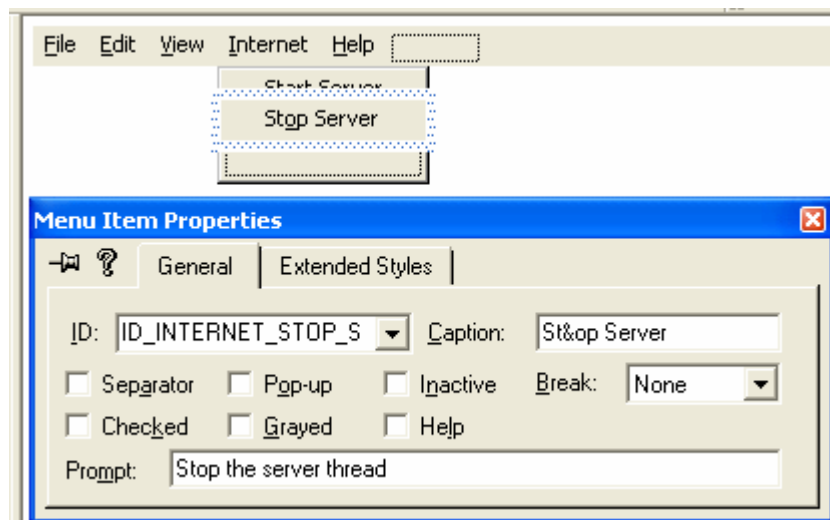


Figure 40: The **Stop Server** menu property.

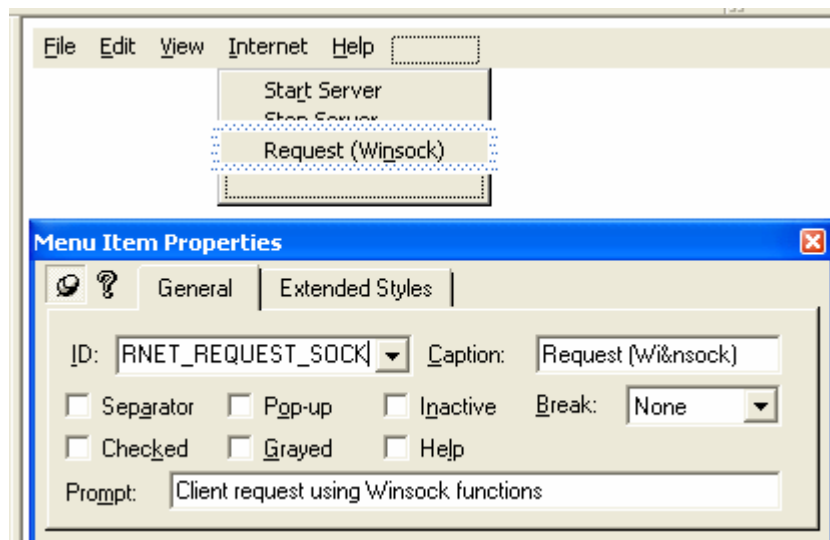


Figure 41: The **Request (Winsock)** menu property.

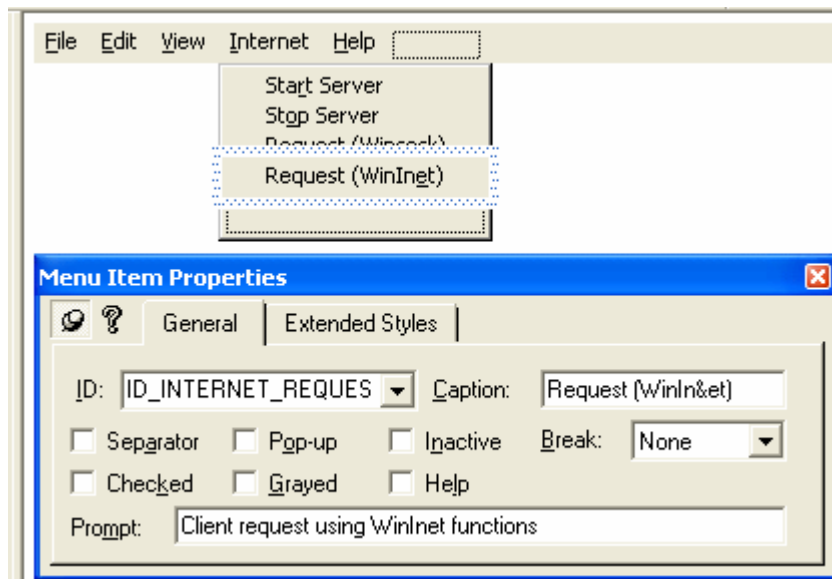


Figure 42: The **Request (WinInet)** menu property.

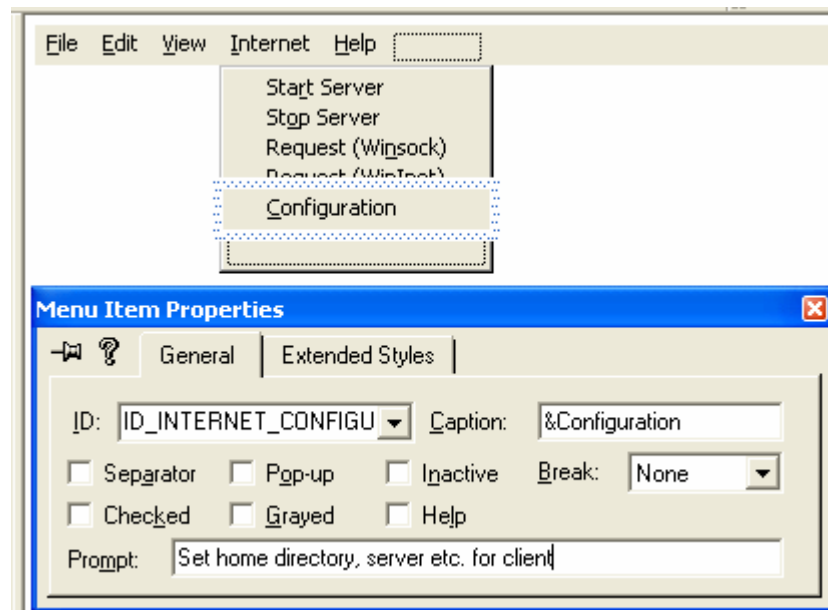


Figure 43: The **Configuration** menu property.

In ResourceView, select the IDD_PROPPAGE_ADV dialog, then, launch ClassWizard. The **Adding a Class** prompt dialog will be displayed. Select **Create a new class** radio button and click **OK**. Add the following classes and all are using the same **Sheetconfig.h** and **Sheetconfig.cpp** header and source files respectively.

ID	Class name	Base class
IDD_PROPPAGE_ADV	CPageAdv	CPropertyPage
IDD_PROPPAGE_SERVER	CPageServer	CPropertyPage
IDD_PROPPAGE_CLIENT	CPageClient	CPropertyPage
-	CSheetConfig	CPropertySheet

Table 22.

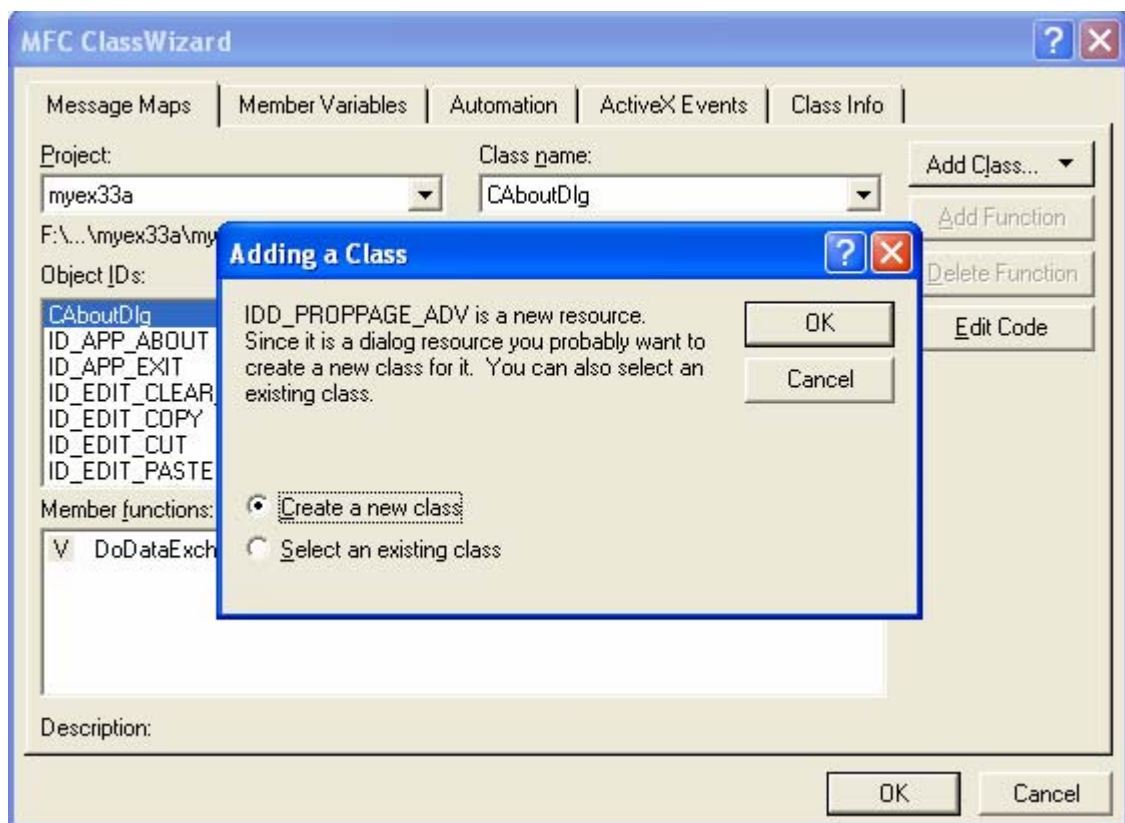


Figure 44: Adding new class dialog prompt.

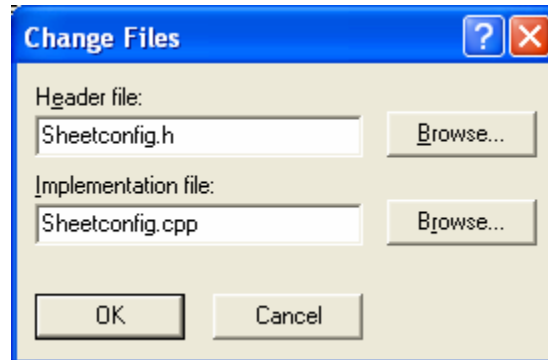


Figure 45: Entering the header and source file names for the classes.

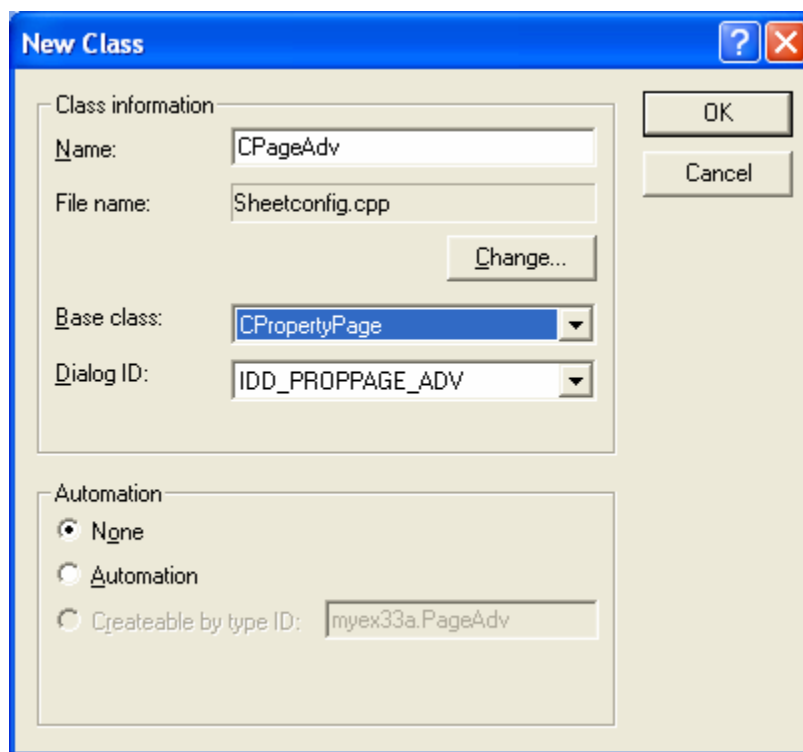


Figure 46: The CPageAdv class information.

Add other classes in the same **Sheetconfig.h** and **Sheetconfig.cpp** files.

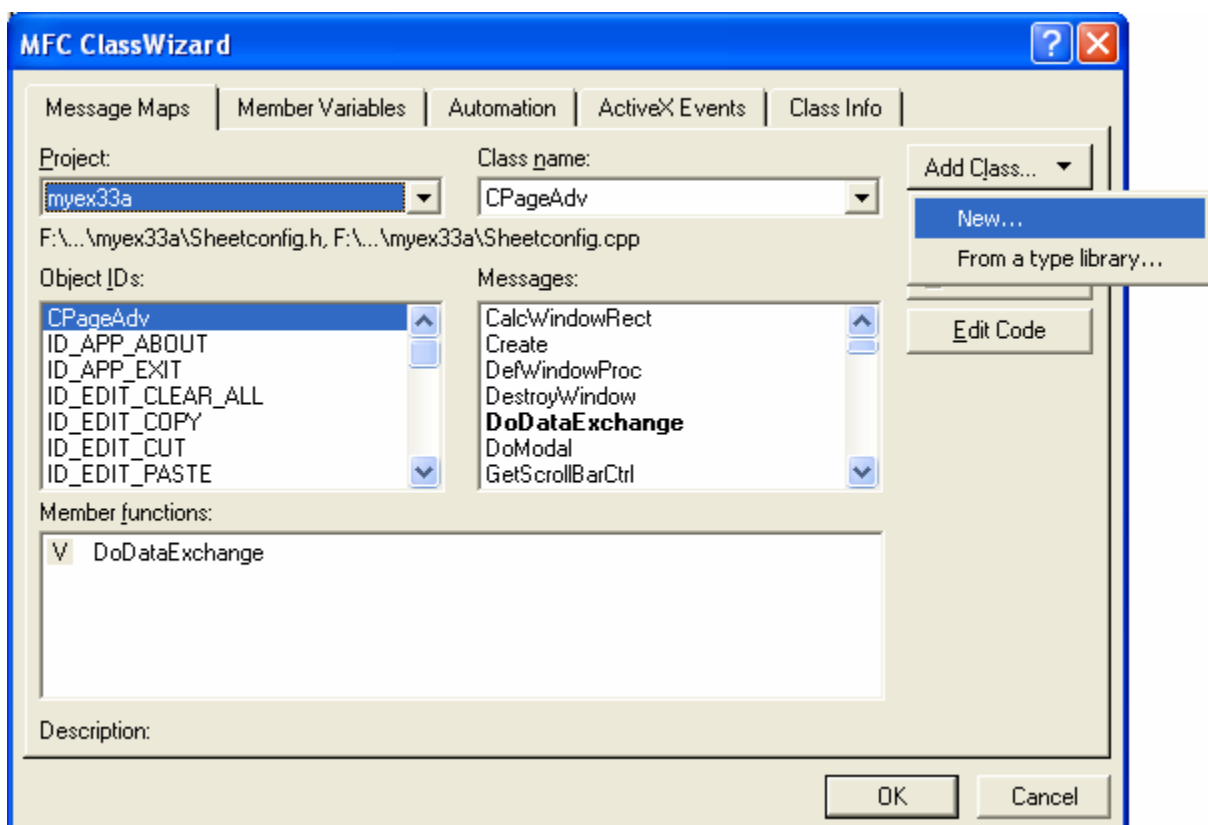
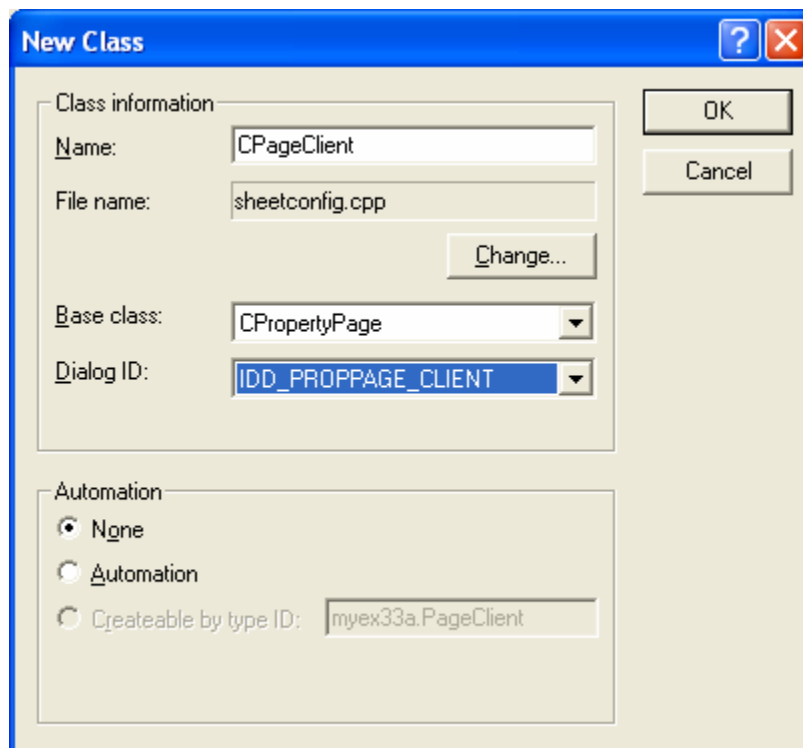


Figure 47: Adding new class through ClassWizard.

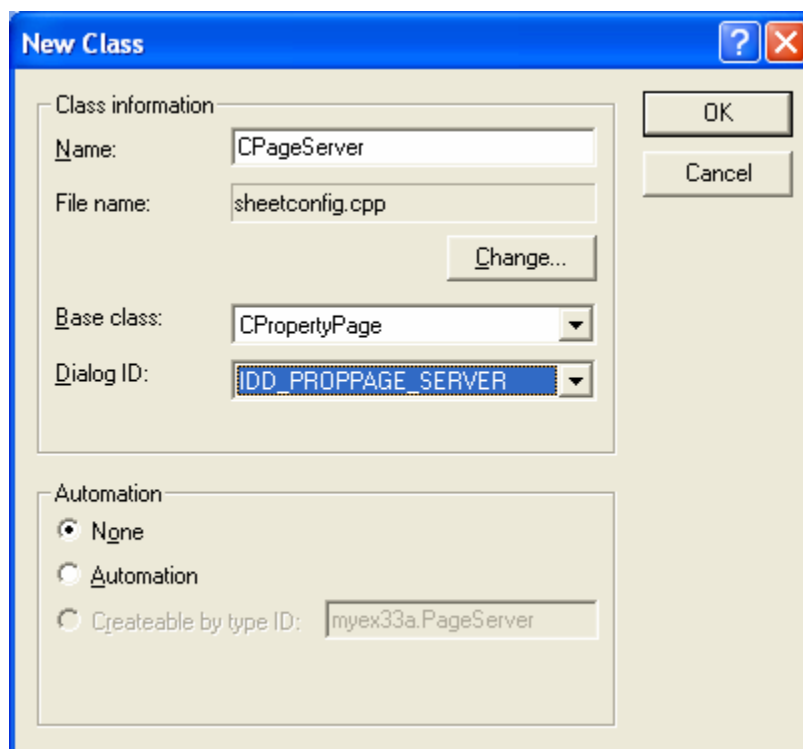


The 'New Class' dialog box is shown with the following settings:

- Class information:**
 - Name: CPageClient
 - File name: sheetconfig.cpp
 - Base class: CPropertyPage
 - Dialog ID: IDD_PROPPAGE_CLIENT
- Automation:**
 - ☒ None
 - ☐ Automation
 - ☐ Createable by type ID: myex33a.PageClient

Buttons: OK, Cancel, Change...

Figure 48: The CPageClient class information.



The 'New Class' dialog box is shown with the following settings:

- Class information:**
 - Name: CPageServer
 - File name: sheetconfig.cpp
 - Base class: CPropertyPage
 - Dialog ID: IDD_PROPPAGE_SERVER
- Automation:**
 - ☒ None
 - ☐ Automation
 - ☐ Createable by type ID: myex33a.PageServer

Buttons: OK, Cancel, Change...

Figure 49: The CPageServer class information.

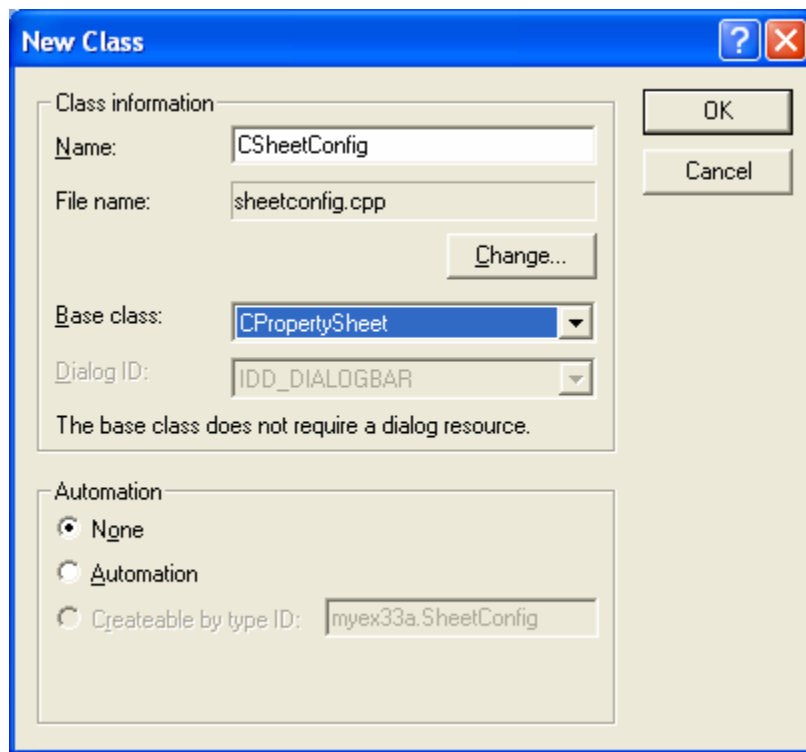


Figure 50: The CSheetConfig class information.

Add member variable to CPageAdv class. In the ClassWizard, click the **Member Variables** page and click the **Add Variable** button. Add the following variables. Follow the shown steps.

ID	Variable name	Type
IDC_IPCLIENT	m_strIPClient	CString
IDC_IPSERVER	m_strIPServer	CString

Table 23.

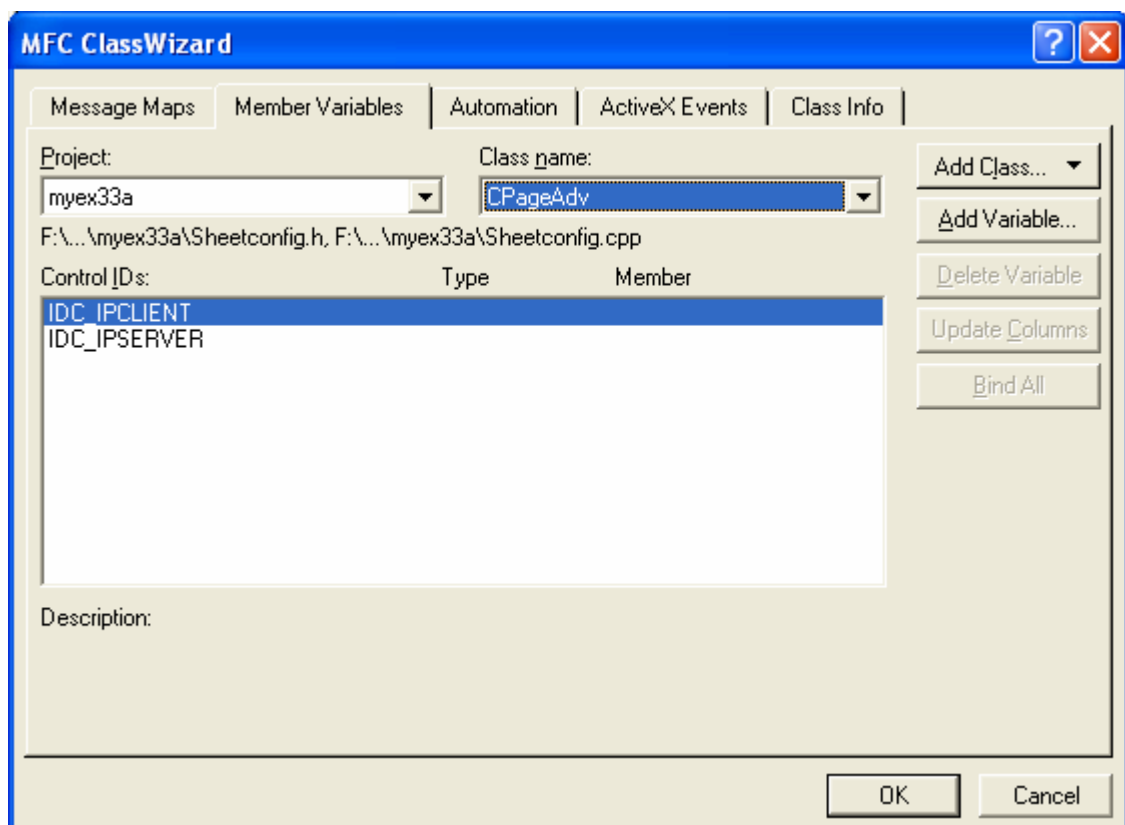


Figure 51: Adding member variables.

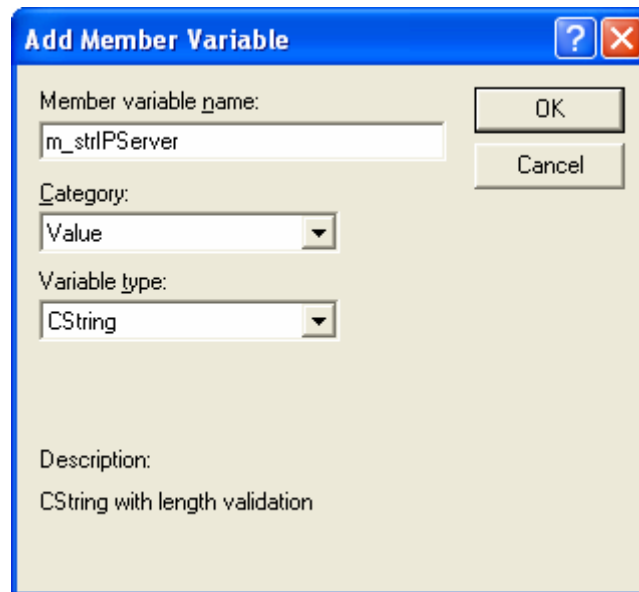


Figure 52: Adding m_strIPServer variable.

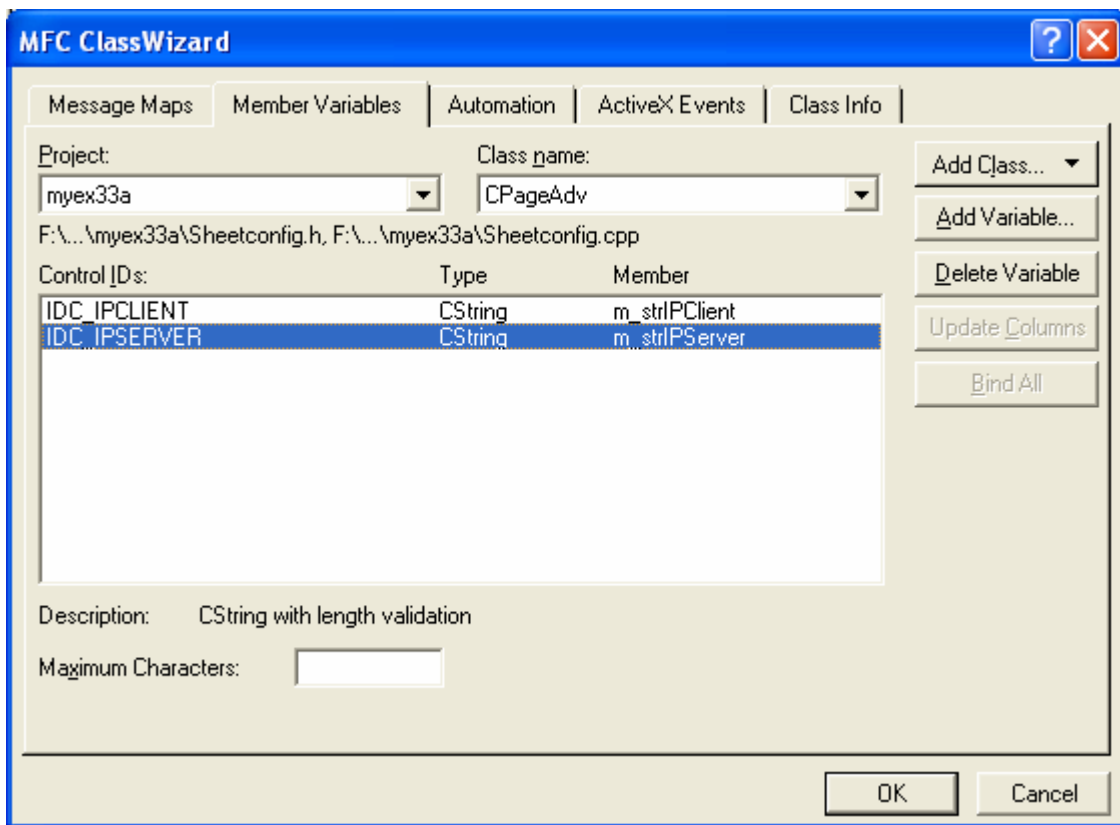


Figure 53: The added variables.

By following the previous steps, add member variable to CPageClient class. Don't forget to save the previous member variables addition by clicking the **Yes** button as shown below.

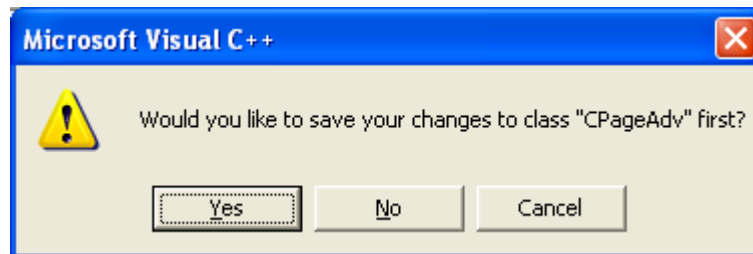


Figure 54: Save changes dialog prompt.

Add the following variables to CPageClient class. Follow the shown steps.

ID	Variable name	Type
IDC_FILE	m_strFile	CString
IDC_IPADDR	m_strServerIP	CString
IDC_PORT	m_nPort	UINT
IDC_PROXY	m_strProxy	CString
IDC_SERVER	m_strServerName	CString
IDC_USEPROXY	m_bUseProxy	BOOL

Table 24.

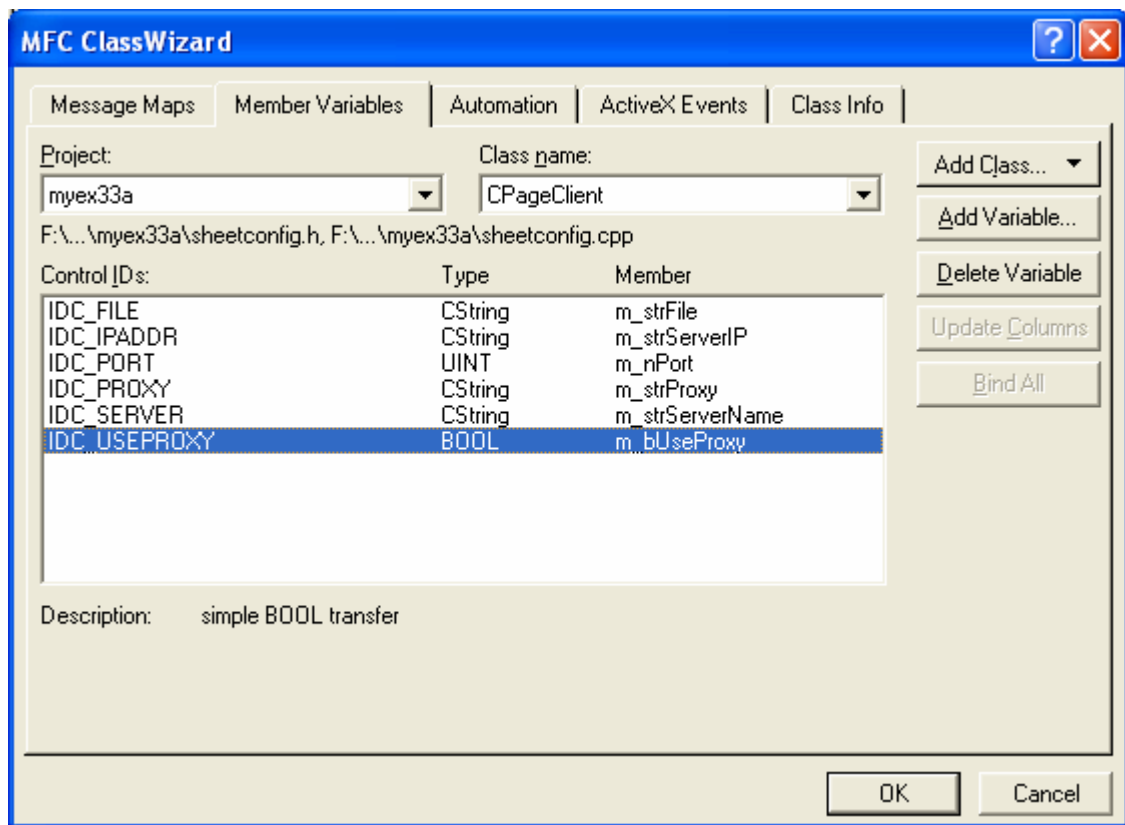


Figure 55: The added CPageClient member variables.

Select CPageServer class in the **Class Name** field, add the following member variables.

ID	Variable name	Type
IDC_DEFAULT	m_strDefault	CString
IDC_DIRECT	m_strDirect	CString
IDC_PORTSERVER	m_nPortServer	UINT

Table 25.

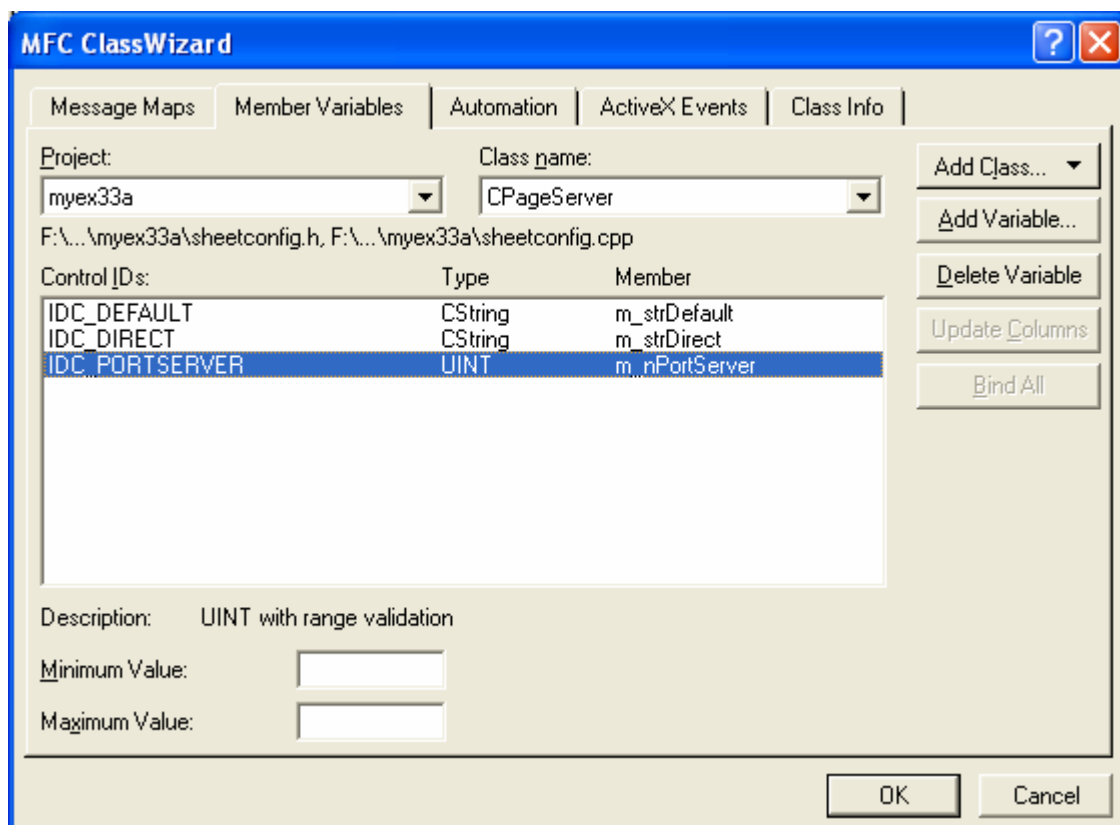


Figure 56: The added CPageServer member variables.

Using ClassView, add the following public member variables to CSheetConfig class.

```
public:
    CPageAdv      m_pageAdv;
    CPageClient   m_pageClient;
    CPageServer   m_pageServer;
```

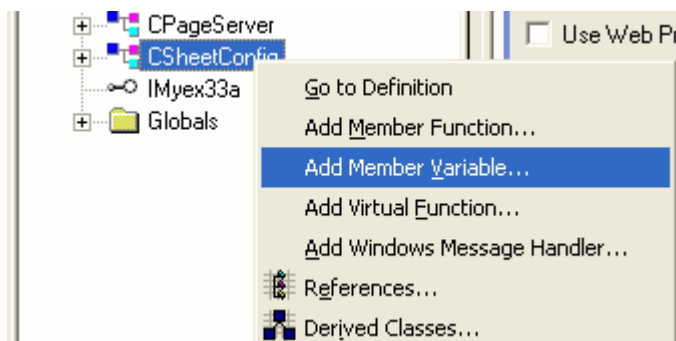


Figure 57: Adding member variables through ClassView.

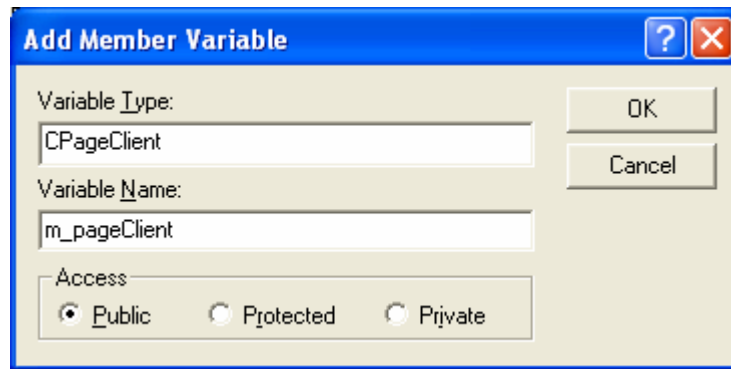


Figure 58: Adding m_pageClient variable.

```
// Implementation
public:
    CPageServer m_pageServer;
    CPageClient m_pageClient;
    CPageAdv m_pageAdv;
    virtual ~CSheetConfig();

    // Generated message map functions
```

Listing 2.

Using ClassView add the following generic class.

Class name	Base class
CSockAddr	sockaddr_in
CBlockingSocket	CObject
CBlockingSocketException	CException
CHttpBlockingSocket	CBlockingSocket

Table 26.

Don't forget to use the same header and source **Blocksock.h** and **Blocksock.cpp** files respectively for all the generic classes. Follow the shown steps.

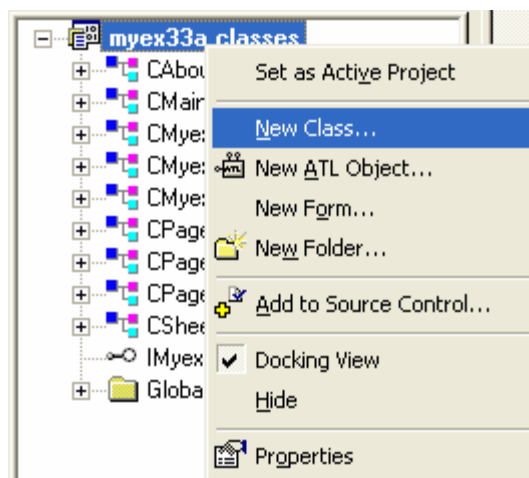


Figure 59: Adding new class through ClassView.

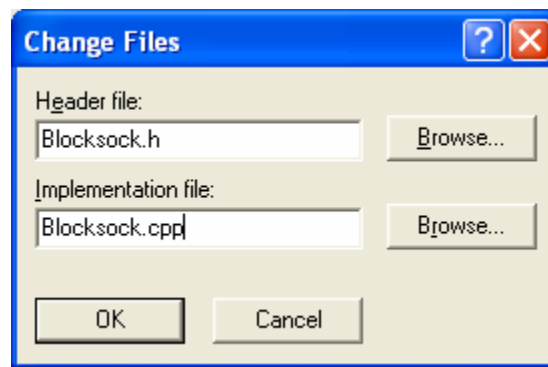


Figure 60: Modifying the header and source file names.

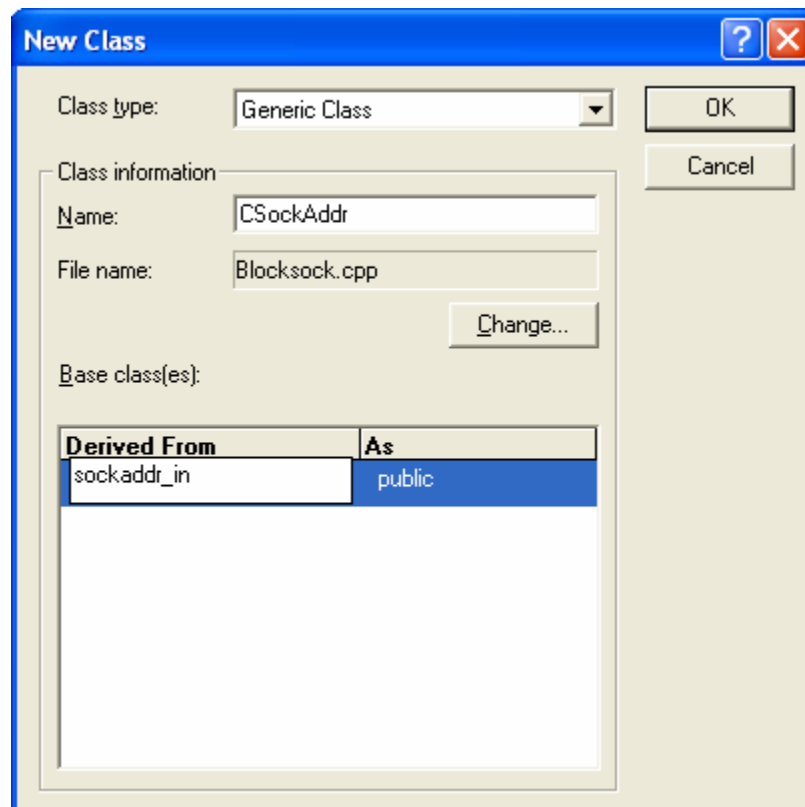


Figure 61: The CSockAddr class information.

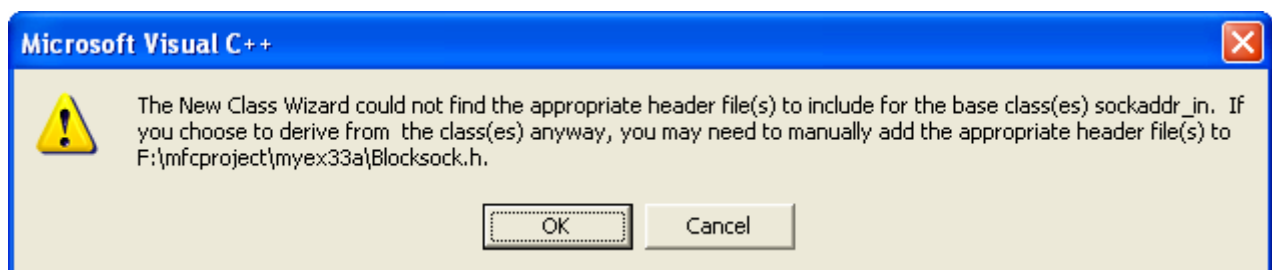


Figure 62: The dialog prompt for non-existence file. Just click OK.

New Class [?] [X]

Class type: [v]

OK Cancel

Class information

Name:

File name:

Base class(es):

Derived From	As
<input type="text" value="CObject"/>	public

Figure 63: The CBlockingSocket class information.

New Class [?] [X]

Class type: [v]

OK Cancel

Class information

Name:

File name:

Base class(es):

Derived From	As
<input type="text" value="CException"/>	public

Figure 64: The CBlockingSocketException class information.

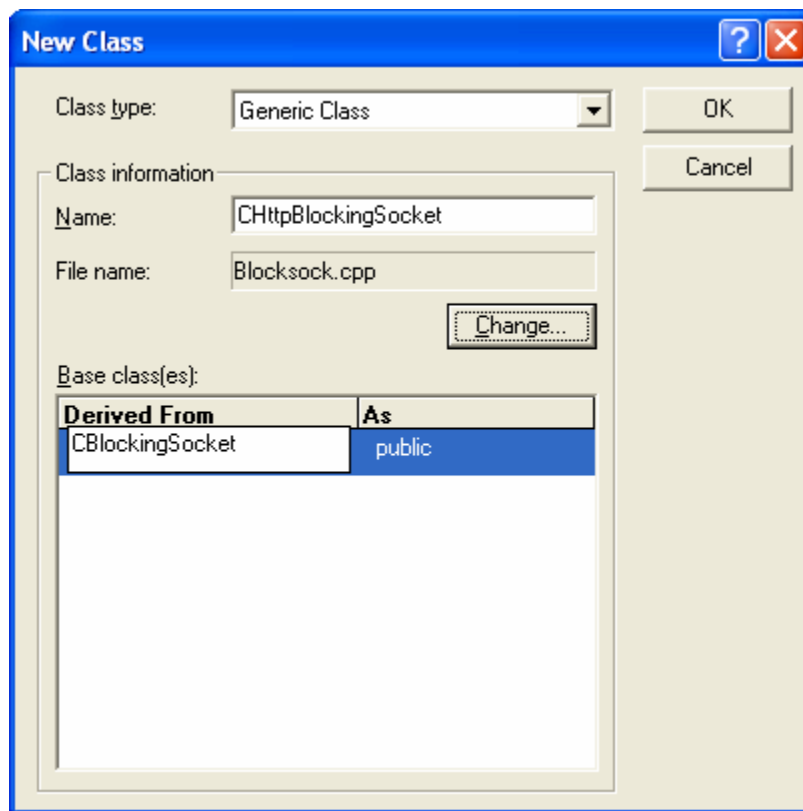


Figure 65: The CHttpRequestBlockingSocket class information.

Add another generic CCallbackInternetSession class using **Utility.h** and **Utility.cpp** as the header and source files respectively.

Class name	Base class
CCallbackInternetSession	CInternetSession

Table 27.

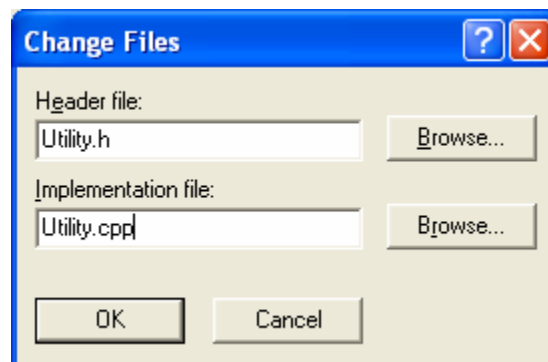


Figure 66: Modifying the header and source file names for new class.

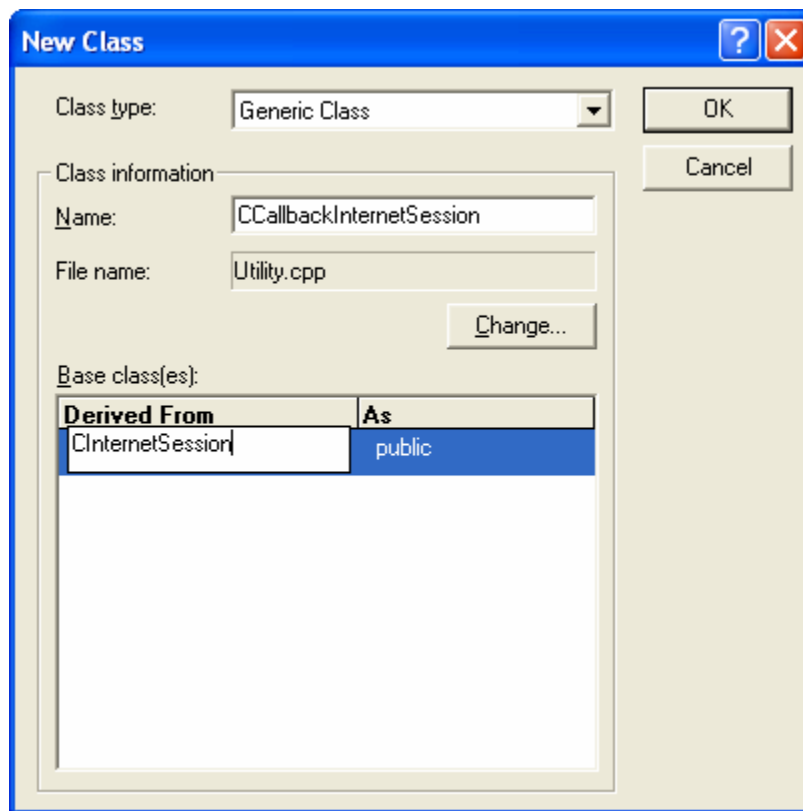


Figure 67: The CCallbackInternetSession class information.

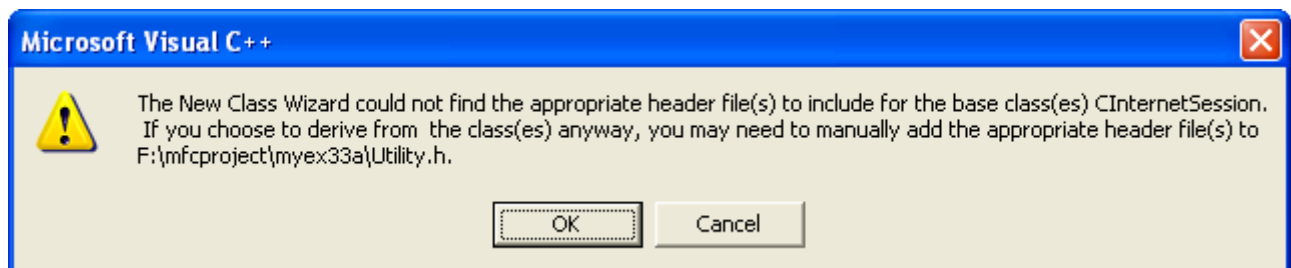


Figure 68: Dialog prompt for the non-existence file, just click the OK.

The Coding Part

Add the following header files in **StdAfx.h**. You can use **winsock2.h**, Winsock version 2. The library for the older Winsock (**ws_32.lib**) cannot be found in Tenouk's Visual C++ 6.0. Later on, you need to add the **ws2_32.lib** (Winsock 2) to your project.

```
#include <afxinet.h>    // MFC WinInet
#include <afxmt.h>      // MFC multi-threading classes
#include <winsock.h>    // Winsock

#ifdef _AFX_NO_AFXCMN_SUPPORT

#include <afxinet.h>    // MFC WinInet
#include <afxmt.h>      // MFC multi-threading classes
#include <winsock.h>    // Winsock

//{{AFX_INSERT_LOCATION}}
```

Listing 3.

We have to add the ID_INDICATOR_LISTENING status indicator to the resource symbol manually. Select **View Resource Symbols** menu.

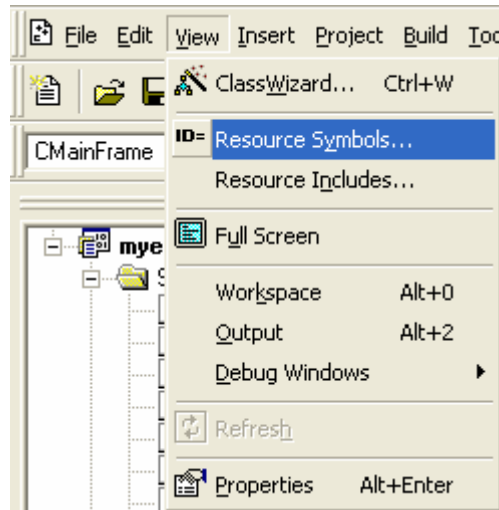


Figure 69: Viewing, adding or modifying resource symbols.

Click the **New** button and add the status indicator as shown below.

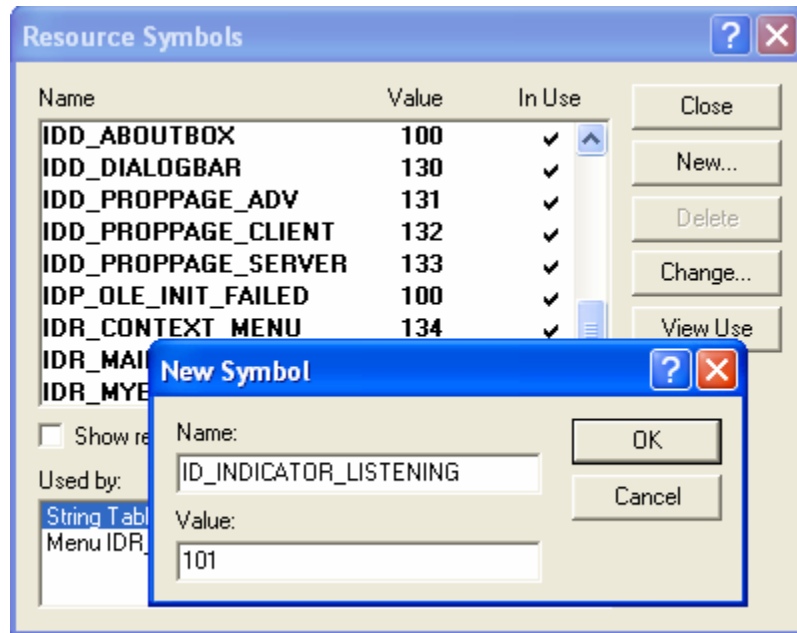


Figure 70: Adding new resource symbol.

Add string in string table. In ResourceView, double click the **String Table** sub directory. Right click the area under IDR_MAINFRAME and select **New String**.

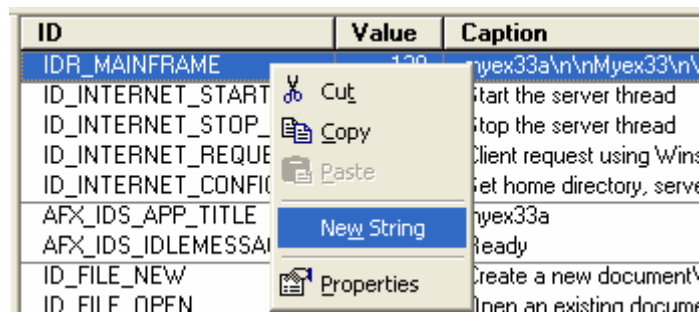


Figure 71: Adding new string to string table.

Add the following string.

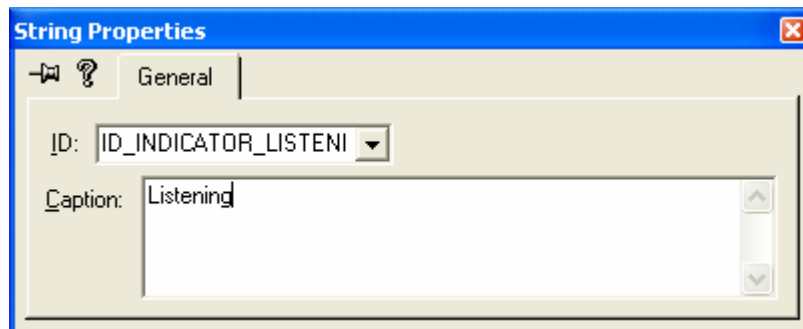


Figure 72: New string information.

Then we are ready to use the status bar indicator. Add the following status bar indicator with separator in **MainFrm.cpp**.

```

        ID_SEPARATOR,           // Wininet status
        ID_INDICATOR_LISTENING, // server listening

static UINT indicators[] =
{
    ID_SEPARATOR,           // status line indicator
    ID_SEPARATOR,           // Wininet status
    ID_INDICATOR_LISTENING, // server listening
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};

```

Listing 4.

Change the following code in **MainFrm.h** file. Later, we need to access those variables.

```

protected: // control bar embedded members
    CStatusBar m_wndStatusBar;
    CToolBar m_wndToolBar;

to

public: // control bar embedded members
    CStatusBar m_wndStatusBar;
    CDialogBar m_wndDialogBar;
protected:
    CToolBar m_wndToolBar;

```

```

public: // control bar embedded members
    CStatusBar m_wndStatusBar;
    CDialogBar m_wndDialogBar;
protected:
    CToolBar m_wndToolBar;
// Generated message map functions

```

Listing 5.

The `ID_INDICATOR_LISTENING` cannot be found in the ClassWizard (though the ClassWizard database has been rebuilt) so we need to add the code manually. Add the following indicator message handler declaration, command and command update, in **MainFrm.h**. Add the code just before the `DECLARE_MESSAGE_MAP()`.

```

afx_msg void OnUpdateListening(CCmdUI* pCmdUI);

// Generated message map functions
protected:
   //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    // NOTE - the ClassWizard will add and remove member functions here.
    // DO NOT EDIT what you see in these blocks of generated code!
   //}}AFX_MSG
    afx_msg void OnUpdateListening(CCmdUI* pCmdUI);
    DECLARE_MESSAGE_MAP()
};

```

Listing 6.

Add the message map for the declared status bar indicator in **MainFrm.cpp**.

```

ON_UPDATE_COMMAND_UI(ID_INDICATOR_LISTENING, OnUpdateListening)

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
   //{{AFX_MSG_MAP(CMainFrame)
    // NOTE - the ClassWizard will add and remove mapping macros here.
    // DO NOT EDIT what you see in these blocks of generated code !
    ON_WM_CREATE()
   //}}AFX_MSG_MAP
    ON_UPDATE_COMMAND_UI(ID_INDICATOR_LISTENING, OnUpdateListening)
END_MESSAGE_MAP()

```

Listing 7.

Then add the implementation code at the end of the **MainFrm.cpp**.

```

void CMainFrame::OnUpdateListening(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(g_bListening);
}

// CMainFrame message handlers
void CMainFrame::OnUpdateListening(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(g_bListening);
}

```


Listing 8.

Next, add the following `#include` directive.

```
#include "Utility.h"

#include "stdafx.h"
#include "myex33a.h"

#include "MainFrm.h"
#include "Utility.h"

#ifdef _DEBUG
#define new DEBUG_NEW
```

Listing 9.

Add/modify the `OnCreate()` as shown below.

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP
        | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;        // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
            sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;        // fail to create
    }

    // TODO: Remove this if you don't want tool tips or a resizable toolbar
    m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
        CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

    if (!m_wndDialogBar.Create(this, IDD_DIALOGBAR, CBRS_TOP, 0xE810))
    {
        TRACE0("Failed to create dialog bar\n");
        return -1;        // fail to create
    }
    m_wndDialogBar.SetDlgItemText(IDC_URL, g_strURL);
    return 0;
}
```

Listing 10.

```

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD
        | WS_VISIBLE | CBRS_TOP | CBRS_GRIPPER | CBRS_TOOLTIPS
        | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;          // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
            sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;          // fail to create
    }

    // TODO: Remove this if you don't want tool tips or a resizable toolbar
    m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
        CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

    if (!m_wndDialogBar.Create(this, IDD_DIALOGBAR, CBRS_TOP, 0xE810))
    {
        TRACE0("Failed to create dialog bar\n");
        return -1;          // fail to create
    }
    m_wndDialogBar.SetDlgItemText(IDC_URL, g_strURL);
    return 0;
}

```

Listing 11.

And the PreCreateWindow().

```

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CFrameWnd::PreCreateWindow(cs);
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CFrameWnd::PreCreateWindow(cs);
}

```

Listing 12.

Using ClassWizard, add/override ExitInstance() and OnIdle() to CMyx33aApp class.

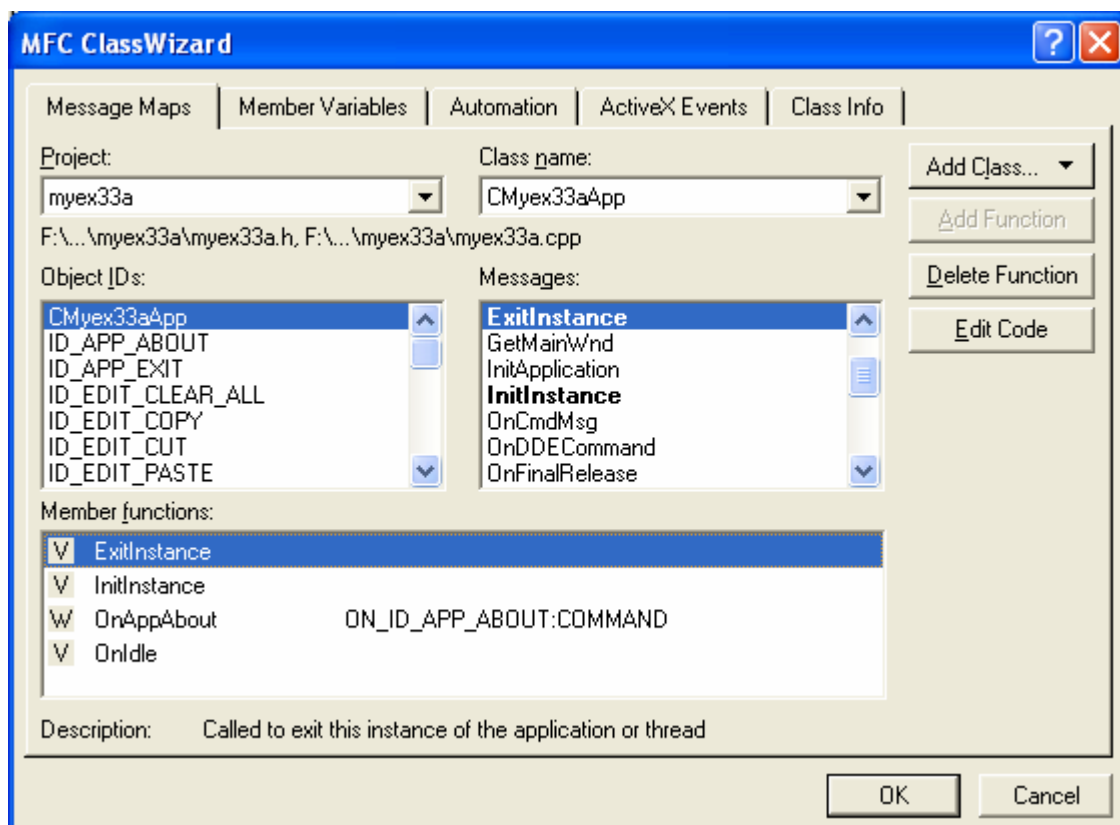


Figure 73: Adding/overriding message handler.

Click the **Edit Code** button, add the following `#include` directive to **myex33a.cpp**.

```
#include "Utility.h"
#include "Blocksock.h"

#include "stdafx.h"
#include "myex33a.h"

#include "MainFrm.h"
#include "myex33aDoc.h"
#include "myex33aView.h"
#include "Utility.h"
#include "Blocksock.h"

#ifdef _DEBUG
```

Listing 13.

Edit **myex33a.cpp** as shown below.

```
// myex33a.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "myex33a.h"

#include "MainFrm.h"
#include "myex33aDoc.h"
#include "myex33aView.h"
#include "Utility.h"
#include "Blocksock.h"
```

```

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

extern CBlockingSocket g_sListen;

////////////////////////////////////
// CMyex33aApp

BEGIN_MESSAGE_MAP(CMyex33aApp, CWinApp)
//{{AFX_MSG_MAP(CMyex33aApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //      DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
// Standard file based document commands
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
// Standard print setup command
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

////////////////////////////////////
// CMyex33aApp construction

CMyex33aApp::CMyex33aApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

////////////////////////////////////
// The one and only CMyex33aApp object

CMyex33aApp theApp;

////////////////////////////////////
// CMyex33aApp initialization

BOOL CMyex33aApp::InitInstance()
{
    AfxEnableControlContainer();
    // Standard initialization
    // If you are not using these features and wish to reduce the size
    // of your final executable, you should remove from the following
    // the specific initialization routines you do not need.

#ifdef _AFXDLL
    Enable3dControls();          // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();    // Call this when linking to MFC statically
#endif

    // Change the registry key under which our settings are stored.
    // TODO: You should modify this string to be something appropriate
    // such as the name of your company or organization.
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));

    LoadStdProfileSettings();    // Load standard INI file options (including MRU)

    // Register the application's document templates. Document templates
    // serve as the connection between documents, frame windows and views.

```

```

CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CMyex33aDoc),
    RUNTIME_CLASS(CMainFrame), // main SDI frame window
    RUNTIME_CLASS(CMyex33aView));
AddDocTemplate(pDocTemplate);

// Enable DDE Execute open
EnableShellOpen();
RegisterShellFileTypes(TRUE);

// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);

// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
    return FALSE;

// socket initialization
WSADATA wsd;
VERIFY(WSAStartup(0x0101, &wsd) == 0);
TRACE("WSAStartup -- min version = %x\n", wsd.wVersion);
g_hMainWnd = m_pMainWnd->m_hWnd;

return TRUE;
}

////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAboutDlg)
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    //{{AFX_MSG(CAboutDlg)
        // No message handlers
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)

```

```

        //}}AFX_DATA_MAP
    }

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
        // No message handlers
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CMyex33aApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

////////////////////////////////////
// CMyex33aApp message handlers

int CMyex33aApp::ExitInstance()
{
    // TODO: Add your specialized code here and/or call the base class
    try {
        if(g_bListening) {
            g_sListen.Close();
            Sleep(30); // wait for thread to exit
        }
        VERIFY(WSACleanup() != SOCKET_ERROR);
    }
    catch(CUserException* e) {
        TRACE("exception in CSock01App::ExitInstance\n");
        e->Delete();
    }
    return CWinApp::ExitInstance();
}

BOOL CMyex33aApp::OnIdle(LONG lCount)
{
    // TODO: Add your specialized code here and/or call the base class
    CStatusBar* pStatus = &((CMainFrame*) m_pMainWnd)->m_wndStatusBar;
    g_csStatus.Lock(); // blocking call in main thread -- could be dangerous
    pStatus->SetPaneText(1, g_pchStatus);
    g_csStatus.Unlock();
    return CWinApp::OnIdle(lCount);
}

```

Listing 14.

Add/modify the **Sheetconfig.cpp** code. Add the following code to the second constructor as shown below.

```

CSheetConfig::CSheetConfig(LPCTSTR pszCaption, CWnd* pParentWnd, UINT
iSelectPage):CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
    AddPage(&m_pageClient);
    AddPage(&m_pageServer);
    AddPage(&m_pageAdv);
}

```

```

CSheetConfig::CSheetConfig(LPCTSTR pszCaption, CWnd* pParentWnd,
    UINT iSelectPage):CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
    AddPage(&m_pageClient);
    AddPage(&m_pageServer);
    AddPage(&m_pageAdv);
}

CSheetConfig::~CSheetConfig()
{
}

```

Listing 15.

Add/modify the **Blocksock.h** code as shown below.

```

// Blocksock.h: interface for the CSockAddr class.
//
// needs winsock.h in the precompiled headers, add ws2_32.lib to the
// project later on...
////////////////////////////////////

#ifndef AFX_BLOCKSOCK_H__68C3054C_1A67_4CAE_8B43_5BEF5CFA1C19__INCLUDED_
#define AFX_BLOCKSOCK_H__68C3054C_1A67_4CAE_8B43_5BEF5CFA1C19__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

typedef const struct sockaddr* LPCSOCKADDR;

class CBlockingSocketException : public CException
{
    DECLARE_DYNAMIC(CBlockingSocketException)
public:
    // Constructor
    CBlockingSocketException(char* pchMessage);

public:
    ~CBlockingSocketException() {}
    virtual BOOL GetErrorMessage(LPCTSTR lpstrError, UINT nMaxError,
        PUINT pnHelpContext = NULL);
private:
    int m_nError;
    CString m_strMessage;
};

extern void LogBlockingSocketException(LPVOID pParam, char* pch,
    CBlockingSocketException* pe);

class CSockAddr : public sockaddr_in {
public:
    // constructors
    CSockAddr()
    { sin_family = AF_INET;
      sin_port = 0;
      // Default
      sin_addr.s_addr = 0; }
    CSockAddr(const SOCKADDR& sa) { memcpy(this, &sa, sizeof(SOCKADDR)); }
    CSockAddr(const SOCKADDR_IN& sin) { memcpy(this, &sin, sizeof(SOCKADDR_IN)); }
    // parms are host byte ordered
    CSockAddr(const ULONG ulAddr, const USHORT ushPort = 0)
    { sin_family = AF_INET;
      sin_port = htons(ushPort);
      sin_addr.s_addr = htonl(ulAddr); }
}

```

```

// dotted IP addr string
CSockAddr(const char* pchIP, const USHORT ushPort = 0)
{
    sin_family = AF_INET;
    sin_port = htons(ushPort);
    // already network byte ordered
    sin_addr.s_addr = inet_addr(pchIP); }
// Return the address in dotted-decimal format
CString DottedDecimal()
{
    // constructs a new CString object
    { return inet_ntoa(sin_addr); }
// Get port and address (even though they're public)
USHORT Port() const
{ return ntohs(sin_port); }
ULONG IPAddr() const
{ return ntohl(sin_addr.s_addr); }
// operators added for efficiency
const CSockAddr& operator=(const SOCKADDR& sa)
{ memcpy(this, &sa, sizeof(SOCKADDR));
  return *this; }
const CSockAddr& operator=(const SOCKADDR_IN& sin)
{ memcpy(this, &sin, sizeof(SOCKADDR_IN));
  return *this; }
operator SOCKADDR()
{ return *((LPSOCKADDR) this); }
operator LPSOCKADDR()
{ return (LPSOCKADDR) this; }
operator LPSOCKADDR_IN()
{ return (LPSOCKADDR_IN) this; }
};

// member functions truly block and must not be used in UI threads
// use this class as an alternative to the MFC CSocket class
class CBlockingSocket : public CObject
{
    DECLARE_DYNAMIC(CBlockingSocket)
public:
    SOCKET m_hSocket;
    CBlockingSocket() { m_hSocket = NULL; }
    void Cleanup();
    void Create(int nType = SOCK_STREAM);
    void Close();
    void Bind(LPSOCKADDR psa);
    void Listen();
    void Connect(LPSOCKADDR psa);
    BOOL Accept(CBlockingSocket& s, LPSOCKADDR psa);
    int Send(const char* pch, const int nSize, const int nSecs);
    int Write(const char* pch, const int nSize, const int nSecs);
    int Receive(char* pch, const int nSize, const int nSecs);
    int SendDatagram(const char* pch, const int nSize, LPSOCKADDR psa,
        const int nSecs);
    int ReceiveDatagram(char* pch, const int nSize, LPSOCKADDR psa,
        const int nSecs);
    void GetPeerAddr(LPSOCKADDR psa);
    void GetSockAddr(LPSOCKADDR psa);
    static CSockAddr GetHostByName(const char* pchName,
        const USHORT ushPort = 0);
    static const char* GetHostByAddr(LPSOCKADDR psa);
    operator SOCKET()
    { return m_hSocket; }
};

class CHttpBlockingSocket : public CBlockingSocket
{
public:
    DECLARE_DYNAMIC(CHttpBlockingSocket)
    // max receive buffer size (> hdr line length)

```



```

        enum {nSizeRecv = 10000};
        CHttpBlockingSocket();
        ~CHttpBlockingSocket();
        int ReadHttpHeaderLine(char* pch, const int nSize, const int nSecs);
        int ReadHttpResponse(char* pch, const int nSize, const int nSecs);
private:
        // read buffer
        char* m_pReadBuf;
        // number of bytes in the read buffer
        int m_nReadBuf;
};

#endif // !defined(AFX_BLOCKSOCKET_H__68C3054C_1A67_4CAE_8B43_5BEF5CFA1C19__INCLUDED_)

```

Listing 16.

Next, add/modify the implementation part of the previous member functions and variables in **Blocksock.cpp**.

```

// Blocksock.cpp: implementation of the
// CBlockingSocketException, CBlockingSocket, CHttpBlockingSocket
////////////////////////////////////

#include "stdafx.h"
#include "myex33a.h"
#include "Blocksock.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

// Class CBlockingSocketException
IMPLEMENT_DYNAMIC(CBlockingSocketException, CException)

CBlockingSocketException::CBlockingSocketException(char* pchMessage)
{
    m_strMessage = pchMessage;
    m_nError = WSAGetLastError();
}

BOOL CBlockingSocketException::GetErrorMessage(LPTSTR lpstrError, UINT nMaxError,
        PUINT pnHelpContext /*= NULL*/)
{
    char text[200];
    if(m_nError == 0)
    { wsprintf(text, "%s error", (const char*) m_strMessage); }
    else
    { wsprintf(text, "%s error #%d", (const char*) m_strMessage, m_nError); }
    strncpy(lpstrError, text, nMaxError - 1);
    return TRUE;
}

// Class CBlockingSocket
IMPLEMENT_DYNAMIC(CBlockingSocket, CObject)

void CBlockingSocket::Cleanup()
{
    // doesn't throw an exception because it's called in a catch block
    if(m_hSocket == NULL) return;
    VERIFY(closesocket(m_hSocket) != SOCKET_ERROR);
    m_hSocket = NULL;
}

```

```

void CBlockingSocket::Create(int nType /* = SOCK_STREAM */)
{
    ASSERT(m_hSocket == NULL);
    if((m_hSocket = socket(AF_INET, nType, 0)) == INVALID_SOCKET) {
        throw new CBlockingSocketException("Create");
    }
}

void CBlockingSocket::Bind(LPCSOCKADDR psa)
{
    ASSERT(m_hSocket != NULL);
    if(bind(m_hSocket, psa, sizeof(SOCKADDR)) == SOCKET_ERROR) {
        throw new CBlockingSocketException("Bind");
    }
}

void CBlockingSocket::Listen()
{
    ASSERT(m_hSocket != NULL);
    if(listen(m_hSocket, 5) == SOCKET_ERROR) {
        throw new CBlockingSocketException("Listen");
    }
}

BOOL CBlockingSocket::Accept(CBlockingSocket& sConnect, LPCTSTR psa)
{
    ASSERT(m_hSocket != NULL);
    ASSERT(sConnect.m_hSocket == NULL);
    int nLengthAddr = sizeof(SOCKADDR);
    sConnect.m_hSocket = accept(m_hSocket, psa, &nLengthAddr);
    if(sConnect == INVALID_SOCKET) {
        // no exception if the listen was canceled
        if(WSAGetLastError() != WSAEINTR) {
            throw new CBlockingSocketException("Accept");
        }
        return FALSE;
    }
    return TRUE;
}

void CBlockingSocket::Close()
{
    if (NULL == m_hSocket)
        return;

    if(closesocket(m_hSocket) == SOCKET_ERROR) {
        // should be OK to close if closed already
        throw new CBlockingSocketException("Close");
    }
    m_hSocket = NULL;
}

void CBlockingSocket::Connect(LPCSOCKADDR psa)
{
    ASSERT(m_hSocket != NULL);
    // should timeout by itself
    if(connect(m_hSocket, psa, sizeof(SOCKADDR)) == SOCKET_ERROR) {
        throw new CBlockingSocketException("Connect");
    }
}

int CBlockingSocket::Write(const char* pch, const int nSize, const int nSecs)
{
    int nBytesSent = 0;
    int nBytesThisTime;

```

```

        const char* pch1 = pch;
        do {
            nBytesThisTime = Send(pch1, nSize - nBytesSent, nSecs);
            nBytesSent += nBytesThisTime;
            pch1 += nBytesThisTime;
        } while(nBytesSent < nSize);
        return nBytesSent;
    }

int CBlockingSocket::Send(const char* pch, const int nSize, const int nSecs)
{
    ASSERT(m_hSocket != NULL);
    // returned value will be less than nSize if client cancels the reading
    FD_SET fd = {1, m_hSocket};
    TIMEVAL tv = {nSecs, 0};
    if(select(0, NULL, &fd, NULL, &tv) == 0) {
        throw new CBlockingSocketException("Send timeout");
    }
    int nBytesSent;
    if((nBytesSent = send(m_hSocket, pch, nSize, 0)) == SOCKET_ERROR) {
        throw new CBlockingSocketException("Send");
    }
    return nBytesSent;
}

int CBlockingSocket::Receive(char* pch, const int nSize, const int nSecs)
{
    ASSERT(m_hSocket != NULL);
    FD_SET fd = {1, m_hSocket};
    TIMEVAL tv = {nSecs, 0};
    if(select(0, &fd, NULL, NULL, &tv) == 0) {
        throw new CBlockingSocketException("Receive timeout");
    }

    int nBytesReceived;
    if((nBytesReceived = recv(m_hSocket, pch, nSize, 0)) == SOCKET_ERROR) {
        throw new CBlockingSocketException("Receive");
    }
    return nBytesReceived;
}

int CBlockingSocket::ReceiveDatagram(char* pch, const int nSize, LPSOCKADDR psa, const
int nSecs)
{
    ASSERT(m_hSocket != NULL);
    FD_SET fd = {1, m_hSocket};
    TIMEVAL tv = {nSecs, 0};
    if(select(0, &fd, NULL, NULL, &tv) == 0) {
        throw new CBlockingSocketException("Receive timeout");
    }

    // input buffer should be big enough for the entire datagram
    int nFromSize = sizeof(SOCKADDR);
    int nBytesReceived = recvfrom(m_hSocket, pch, nSize, 0, psa, &nFromSize);
    if(nBytesReceived == SOCKET_ERROR) {
        throw new CBlockingSocketException("ReceiveDatagram");
    }
    return nBytesReceived;
}

int CBlockingSocket::SendDatagram(const char* pch, const int nSize, LPCSOCKADDR psa,
const int nSecs)
{
    ASSERT(m_hSocket != NULL);
    FD_SET fd = {1, m_hSocket};
    TIMEVAL tv = {nSecs, 0};

```

```

        if(select(0, NULL, &fd, NULL, &tv) == 0) {
            throw new CBlockingSocketException("Send timeout");
        }

        int nBytesSent = sendto(m_hSocket, pch, nSize, 0, psa, sizeof(SOCKADDR));
        if(nBytesSent == SOCKET_ERROR) {
            throw new CBlockingSocketException("SendDatagram");
        }
        return nBytesSent;
    }

void CBlockingSocket::GetPeerAddr(LPSOCKADDR psa)
{
    ASSERT(m_hSocket != NULL);
    // gets the address of the socket at the other end
    int nLengthAddr = sizeof(SOCKADDR);
    if(getpeername(m_hSocket, psa, &nLengthAddr) == SOCKET_ERROR) {
        throw new CBlockingSocketException("GetPeerName");
    }
}

void CBlockingSocket::GetSockAddr(LPSOCKADDR psa)
{
    ASSERT(m_hSocket != NULL);
    // gets the address of the socket at this end
    int nLengthAddr = sizeof(SOCKADDR);
    if(getsockname(m_hSocket, psa, &nLengthAddr) == SOCKET_ERROR) {
        throw new CBlockingSocketException("GetSockName");
    }
}

//static
CsockAddr CBlockingSocket::GetHostByName(const char* pchName, const USHORT ushPort /* = 0
*/)
{
    hostent* pHostEnt = gethostbyname(pchName);
    if(pHostEnt == NULL) {
        throw new CBlockingSocketException("GetHostByName");
    }
    ULONG* pulAddr = (ULONG*) pHostEnt->h_addr_list[0];
    SOCKADDR_IN sockTemp;
    sockTemp.sin_family = AF_INET;
    sockTemp.sin_port = htons(ushPort);
    sockTemp.sin_addr.s_addr = *pulAddr; // address is already in network byte order
    return sockTemp;
}

//static
const char* CBlockingSocket::GetHostByAddr(LPCSOCKADDR psa)
{
    hostent* pHostEnt = gethostbyaddr((char*) &((LPSOCKADDR_IN) psa)
        ->sin_addr.s_addr, 4, PF_INET);
    if(pHostEnt == NULL) {
        throw new CBlockingSocketException("GetHostByAddr");
    }
    return pHostEnt->h_name; // caller shouldn't delete this memory
}

// Class CHttpBlockingSocket
IMPLEMENT_DYNAMIC(CHttpBlockingSocket, CBlockingSocket)

CHttpBlockingSocket::CHttpBlockingSocket()
{
    m_pReadBuf = new char[nSizeRecv];
    m_nReadBuf = 0;
}

```

```

CHttpBlockingSocket::~CHttpBlockingSocket()
{
    delete [] m_pReadBuf;
}

int CHttpBlockingSocket::ReadHttpHeaderLine(char* pch, const int nSize, const int nSecs)
// reads an entire header line through CRLF (or socket close)
// inserts zero string terminator, object maintains a buffer
{
    int nBytesThisTime = m_nReadBuf;
    int nLineLength = 0;
    char* pch1 = m_pReadBuf;
    char* pch2;
    do {
        // look for lf (assume preceded by cr)
        if((pch2 = (char*) memchr(pch1, '\n', nBytesThisTime)) != NULL) {
            ASSERT((pch2) > m_pReadBuf);
            ASSERT(*(pch2 - 1) == '\r');
            nLineLength = (pch2 - m_pReadBuf) + 1;
            if(nLineLength >= nSize) nLineLength = nSize - 1;
            memcpy(pch, m_pReadBuf, nLineLength); // copy the line to caller
            m_nReadBuf -= nLineLength;
            // shift remaining characters left
            memmove(m_pReadBuf, pch2 + 1, m_nReadBuf);
            break;
        }
        pch1 += nBytesThisTime;
        nBytesThisTime = Receive(m_pReadBuf + m_nReadBuf, nSizeRecv - m_nReadBuf,
nSecs);

        if(nBytesThisTime <= 0)
        { // sender closed socket or line longer than buffer
            throw new CBlockingSocketException("ReadHeaderLine");
        }
        m_nReadBuf += nBytesThisTime;
    }
    while(TRUE);
    *(pch + nLineLength) = '\0';
    return nLineLength;
}

int CHttpBlockingSocket::ReadHttpResponse(char* pch, const int nSize, const int nSecs)
// reads remainder of a transmission through buffer full or socket close
// (assume headers have been read already)
{
    int nBytesToRead, nBytesThisTime, nBytesRead = 0;
    if(m_nReadBuf > 0) { // copy anything already in the recv buffer
        memcpy(pch, m_pReadBuf, m_nReadBuf);
        pch += m_nReadBuf;
        nBytesRead = m_nReadBuf;
        m_nReadBuf = 0;
    }
    do { // now pass the rest of the data directly to the caller
        nBytesToRead = min(nSizeRecv, nSize - nBytesRead);
        nBytesThisTime = Receive(pch, nBytesToRead, nSecs);
        if(nBytesThisTime <= 0) break; // sender closed the socket
        pch += nBytesThisTime;
        nBytesRead += nBytesThisTime;
    }
    while(nBytesRead <= nSize);
    return nBytesRead;
}

void LogBlockingSocketException(LPVOID pParam, char* pch, CBlockingSocketException* pe)
{
    // pParam holds the HWND for the destination window (in another thread)
    CString strGmt = CTime::GetCurrentTime().FormatGmt("%m/%d/%y %H:%M:%S GMT");
}

```



```

extern CString g_strFile;
extern char g_pchStatus[];
extern HWND g_hMainWnd;
extern CCriticalSection g_csStatus;
extern CString g_strIPClient;
extern volatile UINT g_nPort;
extern CString g_strProxy;
extern BOOL g_bUseProxy;
extern volatile BOOL g_bListening;
extern CString g_strDirect;
extern CString g_strIPServer;
extern volatile UINT g_nPortServer;
extern CString g_strURL;
extern CString g_strDefault;

extern UINT ClientUrlThreadProc(LPVOID pParam);
extern UINT ServerThreadProc(LPVOID pParam);
extern UINT ClientWinInetThreadProc(LPVOID pParam);
extern UINT ClientSocketThreadProc(LPVOID pParam);

extern void LogInternetException(LPVOID pParam, CInternetException* pe);

class CCallbackInternetSession : public CInternetSession
{
public:
    CCallbackInternetSession( LPCTSTR pstrAgent = NULL, DWORD dwContext = 1,
        DWORD dwAccessType = PRE_CONFIG_INTERNET_ACCESS, LPCTSTR pstrProxyName = NULL,
        LPCTSTR pstrProxyBypass = NULL, DWORD dwFlags = 0 );
protected:
    virtual void OnStatusCallback(DWORD dwContext, DWORD dwInternalStatus,
        LPVOID lpvStatusInformation, DWORD dwStatusInformationLength);
};

#endif // !defined(AFX_UTILITY_H__C57D5E4E_7F33_4249_BEA4_E69BD55D710B__INCLUDED_)

```

Listing 18.

```

// Utility.cpp: implementation of the CCallbackInternetSession class
// and other functions - contains stuff used by more than one thread
//
// ////////////////////////////////////////

#include "stdafx.h"
#include "myex33a.h"
#include "Utility.h"
#include "Blocksock.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

// connection number
volatile int g_nConnection = 0;
// used by both winsock and wininet
CString g_strServerName = "localhost";
CString g_strServerIP;
volatile UINT g_nPort = 80;
CString g_strFile = "/custom";

CCallbackInternetSession::CCallbackInternetSession( LPCTSTR pstrAgent, DWORD dwContext,
    DWORD dwAccessType, LPCTSTR pstrProxyName, LPCTSTR pstrProxyBypass, DWORD
dwFlags) :

```

```

        CInternetSession(pstrAgent, dwContext, dwAccessType, pstrProxyName,
pstrProxyBypass, dwFlags)
{
    EnableStatusCallback();
}

void CCallbackInternetSession::OnStatusCallback(DWORD dwContext, DWORD dwInternalStatus,
        LPVOID lpvStatusInformation, DWORD dwStatusInformationLength)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    int errors[] = {10, 11, 20, 21, 30, 31, 40, 41, 42, 43, 50, 51, 60, 70, 100, 110,
0};
    char* text[] = {
        "Resolving name",
        "Name resolved",
        "Connecting to server",
        "Connected to server",
        "Sending request",
        "Request sent",
        "Receiving response",
        "Response received",
        "Ctl response received",
        "Prefetch",
        "Closing connection",
        "Connection closed",
        "Handle created",
        "Handle closing",
        "Request complete",
        "Redirect",
        "Unknown" };
    int n;
    /* // demonstrates request cancellation
    if(dwInternalStatus == INTERNET_STATUS_REQUEST_SENT) {
        AfxThrowInternetException(dwContext, 999);
    }
    */
    for(n = 0; errors[n] != 0; n++) {
        if(errors[n] == (int) dwInternalStatus) break;
    }
    g_csStatus.Lock();
    strcpy(g_pchStatus, text[n]);
    if(dwInternalStatus == INTERNET_STATUS_RESOLVING_NAME ||
        dwInternalStatus == INTERNET_STATUS_NAME_RESOLVED) {
        strcat(g_pchStatus, "-");
        strcat(g_pchStatus, (char*) lpvStatusInformation);
    }
    TRACE("WININET STATUS: %s\n", g_pchStatus);
    g_csStatus.Unlock();
    // frame doesn't need a handler -- message triggers OnIdle, which updates status
bar
    ::PostMessage(g_hMainWnd, WM_CALLBACK, 0, 0);
}

void LogInternetException(LPVOID pParam, CInternetException* pe)
{
    // pParam holds the HWND for the destination window (in another thread)
    CString strGmt = CTime::GetCurrentTime().FormatGmt("%m/%d/%y %H:%M:% GMT");
    char text1[300], text2[100];
    wsprintf(text1, "CLIENT ERROR: WinInet error #d -- %s\r\n  ",
        pe->m_dwError, (const char*) strGmt);
    pe->GetErrorMessage(text2, 99);
    strcat(text1, text2);
    if(pe->m_dwError == 12152) {
        strcat(text1, " URL not found?\r\n");
    }
    ::SendMessage((HWND) pParam, EM_SETSEL, (WPARAM) 65534, 65535);
    ::SendMessage((HWND) pParam, EM_REPLACESEL, (WPARAM) 0, (LPARAM) text1);
}

```



```
}

```

Listing 19.

Add the server thread source file, **ServerThread.cpp** as shown below.

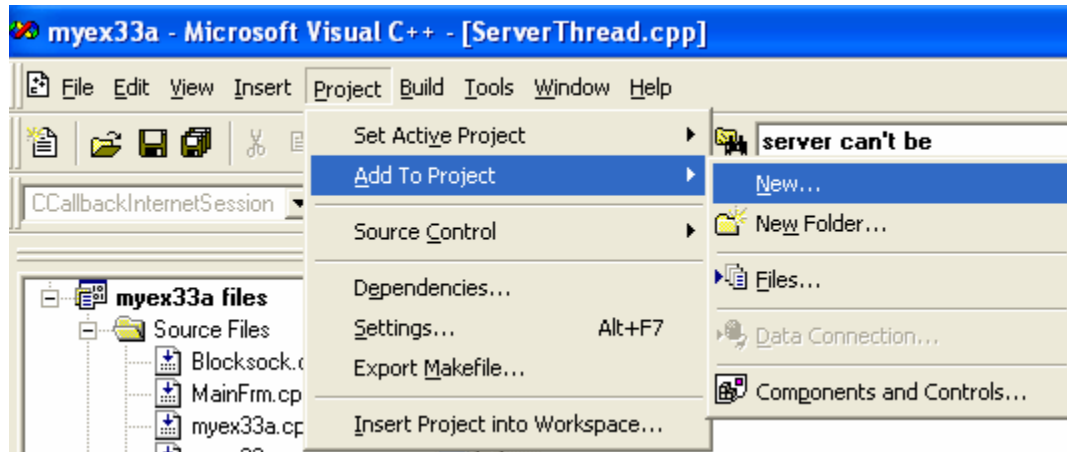


Figure 75: Adding new file to project.

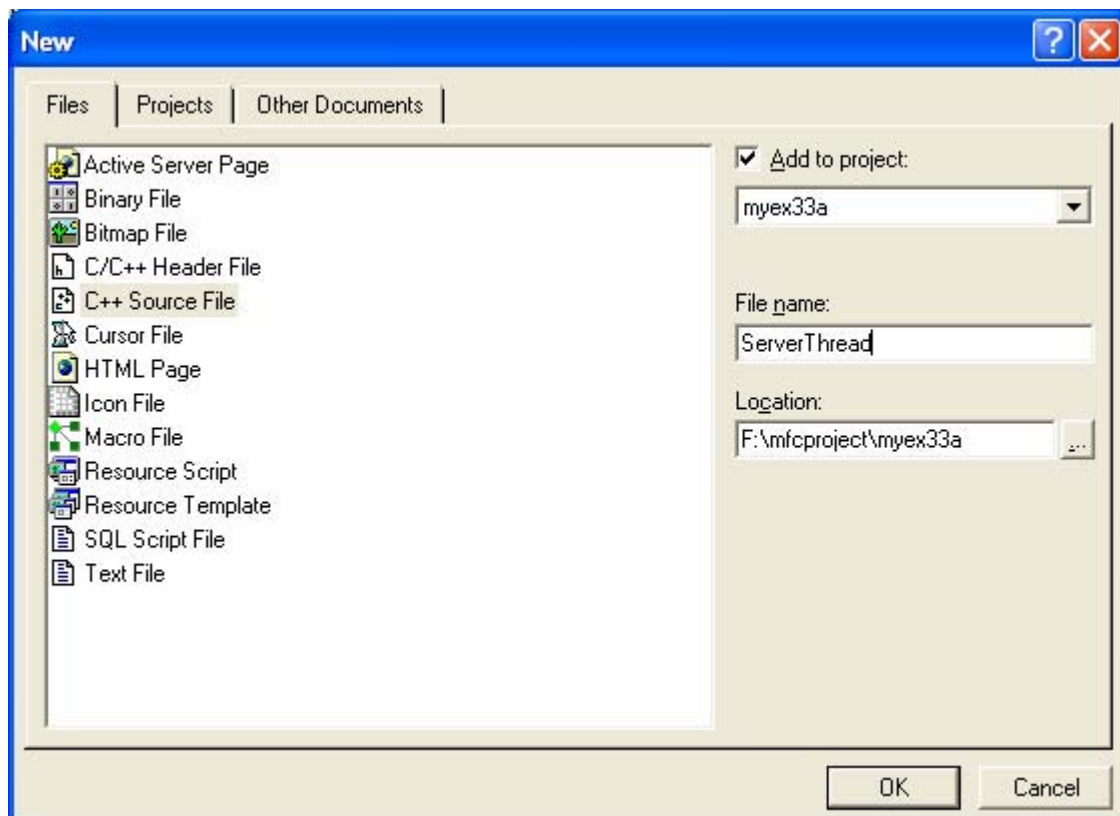


Figure 76: Adding source file to project.

Copy and paste the following code.

```
// serverthread.cpp
#include <stdafx.h>

```

```

#include "Blocksock.h"
#include "Utility.h"
#define SERVERMAXBUF 10000
#define MAXLINELENGTH 200

// #define USE_TRANSMITFILE // uncomment if you have Windows NT

volatile BOOL g_bListening = FALSE;
volatile UINT g_nPortServer = 80;

// Adjust to your programming environment accordingly...
CString g_strDirect = "F:\\mfcproject\\myex33a\\WebSite";
CString g_strIPServer;
// The default document, later, you need to create this file...
CString g_strDefault = "default.htm";
CBlockingSocket g_sListen;

BOOL Parse(char* pStr, char** ppToken1, char** ppToken2)
// really stupid parsing routine
// (must find two tokens, each followed by a space)
{
    *ppToken1 = pStr;
    char* pch = strchr(pStr, ' ');
    if(pch) {
        *pch = '\\0';
        pch++;
        *ppToken2 = pch;
        pch = strchr(pch, ' ');
        if(pch) {
            *pch = '\\0';
            return TRUE;
        }
    }
    return FALSE;
}

void LogRequest(LPVOID pParam, char* pch, CSockAddr sa)
{
    // pParam holds the HWND for the destination window (in another thread)
    CString strGmt = CTime::GetCurrentTime().FormatGmt("%m/%d/%y %H:%M:%S GMT");
    char text1[1000];
    wsprintf(text1, "SERVER CONNECTION # %d: IP addr = %s, port = %d -- %s\\r\\n",
        g_nConnection, sa.DottedDecimal(), sa.Port(), (const char*) strGmt);
    strcat(text1, pch);
    ::SendMessage((HWND) pParam, EM_SETSEL, (WPARAM) 65534, 65535);
    ::SendMessage((HWND) pParam, EM_REPLACESEL, (WPARAM) 0, (LPARAM) text1);
}

CFile* OpenFile(const char* pName)
{
    // if it's really a directory, open the default HTML file
    CFileException e;
    CFile* pFile = new CFile();
    if(*pName == '/') pName++;
    CString strName = pName;
    if(pFile->Open(strName, CFile::modeRead, &e)) {
        return pFile;
    }
    if((e.m_cause == CFileException::accessDenied) ||
        (e.m_cause == CFileException::badPath)) { // directory?
        int nLength;
        // add a / unless it's the "root" directory
        if((nLength = strName.GetLength()) > 1) {
            if(strName[nLength - 1] != '/') {
                strName += '/';
            }
        }
    }
}

```

```

        strName += g_strDefault;
        if(pFile->Open(strName, CFile::modeRead, &e)) {
            return pFile;
        }
    }
    delete pFile;
    return NULL;
}

UINT ServerThreadProc(LPVOID pParam)
{
    CSocketAddr saClient;
    CHttpBlockingSocket sConnect;
    char* buffer = new char[SERVERMAXBUF];
    char message[100], headers[500], request1[MAXLINELENGTH],
request2[MAXLINELENGTH];
    char hdrErr[] = "HTTP/1.0 404 Object Not Found\r\n"
        "Server: Inside Visual C++ SOCK01\r\n"
        "Content-Type: text/html\r\n"
        "Accept-Ranges: bytes\r\n"
        "Content-Length: 66\r\n\r\n" // WinInet wants correct length
        "<html><h1><body>HTTP/1.0 404 Object Not Found</h1></body></html>\r\n";
    char hdrFmt[] = "HTTP/1.0 200 OK\r\n"
        "Server: Inside Visual C++ EX34A\r\n"
        "Date: %s\r\n"
        "Content-Type: text/html\r\n"
        "Accept-Ranges: bytes\r\n"
        "Content-Length: %d\r\n";
    char html1[] = "<html><head><title>Inside Visual C++ \
        Server</title></head>\r\n"
        "<body><body background=\"\"/samples/images/us1.jpg\">\r\n"
        "<h1><center>This is a custom home page</center></h1><p>\r\n"
        "<a href=\"\"/samples/iisdocs.htm\">Click here for iisdocs.htm.</a><p>\r\n"
        "<a href=\"\"/samples/disclaim.htm\">Click here for
disclaim.htm.</a><p>\r\n";
        // custom message goes here
    char html2[] = "</body></html>\r\n\r\n";
    CString strGmtNow = CTime::GetCurrentTime().FormatGmt("%a, %d %b %Y %H:%M:%S
GMT");
    int nBytesSent = 0;
    CFile* pFile = NULL;
    try {
        if(!g_sListen.Accept(sConnect, saClient)) {
            // view or application closed the listing socket
            g_bListening = FALSE;
            delete [] buffer;
            return 0;
        }
        g_nConnection++;
        ::SetCurrentDirectory(g_strDirect);
        AfxBeginThread(ServerThreadProc, pParam, THREAD_PRIORITY_NORMAL);
        // read request from client
        sConnect.ReadHttpHeaderLine(request1, MAXLINELENGTH, 10);
        LogRequest(pParam, request1, saClient);
        char* pToken1; char* pToken2;
        if(Parse(request1, &pToken1, &pToken2)) {
            if(!strcmp(pToken1, "GET")) {
                do { // eat the remaining headers
                    sConnect.ReadHttpHeaderLine(request2, MAXLINELENGTH, 10);
                    TRACE("SERVER: %s", request2);
                }
                while(strcmp(request2, "\r\n"));
                if(!strcmp(pToken2, "/custom")) { // special request
                    // send a "custom" HTML page
                    sprintf(message, "Hi! you are connection #%d on IP %s, port
%d<p>%s", g_nConnection, saClient.DottedDecimal(), saClient.Port(), strGmtNow);

```

```

        wsprintf(headers, hdrFmt, (const char*) strGmtNow, strlen(html1)
        + strlen(message) + strlen(html2));
        // no If-Modified
        strcat(headers, "\r\n"); // blank line
        sConnect.Write(headers, strlen(headers), 10);
        sConnect.Write(html1, strlen(html1), 10);
        sConnect.Write(message, strlen(message), 10);
        sConnect.Write(html2, strlen(html2), 10);
    }
    else if(strchr(pToken2, '?')) { // CGI request
        // Netscape doesn't pass function name in a GET
        TRACE("SERVER: CGI request detected %s\n", pToken2);
        // could load and run the ISAPI DLL here
    }
    else { // must be a file
        // assume this program has already
        // set the default WWW directory
        if((pFile = OpenFile(pToken2)) != NULL) {
            CFileStatus fileStatus;
            pFile->GetStatus(fileStatus);
            CString strGmtMod = fileStatus.m_mtime.FormatGmt("%a, %d
                %b %Y %H:%M:%S GMT");
            char hdrModified[50];
            wsprintf(hdrModified, "Last-Modified: %s\r\n\r\n",
                (const char*) strGmtMod);
            DWORD dwLength = pFile->GetLength();
            // Date: , Content-Length:
            wsprintf(headers, hdrFmt, (const char*) strGmtNow,
                dwLength);
            strcat(headers, hdrModified);
            nBytesSent = sConnect.Write(headers, strlen(headers),

                10);
            TRACE("SERVER: header characters sent = %d\n",
                nBytesSent);
            // would be a good idea to send
            // the file only if the If-
            // Modified-Since date
            // were less than the file's m_mtime
            nBytesSent = 0;
#ifdef USE_TRANSMITFILE
            if(::TransmitFile(sConnect, (HANDLE) pFile->m_hFile, dwLength,
                0, NULL, NULL, TF_DISCONNECT)) {
                nBytesSent = (int) dwLength;
            }
#else
            DWORD dwBytesRead = 0;
            UINT uBytesToRead;
            // send file in small chunks (5K) to avoid big memory alloc overhead
            while(dwBytesRead < dwLength) {
                uBytesToRead = min(SERVERMAXBUF, dwLength - dwBytesRead);
                VERIFY(pFile->Read(buffer, uBytesToRead) == uBytesToRead);
                nBytesSent += sConnect.Write(buffer, uBytesToRead, 10);
                dwBytesRead += uBytesToRead;
            }
#endif
            TRACE("SERVER: full file sent successfully\n");
        }
        else {
            // 404 Object Not Found
            nBytesSent = sConnect.Write(hdrErr, strlen(hdrErr), 10);
        }
    }
    else if(!strcmp(pToken1, "POST")) {
        do { // eat the remaining headers thru blank line

```

```

        sConnect.ReadHttpHeaderLine(request2, MAXLINELENGTH, 10);
        TRACE("SERVER: POST %s", request2);
    }
    while(strcmp(request2, "\r\n"));
    // read the data line sent by the client
    sConnect.ReadHttpHeaderLine(request2, MAXLINELENGTH, 10);
    TRACE("SERVER: POST PARAMETERS = %s\n", request2);
    LogRequest(pParam, request2, saClient);
    // launch ISAPI DLL here?
    // 404 error for now
    nBytesSent = sConnect.Write(hdrErr, strlen(hdrErr),
    10);
}
else {
    TRACE("SERVER: %s (not a GET or POST)\n", pToken1);
    // don't know how to eat the headers
}
}
else {
    TRACE("SERVER: bad request\n");
}
sConnect.Close(); // destructor can't close it
}
catch(CBlockingSocketException* pe) {
    LogBlockingSocketException(pParam, "SERVER:", pe);
    pe->Delete();
}
TRACE("SERVER: file characters sent = %d\n", nBytesSent);
delete [] buffer;
if(pFile) delete pFile;
return 0;
}

```

Listing 20.

Add the following client threads source files for Winsock, WinInet and OpenURL (): **ClientinetThread.cpp**, **ClientsockThread.cpp** and **ClienturlThread.cpp**.

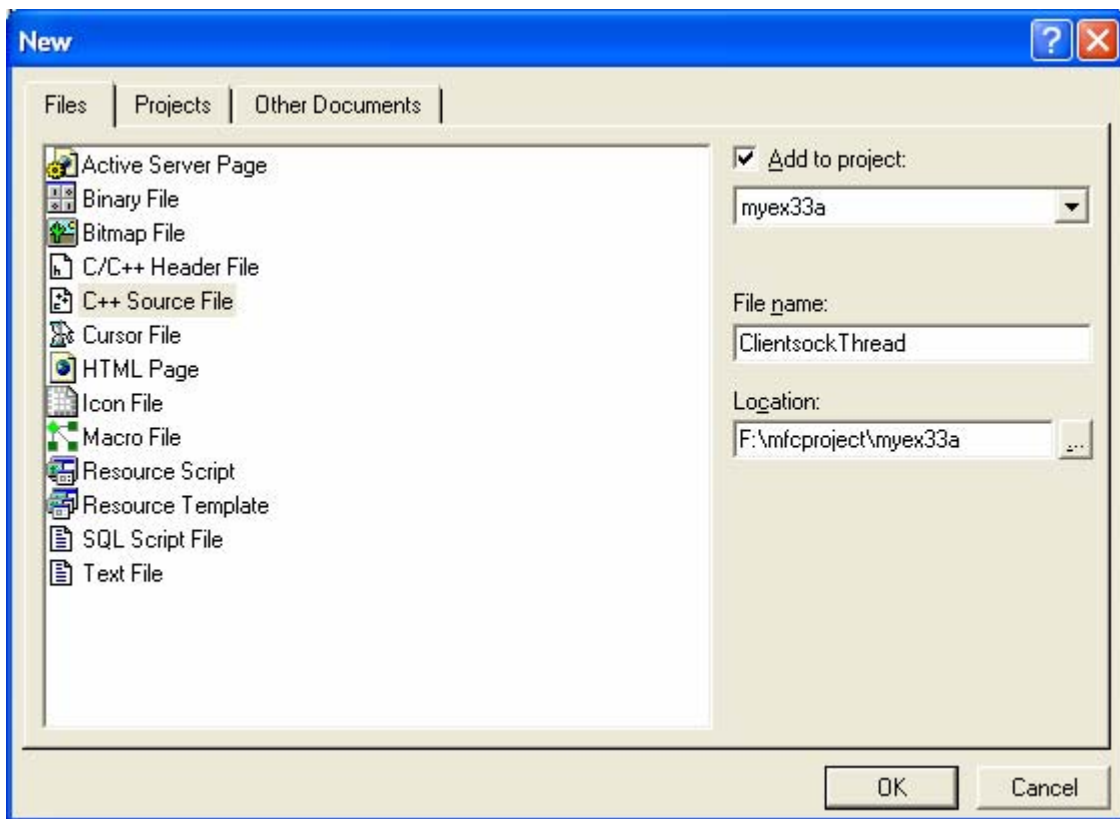


Figure 77: Adding another new source file to project.

Add the codes.

```
// clientinethread.cpp (uses MFC Wininet calls)

#include <stdafx.h>
#include "utility.h"
#define MAXBUF 80000

HWND g_hMainWnd = 0;
char g_pchStatus[25] = "";
CCriticalSection g_csStatus;

UINT ClientWinInetThreadProc(LPVOID pParam)
{
    CCallbackInternetSession session;
    CHttpConnection* pConnection = NULL;
    CHttpFile* pFile1 = NULL;
    char* buffer = new char[MAXBUF];
    UINT nBytesRead = 0;
    DWORD dwStatus;
    try {
        // username/password doesn't work yet
        if(!g_strServerName.IsEmpty()) {
            pConnection = session.GetHttpConnection(g_strServerName,
                (INTERNET_PORT) g_nPort);
        }
        else {
            pConnection = session.GetHttpConnection(g_strServerIP,
                (INTERNET_PORT) g_nPort);
        }
        pFile1 = pConnection->OpenRequest(1, g_strFile, NULL, 1, NULL, NULL, // GET
request
INTERNET_FLAG_KEEP_CONNECTION); // needed for NT
```

Challenge/Response authentication

```
// INTERNET_FLAG_RELOAD forces reload from the server (bypasses client's
// cache)
pFile1->SendRequest();
pFile1->QueryInfoStatusCode(dwStatus);
TRACE("QueryInfoStatusCode = %d\n", dwStatus);
nBytesRead = pFile1->Read(buffer, MAXBUF - 1);
buffer[nBytesRead] = '\0'; // necessary for message box
char temp[100];
if(pFile1->Read(temp, 100) != 0) { // makes caching work if read complete
    AfxMessageBox("File overran buffer -- not cached");
}
::MessageBox(::GetTopWindow(::GetDesktopWindow()), buffer, "WININET
CLIENT", MB_OK);
// could use existing pFile1 to SendRequest again if we wanted to
}
catch(CInternetException* e) {
    LogInternetException(pParam, e);
    e->Delete();
}
// could call OpenRequest again on same connection if we wanted to
if(pFile1) delete pFile1; // does the close -- prints a warning
// why does it print a warning?
if(pConnection) delete pConnection;
delete [] buffer;
g_csStatus.Lock();
// problem with empty string. bug #9897
strcpy(g_pchStatus, "");
g_csStatus.Unlock();
return 0;
}
```

Listing 21.

```
// clientsockthread.cpp (uses Winsock calls only)

#include <stdafx.h>
#include "blocksock.h"
#include "utility.h"
#define MAXBUF 80000

CString g_strIPClient;
CString g_strProxy = "ITGPROXY";
BOOL g_bUseProxy = FALSE;

UINT ClientSocketThreadProc(LPVOID pParam)
{
    // sends a blind request, followed by a request for a specific URL
    CHttpBlockingSocket sClient;
    char* buffer = new char[MAXBUF];
    int nBytesReceived = 0;
    // We're doing a blind GET, but we must provide server name if we're using a
    proxy.
    // A blind GET is supposed to retrieve the server's default HTML document.
    // Some servers don't have a default document but return a document name
    // in the Location header.
    char request[] = "GET %s%s%s HTTP/1.0\r\n";
    char headers[] =
        "User-Agent: Mozilla/1.22 (Windows; U; 32bit)\r\n"
        "Accept: */*\r\n"
        "Accept: image/gif\r\n"
        "Accept: image/x-xbitmap\r\n"
        "Accept: image/jpeg\r\n"
    // following line tests server's ability to not send the URL
}
```

```

//          "If-Modified-Since: Wed, 11 Sep 1996 20:23:04 GMT\r\n"
        "\r\n"; // need this
CSockAddr saServer, saPeer, saTest, saClient;
try {
    sClient.Create();
    if(!g_strIPClient.IsEmpty()) {
        // won't work if network is assigning us our IP address
        // good only for intranets where client computer
        // has several IP addresses
        saClient = CSockAddr(g_strIPClient);
        sClient.Bind(saClient);
    }
    if(g_bUseProxy) {
        saServer = CBlockingSocket::GetHostByName(g_strProxy, 80);
    }
    else {
        if(g_strServerIP.IsEmpty()) {
            saServer = CBlockingSocket::GetHostByName(g_strServerName, g_nPort);
        }
        else {
            saServer = CSockAddr(g_strServerIP, g_nPort);
        }
    }
    sClient.Connect(saServer);
    sClient.GetSockAddr(saTest);
    TRACE("SOCK CLIENT: GetSockAddr = %s, %d\n", saTest.DottedDecimal(),
saTest.Port());
    if(g_bUseProxy) {
        wsprintf(buffer, request, "http://" , (const char*) g_strServerName,
g_strFile);
    }
    else {
        wsprintf(buffer, request, "", "", g_strFile);
    }
    sClient.Write(buffer, strlen(buffer), 10);
    sClient.Write(headers, strlen(headers), 10);
    // read all the server's response headers
    do {
        nBytesReceived = sClient.ReadHttpHeaderLine(buffer, MAXBUF, 10);
        TRACE("SOCK CLIENT: %s", buffer);
    } while(strcmp(buffer, "\r\n"));
    // read the server's file
    nBytesReceived = sClient.ReadHttpResponse(buffer, MAXBUF, 10);
    TRACE("SOCK CLIENT: bytes received = %d\n", nBytesReceived);
    if(nBytesReceived == 0) {
        AfxMessageBox("No response received. Bad URL?");
    }
    else {
        buffer[nBytesReceived] = '\0';
        ::MessageBox(::GetTopWindow(::GetDesktopWindow()), buffer, "WINSOCK
CLIENT", MB_OK);
    }

    // could do another request on sClient by calling Close, then Create, etc.
}
catch(CBlockingSocketException* e) {
    LogBlockingSocketException(pParam, "CLIENT:", e);
    e->Delete();
}
sClient.Close();
delete [] buffer;
return 0;
}

```

Listing 22.


```

// clienturlthread.cpp (uses CInternetSession::OpenURL)

#include <stdafx.h>
#include "utility.h"
#define MAXBUF 80000

CString g_strURL = "http://";

UINT ClientUrlThreadProc(LPVOID pParam)
{
    char* buffer = new char[MAXBUF];
    UINT nBytesRead = 0;

    CInternetSession session; // can't get status callbacks for OpenURL
    CStdioFile* pFile1 = NULL; // could call ReadString to get 1 line
    try {
        pFile1 = session.OpenURL(g_strURL, 0, INTERNET_FLAG_TRANSFER_BINARY |
                                INTERNET_FLAG_KEEP_CONNECTION);
        // if OpenURL fails, we won't get past here
        nBytesRead = pFile1->Read(buffer, MAXBUF - 1);
        buffer[nBytesRead] = '\\0'; // necessary for message box
        char temp[100];
        if(pFile1->Read(temp, 100) != 0) { // makes caching work if read complete
            AfxMessageBox("File overran buffer -- not cached");
        }
        ::MessageBox(::GetTopWindow(::GetDesktopWindow()), buffer, "URL CLIENT",
MB_OK);
    }
    catch(CInternetException* e) {
        LogInternetException(pParam, e);
        e->Delete();
    }
    if(pFile1) delete pFile1;
    delete [] buffer;
    return 0;
}

```

Listing 23.

Using ClassWizard, add SaveModified() virtual function to CMyex33aDoc class.

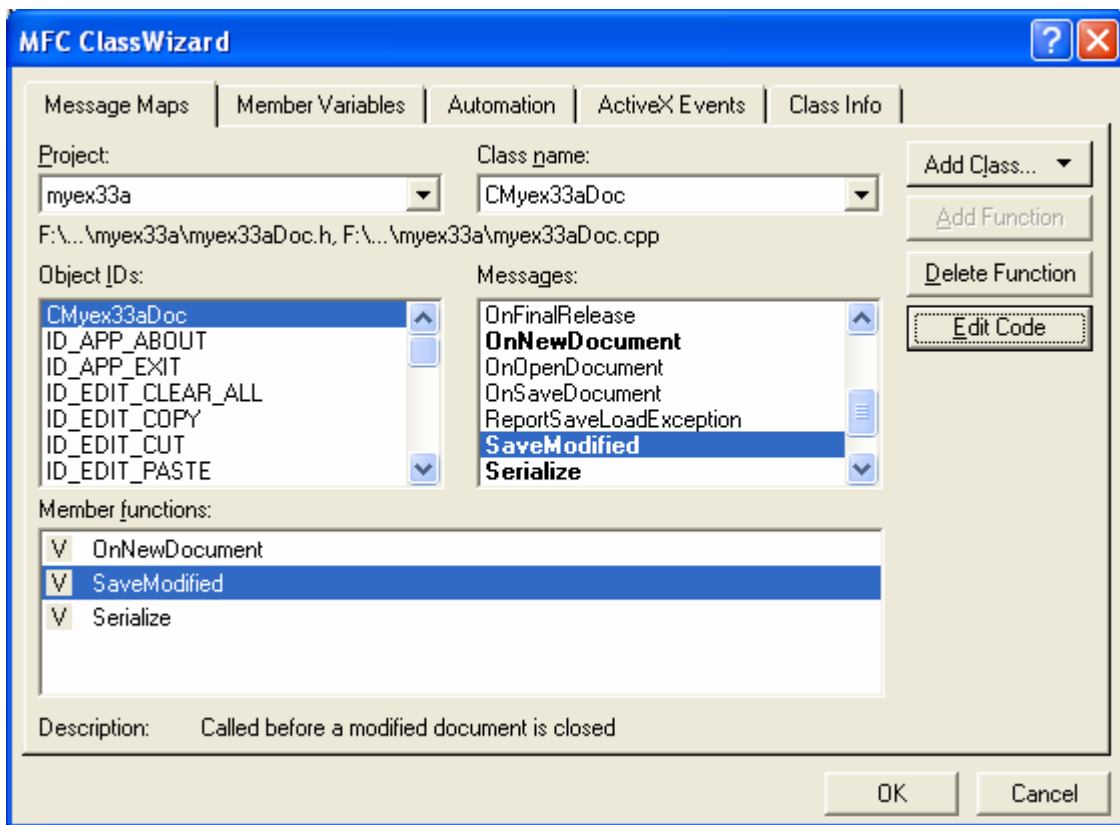


Figure 78: Adding SaveModified() function to CMyex33aDoc class.

Click the **Edit Code** button and edit the SaveModified() as shown below.

```

BOOL CMyex33aDoc::SaveModified()
{
    // TODO: Add your specialized code here and/or call the base class
    return TRUE; // eliminates "save doc" message on exit
}

```

Using ClassWizard, add the following message map to **CMyex33aView** class. Before that, open the **ResourceView**, select the **IDD_DIALOGBAR** under the **dialog** folder and launch ClassWizard. When the **Adding a class** prompt displayed, choose **Select an existing class** radio button and select **CMyex33aView** class. Just click **Yes** button if warning/prompt dialog displayed.

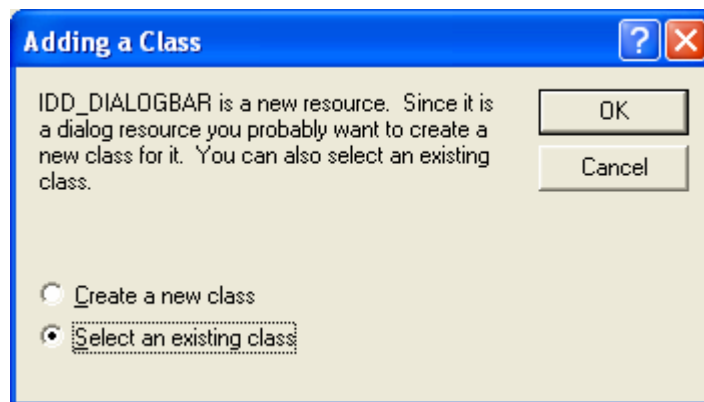


Figure 79: Adding new class prompt dialog.

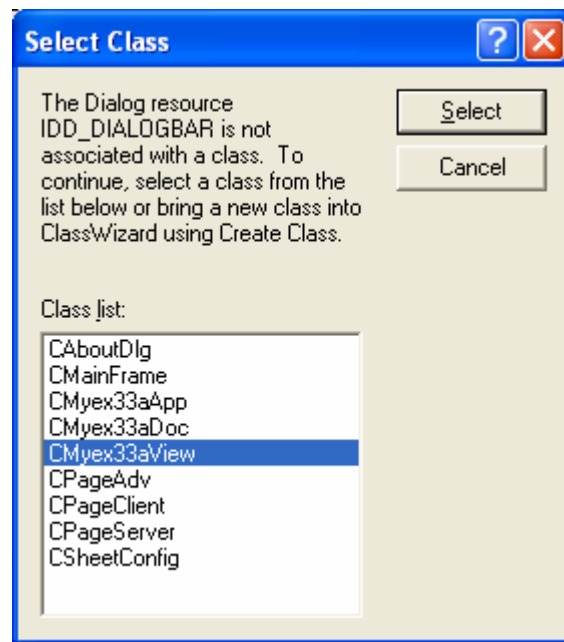


Figure 80: Selecting the existing class.

Next, add the following message handlers.

ID	Message	Function
IDR_REQUEST	BN_CLICK	OnRequest ()

Table 28.

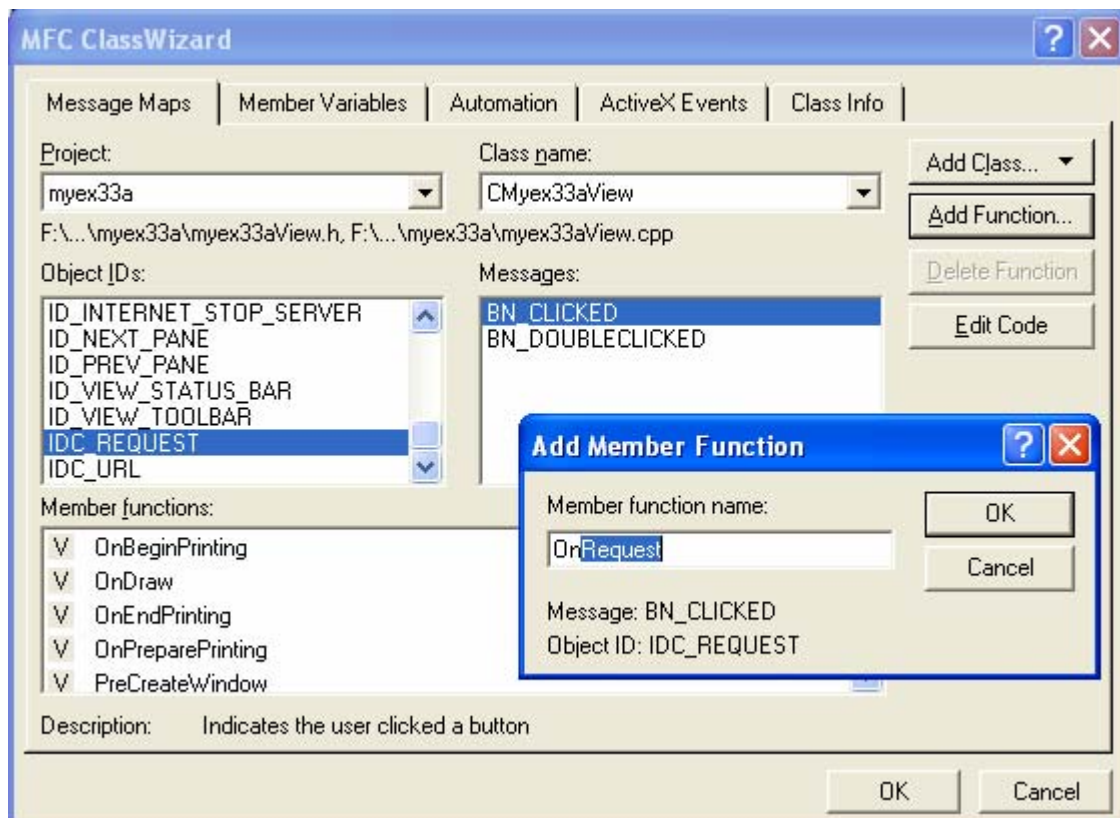


Figure 81: Adding message handler to CMyex33aView class.

Next, add the following command, command updates and Windows message handler to the CMyex33aView class.

ID	Message map	Function
ID_INTERNET_START_SERVER	Command	OnInternetStartServer()
	Command Update	OnUpdateInternetStartServer(CCmdUI* pCmdUI)
ID_INTERNET_REQUEST SOCK	Command	OnInternetRequestSocket()
ID_INTERNET_REQUEST_INET	Command	OnInternetRequestWininet()
ID_INTERNET_STOP_SERVER	Command	OnInternetStopServer()
	Command Update	OnUpdateInternetStopServer(CCmdUI* pCmdUI)
ID_INTERNET_CONFIGURATION	Command	OnInternetConfiguration()
	Command Update	OnUpdateInternetConfiguration(CCmdUI* pCmdUI)
ID_EDIT_CLEAR_ALL	Command	OnEditClearAll()
IDR_CONTEXT_MENU	WM_CONTEXTMENU	OnContextMenu(CWnd* pWnd, CPoint point)

Table 29.

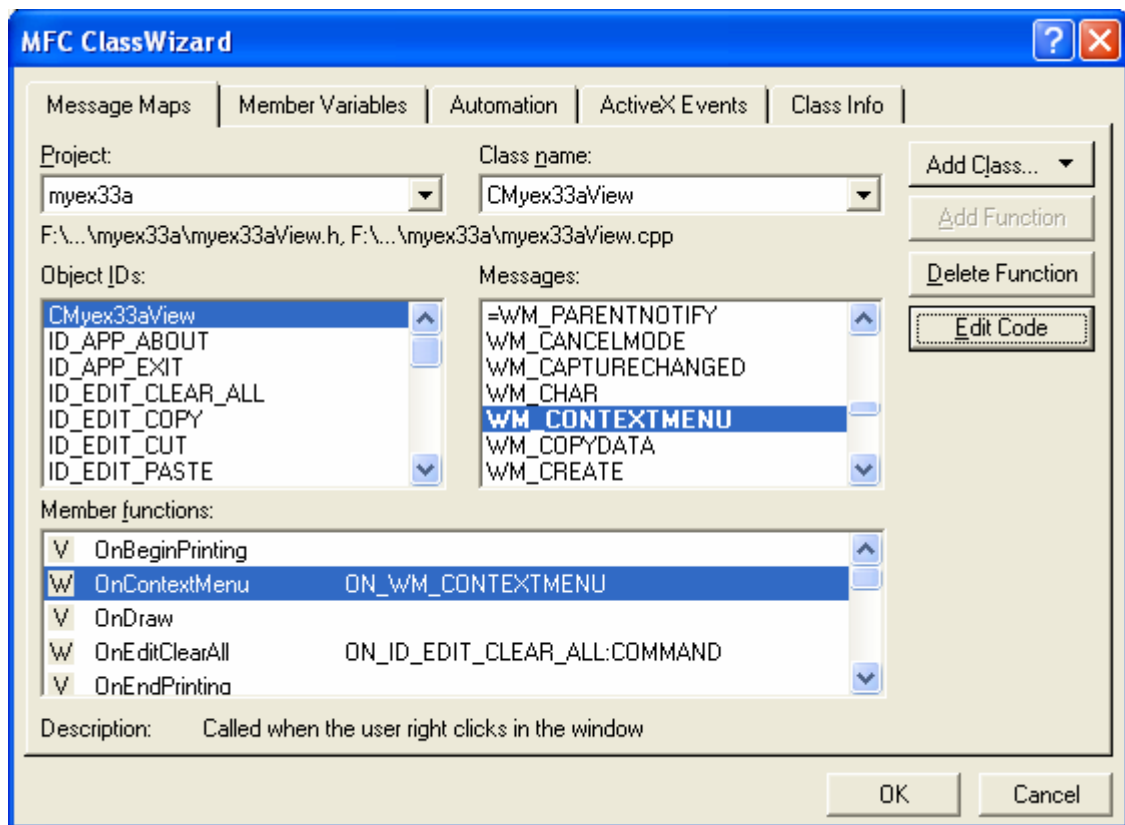


Figure 82: Adding command, command update and Windows message handler.

You can verify the result in **Myex33aView.h** as shown below.

```

// Generated message map functions
protected:
   //{{AFX_MSG(CMyex33aView)
    afx_msg void OnRequest();
    afx_msg void OnInternetStartServer();
    afx_msg void OnUpdateInternetStartServer(CCmdUI* pCmdUI);
    afx_msg void OnInternetRequestSocket();
    afx_msg void OnInternetRequestWininet();
    afx_msg void OnInternetStopServer();
    afx_msg void OnUpdateInternetStopServer(CCmdUI* pCmdUI);
    afx_msg void OnInternetConfiguration();
    afx_msg void OnUpdateInternetConfiguration(CCmdUI* pCmdUI);
    afx_msg void OnEditClearAll();
    afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
   //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

```

Listing 24.

And in **Myex33aView.cpp** as shown below.

```

BEGIN_MESSAGE_MAP(CMyex33aView, CEditView)
   //{{AFX_MSG_MAP(CMyex33aView)
    ON_BN_CLICKED(IDC_REQUEST, OnRequest)
    ON_COMMAND(ID_INTERNET_START_SERVER, OnInternetStartServer)
    ON_UPDATE_COMMAND_UI(ID_INTERNET_START_SERVER, OnUpdateInternetStartServer)
    ON_COMMAND(ID_INTERNET_REQUEST SOCK, OnInternetRequestSocket)
    ON_COMMAND(ID_INTERNET_REQUEST_INET, OnInternetRequestWininet)
    ON_COMMAND(ID_INTERNET_STOP_SERVER, OnInternetStopServer)
    ON_UPDATE_COMMAND_UI(ID_INTERNET_STOP_SERVER, OnUpdateInternetStopServer)
    ON_COMMAND(ID_INTERNET_CONFIGURATION, OnInternetConfiguration)
    ON_UPDATE_COMMAND_UI(ID_INTERNET_CONFIGURATION, OnUpdateInternetConfiguration)
    ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
    ON_WM_CONTEXTMENU()
   //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CEditView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CEditView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CEditView::OnFilePrintPreview)
END_MESSAGE_MAP()

```

Listing 25.

Next, add the implementation codes to **CMyex33aView.cpp**. Don't forget the related header files that need to be included as shown below.

```

// myex33aView.cpp : implementation of the CMyex33aView class
//

#include "stdafx.h"
#include "myex33a.h"
#include "MainFrm.h"

#include "myex33aDoc.h"
#include "myex33aView.h"
#include "Utility.h"
#include "Sheetconfig.h"
#include "Blocksock.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

```

```

extern CBlockingSocket g_sListen;

////////////////////////////////////
// CMyex33aView

IMPLEMENT_DYNCREATE(CMyex33aView, CEditView)

BEGIN_MESSAGE_MAP(CMyex33aView, CEditView)
    //{AFX_MSG_MAP(CMyex33aView)
    ON_BN_CLICKED(IDC_REQUEST, OnRequest)
    ON_COMMAND(ID_INTERNET_START_SERVER, OnInternetStartServer)
    ON_UPDATE_COMMAND_UI(ID_INTERNET_START_SERVER, OnUpdateInternetStartServer)
    ON_COMMAND(ID_INTERNET_REQUEST_SOCKET, OnInternetRequestSocket)
    ON_COMMAND(ID_INTERNET_REQUEST_INET, OnInternetRequestWininet)
    ON_COMMAND(ID_INTERNET_STOP_SERVER, OnInternetStopServer)
    ON_UPDATE_COMMAND_UI(ID_INTERNET_STOP_SERVER, OnUpdateInternetStopServer)
    ON_COMMAND(ID_INTERNET_CONFIGURATION, OnInternetConfiguration)
    ON_UPDATE_COMMAND_UI(ID_INTERNET_CONFIGURATION, OnUpdateInternetConfiguration)
    ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
    //}}AFX_MSG_MAP
    ON_WM_CONTEXTMENU()
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CEditView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CEditView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CEditView::OnFilePrintPreview)
END_MESSAGE_MAP()

////////////////////////////////////
// CMyex33aView construction/destruction

CMyex33aView::CMyex33aView()
{
    // TODO: add construction code here
}

CMyex33aView::~CMyex33aView()
{
}

BOOL CMyex33aView::PreCreateWindow(CREATESTRUCT& cs)
{
    BOOL bPreCreated = CEditView::PreCreateWindow(cs);
    cs.style &= ~(ES_AUTOHSCROLL|WS_HSCROLL); // Enable word-wrapping
    cs.style |= ES_READONLY;
    return bPreCreated;
}

////////////////////////////////////
// CMyex33aView drawing

void CMyex33aView::OnDraw(CDC* pDC)
{
    CMyex33aDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
}

////////////////////////////////////
// CMyex33aView printing

BOOL CMyex33aView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default CEditView preparation
    return CEditView::OnPreparePrinting(pInfo);
}

```

```

void CMyex33aView::OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo)
{
    // Default CEditView begin printing.
    CEditView::OnBeginPrinting(pDC, pInfo);
}

void CMyex33aView::OnEndPrinting(CDC* pDC, CPrintInfo* pInfo)
{
    // Default CEditView end printing
    CEditView::OnEndPrinting(pDC, pInfo);
}

////////////////////////////////////
// CMyex33aView diagnostics

#ifdef _DEBUG
void CMyex33aView::AssertValid() const
{
    CEditView::AssertValid();
}

void CMyex33aView::Dump(CDumpContext& dc) const
{
    CEditView::Dump(dc);
}

CMyex33aDoc* CMyex33aView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMyex33aDoc));
    return (CMyex33aDoc*)m_pDocument;
}
#endif // _DEBUG

////////////////////////////////////
// CMyex33aView message handlers

void CMyex33aView::OnInternetStartServer()
{
    try {
        CSockAddr saServer;
        if(g_strIPServer.IsEmpty()) { // first or only IP
            saServer = CSockAddr(INADDR_ANY, (USHORT) g_nPortServer);
        }
        else { // if our computer has multiple IP addresses...
            saServer = CSockAddr(g_strIPServer, (USHORT) g_nPortServer);
        }
        g_sListen.Create();
        g_sListen.Bind(saServer);
        g_sListen.Listen();// start listening
        g_bListening = TRUE;
        g_nConnection = 0;
        AfxBeginThread(ServerThreadProc, GetSafeHwnd(), THREAD_PRIORITY_NORMAL);
    }
    catch(CBlockingSocketException* e) {
        g_sListen.Cleanup();
        LogBlockingSocketException(GetSafeHwnd(), "VIEW:", e);
        e->Delete();
    }
}

void CMyex33aView::OnUpdateInternetStartServer(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!g_bListening);
}

```

```

void CMyex33aView::OnInternetRequestSocket()
{
    AfxBeginThread(ClientSocketThreadProc, GetSafeHwnd(), THREAD_PRIORITY_NORMAL);
}

void CMyex33aView::OnInternetRequestWininet()
{
    AfxBeginThread(ClientWinInetThreadProc, GetSafeHwnd(), THREAD_PRIORITY_NORMAL);
}

void CMyex33aView::OnInternetStopServer()
{
    try {
        if(g_bListening) {
            g_sListen.Close();
        }
    }
    catch(CBlockingSocketException* e) {
        LogBlockingSocketException(GetSafeHwnd(), "VIEW:", e);
        e->Delete();
    }
}

void CMyex33aView::OnUpdateInternetStopServer(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(g_bListening);
}

void CMyex33aView::OnInternetConfiguration()
{
    CSheetConfig sh("Configuration");
    sh.m_pageServer.m_strDirect = g_strDirect;
    sh.m_pageServer.m_strDefault = g_strDefault;
    sh.m_pageServer.m_nPortServer = g_nPortServer;
    sh.m_pageClient.m_strServerIP = g_strServerIP;
    sh.m_pageClient.m_nPort = g_nPort;
    sh.m_pageClient.m_strServerName = g_strServerName;
    sh.m_pageClient.m_strFile = g_strFile;
    sh.m_pageClient.m_strProxy = g_strProxy;
    sh.m_pageClient.m_bUseProxy = g_bUseProxy;
    sh.m_pageAdv.m_strIPClient = g_strIPClient;
    sh.m_pageAdv.m_strIPServer = g_strIPServer;
    if(sh.DoModal() == IDOK) {
        g_strDirect = sh.m_pageServer.m_strDirect;
        g_strDefault = sh.m_pageServer.m_strDefault;
        g_nPortServer = sh.m_pageServer.m_nPortServer;
        g_strServerIP = sh.m_pageClient.m_strServerIP;
        g_nPort = sh.m_pageClient.m_nPort;
        g_strServerName = sh.m_pageClient.m_strServerName;
        if(sh.m_pageClient.m_strFile.IsEmpty()) {
            g_strFile = "/";
        }
        else {
            g_strFile = sh.m_pageClient.m_strFile;
        }
        g_strProxy = sh.m_pageClient.m_strProxy;
        g_bUseProxy = sh.m_pageClient.m_bUseProxy;
        g_strIPClient = sh.m_pageAdv.m_strIPClient;
        g_strIPServer = sh.m_pageAdv.m_strIPServer;
        if(!g_strIPClient.IsEmpty() && g_bUseProxy) {
            AfxMessageBox("Warning: you can't assign a client IP address if "
                "you are using a proxy server");
        }
        if(!g_strServerIP.IsEmpty() && g_bUseProxy) {
            AfxMessageBox("Warning: you must specify the server by name if "
                "you are using a proxy server");
        }
    }
}

```



```

    }
    if(g_strServerIP.IsEmpty() && g_strServerName.IsEmpty()) {
        AfxMessageBox("Warning: you must specify either a server name or "
            "a server IP address");
    }
    if(!g_strServerIP.IsEmpty() && !g_strServerName.IsEmpty()) {
        AfxMessageBox("Warning: you cannot specify both a server name "
            "and a server IP address");
    }
    ::SetCurrentDirectory(g_strDirect);
}

}

void CMyex33aView::OnUpdateInternetConfiguration(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!g_bListening);
}

void CMyex33aView::OnEditClearAll()
{
    SetWindowText("");
}

void CMyex33aView::OnRequest()
{
    CWnd& rBar = ((CMainFrame*) AfxGetApp()->m_pMainWnd)->m_wndDialogBar;
    // g_strURL: thread sync?
    rBar.GetDlgItemText(IDC_URL, g_strURL);
    TRACE("CMyex33aView::OnRequest -- URL = %s\n", (const char*) g_strURL);
    AfxBeginThread(ClientUrlThreadProc, GetSafeHwnd(), THREAD_PRIORITY_NORMAL);
}

void CMyex33aView::OnContextMenu(CWnd* pWnd, CPoint point)
{
    // clear-all menu activated on right button
    CMenu menu;
    menu.LoadMenu(IDR_CONTEXT_MENU);
    menu.GetSubMenu(0)->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
        point.x, point.y, this);
}

```

Listing 26.

Create a directory named **WEBSITE** under your project directory and copy the [following sample files](#). You can create your own files if needed, provided the html file names are same, else you have to do some source code modifications.

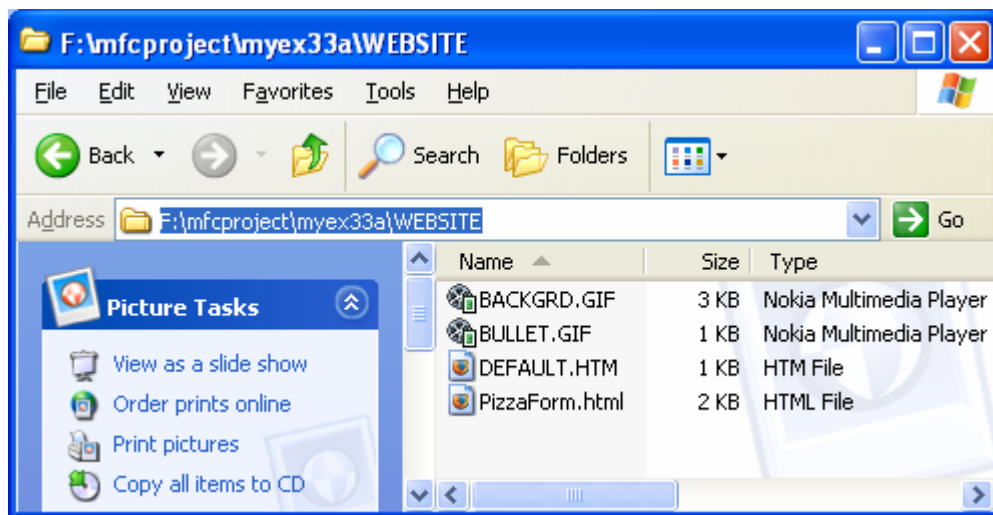


Figure 83: Creating the root directory of the web for MYEX33A.

Build and run this program. Let start our web server by selecting the **Internet Start Server** menu and make sure the IIS/World Wide Web is not running, else you need to stop it.

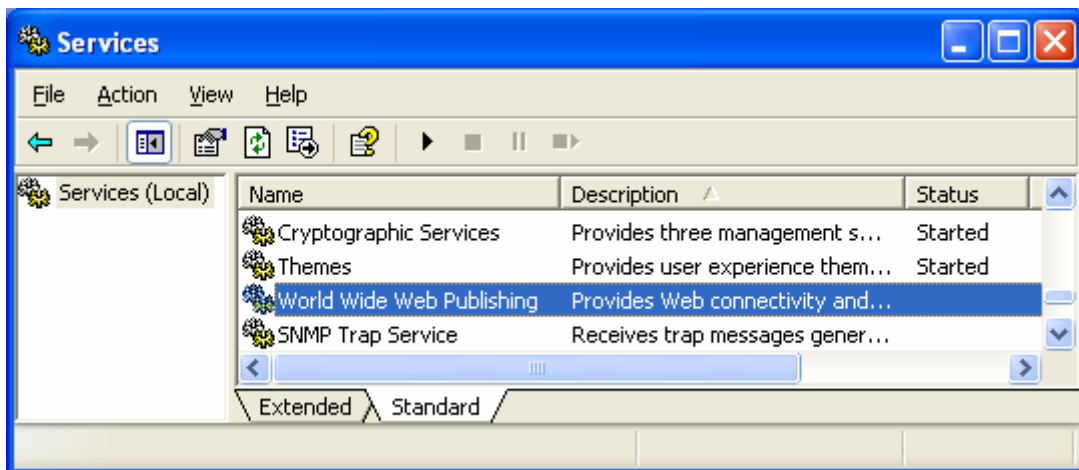


Figure 84: The IIS/WWW Publishing service was stopped to give a way for our own web server application.

Next, start the server.

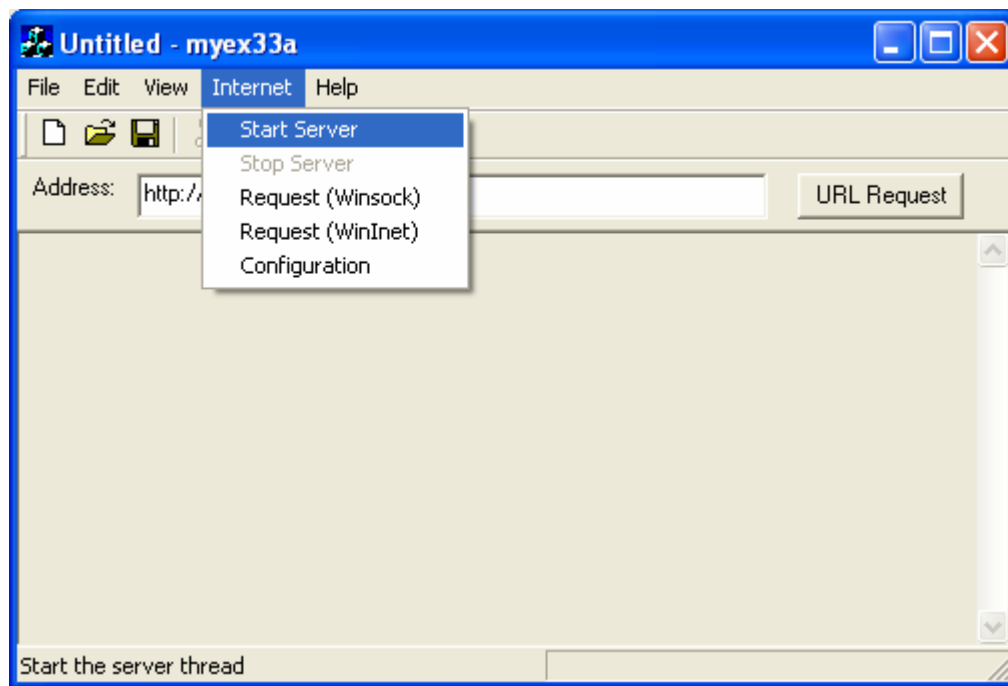


Figure 85: MYEX33A in action, starting the web server.

It seems that our status indicator not working lol! Forget it; next, select the **Request (Winsock)** client menu.

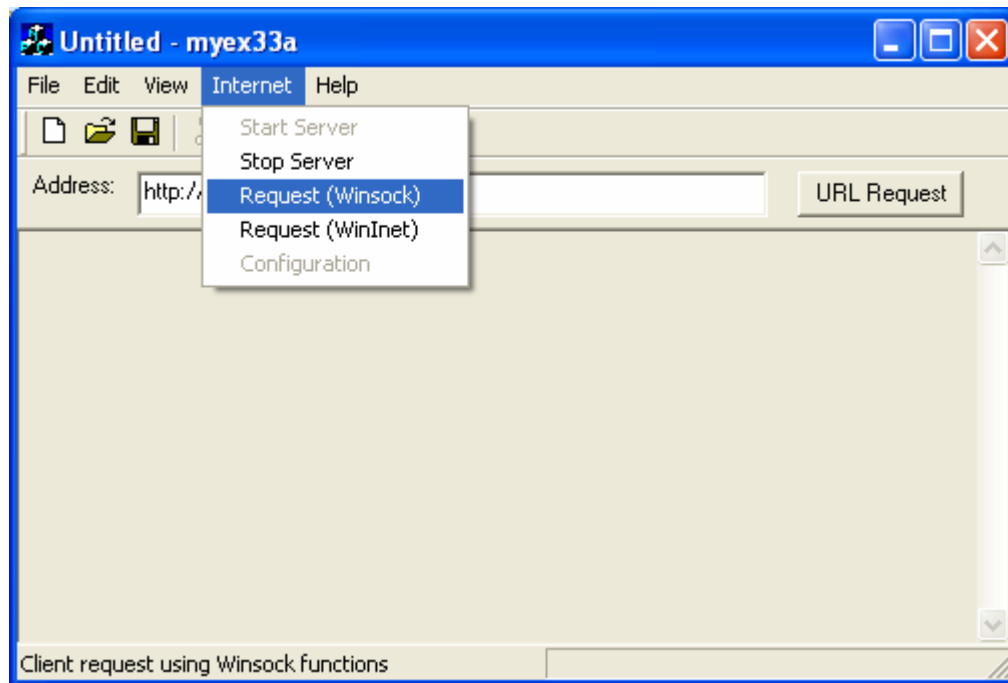


Figure 86: Testing the Winsock client request

The following message should be displayed.

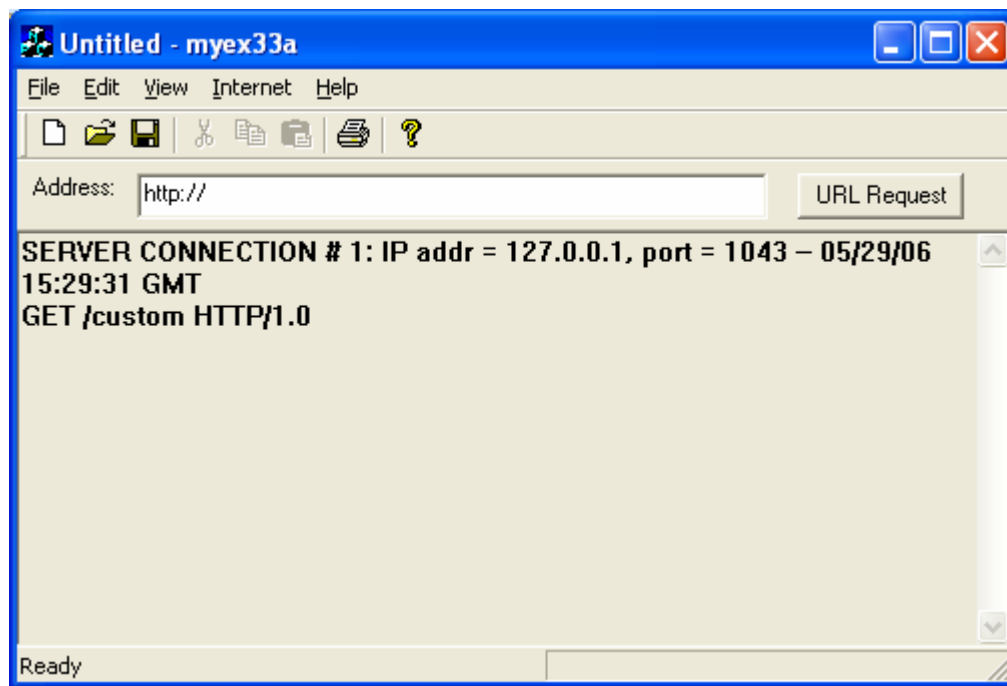


Figure 87: The Winsock client message.

Together with the following dialog (the html source code).

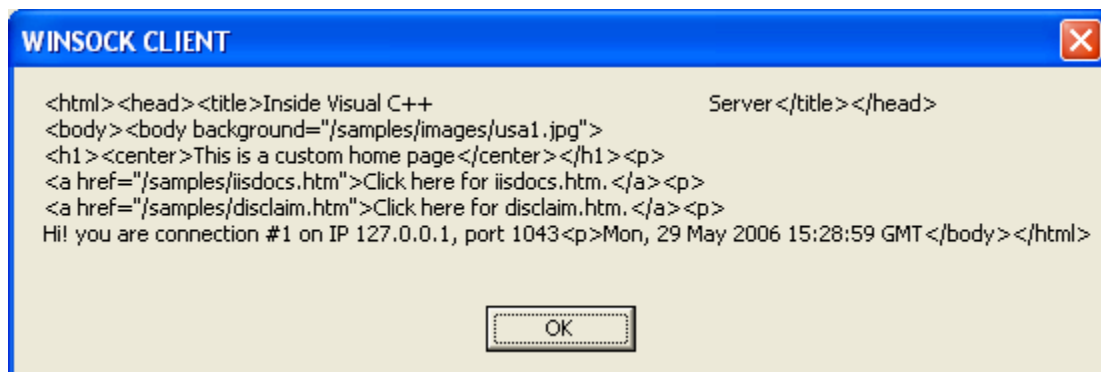


Figure 88: Winsock client request html code.

Next, test the **Request (WinInet)** client menu.

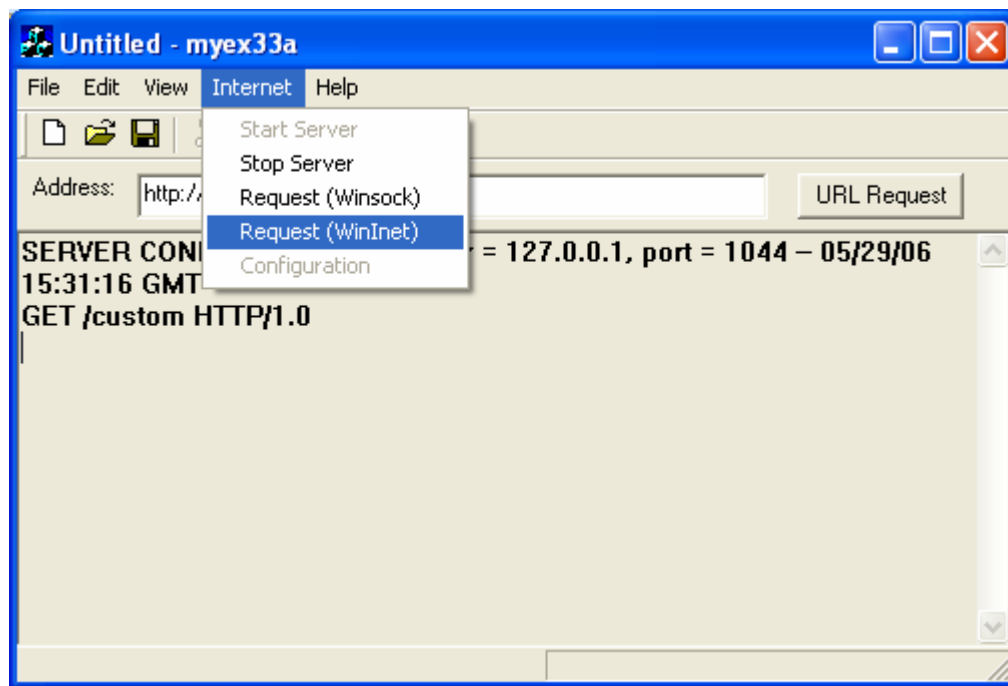


Figure 89: Testing the WinInet client request.

The **WinInet** client html source should be displayed.

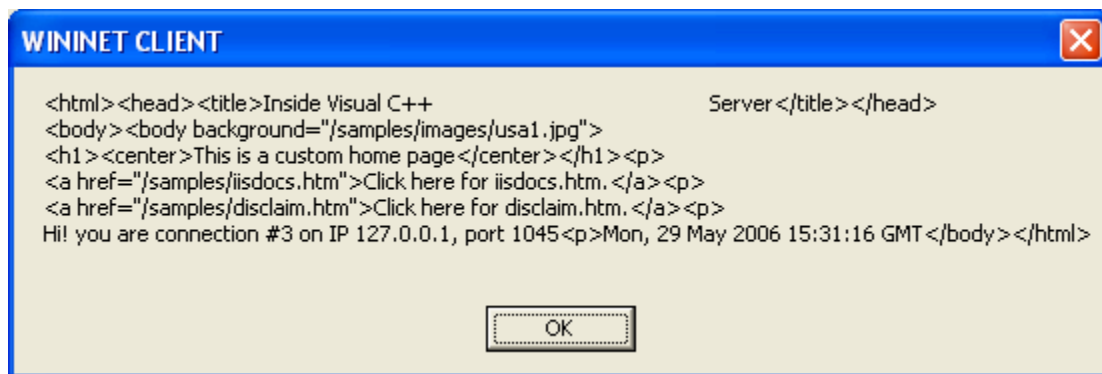


Figure 90: The WinInet client request html source code.

And the **WinInet** message is shown below.

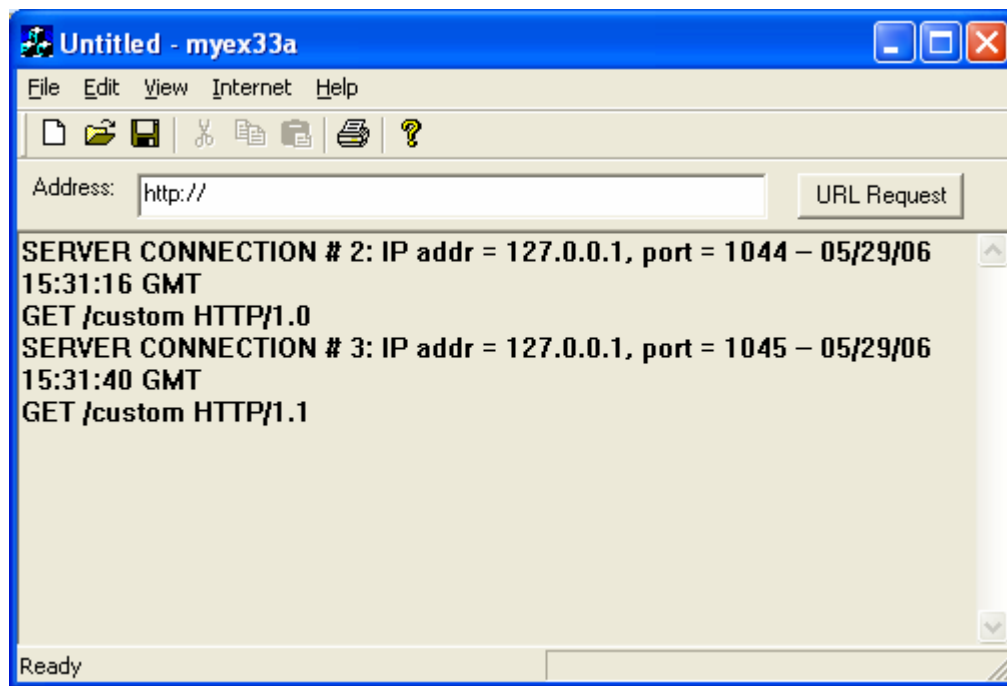


Figure 91: The WinInet client request message.

To change the configuration through the **Configuration** menu, you have to stop the server (select **Stop Server** menu). The following is the property page of the **Configuration** menu.

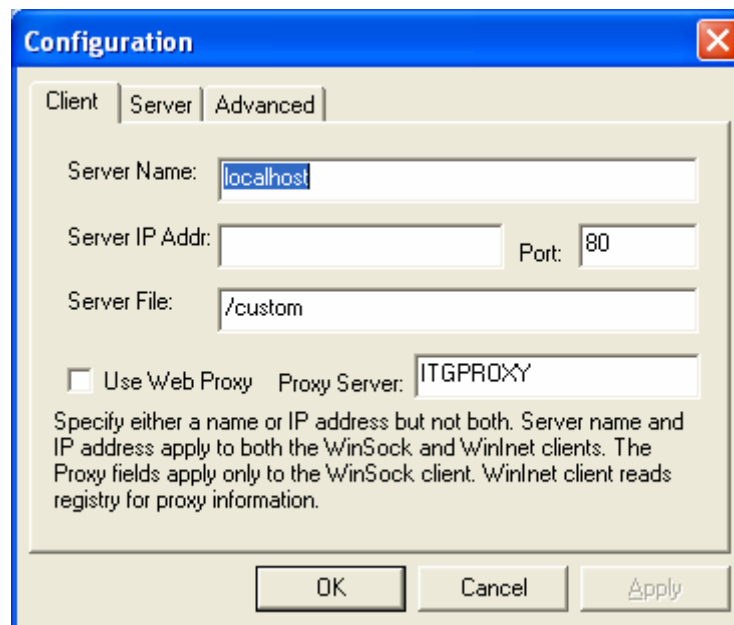


Figure 92: The Configuration property page.

You can clear all the child window by using the **Clear All** context menu by right clicking anywhere in the child window.

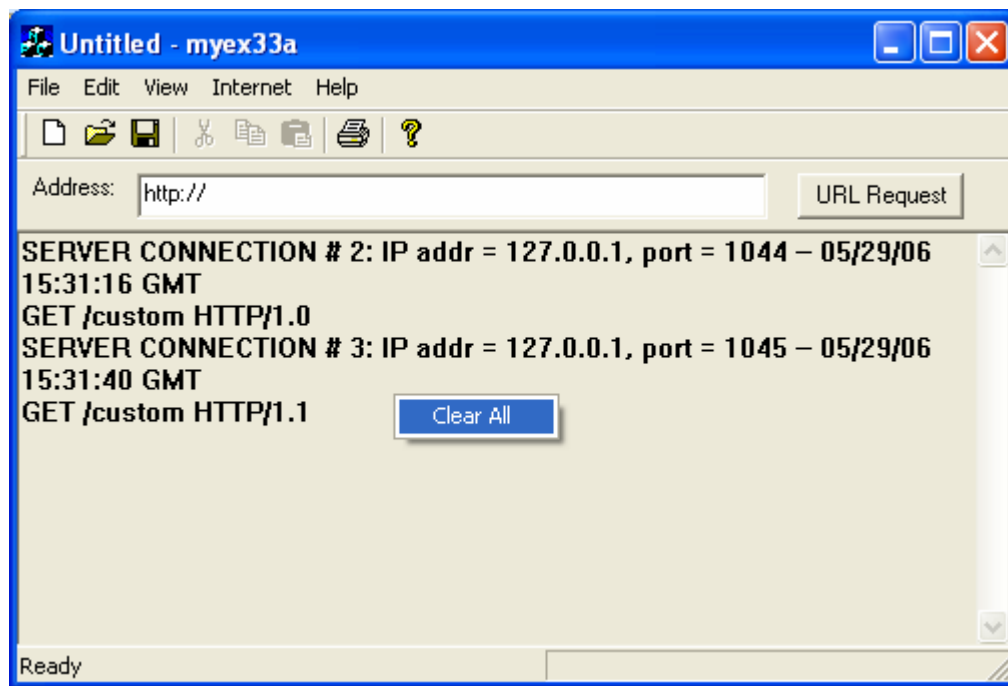


Figure 93: **Clear All** context menu in action.

Finally let test the **Address** bar. Make sure you are online. Type any URL such as shown below. Just click the **OK** for the warning dialog as shown below, the html size not fit to the declared buffer size in our code.

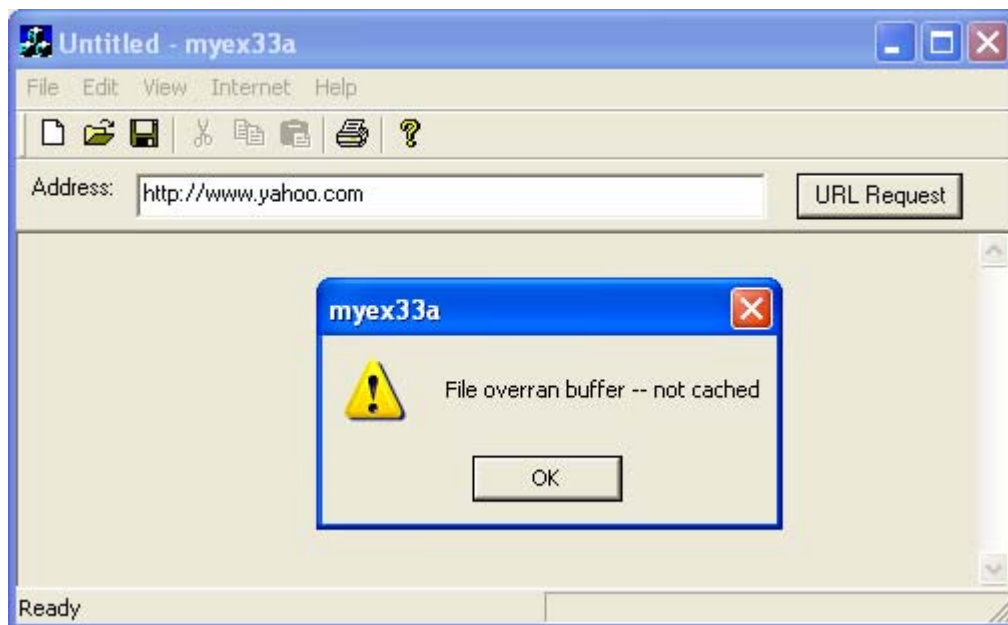


Figure 94: Testing the URL client request.

The following is the URL client html source.



Figure 95: www.yahoo.com html source code.

Try our own test page: <http://localhost/>.

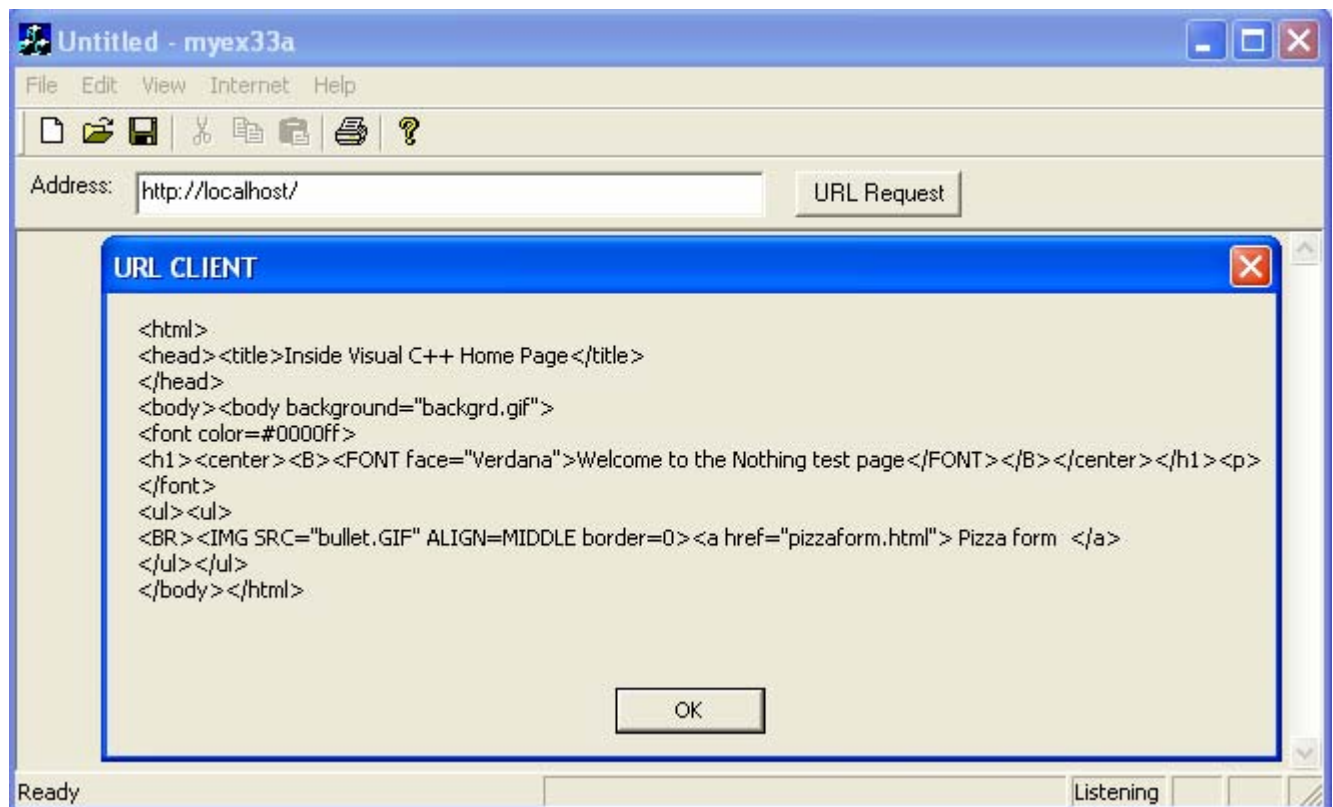


Figure 96: The local test page

Check your project Debug window to see the previous activities. Well, that all!

Back to the Story

Building and Testing MYEX33A

Open the myex33a project in Visual C++, and then build the project. A directory under MYEX33A, called **Website**, contains some HTML files and is set up as the MYEX33A server's home directory, which appears to clients as the server's root directory.

If you have another HTTP server running on your computer, stop it now. If you have installed IIS along with Windows NT Server, it is probably running now, so you must run the **Internet Service Manager (Internet Information Services)** program from the **Administrative Tools** menu. Select the web site folder for example, Default Web Site as shown below and then click the stop button (the one with the square). MYEX33A reports a bind error (10048) if another server is already listening on port 80.

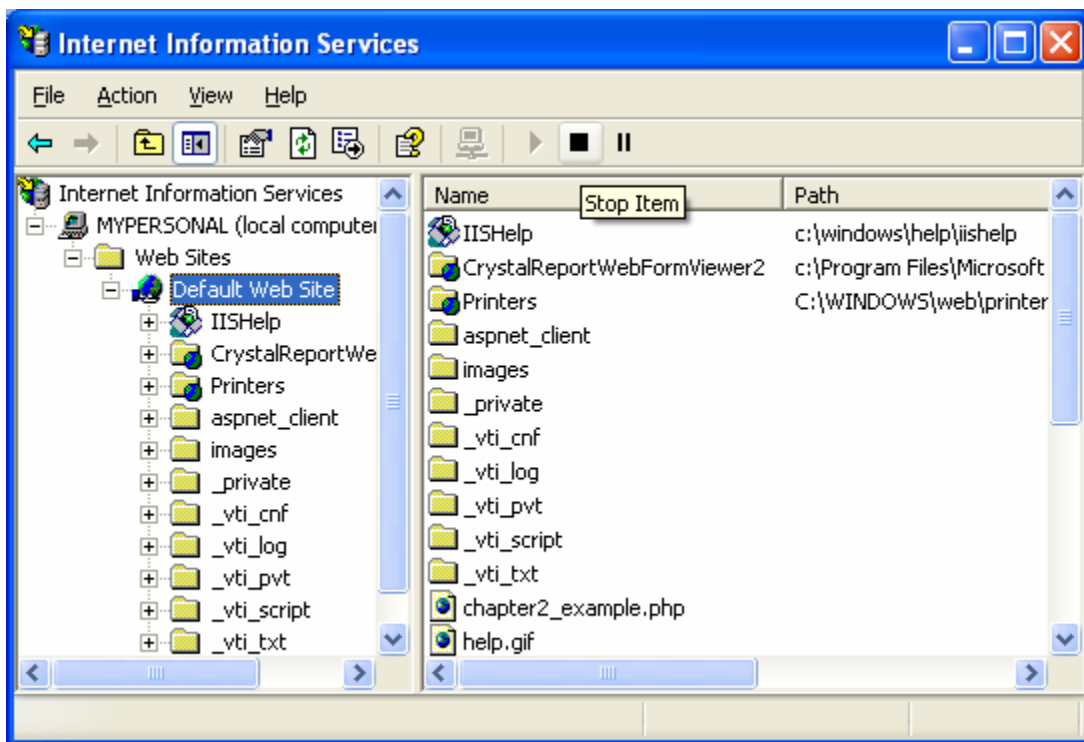


Figure 97: Stopping the web server.

Run the program from the debugger, and then choose **Start Server** from the **Internet** menu.

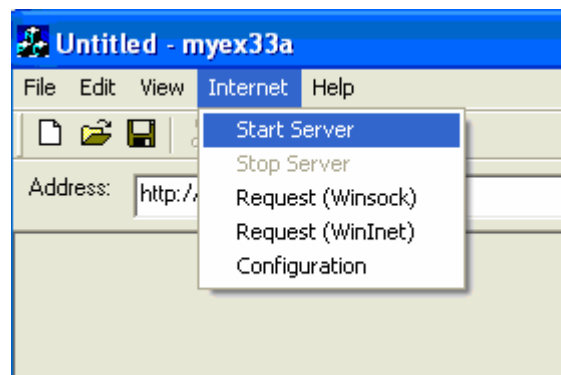


Figure 98: Starting our own, MYEX33A web server.

Now go to your Web browser and type **localhost**. You should see the **Welcome to the Nothing test page** complete with simple graphics.

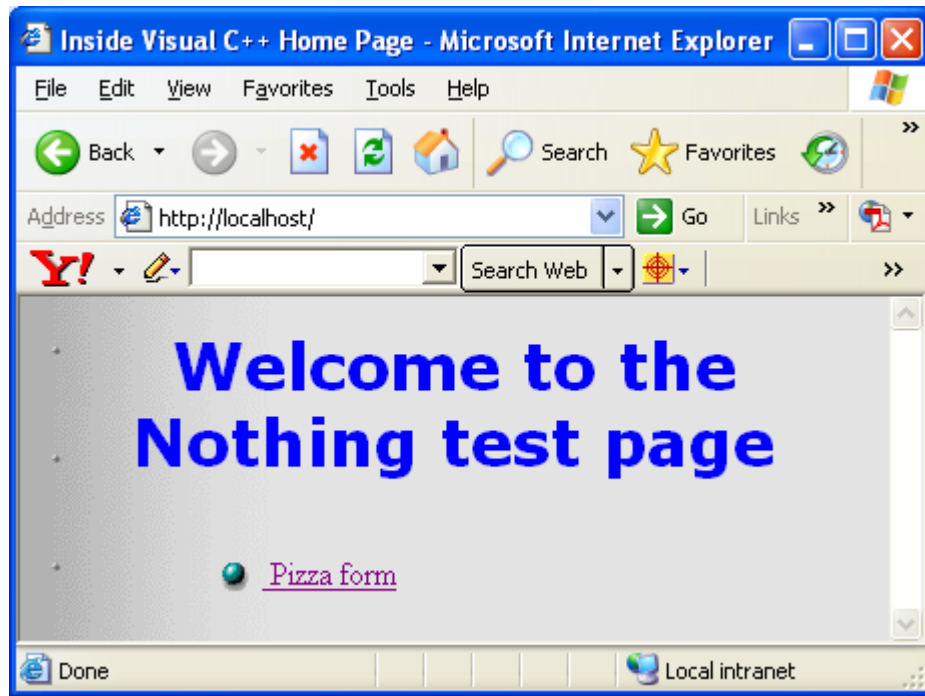


Figure 99: MYEX33A – accessing web through browser.

The MYEX33A window should look like this.

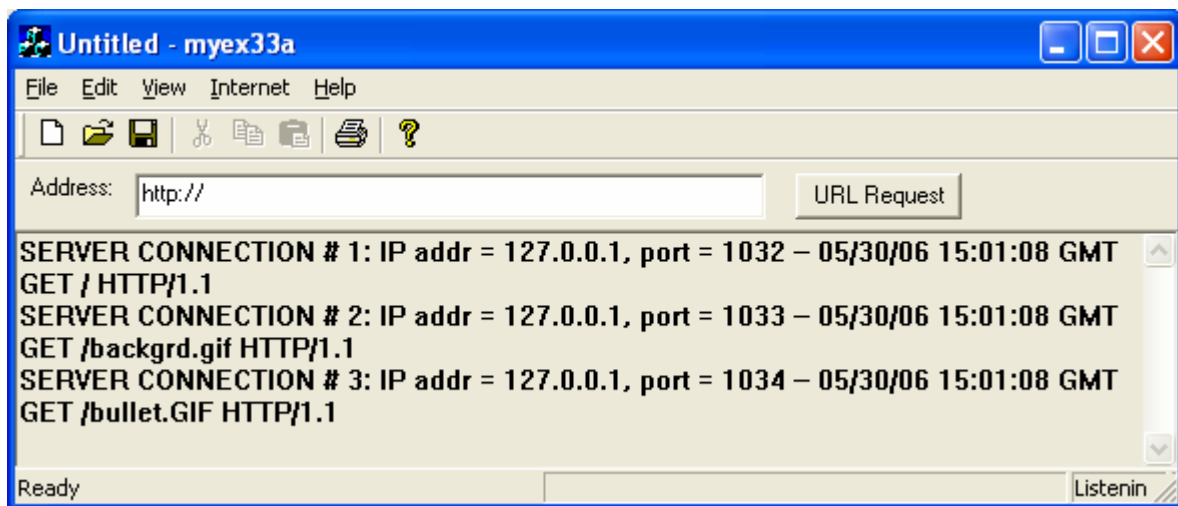


Figure 100: MYEX33A – the web server activities.

Look at the Visual C++ debug window for a listing of the client's request headers. If you click the browser's **Refresh** button, you might notice MYEX33A error messages like this:

```
WINSOCK ERROR--SERVER: Send error #10054 -- 10/05/96 04:34:10 GMT
```

This tells you that the browser read the file's modified date from the server's response header and figured out that it didn't need the data because it already had the file in its cache. The browser then closed the socket, and the server

detected an error. If the MYEX33A server were smarter, it would have checked the client's If-Modified-Since request header before sending the file.

Using Telnet

The [Telnet](#) utility is included with Windows Operating Systems. It's useful for testing server programs such as MYEX33A. With Telnet, you're sending one character at a time, which means that the server's `CBlockingSocket::Receive` function is receiving one character at a time. The Telnet window is shown here.

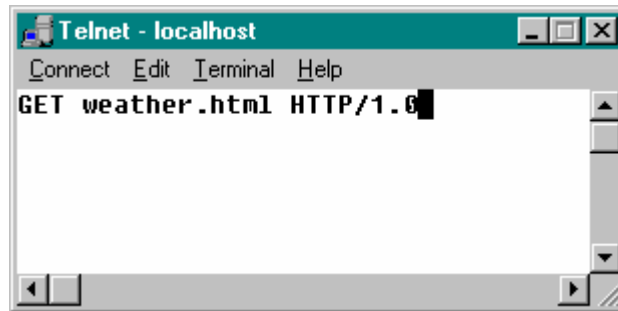


Figure 101: Using Telnet to grab a web page.

The first time you run Telnet, choose **Preferences** from the **Terminal** menu and turn on **Local Echo**. Each time thereafter, choose **Remote System** from the **Connect** menu and then type your server name and port number 80. You can type a GET request (followed by a double carriage return), but you'd better type fast because the MYEX33A server's `Receive()` calls are set to time-out after 10 seconds.

Building a Web Client with `CHttpBlockingSocket`

If you had written your own Internet browser program a few years ago, you could have made a billion dollars by now. But these days, you can download browsers for free, so it doesn't make sense to write one. It does make sense, however, to add Internet access features to your Windows applications. Winsock is not the best tool if you need HTTP or FTP access only, but it's a good learning tool.

The MYEX33A Winsock Client

The MYEX33A program implements a Winsock client in the file `ClientSockThread.cpp`. The code is similar to the code for the simplified HTTP client. The client thread uses global variables set by the **Configuration** property sheet, including server filename, server host name, server IP address and port, and client IP address. The client IP address is necessary only if your computer supports multiple IP addresses. When you run the client, it connects to the specified server and issues a GET request for the file that you specified. The Winsock client logs error messages in the MYEX33A main window.

MYEX33A Support for Proxy Servers

If your computer is connected to a LAN at work, chances are it's not exposed directly to the Internet but rather connected through a proxy server, sometimes called a firewall (or proxy with firewall). There are two kinds of proxy servers: Web and Winsock. Web proxy servers, sometimes called **CERN proxies**, support only the HTTP, FTP, and gopher protocols. (The gopher protocol, which predates HTTP, allows character-mode terminals to access Internet files.) A Winsock client program must be specially adapted to use a Web proxy server. A Winsock proxy server is more flexible and thus can support protocols such as RealAudio. Instead of modifying your client program source code, you link to a special **Remote Winsock DLL** that can communicate with a Winsock proxy server.

The MYEX33A client code can communicate through a Web proxy if you check the **Use Proxy** check box in the **Client Configuration** page. In that case, you must know and enter the name of your proxy server. From that point on, the client code connects to the proxy server instead of to the real server. All GET and POST requests must then specify the full Uniform Resource Locator (URL) for the file. If you were connected directly to SlowSoft's server, for example, your GET request might look like this:

```
GET /customers/newproducts.html HTTP/1.0
```

But if you were connected through a Web proxy server, the GET would look like this:

```
GET http://slowsoft.com/customers/newproducts.html HTTP/1.0
```

Testing the MYEX33A Winsock Client

The easiest way to test the Winsock client is by using the built-in Winsock server. Just start the server as before, and then choose **Request** (Winsock) from the **Internet** menu. You should see some HTML code in a message box. You can also test the client against IIS, the server running in another MYEX33A process on the same computer, the MYEX33A server running on another computer on the Net, and an Internet server. Ignore the "Address" URL on the dialog bar for the time being; it's for one of the WinInet clients. You must enter the server name and filename in the **Client** page of the **Configuration** dialog.

WinInet

WinInet is a higher-level API than Winsock, but **it works only for HTTP, FTP, and gopher client programs** in both asynchronous and synchronous modes. You can't use it to build servers. The **WININET** DLL is independent of the **WINSOCK32** DLL. Microsoft Internet Explorer 3.0 (IE3) uses WinInet, and so do ActiveX controls.

WinInet's Advantages over Winsock

WinInet far surpasses Winsock in the support it gives to a professional-level client program. Following are just some of the WinInet benefits:

- Caching: Just like IE3, your WinInet client program caches HTML files and other Internet files. You don't have to do a thing. The second time your client requests a particular file, it's loaded from a local disk instead of from the Internet.
- Security: WinInet supports basic authentication, Windows NT challenge/response authentication, and the **Secure Sockets Layer** (SSL). Authentication is described in [Module 33](#).
- Web proxy access: You enter proxy server information through the **Control Panel** (click on the Internet icon), and it's stored in the Registry. WinInet reads the Registry and uses the proxy server when required.
- Buffered I/O: WinInet's read function doesn't return until it can deliver the number of bytes you asked for. (It returns immediately, of course, if the server closes the socket.) Also, you can read individual text lines if you need to.
- Easy API: Status callback functions are available for UI update and cancellation. One function, `CInternetSession::OpenURL`, finds the server's IP address, opens a connection, and makes the file ready for reading, all in one call. Some functions even copy Internet files directly to and from disk.
- User friendly: WinInet parses and formats headers for you. If a server has moved a file to a new location, it sends back the new URL in an HTTP Location header. WinInet seamlessly accesses the new server for you. In addition, WinInet puts a file's modified date in the request header for you.

The MFC winInet Classes

WinInet is a modern API available only for Win32. The MFC wrapping is quite good, which means we didn't have to write our own WinInet class library. Yes, MFC WinInet supports blocking calls in multithreaded programs, and by now you know that makes us happy.

The MFC classes closely mirror the underlying WinInet architecture, and they add exception processing. These classes are summarized in the sections on the following pages.

CInternetSession

You need only one `CInternetSession` object for each thread that accesses the Internet. After you have your `CInternetSession` object, you can establish HTTP, FTP, or gopher connections or you can open remote files directly by calling the `OpenURL()` member function. You can use the `CInternetSession` class directly, or you can derive a class from it in order to support status callback functions.

The `CInternetSession` constructor calls the WinInet `InternetOpen()` function, which returns an `HINTERNET` session handle that is stored inside the `CInternetSession` object. This function initializes your application's use of the WinInet library, and the session handle is used internally as a parameter for other WinInet calls.

CURLConnection

An object of class `CURLConnection` represents a "permanent" HTTP connection to a particular host. You know already that HTTP doesn't support permanent connections and that FTP doesn't either. (The connections last only for the duration of a file transfer.) WinInet gives the appearance of a permanent connection because it remembers the host name.

After you have your `CInternetSession` object, you call the `GetURLConnection()` member function, which returns a pointer to a `CURLConnection` object. (Don't forget to delete this object when you are finished with it.) The `GetURLConnection()` member function calls the WinInet `InternetConnect()` function, which returns an `HINTERNET` connection handle that is stored inside the `CURLConnection` object and used for subsequent WinInet calls.

CFTPConnection, CGopherConnection

These classes are similar to `CURLConnection`, but they use the FTP and gopher protocols. The `CFTPConnection` member functions `GetFile()` and `PutFile()` allow you to transfer files directly to and from your disk.

CInternetFile

With HTTP, FTP, or gopher, your client program reads and writes byte streams. The MFC WinInet classes make these byte streams look like ordinary files. If you look at the class hierarchy, you'll see that `CInternetFile` is derived from `CStdioFile`, which is derived from `CFile`. Therefore, `CInternetFile` and its derived classes override familiar `CFile` functions such as `Read()` and `Write()`. For FTP files, you use `CInternetFile` objects directly, but for HTTP and gopher files, you use objects of the derived classes `CHttpFile` and `CGopherFile`. You don't construct a `CInternetFile` object directly, but you call `CFTPConnection::OpenFile` to get a `CInternetFile` pointer.

If you have an ordinary `CFile` object, it has a 32-bit `HANDLE` data member that represents the underlying disk file. A `CInternetFile` object uses the same `m_hFile` data member, but that data member holds a 32-bit Internet file handle of type `HINTERNET`, which is not interchangeable with a `HANDLE`. The `CInternetFile` overridden member functions use this handle to call WinInet functions such as `InternetReadFile()` and `InternetWriteFile()`.

CHttpFile

This Internet file class has member functions that are unique to HTTP files, such as `AddRequestHeaders()`, `SendRequest()`, and `GetFileURL()`. You don't construct a `CHttpFile` object directly, but you call the `CURLConnection::OpenRequest` function, which calls the WinInet function `HttpOpenRequest()` and returns a `CHttpFile` pointer. You can specify a GET or POST request for this call. Once you have your `CHttpFile` pointer, you call the `CHttpFile::SendRequest` member function, which actually sends the request to the server. Then you call `Read()`.

CFTPFileFind, CGopherFileFind

These classes let your client program explore FTP and gopher directories.

CInternetException

The MFC WinInet classes throw `CInternetException` objects that your program can process with try/catch logic.

Internet Session Status Callbacks

WinInet and MFC provide callback notifications as a WinInet operation progresses, and these status callbacks are available in both synchronous (blocking) and asynchronous modes. In synchronous mode (which we're using exclusively

here), your WinInet calls block even though you have status callbacks enabled. Callbacks are easy in C++. You simply derive a class and override selected virtual functions. The base class for WinInet is CInternetSession. Now let's derive a class named CCallbackInternetSession:

```
class CCallbackInternetSession : public CInternetSession
{
public:
    CCallbackInternetSession( LPCTSTR pstrAgent = NULL, DWORD dwContext =
1,
        DWORD dwAccessType = PRE_CONFIG_INTERNET_ACCESS,
        LPCTSTR pstrProxyName = NULL, LPCTSTR pstrProxyBypass = NULL,
        DWORD dwFlags = 0 ) { EnableStatusCallback() }
protected:
    virtual void OnStatusCallback(DWORD dwContext, DWORD dwInternalStatus,
        LPVOID lpvStatusInformation, DWORD dwStatusInformationLength);
};
```

The only coding that's necessary is a constructor and a single overridden function, OnStatusCallback(). The constructor calls CInternetSession::EnableStatusCallback to enable the status callback feature. Your WinInet client program makes its various Internet blocking calls, and when the status changes, OnStatusCallback() is called. Your overridden function quickly updates the UI and returns, and then the Internet operation continues. For HTTP, most of the callbacks originate in the CHttpFile::SendRequest function. What kind of events trigger callbacks? A list of the codes passed in the dwInternalStatus parameter is shown here.

Code Passed	Action Taken
INTERNET_STATUS_RESOLVING_NAME	Looking up the IP address of the supplied name. The name is now in lpvStatusInformation.
INTERNET_STATUS_NAME_RESOLVED	Successfully found the IP address. The IP address is now in lpvStatusInformation.
INTERNET_STATUS_CONNECTING_TO_SERVER	Connecting to the socket.
INTERNET_STATUS_CONNECTED_TO_SERVER	Successfully connected to the socket.
INTERNET_STATUS_SENDING_REQUEST	Send the information request to the server.
INTERNET_STATUS_REQUEST_SENT	Successfully sent the information request to the server.
INTERNET_STATUS_RECEIVING_RESPONSE	Waiting for the server to respond to a request.
INTERNET_STATUS_RESPONSE_RECEIVED	Successfully received a response from the server.
INTERNET_STATUS_CLOSING_CONNECTION	Closing the connection to the server.
INTERNET_STATUS_CONNECTION_CLOSED	Successfully closed the connection to the server.
INTERNET_STATUS_HANDLE_CREATED	Program can now close the handle.
INTERNET_STATUS_HANDLE_CLOSING	Successfully terminated this handle value.
INTERNET_STATUS_REQUEST_COMPLETE	Successfully completed the asynchronous operation.

Table 30.

You can use your status callback function to interrupt a WinInet operation. You could, for example, test for an event set by the main thread when the user cancels the operation.

A Simplified WinInet Client Program

And now for the WinInet equivalent of our Winsock client program that implements a blind GET request. Because you're using WinInet in blocking mode, you must put the code in a worker thread. That thread is started from a command handler in the main thread:

```
AfxBeginThread(ClientWinInetThreadProc, GetSafeHwnd());
```

Here's the client thread code:

```

CString g_strServerName = "localhost"; // or some other host name
UINT ClientWinInetThreadProc(LPVOID pParam)
{
    CInternetSession session;
    CHttpConnection* pConnection = NULL;
    CHttpFile* pFile1 = NULL;
    char* buffer = new char[MAXBUF];
    UINT nBytesRead = 0;
    try
    {
        pConnection = session.GetHttpConnection(g_strServerName, 80);
        pFile1 = pConnection->OpenRequest(1, "/"); // blind GET
        pFile1->SendRequest();
        nBytesRead = pFile1->Read(buffer, MAXBUF - 1);
        buffer[nBytesRead] = '\\0'; // necessary for message box
        char temp[10];
        if(pFile1->Read(temp, 10) != 0) {
            // makes caching work if read complete
            AfxMessageBox("File overran buffer - not cached");
        }
        AfxMessageBox(buffer);
    }
    catch(CInternetException* e)
    {
        // Log the exception
        e->Delete();
    }
    if(pFile1) delete pFile1;
    if(pConnection) delete pConnection;
    delete [] buffer;
    return 0;
}

```

The second Read () call needs some explanation. It has two purposes. If the first Read () doesn't read the whole file, that means that it was longer than MAXBUF -1. The second Read () will get some bytes, and that lets you detect the overflow problem. If the first Read () reads the whole file, you still need the second Read () to force WinInet to cache the file on your hard disk. Remember that WinInet tries to read all the bytes you ask it to, through the end of the file. Even so, you need to read 0 bytes after that.

Building a Web Client with the MFC WinInet Classes

There are two ways to build a Web client with WinInet. The first method, using the CHttpConnection class, is similar to the simplified WinInet client on the preceding page. The second method, using CInternetSession::OpenURL, is even easier. We'll start with the CHttpConnection version.

The MYEX33A WinInet Client #1: Using CHttpConnection

The MYEX33A program implements a WinInet client in the file **ClientInetThread.cpp**. Besides allowing the use of an IP address as well as a host name, the program uses a status callback function. That function, CCallbackInternetSession::OnStatusCallback in the file **Utility.cpp**, puts a text string in a global variable g_pchStatus, using a critical section for synchronization. The function then posts a user-defined message to the application's main window. The message triggers an **Update Command UI** handler (called by CWinApp::OnIdle), which displays the text in the second status bar text pane.

Testing the WinInet Client #1

To test the WinInet client #1, you can follow the same procedure you used to test the Winsock client. Note the status bar messages as the connection is made (failed in the example!). Note that the file appears more quickly the second time you request it.

The MYEX33A WinInet Client #2: Using OpenURL()

The MYEX33A program implements a different WinInet client in the file **ClientUrlThread.cpp**. This client uses the "Address" URL (that you type to access the Internet site). Here's the actual code:

```
CString g_strURL = "http:// ";

UINT ClientUrlThreadProc(LPVOID pParam)
{
    char* buffer = new char[MAXBUF];
    UINT nBytesRead = 0;

    CInternetSession session; // can't get status callbacks for OpenURL
    CStdioFile* pFile1 = NULL; // could call ReadString to get 1 line
    try {
        pFile1 = session.OpenURL(g_strURL, 0,
            INTERNET_FLAG_TRANSFER_BINARY|INTERNET_FLAG_KEEP_CONNECTION);
        // If OpenURL fails, we won't get past here
        nBytesRead = pFile1->Read(buffer, MAXBUF - 1);
        buffer[nBytesRead] = '\0'; // necessary for message box
        char temp[100];
        if(pFile1->Read(temp, 100) != 0)
        {
            // makes caching work if read complete
            AfxMessageBox("File overran buffer - not cached");
        }
        ::MessageBox(::GetTopWindow(::GetDesktopWindow()), buffer,
            "URL CLIENT", MB_OK);
    }
    catch(CInternetException* e)
    {
        LogInternetException(pParam, e);
        e->Delete();
    }
    if(pFile1) delete pFile1;
    delete [] buffer;
    return 0;
}
```

Note that OpenURL() returns a pointer to a CStdioFile object. You can use that pointer to call Read() as shown, or you can call ReadString() to get a single line. The file class does all the buffering. As in the previous WinInet client, it's necessary to call Read() a second time to cache the file. The OpenURL() INTERNET_FLAG_KEEP_CONNECTION parameter is necessary for Windows NT challenge/response authentication, which is described in [Module 33](#). If you added the flag INTERNET_FLAG_RELOAD, the program would bypass the cache just as the browser does when you click the Refresh() button.

Testing the WinInet Client #2

You can test the WinInet client #2 against any HTTP server. You run this client by typing in the URL address, not by using the menu. You must include the protocol (**http://** or **ftp://**) in the URL address. Type **http://localhost**. You should see the same HTML code in a message box. No status messages appear here because the status callback doesn't work with OpenURL().

Asynchronous Moniker Files

Just when you thought you knew all the ways to download a file from the Internet, you're going to learn about another one. With asynchronous moniker files, you'll be doing all your programming in your application's main thread without blocking the user interface. Sounds like magic, doesn't it? The magic is inside the Windows **URLMON** DLL, which depends on WinInet and is used by Microsoft Internet Explorer. The MFC **CAsyncMonikerFile** class makes the programming easy, but you should know a little theory first.

Monikers

A moniker is a "surrogate" COM object that holds the name (URL) of the "real" object, which could be an embedded component but more often is just an Internet file (HTML, JPEG, GIF, PNG and so on). Monikers implement the **IMoniker** interface, which has two important member functions: **BindToObject()** and **BindToStorage()**. The **BindToObject()** function puts an object into the running state, and the **BindToStorage()** function provides an **IStream** or an **IStorage** pointer from which the object's data can be read. A moniker has an associated **IBindStatusCallback** interface with member functions such as **OnStartBinding()** and **OnDataAvailable()**, which are called during the process of reading data from a URL. The callback functions are called in the thread that created the moniker. This means that the **URLMON** DLL must set up an invisible window in the calling thread and send the calling thread messages from another thread, which uses **WinInet** functions to read the URL. The window's message handlers call the callback functions.

The MFC **CAsyncMonikerFile** Class

Fortunately, MFC can shield you from the COM interfaces described above. The **CAsyncMonikerFile** class is derived from **CFile**, so it acts like a regular file. Instead of opening a disk file, the class's **Open()** member function gets an **IMoniker** pointer and encapsulates the **IStream** interface returned from a call to **BindToStorage()**. Furthermore, the class has virtual functions that are tied to the member functions of **IBindStatusCallback**. Using this class is a breeze; you construct an object or a derived class and call the **Open()** member function, which returns immediately. Then you wait for calls to overridden virtual functions such as **OnProgress()** and **OnDataAvailable()**, named, not coincidentally, after their **IBindStatusCallback** equivalents.

Using the **CAsyncMonikerFile** Class in a Program

Suppose your application downloads data from a dozen URLs but has only one class derived from **CAsyncMonikerFile**. The overridden callback functions must figure out where to put the data. That means you must associate each derived class object with some UI element in your program. The steps listed below illustrate one of many ways to do this. Suppose you want to list the text of an HTML file in an edit control that's part of a form view. This is what you can do:

1. Use ClassWizard to derive a class from **CAsyncMonikerFile**.
2. Add a character pointer data member **m_buffer**. Invoke **new** for this pointer in the constructor; invoke **delete** in the destructor.
3. Add a public data member **m_edit** of class **CEdit**.
4. Override the **OnDataAvailable()** function thus:

```
void CMyMonikerFile::OnDataAvailable(DWORD dwSize, DWORD bscfFlag)
{
    try {
        UINT nBytesRead = Read(m_buffer, MAXBUF - 1);
        TRACE("nBytesRead = %d\n", nBytesRead);
        m_buffer[nBytesRead] = '\0'; // necessary for edit control
        // The following two lines add text to the edit control
        m_edit.SendMessage(EM_SETSEL, (WPARAM) 999999, 1000000);
        m_edit.SendMessage(EM_REPLACESEL, (WPARAM) 0,
            (LPARAM) m_buffer);
    }
    catch(CFileException* pe) {
        TRACE("File exception %d\n, pe->m_cause");
    }
}
```

```

        pe->Delete();
    }
}

```

5. Embed an object of your new moniker file class in your view class.
6. In you view's `OnInitialUpdate()` function, attach the `CEdit` member to the edit control like this:

```
m_myEmbeddedMonikerFile.m_edit.SubClassDlgItem(ID_MYEDIT, this);
```

7. In your view class, open the moniker file like this:

```
m_myEmbeddedMonikerFile.Open("http://host/filename");
```

For a large file, `OnDataAvailable()` will be called several times, each time adding text to the edit control. If you override `OnProgress()` or `OnStopBinding()` in your derived moniker file class, your program can be alerted when the transfer is finished. You can also check the value of `bscfFlag` in `OnDataAvailable()` to determine whether the transfer is completed. Note that everything here is in your main thread and - most important - the moniker file object must exist for as long as the transfer is in progress. That's why it's a data member of the view class.

Asynchronous Moniker Files vs. WinInet Programming

In the WinInet examples earlier in this module, you started a worker thread that made blocking calls and sent a message to the main thread when it was finished. With asynchronous moniker files, the same thing happens; the transfer takes place in another thread, which sends messages to the main thread. You just don't see the other thread. There is one very important difference, however, between **asynchronous moniker files** and **WinInet programming**: with blocking WinInet calls, you need a separate thread for each transfer; with asynchronous moniker files, only one extra thread handles all transfers together. For example, if you're writing a browser that must download 50 bitmaps simultaneously, using asynchronous moniker files saves 49 threads, which makes the program much more efficient. Of course, you have some extra control with WinInet, and it's easier to get information from the response headers, such as total file length. Your choice of programming tools, then, depends on your application. The more you know about your options, the better your choice will be.

-----End Winsock, WinInet, IIS-----

Further reading and digging:

1. [MSDN MFC 6.0 class library online documentation](#) - used throughout this Tutorial.
2. [MSDN MFC 7.0 class library online documentation](#) - used in .Net framework and also backward compatible with 6.0 class library.
3. [MSDN Library](#)
4. [Windows data type](#).
5. [Win32 programming Tutorial](#).
6. [The best of C/C++, MFC, Windows and other related books](#).
7. Unicode and Multibyte character set: [Story](#) and [program examples](#).