An Introduction To TCP/IP Programming With SocketWrench/VB™

1. Introduction

With the acceptance of TCP/IP as a standard platform-independant network protocol, and the explosive growth of the Internet, the Windows Sockets API (application program interface) has emerged as the standard for network programming in the Windows environment. This document will introduce the basic concepts behind Windows Sockets programming and get you started with your first application. If you're already familiar with sockets programming, feel free to skip this section. It is assumed that the reader is familiar with Visual Basic and has installed the SocketWrench/VB custom control.

The Windows Sockets specification was created by a group of companies, including Microsoft, in an effort to standardize the TCP/IP suite of protocols under Windows. Prior to Windows Sockets, each vendor developed their own proprietary libraries, and although they all had similar functionality, the differences were significant enough to cause problems for the software developers that used them. The biggest limitation was that, upon choosing to develop against a specific vendor's library, the developer was "locked" into that particular implementation. A program written against one vendor's product would not work with another's. Windows Sockets was offered as a solution, leaving developers and their end-users free to choose any vendor's implementation with the assurance that the product will continue to work.

There are two general approaches that you can take when creating a Visual Basic program that uses Windows Sockets: code directly against the API, or use a custom control (VBX) which provides an interface to the library by setting properties and responding to events. The first approach, while possible, is not recommended. The Windows Sockets API was designed for use by C programmers, and some of the functions are fairly unpleasant to use in the Visual Basic environment. You'll end up writing a great deal of code geared towards working around the incompatibilities between the API and Visual Basic, rather than writing the code for your application.

A custom control provides a more "natural" programming interface, and allows you to avoid much of the error-prone drudgery commonly associated with sockets programming. By dropping the control on a form, setting some properties and responding to events, you can quickly and easily write an Internet-enabled application. And because of the nature of custom controls in general, the learning curve is low and experimentation is easy. SocketWrench/VB provides a comprehensive interface to the Windows Sockets library and will be used to build a simple client-server application in the next section of this document. Before we get started with the control, however, we'll cover the basic terminology and concepts behind sockets programming in general.

1.1 Transmission Control Protocol

When two computers wish to exchange information over a network, there are several components that must be in place before the data can actually be sent and received. Of course, the physical hardware must exist, which typically includes network interface cards (NICs) and wiring of some type to connect them. Beyond this physical connection, however, computers also need to use a *protocol* which defines the parameters of the communication between them. In short, a protocol defines the "rules of the road" that each computer must follow so that all of the systems in the network can exchange data. One of the most popular protocols in use today is TCP/IP, which stands for Transmission Control Protocol/Internet Protocol.

By convention, TCP/IP is used to refer to a suite of protocols, all based on the Internet Protocol (IP). Unlike a single local network, where every system is directly connected to each other, an *internet* is a collection of networks, combined into a single, virtual network. The Internet Protocol provides the means by which any system on any network can communicate with another as easily as if they were on the same physical network. Each system, commonly referred to as a *host*, is assigned a unique 32-bit number which can be used to identify it over the internetwork. Typically, this address is broken into four 8-bit numbers separated by periods. This is called *dot-notation*, and looks something like "192.43.19.64". Some parts of the address are used to identify the network that the system is connected to, and the remainder identifies the system itself. Without going into the minutia of the Internet addressing scheme, just be aware that there are three “classes” of addresses, referred to as “A”, “B” and “C”. The rule of thumb is that class “A” addresses are assigned to very large networks, class “B” addresses are assigned to medium sized networks, and class “C” addresses are assigned to smaller networks (networks with less than approximately 250 hosts).

When a system sends data over the network using the Internet Protocol, it is sent in discrete units called *datagrams*, also commonly referred to as *packets*. A datagram consists of a header followed by application-defined data. The header contains the addressing information which is used to deliver the datagram to it’s destination, much like an envelope is used to address and contain postal mail. And like postal mail, there is no guarantee that a datagram will actually arrive at it’s destination. In fact, datagrams may be lost, duplicated or delivered out of order during their travels over the network. Needless to say, this kind of unreliability can cause a lot of problems for software developers. What’s really needed is a reliable, straight-forward way to exchange data without having to worry about lost packets or jumbled data.

To fill this need, the Transmission Control Protocol (TCP) was developed. Built on top of IP, TCP offers a reliable, full-duplex byte stream which may be read and written to in a fashion similar to reading and writing a file. The advantages to this are obvious: the application programmer doesn’t need to write code to handle dropped or out-of-order datagrams, and instead can focus on the application itself. And because the data is presented as a stream of bytes, existing code can be easily adopted and modified to use TCP.

TCP is known as a *connection-oriented* protocol. In other words, before two programs can begin to exchange data they must establish a “connection” with each other. This is done with a three-way handshake in which both sides exchange packets and establish the initial packet sequence numbers (the sequence number is important because, as mentioned above, datagrams can arrive out of order; this number is used to ensure that data is received in the order that it was sent). When establishing a connection, one program must assume the role of the *client*, and the other the *server*. The client is responsible for initiating the connection, while the server’s responsibility is to wait, listen and respond to incoming connections. Once the connection has been established, both sides may send and receive data until the connection is closed.

1.2 User Datagram Protocol

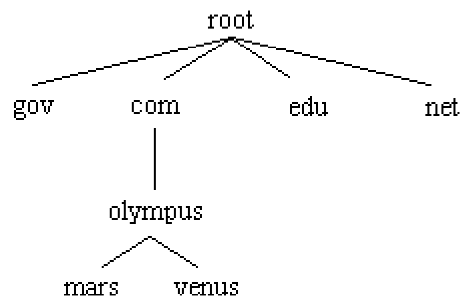
Unlike TCP, the User Datagram Protocol (UDP) does not present data as a stream of bytes, nor does it require that you establish a connection with another program in order to exchange information. Data is exchanged in discrete units called datagrams, which are similar to IP datagrams. In fact, the only features that UDP offers over raw IP datagrams are port numbers and an optional checksum.

UDP is sometimes referred to as an *unreliable protocol* because when a program sends a UDP datagram over the network, there is no way for it to know that it actually arrived at it’s destination. This means that the sender and receiver must typically implement their own application protocol on top of UDP. Much of the work that TCP does transparently (such as generating checksums, acknowledging the receipt of packets, retransmitting lost packets and so on) must be performed by the application itself.

With the limitations of UDP, you might wonder why it’s used at all. UDP has the advantage over TCP in two critical areas: speed and packet overhead. Because TCP is a reliable protocol, it goes through great lengths to insure that data arrives at it’s destination intact, and as a result it exchanges a fairly high number of packets over the network. UDP doesn’t have this overhead, and is considerably faster than TCP. In those situations where speed is paramount, or the number of packets sent over the network must be kept to a minimum, UDP is the solution.

1.3 Hostnames

In order for an application to send and receive data with a remote process, it must have several pieces of information. The first is the IP address of the system that the remote program is running on. Although this address is internally represented by a 32-bit number, it is typically expressed in either dot-notation or by a logical name called a *hostname*. Like an address in dot-notation, hostnames are divided into several pieces separated by periods, called *domains*. Domains are hierarchical, with the top-level domains defining the type of organization that network belongs to, with sub-domains further identifying the specific network.



In this figure, the top-level domains are “gov” (government agencies), “com” (commercial organizations), “edu” (educational institutions) and “net” (Internet service providers). The *fully qualified domain name* is the specified by naming the host and each parent sub-domain above it, separating them with periods. For example, the fully qualified domain name for the “mars” host would be “mars.olympus.com”. In other words, the system “mars” is part of the “olympus” domain (a company’s local network) which in turn is part of the “com” domain (a domain used by all commercial enterprises).

In order to use a hostname instead of a dot-address to identify a specific system or network, there must be some correlation between the two. This is accomplished by one of two means: a local host table or a name server. A host table is a text file that lists the IP address of a host, followed by the names that it’s known by. Typically this file is named **hosts** and is found in the same directory in which the TCP/IP software has been installed. A name server, on the other hand, is a system (actually, a program running on a system) which can be presented with a hostname and will return that host’s IP address. This approach is advantageous because the host information for the entire network is maintained in one centralized location, rather than being scattered about on every host on the network.

1.4 Service Ports

In addition to the IP address of the remote system, an application also needs to know how to address the specific program that it wishes to communicate with. This is accomplished by specifying a *service port*, a 16-bit number that uniquely identifies an application running on the system. Instead of numbers, however, service names are usually used instead. Like hostnames, service names are usually matched to port numbers through a local file, commonly called **services**. This file lists the logical service name, followed by the port number and protocol used by the server.

A number of standard service names are used by Internet-based applications and these are referred to as *well-known services*. Some common services are:

Service Name	Function
echo	Used to echo data back to the program that sent it. This is commonly used to test an application to make sure that a network connection can be established successfully.
ftp	Used to transfer files between computer systems using the File Transfer Protocol.
telnet	Used to provide terminal emulation services for the remote host.
smtp	Used to send electronic mail to a remote host using the Simple Mail Transfer Protocol.

Remember that a service name or port number is a way to *address* an application running on a remote host. Because a particular service name is used, it doesn't guarantee that the service is available, just as dialing a telephone number doesn't guarantee that there is someone at home to answer the call.

1.5 Sockets

The previous sections described what information a program needs to communicate over a TCP/IP network. The next step is for the program to create what is called a *socket*, a communications end-point that can be likened to a telephone. However, creating a socket by itself doesn't let you exchange information, just like having a telephone in your house doesn't mean that you can talk to someone by simply taking it off the hook. You need to establish a connection with the other program, just as you need to dial a telephone number, and to do this you need the *socket address* of application that you want to connect to. This address consists of three key parts: the *protocol family*, *Internet Protocol (IP) address* and the *service port number*.

We've already talked about the IP address and service port, but what's the protocol family? It's a number which is used to logically designate the group that a given protocol belongs to. Since the socket interface is general enough to be used with several different protocols, the protocol family tells the underlying network software which protocol is being used by the socket. In our case, the Internet Protocol family will always be used when creating sockets. With the protocol family, IP address of the system and the service port number for the program that you want to exchange data with, you're ready to establish a connection.

1.6 Client-Server Applications

Programs written to use TCP are developed using the *client-server model*. As mentioned previously, when two programs wish to use TCP to exchange data, one of the programs must assume the role of the client, while the other must assume the role of the server. The client application initiates what is called an *active open*. It creates a socket and actively attempts to connect to a server program. On the other hand, the server application creates a socket and passively listens for incoming connections from clients, performing what is called a *passive open*. When the client initiates a connection, the server is notified that some process is attempting to connect with it. By *accepting* the connection, the server completes what is called a *virtual circuit*, a logical communications pathway between the two programs. It's important to note that the act of accepting a connection creates a new socket; the original socket remains unchanged so that it can continue to be used to listen for additional connections. When the server no longer wishes to listen for connections, it closes the original passive socket.

To review, there are five significant steps that a program which uses TCP must take to establish and complete a connection. The server side would follow these steps:

1. Create a socket.
2. Listen for incoming connections from clients.
3. Accept the client connection.
4. Send and receive information.
5. Close the socket when finished, terminating the conversation.

In the case of the client, these steps are followed:

1. Create a socket.
2. Specify the address and service port of the server program.
3. Establish the connection with the server.
4. Send and receive information.
5. Close the socket when finished, terminating the conversation.

Only steps two and three are different, depending on if it's a client or server application.

1.7 Blocking vs. Non-Blocking Sockets

One of the first issues that you'll encounter when developing your Windows Sockets applications is the difference between blocking and non-blocking sockets. Whenever you perform some operation on a socket, it may not be able to complete immediately and return control back to your program. For example, a read on a socket cannot complete until some data has been sent by the remote host. If there is no data waiting to be read, one of two things can happen: the function can wait until some data has been written on the socket, or it can return immediately with an error that indicates that there is no data to be read.

The first case is called a *blocking socket*. In other words, the program is "blocked" until the request for data has been satisfied. When the remote system does write some data on the socket, the read operation will complete and execution of the program will resume. The second case is called a *non-blocking socket*, and requires that the application handle the situation appropriately. Programs that use non-blocking sockets typically use one of two methods when sending and receiving data. The first method, called polling, is when the program periodically attempts to read or write data from the socket using a timer. The second, and preferred method, is to use what is called *asynchronous notification*. This means that the program is notified whenever a socket event takes place, and in turn can respond to that event. For example, if the remote program writes some data to the socket, a "read event" is generated so that program knows it can read the data from the socket at that point.

For historical reasons, the default behavior is for socket functions to "block" and not return until the operation has completed. Under Windows this can introduce some special problems because when the program blocks, it enters what is called a "message loop", where it continues to process messages sent to it by Windows and other applications. Since messages are being processed, this means that the program can be re-entered at a different point, with the blocked operation "parked" on the program's stack. For example, consider a program that attempts to read some data from the socket when a button is pressed. Because no data has been written yet, it blocks and the program goes into a message loop. The user then presses a different button, which causes code to be executed, which in turn attempts to read data from the socket, and so on. Obviously, this can pose a big problem. To resolve it, the Windows Sockets standard states that there may only be one outstanding blocked call per thread of execution. This means that, in the example given above, the second read would fail with an error indicating that a blocking operation is already in progress. While blocking sockets may be convenient in some situations, their use is generally discouraged because of the complications that they can introduce into the program.

The SocketWrench control facilitates the use of non-blocking sockets by firing events when appropriate. For example, a Read event is generated whenever the remote host writes on the socket, which tells your application that there is data waiting to be read. The use of non-blocking sockets will be demonstrated in the next section, and is one of the key areas in which a control has a distinct advantage over coding directly against the Windows Sockets API.

2. SocketWrench/VB

Because SocketWrench has a large number of properties, you might feel overwhelmed when you start reading through the technical reference material. Don’t worry -- you only need to understand how to use a handful of properties and events to get started. Once you’ve become more comfortable and knowledgeable about sockets programming, you’ll appreciate the power and flexibility that SocketWrench gives you.

Each control that you use corresponds to one socket, which may or may not be connected to a remote host. If you need access to multiple sockets, you must use multiple controls, typically as a control array. This is most commonly needed when your application acts a server and must be able to handle several connections at one time.

2.1 System Requirements

The SocketWrench control requires Microsoft Windows 3.1 or later, Visual Basic 3.0 or later and a TCP/IP protocol stack with a Windows Sockets 1.1 compliant library. Microsoft has a free TCP/IP implementation for Windows for Workgroups 3.11 and Windows 95 includes TCP/IP as part of the base operating system.

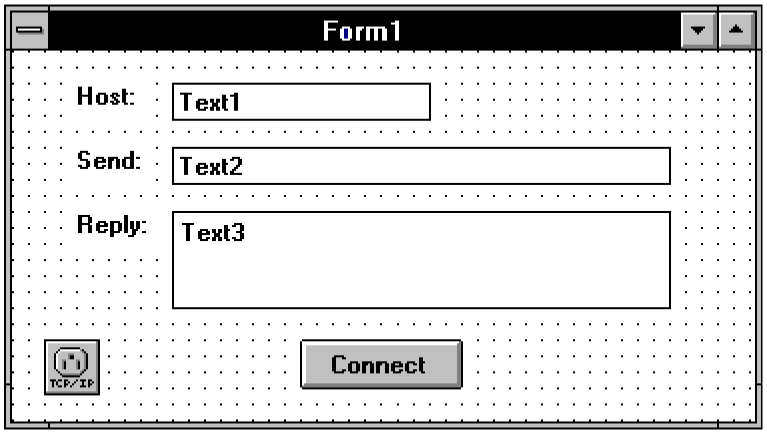
For those systems that do not have a network connection, Catalyst Software includes a “loopback” library that supports the complete Windows Sockets API. It can be used to simulate a network connection so that applications using SocketWrench can function. Refer to the release notes for further information on installing and configuring the loopback library.

2.2 A Sample Client Program

The sample program that will be used throughout this document is a simple tool that can be used to connect with an *echo server*, a program which echoes back any data that’s sent to it. Later on, we’ll also cover how to implement your own echo server.

The first step, after starting Visual Basic, is to include the SocketWrench control in your new project. Select the **File...Add File** option from the Visual Basic menu. A file selection dialog will be displayed. Enter the complete pathname of the control, such as **`c:\windows\system\cswsock.vbx`**. After the control has been added to the tool palette. In addition to adding the control to the project, you will also need to include the **`cswsock.bas`** file that was copied to the directory that you specified during installation. This file is included in the same fashion as the control.

To begin, you’ll need to start Visual Basic and create a form that has three labels, three text controls, a button and the SocketWrench control. The form might look something like this:



When executed, the user will enter the name or IP address of the system in the `Text1` control, the text that is to be echoed in the `Text2` control, and the server's reply will be displayed in the `Text3` control. The `Command1` button will be used to establish a connection with the remote server. The `Text2` and `Text3` controls should be created with their Enabled properties initially set to False.

Some essential properties of the SocketWrench control, called Socket1, needs to be initialized.. The best place to do this is in the form's Load subroutine. The code should look like this:

```
Sub Form_Load ()
    Socket1.AddressFamily = AF_INET
    Socket1.Protocol = IPPROTO_IP
    Socket1.Type = SOCK_STREAM
    Socket1.Binary = False
    Socket1.BufferSize = 1024
    Socket1.Blocking = False
End Sub
```

These six properties should be set for every instance of the SocketWrench control:

AddressFamily	This property is part of the socket address, and should always be set to a value of AF_INET, which is global constant with the integer value of 2. At this time, the Windows Sockets interface only supports the IP address family.
Protocol	This property determines which protocol is going to be used to communicate with the remote application. Most commonly, the value IPPROTO_IP is used, which means that the protocol appropriate for the socket type will be used.
Type	This property specifies the type of socket that is to be created. It may be either of type SOCK_STREAM or SOCK_DGRAM. The stream-based socket uses the TCP protocol, and data is read and written on the socket as a stream of bytes, similar to how data in a file is accessed. The datagram-based socket uses the UDP protocol, and data is read and written in discrete units called datagrams. Most sockets that you will create will be of the stream variety.
Binary	This property determines how data should be read from the socket. If set to a value of True, then the data is received unmodified. If set to False, the data is interpreted as text, with the carriage return and linefeed characters stripped from the data stream. Each receive returns exactly one line of text.
BufferSize	This property is used only for stream-based (TCP) sockets. It specifies the

amount of memory, in bytes, that should be allocated for the socket's send and receive buffers.

Blocking	This property specifies if the application should wait for a socket operation to complete before continuing. By setting this property to False, that indicates that the application will not wait for the operation to complete, and instead will respond to events generated by the control. This is the recommended approach to take when designing your application.

The next step is to establish a connection with the echo server. This is done by including code in the `Command1` button's Click event. The code should look like this:

```
Sub Command1_Click ()
    Socket1.HostName = Trim$(Text1.Text)
    Socket1.RemotePort = IPPORT_ECHO
    Socket1.Action = SOCKET_CONNECT
End Sub
```

The `Text1` edit control should contain the name or IP address of a system that has an echo server running (most UNIX and Windows NT based systems do have such a server). The properties which have been used to establish the connection are:

HostName	This property should be set to either the host name of the system that you want to connect to, or it's IP address in dot-notation.
RemotePort	This property should be set to the number of the port which the remote application is listening on. Port numbers below 1024 are considered reserved by the system. In this example, the echo server port number is 7, which is specified by using the global constant IPPORT_ECHO.
Action	This property initiates some action on the socket. The SOCKET_CONNECT action tells the control to establish a connection using the appropriate properties that have been set. A related action, SOCKET_CLOSE, instructs the control to close the connection, terminating the conversation with the remote server.

Both HostName and RemotePort are known as *reciprocal properties*. This means that by changing the property, the another related property will also change to match it. For example, when you assign a value to the HostName property, the control will determine it's IP address and automatically set the HostAddress property to the correct value. The reciprocal property for RemotePort is the RemoteService property. For more information about these properties, refer to the *SocketWrench Technical Reference*.

Because the socket is non-blocking (i.e.: the Blocking property has been set to a value of False), the program will not wait for the connection to be established. Instead, it will return immediately and respond to the Connect event in the SocketWrench control. The code for that event should look like this:

```
Sub Socket1_Connect ()
    Text2.Enabled = True
    Text3.Enabled = True
End Sub
```

This tells the application that when a connection has been established, enable the edit controls so that the user can send and receive information from the server.

Now that the code to establish the connection has been included, the next step is to actually send and receive data to and from the server. To do this, the `Text2` control should have the following code added to it's KeyPress event:

```
Sub Text2_KeyPress (KeyAscii As Integer)
    If KeyAscii = 13 Then
        Socket1.SendLen = Len(Text2.Text)
        Socket1.SendData = Text2.Text
        KeyAscii = 0: Text2.Text = ""
    End If
End Sub
```

If the user presses the Enter key in the Text2 control, then that text is sent down to the echo server. The properties used to send data are as follows:

SendLen	This property specifies the length of the data being sent to the server. It should *always* be set before the data is written to the socket. After the data has been sent, the value of the property is adjusted to indicate the actual number of bytes that have been written.
SendData	Setting this property causes the data assigned to it to be written to the socket. The number of bytes actually written may be less than the amount specified in the SendLen property if the socket buffers become full.

Once the data has been sent to the server, it immediately sends the data back to the client. This generates a Read event in SocketWrench, which should have the following code:

```
Sub Socket1_Read (DataLength As Integer, IsUrgent As Integer)
    Socket1.RecvLen = DataLength
    Text3.Text = Socket1.RecvData
End Sub
```

The properties used to receive the data are as follows:

RecvLen	This property specifies the maximum number of bytes that should be read from the socket. After the data has been received, the value is changed to reflect the number of bytes actually read.
RecvData	Reading this property causes data to be read from the socket, up to the maximum number of bytes specified by the RecvLen property. If the socket is non-blocking and there is no data to be read, an error is generated.

The Read event is passed two parameters, the number of bytes that are available to be read, and a flag that specifies if the data is urgent (also known as “out-of-band” data, the use of urgent data is an advanced topic outside of the scope of this document). For more information about the Read event, please refer to the *SocketWrench Technical Reference*.

The last piece of code to add to the sample is to handle closing the socket when the program is terminated by selecting Close on the system menu. The best place to put socket cleanup code in the form's Unload event, such as:

```
Sub Form_Unload (Cancel As Integer)
    If Socket1.Connected Then Socket1.Action = SOCKET_CLOSE
    End
End Sub
```

This should be rather self-explanatory. The only new property that has been introduced is the Connected property, which is a Boolean flag. If it is True, then a connection has been established. With all of the properties and event code needed for the sample client application completed, all that's left to do is run the program!

The first thing that you'll probably notice is that if you specify an incorrect host name or address, an error will be generated and the program will stop. Of course, in a real application you'd need to provide extensive error checking. SocketWrench errors start at 24,000 and correspond to the error codes used by the Windows Sockets library. Most errors will occur when setting the host name, address, service port or Action property.

2.3 Building An Echo Server

The next step is to implement your own echo server. To accomplish this, we'll modify the client application to function as a server as well. The side benefit is that this will allow you to test both the client and server application on your local system.

Remember that the first thing that a server application must do is *listen* on a local port for incoming connections. You know that an application is attempting to connect with you when the Accept event is generated for the SocketWrench control. There are two methods which you can use to accept an incoming connection: set the Action property to the value SOCKET_ACCEPT, or set the Accept property.
Setting the Action property is the simplest of the two methods. As you'll recall, the act of accepting a connection causes a *second* socket to be created. The original listening socket continues to listen for more connections, while the second socket can be used to communicate with the client that connected to you. When you set the Action property to SOCKET_ACCEPT, what you're telling the control to do is to *close* the original listening socket, and from that point on, the control can be used to communicate with the client. While this is convenient, it is also limiting -- since the listening socket has been closed, no more clients can connect with your program, effectively limiting it to a single client connection.

The more flexible approach is to set the Accept property to the value passed as an argument to the Accept event. However, this cannot be done by the control that is listening for connections because it is in use. You have to use another, unused control to accept the connection. The problem is, how many clients are going to attempt to connect to you? Of course, you could drop a fixed number of SocketWrench controls on your form, thereby limiting the number of connections, but that's not a very good design. The better approach is to create a *control array* which can be dynamically loaded when a connection is attempted by a client, and unloaded when the connection is closed. This is the approach that we'll take in our server code sample.

The first thing to do is to add a second SocketWrench control to your form, and make it a control array. Initially there will only be one control in the array, identified as `Socket2(0)`. This control will be responsible for listening for client connections. Just as with the client socket control, several of the control's properties should be initialized in the form's Load subroutine. The new subroutine should look like this:

```
Sub Form_Load ()
    Socket1.AddressFamily = AF_INET
    Socket1.Protocol = IPPROTO_IP
    Socket1.Type = SOCK_STREAM
    Socket1.Binary = False
    Socket1.BufferSize = 1024
    Socket1.Blocking = False

    Socket2(0).AddressFamily = AF_INET
    Socket2(0).Protocol = IPPROTO_IP
    Socket2(0).Type = SOCK_STREAM
    Socket2(0).Blocking = False
    Socket2(0).LocalPort = IPPORT_ECHO
    Socket2(0).Action = SOCKET_LISTEN
    LastSocket = 0
End Sub
```

The only thing that is new here is the LocalPort property and the NextSocket variable. The LocalPort property is used by server applications to specify the local port that it's listening on for connections. By specifying the standard port used by echo servers, any other system can connect to yours and expect the program to echo back whatever is sent to it.

The LastSocket variable is defined in the "general" section of the Visual Basic application as an integer. It is used to keep track of the next index value that can be used in the control array.

By setting the Action property to SOCKET_LISTEN in the form's Load event, the program will start listening for connections as soon as the program starts executing. When a client tries to connect with your server, Socket2's Accept event will fire. The code for this event should look like this:

```
Sub Socket2_Accept (Index As Integer, SocketId As Integer)
    Dim I As Integer
    For I = 1 To LastSocket
        If Not Socket2(I).Connected Then Exit For
    Next I
    If I > LastSocket Then
        LastSocket = LastSocket + 1: I = LastSocket
        Load Socket2(I)
    End If
    Socket2(I).AddressFamily = AF_INET
    Socket2(I).Protocol = IPPROTO_IP
    Socket2(I).Type = SOCK_STREAM
    Socket2(I).Binary = True
    Socket2(I).BufferSize = 1024
    Socket2(I).Blocking = False
    Socket2(I).Accept = SocketId
End Sub
```

The first statement loads a new instance of the SocketWrench control as part of the `Socket2` control array. The next six lines initialize the control's properties, and then the Accept property is set to the value of the SocketId parameter that is passed to the control. After executing this statement, the control is now ready to start communicating with the client program.

Since it's the job of an echo server to echo back whatever is sent to it, we have to add code to the control's Read event, which tells it that the client has sent some data to us. It looks like this:

```
Sub Socket2_Read (Index As Integer, DataLength As Integer,
                  IsUrgent As Integer)
    Socket2(Index).RecvLen = DataLength
    Socket2(Index).SendLen = DataLength
    Socket2(Index).SendData = Socket2(Index).RecvData
End Sub
```

Finally, when the client closes the connection, the socket control must also close it's end of the connection. This is accomplished by adding a line of code in the socket's Close event:

```
Sub Socket2_Close (Index As Integer)
    Socket2(Index).Action = SOCKET_CLOSE
End Sub
```

To make sure that all of the socket connections are closed when the application is terminated, the following code should be included in the form's Unload event:

```
Sub Form_Unload (Cancel As Integer)
    Dim I As Integer
    If Socket1.Connected Then Socket1.Action = SOCKET_CLOSE
    If Socket2(0).Listening Then Socket2(0).Action = SOCKET_CLOSE
    For I = 1 To LastSocket
        If Socket2(I).Connected Then Socket2(I).Action = SOCKET_CLOSE
    Next I
    End
End Sub
```

The only new property shown here is the Listening property, which like the Connected property, is a Boolean flag. If the control is listening for incoming connections, this property will return True, otherwise it returns False. This is added only as an extra "sanity check", and the property should always return True for this instance of the control.

2.4 Putting It All Together

This document has introduced you to the basic concepts behind socket programming, and how to use SocketWrench to get started developing your own TCP/IP applications. Although the echo client and server sample program is fairly basic, it does illustrate many of the key issues that you'll encounter when developing your own software.

Now is a good time to review the SocketWrench Technical Reference and the other sample programs included with the SocketWrench distribution. Also, the help file included with SocketWrench includes all of the information in the Technical Reference at your fingertips.

We believe that SocketWrench is the most complete and powerful Windows Sockets custom control available, and hope that your time using it is both productive and enjoyable.