Catalyst Development Corporation
www.catalyst.com

An Introduction to TCP/IP Programming with SocketWrench™ Freeware Edition

Introduction

With the acceptance of TCP/IP as a standard platform-independent 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 created with SocketWrench. The tutorial section of this document requires that that the reader be familiar with Visual Basic and has installed the SocketWrench Freeware Edition control.

The **SocketWrench Freeware Edition** is the most popular, freely available Internet control for Microsoft Windows. For developers who are new to Internet software development, SocketWrench greatly reduces the learning curve typically associated with network programming and enables developers to quickly build client and server applications. Included in the Freeware Edition is the SocketWrench Windows Sockets control for network programming, the Remote Access Dialer control for establishing dial-up networking connections to service providers, and a file encoding/decoding control which supports the uucode, base64 and quoted-printable formats. The Freeware Edition is ideal for students, hobbyists and programmers who are new to Internet software development.

The **SocketWrench Secure Edition** is new, commercial release of SocketWrench that supports standard and secure (SSL/TLS) network connections. The Secure Edition is a complete rewrite of the control, making Windows Sockets programming even easier than before. Designed for the professional commercial software developer, the Secure Edition is optimized for 32-bit platforms and implements secure protocols with support for up to 128-bit encryption. This new release includes both ActiveX controls, standard dynamic link libraries (DLLs) and C++ class wrappers in the same package, along with new samples and over 400 pages of documentation. For professional developers, the SocketWrench Secure Edition provides all of the features, documentation and technical support needed to develop complete Internet applications, without the complexities of learning the Windows Sockets API or working around the limitations of other Internet controls.

The **SocketTools Visual and Library Editions** provide a complete collection of controls and libraries for many of the popular Internet application protocols such as FTP, POP3, SMTP and HTTP. Secure editions of these components are also available that support both standard and secure (SSL/TLS) network connections. You'll find the same features, functionality and stability in the SocketTools package without having to learn how to implement complex application protocols or decipher cryptic standards documents. With SocketTools, adding features like file transfer, sending and retrieving e-mails, and accessing web pages can be done in just a few minutes. Instead of reinventing the wheel, you can spend your time working on your core application and increasing your productivity without sacrificing the features that your users expect.

To learn more about the SocketWrench Secure Edition and SocketTools family of products, please visit the Catalyst Development website at www.catalyst.com

Windows Sockets API

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 program that uses Windows Sockets. One is to code directly against the API. The other is to use a component which provides a higher-level interface to the library by setting properties and responding to events. This can provide a more "natural" programming interface, and it allows you to avoid much of the error-prone drudgery commonly associated with sockets programming. By including the control in a project, 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 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.

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 is typically either a network interface card (NIC) or a serial communications port for dial-up networking connections. 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.

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.

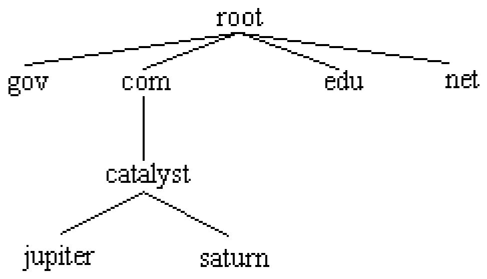
UDP is sometimes referred to as an *unreliable protocol* because when a program sends a UDP datagram over the network, there is no way for it to know that it actually arrived at it's destination. This means that the sender and receiver must typically implement their own application protocol on top of 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.

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 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 "jupiter" host would be "jupiter.catalyst.com". In other words, the system "jupiter" is part of the "catalyst" 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.

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*. These services are defined by a standards document and include common application protocols such as FTP, POP3, SMTP and HTTP.

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.

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 the 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.

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.

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 recognize the error condition and 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 (typically 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. However, blocking sockets in Windows can introduce some special problems. For 16-bit applications, the blocking function will enter 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.

Blocking socket functions can introduce a different type of problem in 32-bit applications because blocking functions will prevent the calling thread from processing any messages sent to it. Since many applications are single-threaded, this can result in the application being unresponsive to user actions.[1] To resolve the general problems with blocking sockets, the Windows Sockets standard states that there may only be one outstanding blocked call per thread of execution. This means that 16-bit applications that are re-entered (as in the example above) will encounter errors whenever they try to take some action while a blocking function is already in progress. With 32-bit programs, the creation of *worker threads* to perform blocking socket operations is a common approach, although it introduces additional complexity into the application.

It should be noted that there are advantages to using blocking sockets. In most cases, the application design and implementation is simpler, and raw throughput (the rate at which data is sent and received) is generally higher with blocking sockets because it does not have to go through an event mechanism to notify the application of a change in status. In general, if your application is designed as a client, and does not have the need to establish multiple simultaneous connections then blocking sockets may be appropriate. However, if your application functions as a server or needs to establish multiple connections then an asynchronous, event-driven design is more appropriate.

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.

In summary, there are three general approaches that can be taken when building an application with the control in regard to blocking or non-blocking sockets:

- Use a blocking (synchronous) socket. In this mode, the program will not resume execution until the socket operation has completed. This method is only recommended for relatively small applications. Blocking sockets in 16-bit application will allow it to be re-entered at a different point, and 32-bit applications will stop responding to user actions. Blocking sockets can lead to complex interactions (and difficult debugging) if there are multiple active controls in use by the application.

- Use a non-blocking (asynchronous) socket, which allows your application to respond to events. For example, when the remote system writes data to the socket, a Read event is generated for the control. Your application can respond by reading the data from the socket, and perhaps send some data back, depending on the context of the data received.

- Use a combination of blocking and non-blocking socket operations. The ability to switch between blocking and non-blocking modes "on the fly" provides a powerful and convenient way to perform socket operations. Note that the warning regarding blocking sockets also applies here.

[1] Fortunately blocked 32-bit applications do not prevent other programs from running, as is the case with 16-bit platforms such as Windows 3.1.

If you decide to use non-blocking sockets in your application, it's important to keep in mind that you must check the return value from every read and write operation, since it is possible that you may not be able to send or receive all of the specified data. Frequently, developers encounter problems when they write a program that assumes a given number of bytes can always be written to, or read from, the socket. In many cases, the program works as expected when developed and tested on a local area network, but fails unpredictably when the program is released to a user that has a slower network connection (such as a serial dial-up connection to the Internet). By always checking the return values of these operations, you insure that your program will work correctly, regardless of the speed or configuration of the network.

Programming With SocketWrench in Visual Basic

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 as a server and must be able to handle several connections at one time.

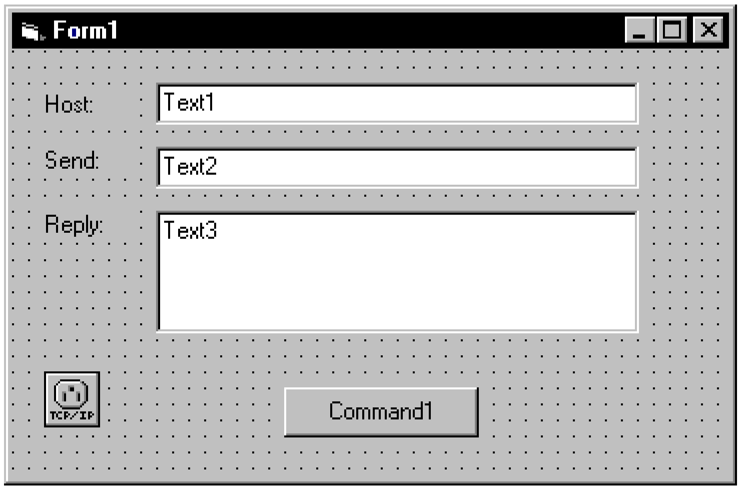
A Sample Client Program

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. In Visual Basic 3.0, select the **File|Add File** option from the menu and a file selection dialog will be displayed. Enter the complete pathname of the control, such as c:\windows\system\cswskctl.vbx. To add the control to the list of available controls, click on the **Browse** button and enter the complete pathname of the control. In Visual Basic 4.0, you should select **Tools|Custom Controls**, while in Visual Basic 5.0 and Visual Basic 6.0, you should select **Project|Components**. A dialog will display all of the available ActiveX controls, then select the Catalyst SocketWrench Control.

After the control has been added to the tool palette, you will also need to include some constants used by SocketWrench into your application. These can be defined in your form, or in another module. For a list of the constants that you can use, go to the SAMPLES directory and select the CONSTANTS.TXT file that is appropriate for your version of Visual Basic. You can copy the entire file into your project if you wish.

To begin, you'll need to 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, need to be initialized. The best place to do this is in the form's Load subroutine. The code should look like this[2]:

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

[2] The code examples shown here use keywords like **Private** which are used in Visual Basic 4.0 and later. If you are using an earlier version of Visual Basic, these keywords should be ignored.

SocketType	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:

```
Private Sub Command1_Click()

    On Error GoTo Failed
    Socket1.HostName = Trim$(Text1.Text)
    Socket1.RemotePort = IPPORT_ECHO
    Socket1.Action = SOCKET_CONNECT
    Exit Sub

Failed:
    MsgBox "Unable to connect to remote host"
    Exit Sub

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, 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 *Technical Reference* section.

When using the ActiveX controls, an alternative to setting the Action property is to use the Connect method. This method performs the same function, but would be called like this:

```
Private Sub Command1_Click()

    Socket1.HostName = Trim$(Text1.Text)
    Socket1.RemotePort = IPPORT_ECHO

    If Socket1.Connect() <> 0 Then
        MsgBox "Unable to connect to remote host"
        Exit Sub
    End If

End Sub
```

This demonstrates one of the principal differences between using the Action property and using the methods in the ActiveX controls. When you set the Action property and an error occurs, a Visual Basic error is generated which causes any error trapping code to be executed. However, calling a method in the control will not generate a Visual Basic error if the method fails. Instead, the error code is returned by the method, and the application is responsible for handling that error condition.

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:

```
Private 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 written, 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:

```
Private 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.

When using ActiveX controls, as with the **Action** property, there is a method called Write which can be used instead of the **SendLen** and **SendData** properties. The **Write** method has two arguments, the string buffer to write to the socket and the number of bytes to write. For example, the code would look like this:

```
Private Sub Text2_KeyPress(KeyAscii As Integer)
    Dim strText As String
    If KeyAscii = 13 Then
        strText = Text2.Text
        Socket1.Write strText, Len(strText)
        KeyAscii = 0: Text2.Text = ""
    End If
End Sub
```

Because we have connected to the echo service, once the data has been sent to the remote host, it immediately sends the data back to the client. This generates a Read event in SocketWrench, which should have the following code:

```
Private 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.

When using ActiveX controls, the Read method can be used instead of the RecvLen and RecvData properties. The method has two arguments, the string buffer to copy the data into and the number of bytes to read from the socket. For example, the code would look like this:

```
Private Sub Socket1_Read(DataLength As Integer, IsUrgent As Integer)
    Dim strBuffer As String
    Socket1.Read strBuffer, DataLength
    Text3.Text = strBuffer
End Sub
```

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 *Technical Reference* section.

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 is 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! 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.

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.SocketType = SOCK_STREAM
    Socket1.Binary = False
    Socket1.BufferSize = 1024
    Socket1.Blocking = False

    Socket2(0).AddressFamily = AF_INET
    Socket2(0).Protocol = IPPROTO_IP
    Socket2(0).SocketType = 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 LastSocket 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:

```
Private 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).SocketType = 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:

```
Private 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 Disconnect event:

```
Private Sub Socket2_Disconnect(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:

```
Private 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.

Putting It All Together

This guide has introduces you to the basic concepts behind socket programming and how to use SocketWrench to get started developing your own Windows Sockets applications. Although the echo client and server sample program is fairly basic, it does examine 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 in the package. The help file included with SocketWrench also includes the complete technical reference, and can be accessed directly within your development environment.

SocketWrench Secure Edition

The SocketWrench Secure Edition is new, commercial release of SocketWrench that supports standard and secure (SSL/TLS) network connections. The Secure Edition is a complete rewrite of the control, making Windows Sockets programming even easier than before. Designed for the professional commercial software developer, some of the new features are:

- Support for secure communications, allowing developers to create their own custom secure client and server applications as well as connect to standard secure servers.
- Implements the standard SSL 2.0 and 3.0 protocols, the TLS 1.0 protocol and the PCT 1.0 protocol for secure communications, with support for up to 128-bit encryption.
- Optimized for 32-bit platforms, the control and library has improved performance and reliability with less overhead, reduced memory requirements and higher overall throughput. SocketWrench is compatible with all 32-bit Intel based Windows platforms from Windows 95 to Windows XP.
- Improved interface allows developers to implement the same functionality with fewer lines of code, which means fewer mistakes and easier to read source code.
- Simplified distribution means that you only need to redistribute the single SocketWrench component; there are no external third-party file dependencies which can complicate installations on target platforms.
- Includes ActiveX controls, standard dynamic-link libraries (DLLs) and C++ class wrappers in the same package. SocketWrench can be used with virtually any Windows software development tool.
- Over 400 pages of documentation, including a revised tutorial that covers secure sockets programming and a new tutorial for building your first client/server application using SocketWrench. Also includes a guide to migrating applications which use the Freeware Edition control.
- New and revised sample programs to demonstrate how to use SocketWrench, including a complete FTP client, secure HTTP client and an HTTP server example.

For professional developers, the SocketWrench Secure Edition provides all of the features, documentation and technical support needed to develop complete Internet applications, without the complexities of learning the Windows Sockets API or working around the limitations of other Internet controls.

For more information about the SocketWrench Secure Edition, visit the Catalyst Development website at www.catalyst.com/products/socketwrench

Advanced Development Using SocketTools

SocketWrench is part of a package developed by Catalyst called SocketTools. In addition to the comprehensive, but fairly low-level, access that SocketWrench provides, SocketTools includes components and libraries for many of the popular Internet application protocols. There are six different editions of SocketTools available, and all editions provide royalty-free redistribution licensing and a thirty day money-back guarantee. Evaluation copies of all editions are available for downloading from our website and we provide unlimited free technical support.

SocketTools Visual Edition

The SocketTools Visual Edition consists of 16-bit Visual Basic (VBX) controls and both 16-bit and 32-bit ActiveX (OCX) controls for use with visual development environments such as Visual Basic, Visual C++ and Delphi. A total of nineteen controls provide client interfaces for the major application protocols such as the File Transfer Protocol, Simple Mail Transfer Protocol, Domain Name Service and Telnet. All versions of Visual Basic from 2.0 and later are supported, and the ActiveX controls can be used with any 32-bit development tool that supports COM and the ActiveX control specification. The network controls support both synchronous (blocking) and asynchronous modes of operation and include trace debugging facilities. All of the controls are thread-safe and can be used in multithreaded containers, such as Internet Explorer.

SocketTools Secure Visual Edition

The SocketTools Secure Visual Edition consists of the same 32-bit ActiveX components in the standard Visual Edition, including components which support secure data communications over the Internet or a local network. The Secure Visual Edition supports three standard security protocols: Secure Sockets Layer (SSL) versions 2.0 and 3.0, Private Communication Technology (PCT) version 1.0 and Transport Layer Security (TLS) version 1.0. The protocols supported are HTTPS, FTPS, SMTPS, POP3S, NNTPS and TELNETS.

SocketTools Library Edition

The SocketTools Library Edition consists of 16-bit and 32-bit dynamic link libraries (DLLs), and can be used by virtually any Windows programming language or scripting tool. A total of sixteen libraries provide client interfaces for application protocols such as the File Transfer Protocol, Simple Mail Transfer Protocol and Telnet protocol. The application program interface for the Library Edition is implemented with a simple elegance that makes it easy to use with any language, not just C or C++. All of the libraries are thread-safe and can be used in multithreaded applications.

SocketTools Secure Library Edition

The SocketTools Secure Library Edition consists of the same 32-bit dynamic link libraries in the standard Library Edition, including libraries which support secure data communications over the Internet or a local network. The Secure Library Edition supports three standard security protocols: Secure Sockets Layer (SSL) versions 2.0 and 3.0, Private Communication Technology (PCT) version 1.0 and Transport Layer Security (TLS) version 1.0. The protocols supported are HTTPS, FTPS, SMTPS, POP3S, NNTPS and TELNETS.

SocketTools Enterprise Edition

The SocketTools Enterprise Edition offers the best of both worlds for the corporate developer who needs visual controls for rapid application development, as well as the power and flexibility of dynamic-link libraries for developing core application systems.

Including 16-bit Visual Basic controls, 16/32-bit ActiveX controls and 16/32-bit dynamic link libraries (DLLs), the Enterprise Edition is suitable for use with virtually any Windows development environment.

SocketTools Secure Enterprise Edition

The SocketTools Secure Enterprise Edition consists of the same 32-bit ActiveX controls and dynamic link libraries in the standard Enterprise Edition, including libraries and controls which support secure data communications over the Internet or a local network. The Secure Library Edition supports three standard security protocols: Secure Sockets Layer (SSL) versions 2.0 and 3.0, Private Communication Technology (PCT) version 1.0 and Transport Layer Security (TLS) version 1.0. The protocols supported are HTTPS, FTPS, SMTPS, POP3S, NNTPS and TELNETS.

For more information about SocketTools, visit the Catalyst Development website at www.catalyst.com/products/sockettools