An Introduction to TCP/IP Programming

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. It is assumed that the reader is familiar with Visual Basic and has installed the SocketWrench Secure Edition control.

Designed for the professional commercial software developer, SocketWrench 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++ classes in the same package, along with new samples and over 400 pages of documentation. For professional developers, SocketWrench 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 SocketWrench 4.0 and the SocketTools family of products, please visit the Catalyst Development website at **http://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 its 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 its 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, straightforward 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.

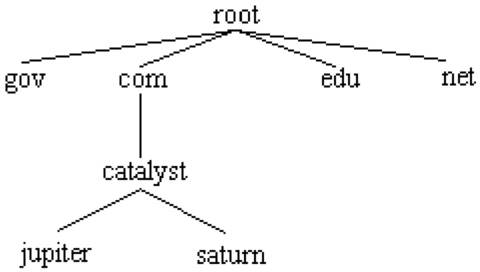
UDP is sometimes referred to as an unreliable protocol because when a program sends a UDP datagram over the network, there is no way for it to know that it actually arrived at its destination. This means that the sender and receiver must typically implement their own application protocol on top of UDP.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

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 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. The creation of worker threads to perform blocking socket operations is a common approach to address this issue, 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, an **OnRead** 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. Blocking operations can cause code to be re-entered at a different point, leading 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, an **OnRead** 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.

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.

Secure Network Communication

Security and privacy is a concern for everyone who uses the Internet, and the ability to provide secure transactions over the Internet has become one of the key requirements for many business applications.

SocketWrench has the ability to establish secure connections with remote servers, as well as function as a secure server itself. Although most of the technical issues such as data encryption are handled internally by the control and library, a general understanding of the standard security protocols is useful when designing your own applications.

When you establish a connection to a server over the Internet (for example, a web server), the data that you exchange is typically routed over dozens of computer systems until it reaches its destination. Any one of these systems may monitor and log the data that it forwards, and there is no way for either the sender or receiver of that data to know if this has been done. Exchanging information over the Internet could be likened to talking with someone in a public restaurant. Anyone can choose to listen to what you're saying, and unless they introduce themselves, you have no idea who they are or if they've even heard what you said.

To ensure that private information can be securely exchanged over the Internet, two basic requirements must be met: there must be a way to send that information so that only the sender and the receiver can understand what is being exchanged, and there must be a way for them to determine that they each are in fact who they claim to be. The solution to the first problem is to use encryption, where a key is used to encrypt and decrypt the data using a mathematical formula. The second problem is addressed by using digital certificates. These certificates are issued by a certificate authority (CA), which is a trusted third-party organization who verifies that the individual or company which is issued a certificate is who they claim to be. These two concepts, encryption and digital certificates, are combined to provide the means to send and receive secure information over the Internet.

The Secure Sockets Layer (SSL) protocol was originally developed by Netscape as a way to exchange information securely over the Internet, and is still the most common protocol in use today. The latest improvements to SSL have resulted in the Transport Layer Security (TLS) protocol, and it is beginning to replace SSL as the standard for secure communications over the Internet. Microsoft also developed a protocol similar to SSL called the Private Communication Technology (PCT) protocol. All of these protocols were designed to provide essentially the same thing: a private exchange of encrypted data between the sender and receiver, making it unreadable by an intermediate system. Using the restaurant analogy, it would be as if two people were speaking in a language that only they could understand. Although someone sitting at the next table could listen in on the conversation, they wouldn't have any idea what was actually being said.

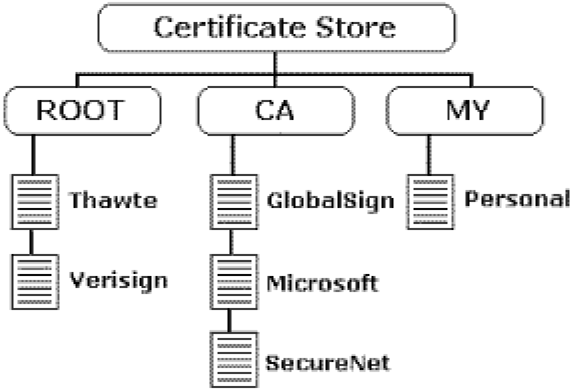
A secure connection, for example between a web browser and a server, begins with what is called the handshake phase where the client and server identify themselves. When the client first connects with the server it sends a block of data to the server and the server responds with its digital certificate, along with its public key and information about what type of encryption it would like to use. Next, the client generates a master key and sends this key to the server, which authenticates it. Once the client and server have completed this exchange, keys are generated which are used to encrypt and decrypt the data that is exchanged. With the handshake completed, a secure connection between the client and server is established. SocketWrench handles the handshake phase of the secure connection automatically and does not require any additional programming. If a secure connection cannot be established, an error is returned and the network connection is closed.

After the handshake phase has completed, the client may choose to examine the digital certificate that has been returned by the server. The information contained in the certificate includes the date that it was issued, the date that it expires, information about the organization who issued the certificate (called the issuer) and who the certificate was issued to (called the subject of the certificate). The client may also validate the status of the certificate, determining if it was issued by a trusted certificate authority and was returned by the same company or individual it was issued to. There may be certain cases where the client determines that there's a problem with the certificate (for example, if the certificate's common name does not match the domain name of the server), but chooses to continue communicating with the server. Note that the connection with the server will still be secure in this case. In other cases, for example if the certificate has expired, the client may choose to terminate the connection and warn the user.

Digital Certificates

With secure connections, digital certificates are used to exchange public keys for data encryption and to provide identification information. This information typically includes the organization that was issued the certificate, its physical location and so on. The certificate itself is used to validate that the public key actually belongs to the entity that it was issued to. The certificate also includes information about the Certification Authority (CA) who issued the certificate. The CA is responsible for validating the information provided by that organization, and then digitally signing the certificate. This establishes a relationship between the two so that when others validate the certificate, they know that it has been issued by a trusted third-party. For example, let's say that a company wants to implement a secure site so that people can order products online. They would provide information about their company (organizational contacts, financial information and so on) to a trusted third party organization such as Verisign. Verisign would then verify that the information they provided was complete and correct, and then would issue a signed certificate to them, which they install on their server. When a user (client system) connects to their server and checks the certificate, they see that it was issued by Verisign, a trusted Certification Authority. In essence, the user is saying that because they trust Verisign, and Verisign trusts the company the certificate was issued to, they will trust the company as well.

To establish this relationship between the Certification Authority and the organization a certificate is issued to, there needs to be a root certificate which has been signed by the same trusted organization. This serves as the beginning of the certification path that is used to validate signed certificates. Using the above example, on the user's system there is a root certificate for Verisign, signed by Verisign. Root certificates are maintained in the local system's certificate store which is essentially a database of digital certificates. This database is structured so that different types of certificates can be organized in one central location on the system, and a standard interface is provided to enumerate and validate these certificates. Certificates are associated with a store name, allowing them to be easily categorized. For example, root certificates are stored under the name "root", while a user's personal certificates (along with their private keys) are stored under the name "my".



When the Windows operating system is installed, there is a certificate store that contains the root certificates for the major Certification Authorities. However, there are situations where additional certificates may need to be added to the system. To facilitate this, there is a tool called CertMgr which allows a user to install certificates, as well as export or remove certificates from the certificate store. When managing your system's certificate store, you should take the same care that you do when making changes to the system registry. Inadvertently removing a certificate could result in errors when attempting to access secure systems.

In general, the one situation where certificate management becomes important is when you want to develop your own secure server. This is because your server needs to have a signed certificate to send to the client in order to establish the secure connection. For general-purpose commercial applications, this generally means you would need to obtain a certificate that has been signed by a Certification Authority such as Verisign. This certificate would then be installed in the certificate store on the server. However, for development purposes it may be inconvenient to purchase a certificate. There also may be situations in which an organization wishes to function as its own Certification Authority and issue certificates themselves. This allows the organization to control how certificates are managed and can be ideal for secure applications that are designed for the corporate intranet. A utility for creating self-signed root certificates and server certificates is included with SocketWrench.

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.

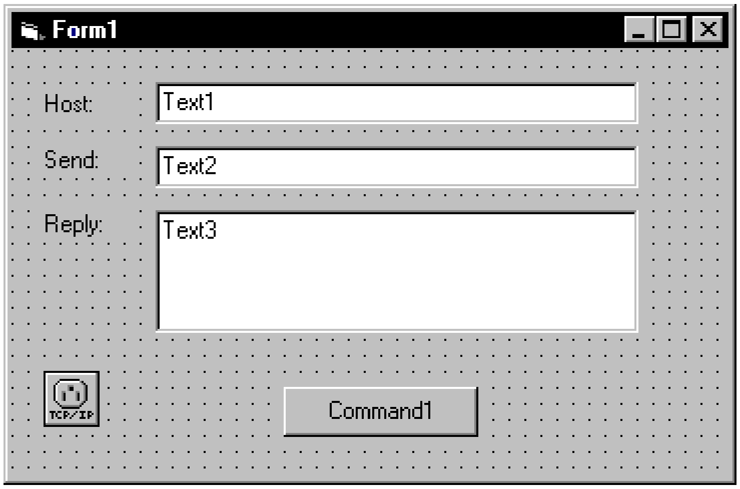
Each control that you use corresponds to one socket, which may or may not be connected to a remote host. If you need access to multiple sockets, you must use multiple controls, typically as a control array. This is most commonly needed when your application acts as a server and must be able to handle 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 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. Note that if you have a previous version of SocketWrench installed on the system, both versions will be displayed. You should select version 4.0 for this example.

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. When you save your project, call it "Client".

First we should initialize the controls in the Form's Load subroutine. Note that we want to disable the Text2 and Text3 controls, since they only should be usable once a connection to a server has been established. The code should look like this:

```
Private Sub Form_Load()
    Command1.Caption = "Connect"
    Command1.Enabled = True
    Text1.Enabled = True
    Text2.Enabled = False
    Text3.Enabled = False
End Sub
```

The next step is to write the code that actually establishes a connection with the remote server in the **Click** event for the Command1 button. The code should look like this:

```
Private Sub Command1_Click()

    If Not SocketWrench1.Connected Then
        Dim strRemoteHost As String
        Dim nError As Long

        strRemoteHost = Trim(Text1.Text)

        SocketWrench1.AutoResolve = False
        SocketWrench1.Blocking = False
        SocketWrench1.Protocol = swProtocolTcp

        nError = SocketWrench1.Connect(strRemoteHost, swPortEcho)

        If nError <> 0 Then
            MsgBox "Unable to connect to remote host", vbExclamation
            Exit Sub
        End If

        Command1.Enabled = False
    Else
        SocketWrench1.Disconnect
        Command1.Caption = "Connect"
        Text2.Enabled = False
        Text3.Enabled = False
    End If

End Sub
```

The first SocketWrench property that we encounter is the **Connected** property. This is a boolean flag which tells us if the control has established a connection to a remote host. We're using this to allow the Command1 button to function in one of two ways: if no connection has been established, then pressing the button will cause the client to make a connection to the server entered in the Text1 control. However, if there is an active connection, then pressing the button will disconnect the client from the server.

These next three SocketWrench properties are used to define some basic functions of the control, such as how host names are resolved and what network protocol is used. These properties are:

AutoResolve	This property specifies that the control should not immediately attempt to resolve host names into IP addresses if the **HostName** and/or **HostAddress** property are set. In general it is recommended that you initialize this property value to False unless your application has a specific to automatically resolve host names.
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.

Protocol	This property determines which protocol is going to be used to communicate with the remote application. Most commonly, the value **swProtocolTcp** is specified, which means that the stream-based Transmission Control Protocol will be used. To send UDP datagrams, this property can be set to the value **swProtocolUdp**.

To establish the connection to the server, the **Connect** method is called, passing the name of the server to connect to and the port number of the echo server. It should be noted that there are a number of optional arguments to this method, but for the purposes of this example, only the host name and port number are needed. If the connection attempt is successful, the method will return a value of zero. However, if an error occurs the method will return a non-zero value which specifies an error code.

If the connection attempt is successful, then the Command1 button is disabled. Because the socket is non-blocking (that is, the **Blocking** property is False), when the **Connect** method returns it does not mean that the connection has actually completed. Instead, it means that the connection process has begun, and completion is signaled by the control's **OnConnect** event firing. So between the time that the **Connect** method is called to establish a connection and the time that the **OnConnect** event is fired to indicate that the connection has been completed, the user should not be able to press the Command1 button because it would result in the **Connect** method being called again.

To update our form when a connection has been established, we need to add some code to the control's **OnConnect** event. Remember, this event is only called after a connection attempt has completed on a non-blocking socket:

```
Private Sub SocketWrench1_Connect()
    Command1.Caption = "Disconnect"
    Command1.Enabled = True
    Text2.Enabled = True
    Text3.Enabled = True
    MsgBox "Connect to remote host", vbInformation
End Sub
```

This will change the caption of our Command1 button to "Disconnect" (informing the user that when they press it, now it will disconnect the current session), and enable our Text2 and Text3 controls. We also display a message box indicating that the connection has completed.

There is a possibility that the remote host may terminate our connection, and our client application needs to be able to handle this. If this happens, for example if the server is stopped, then the control's **OnDisconnect** event will fire. In our code, we'll reset our command button's caption, disable the Text2 and Text3 controls and display a message box indicating that the connection has been lost. The code would look like this:

```
Private Sub SocketWrench1_Disconnect()
    SocketWrench1.Disconnect
    Command1.Caption = "Connect"
    Command1.Enabled = True
    Text2.Enabled = False
    Text3.Enabled = False
    MsgBox "Disconnected from remote host", vbInformation
End Sub
```

The one thing that may seem strange here is calling the **Disconnect** method. After all, the connection has been closed, so this appears to be redundant. The thing to remember is that a socket is a communications endpoint; for every conversation between a client and a server, there are two sockets: one on your end (the client) and one on theirs (the server). When the **OnDisconnect** event fires, what the control is telling you is that the other socket, in this case the server's socket, has been closed. However, until you call the **Disconnect** method it will remain open on the client side. For the connection to be completely terminated, the sockets on both ends of the connection need to be closed.

What happens if there is an error while the client attempts to connect to the server? It is possible for the **Connect** method to return zero (indicating success), and then once the connection attempt begins, an error occurs. For example, this can happen if there is no server listening on the specified port number. To be able to handle this, the control has an event called **OnError** which is fired whenever an error such as this occurs. Let's add some code to the event to report any errors:

```
Private Sub SocketWrench1_Error(ByVal Error As Variant, _
                                    ByVal Description As Variant)

    If Error <> swErrorOperationWouldBlock Then
        SocketWrench1.Disconnect
        Command1.Caption = "Connect"
        Command1.Enabled = True
        Text2.Enabled = True
        Text3.Enabled = True
        MsgBox Description, vbExclamation, "Error " & CStr(Error)
    End If

End Sub
```

The **OnError** event has two arguments passed to it, an error code and a textual description of the error. The first thing that we do is compare this error against one of our predefined error constants `swErrorOperationWouldBlock` which occurs if a socket operation would cause a non-blocking socket to block. For example, attempting to read data from a non-blocking socket and there is no data available at that time would result in this error. The reason that we're specifically checking for it is because this particular error code is really more of a warning to the application, not a fatal error. In all other cases, we disconnect the client session and report the error.

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 its **KeyPress** event:

```
Private Sub Text2_KeyPress(KeyAscii As Integer)
    If KeyAscii = 13 Then
        Dim strBuffer As String
        Dim cchBuffer As Long, nResult As Long

        strBuffer = Text2.Text & vbCrLf
        cchBuffer = Len(strBuffer)
        Text2.Text = ""
        KeyAscii = 0

        nResult = SocketWrench1.Write(strBuffer, cchBuffer)
        If nResult = -1 Then
            MsgBox "Unable to send data to server"
            Exit Sub
        End If
    End If
End Sub
```

The **Write** method is used to send data to the remote server. The first argument is the buffer that contains the data (in this case, a string variable) and the second argument is the number of bytes to write. Note that the second argument is optional and if it is omitted the entire buffer is written. For clarity, it is recommended that the buffer length be specified. Note that in addition to strings, the **Write** method will also accept bytes and byte arrays as parameters.

Because our example is connecting to an echo service, once the data has been sent to the remote host, it immediately sends the data back to the client. This generates an **OnRead** event in SocketWrench, which should have the following code:

```
Private Sub SocketWrench1_Read()
    Dim strBuffer As String
    Dim nResult As Long

    nResult = SocketWrench1.Read(strBuffer, 1024)
    If nResult > 0 Then
        Text3.Text = Text3.Text + strBuffer
    End If
End Sub
```

The **OnRead** event indicates that data has arrived and is available to be read by the control. The **Read** method then reads the data sent by the server and stores it in the buffer specified in the first parameter. The second parameter specifies the maximum number of bytes to read. Note that in this case, it is an arbitrary value of 1,024 bytes. One important thing to note is that requesting to read a specified number of bytes does not guarantee that you will actually receive that amount. Because TCP is a stream-oriented protocol, there is no concept of a "message boundary" or a one-to-one relationship between the amount of data written to the socket and the amount of data read from it. In other words, the server sends four pieces of data in 512 byte blocks, there is no guarantee that your program will get four **OnRead** events for that number of bytes per read. Instead, you may get more than four events (in which the data sent is received in smaller blocks) or you may get fewer events, with the data being combined. This is the nature of how TCP/IP works, and must be accounted for in the design of you application. Typically this means buffering the data in the program and either looking for special "end of message" characters in the data stream, accumulating data in fixed sizes and processing it as the buffer is filled.

A good rule of thumb to follow when considering your design is thinking about how your program would work if, after asking for some arbitrary number of bytes of data, it received only a single byte. If your program is robust enough to handle this situation, then it will function correctly under a wide variety of networking environments (such as low throughput or high latency networks). On the other hand, if your program expects that it will be able to read a specific number of bytes of data at any given time, then you may find that it works correctly in under most circumstances, but intermittently fails under low bandwidth or high network load conditions.

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:

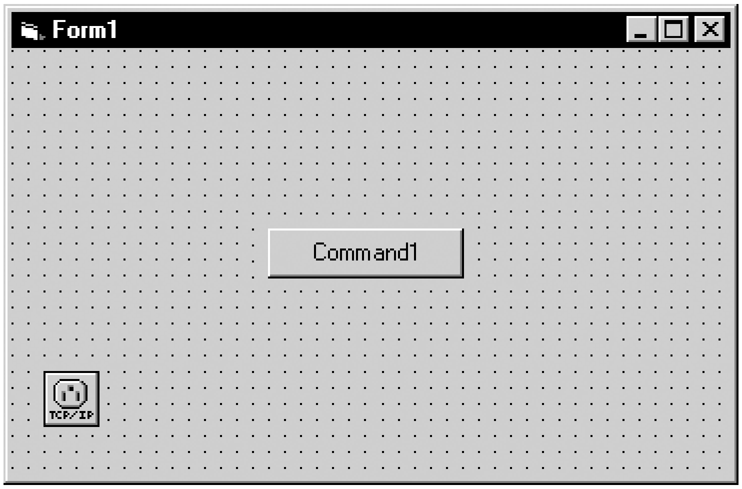
```
Sub Form_Unload (Cancel As Integer)
    If SocketWrench1.Connected Then SocketWrench1.Disconnect
    End
End Sub
```

If the **Connected** property returns True, then a connection has been established and we should disconnect from the server before the program terminates. 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 10,000 and correspond to the error codes used by the Windows Sockets API. Most errors will occur when setting the host name, address, service port or using one of the methods.

Note that if you don't have access to an echo server, then you won't be able to test your client program just yet. But don't worry, the next step is building your own server application and you can use the client to communicate with it.

Building An Echo Server

The next step is to implement your own echo server. The server will listen on the echo port, accept connections from one or more clients and echo back any data that is sent to it. First, start a new Visual Basic project with a single form, a button in the center of the form and the SocketWrench control. It might look something like this:



The first that we need to do is create a global variable called **LastSocket** which we will use to keep track of the number of clients that have connected to our server. This should be done in the general declaration section, as follows:

```
Dim LastSocket As Integer
```

Next, we will initialize the form in the Load subroutine with the following code:

```
Private Sub Form_Load()
    Command1.Caption = "Listen"
    LastSocket = 0
End Sub
```

When the user presses the command button, we want the server to begin listening for connections. Remember that the first thing that a server application must do is listen on a local port for incoming connections from a client. You'll know that a client is attempting to connect with you when the **OnAccept** event is generated for the SocketWrench control.

To accept the connection, your program calls the **Accept** method, passing the listening socket handle as a parameter. As you'll recall from the TCP/IP tutorial, 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. If you use the **Accept** method to accept the connection on the same instance of the control, you're effectively telling the control 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.

A better approach is to create an additional instance of the control and have it accept the connection, leaving the original listening socket available so that more clients can establish a connection with your server. 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 echo server.

In order to implement a dynamically-loaded control array, set the Index property of SocketWrench1 to 0. This will also cause VB to include the parameter Index in events that you implement.

To have the server begin listening when the button is pressed, we need to add code to the button's **Click** event. Initially there will only be one instance of the control in our control array, identified as SocketWrench1(0) and it will be used to listen for connections:

```
Private Sub Command1_Click()
    If Not SocketWrench1(0).Listening Then
        Dim nError As Long

        SocketWrench1(0).AutoResolve = False
        SocketWrench1(0).Blocking = False
        SocketWrench1(0).Protocol = swProtocolTcp
        SocketWrench1(0).LocalPort = swPortEcho

        nError = SocketWrench1(0).Listen()
        If nError <> 0 Then
            MsgBox "Unable to listen for connections", vbExclamation
            Exit Sub
        End If

        Command1.Caption = "Disconnect"
    Else
        SocketWrench1(0).Disconnect
        Command1.Caption = "Listen"
    End If
End Sub
```

There are two new properties here, the **Listening** property and the **LocalPort** property. The **Listening** property is a boolean flag, similar to the **Connected** property in our client example. It will return True if the control is currently listening for client connections. 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 (port 7), any other system can connect to yours and expect the program to echo back whatever is sent to it.

If the control is listening for connections and you press the button, it will disconnect the socket. This stops the control from listening for new client connections, however it will not interrupt any clients that have already connected to the server. The reason for this is because the client connections are actually managed on separate sockets which are not affected by closing the listening socket.

When our server program is executed and you press the button, the control will begin listening for client connections. When this occurs, the control's **OnAccept** event will fire. The code for this event should look like this:

```
Private Sub SocketWrench1_Accept(Index As Integer, ByVal Handle As Variant)
    Dim I As Integer

    For I = 1 To LastSocket
        If Not SocketWrench1(I).Connected Then Exit For
    Next I

    If I > LastSocket Then
        LastSocket = LastSocket + 1: I = LastSocket
        Load SocketWrench1(I)
    End If

    SocketWrench1(I).AutoResolve = False
    SocketWrench1(I).Blocking = False
    SocketWrench1(I).Protocol = swProtocolTcp
    SocketWrench1(I).Accept Handle
End Sub
```

The first thing that we do is iterate through each instance of the control in our control array, checking to see if it is connected to a client. If we find a control that is not connected to a client (in other words, the client has disconnected from the server) then we will re-use that control. If all of the controls are currently being used, then we need to increment the **LastSocket** variable and dynamically load another instance of the control.

Next, we initialize the control's properties, and then the Accept method is called with the Handle 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 **OnRead** event, which tells it that the client has sent some data to us:

```
Private Sub SocketWrench1_Read(Index As Integer)
    Dim strBuffer As String
    Dim cbBuffer As Long

    cbBuffer = SocketWrench1(Index).Read(strBuffer, 1024)
    If cbBuffer > 0 Then
        SocketWrench1(Index).Write strBuffer, cbBuffer
    End If
End Sub
```

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

```
Private Sub SocketWrench1_Disconnect(Index As Integer)
    SocketWrench1(Index).Disconnect
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 SocketWrench1(0).Listening Then SocketWrench1( (0).Disconnect
  For I = 1 To LastSocket
   If SocketWrench1( (I).Connected Then SocketWrench1( (I).Disconnect
  Next I
  End
End Sub
```

This will disconnect the listening socket so that no more clients can establish connections, and will then disconnect from each of the clients.

A Secure Echo Client

In the SocketWrench control, a number of properties have been added to support secure connections. Note that a secure connection requires a Secure Edition development license. The boolean **Secure** property controls whether or not a secure connection is established, and must be explicitly set in code prior to making the connection attempt or accepting a connection from a client. The default value for this property is False, which means that the control should establish a standard (non-secure) connection to the server. By setting this property to True, the control will attempt to establish a secure connection.

To use the secure features of SocketWrench, let's modify our echo client/server example to establish a secure connection. The first thing that we need to do is use a different port number than the standard echo port. For purposes of this example, we'll use port 7000 for our secure client and servers program. Modify the command button's Click event to use this new port number and set the **Secure** property to True.

```
Private Sub Command1_Click()

    If Not SocketWrench1.Connected Then
        Dim strRemoteHost As String
        Dim nError As Long

        strRemoteHost = Trim(Text1.Text)

        SocketWrench1.AutoResolve = False
        SocketWrench1.Blocking = False
        SocketWrench1.Secure = True

        nError = SocketWrench1.Connect(strRemoteHost, 7000)

        If nError <> 0 Then
            MsgBox "Unable to connect to remote host", vbExclamation
            Exit Sub
        End If

        Command1.Enabled = False
    Else
        SocketWrench1.Disconnect
        Command1.Caption = "Connect"
        Text2.Enabled = False
        Text3.Enabled = False
    End If

End Sub
```

Once a secure connection has been established, a number of other security related properties become available to the control. These fall into two general groups, returning information either about the secure connection itself, or about the server's digital certificate. The properties which provide information about the connection are:

CipherStrength	This property returns information about the relative strength of the encryption that is being used to secure the data. The value returned is actually the length of the key (in bits) used by the encryption algorithm, and will typically be 40, 56 or 128. A key length of 40 bits is considered to be weak, while a key length of 56 bits is considered to be moderate and 128 bit keys are considered to be very secure.
HashStrength	This property returns information about the strength of the message digest (hash) that was selected. Common values returned by this property are 128 and 160.
SecureCipher	This property identifies the encryption algorithm that was selected. The algorithms supported are RC2, RC4, DES, and Triple DES. The most commonly used algorithm is RC4.

SecureHash	This property identifies the message digest (hash) algorithm that was selected. The algorithms supported are SHA and MD5. The most commonly used message digest is MD5. This algorithm is used during the handshake phase between the client and server, and is made available to the client for informational purposes.
SecureKeyExchange	This property identifies the key exchange algorithm that was selected. The algorithms supported are RSA, KEA and Diffie-Hellman. The most commonly used key exchange algorithm is RSA.
SecureProtocol	This property identifies the protocol used to establish the secure connection. The protocols supported are SSL 2.0, SSL 3.0, PCT 1.0 and TLS 1.0.

In addition to information about the secure connection, there are several properties which return information about the remote server's digital certificate. These properties are:

CertificateExpires	This property returns the date that the server's certificate expires.
CertificateIssued	This property returns the date that the server's certificate was issued by the certificate authority.
CertificateIssuer	This property returns information about the organization that issued the certificate. The data is returned as a string which contains one or more tagged name and value pairs.
CertificateStatus	This property returns information about the status of the certificate. The client is responsible for checking this value, and based on the value returned, decide if the connection should be terminated or not.
CertificateSubject	This property returns information about the organization that the certificate was issued to. Like the CertificateIssuer property, this property returns a string which contains one or more tagged name and value pairs.

It is recommended that your application immediately check the value of the **CertificateStatus** property after the secure connection has been established. This allows your application to make the decision as to whether or not it is safe to communicate with the server based on the status of the digital certificate it returns. For example, using the above code the **CertificateStatus** property would return a value of 1 (swCertificateValid), which indicates that the certificate is valid. However, let's modify the code slightly, changing the value of the host name to the IP address for that same server. If you executed the program again, this time the **CertificateStatus** property would return a value of 2 (swCertificateNoMatch). What this tells you is that, although the certificate is valid, the name in the certificate does not match the host name that you used to connect to the server.

This is because you've specified an IP address, but the name in the certificate is normally the domain name for the server. Note that you can try the same thing with your web browser specifying the IP address instead of the domain name in the URL and you should get a warning that the site name doesn't match the certificate name.

So, let's say that for your application it is acceptable to connect to a site if the certificate is valid, or if the domain name does not match the name in the certificate; however, it is not acceptable to connect to a site where the certificate has expired, has been revoked, is untrusted or invalid. You could write code that looks like this:

```
Private Sub SocketWrench1_Connect()
    Command1.Caption = "Disconnect"
    Command1.Enabled = True
    MsgBox "Connect to remote host", vbInformation
    Text2.Enabled = True
    Text3.Enabled = True
    If SocketWrench1.CertificateStatus > swCertificateNoMatch Then
        MsgBox "The certificate could not be validated", vbExclamation
        SocketWrench1_Disconnect
        Exit Sub
    End If

End Sub
```

If you wanted to display specific information about a certificate, for example, the name of the organization that issued the certificate or the name of the company that it was issued to, you would need to use the **CertificateIssuer** and **CertificateSubject** properties. These are strings that consist of one or more values, separated by a comma. Each comma-separated value is a tagged pair of values which provides information about the certificate. For example:

C=US, O="RSA Data Security, Inc."

In this case, there are two values:

1. C=US
2. O="RSA Data Security, Inc."

Each of these values consist of an identifier called an RDN (Relative Distinguished Name) and its data. Since the second value contains a comma, it is enclosed in quotes, so this needs to be accounted for when parsing the string. There are a predefined set of RDNs defined by the X.500 standard which are used in certificates. The most commonly used RDNs in X.509 certificates are:

C	The ISO standard two character country code
S	The name of the state or province
L	The name of the city or locality
O	The name of the company or organization
OU	The name of the department or organizational unit
CN	The common name; with X.509 certificates, this is the domain name of the site the certificate was issued for

So, for example, if you wanted to determine the domain name that a certificate was issued for, you would need to read the value of the **CertificateSubject** property and parse the resulting string for the "CN" (Common Name) RDN.

The samples included with SocketWrench include code which displays certificate information and provides some general purpose routines for parsing the **CertificateIssuer** and **CertificateSubject** properties.

A Secure Echo Server

The next step is to modify the server example so that when a secure client connects to it, the appropriate certificate is passed to the client. To do this, first modify the command button's **Click** event to listen on the new port number:

```
Private Sub Command1_Click()
    If Not SocketWrench1(0).Listening Then
        Dim nError As Long

        SocketWrench1(0).AutoResolve = False
        SocketWrench1(0).Blocking = False
        SocketWrench1(0).Protocol = swProtocolTcp
        SocketWrench1(0).LocalPort = 7000

        nError = SocketWrench1(0).Listen()
        If nError <> 0 Then
            MsgBox "Unable to listen for connections", vbExclamation
            Exit Sub
        End If

        Command1.Caption = "Disconnect"
    Else
        SocketWrench1(0).Disconnect
        Command1.Caption = "Listen"
    End If
End Sub
```

Next, modify the **OnAccept** event to set the control into secure mode, and to specify the name of the server certificate that is going to be passed to the client:

```
Private Sub SocketWrench1_Accept(Index As Integer, ByVal Handle As Variant)
    Dim I As Integer

    For I = 1 To LastSocket
        If Not SocketWrench1(I).Connected Then Exit For
    Next I

    If I > LastSocket Then
        LastSocket = LastSocket + 1: I = LastSocket
        Load SocketWrench1(I)
    End If

    SocketWrench1(I).AutoResolve = False
    SocketWrench1(I).Blocking = False
    SocketWrench1(I).Protocol = swProtocolTcp
    SocketWrench1(I).CertificateName = "localhost"
    SocketWrench1(I).Secure = True
    SocketWrench1(I).Accept Handle
End Sub
```

In this case, we've set the name of the certificate to "localhost", which we'll use for the name of the local system. This can be any string which uniquely identifies a certificate installed in the system certificate store. When the client connects to the server, this is the digital certificate that will be passed to it during the handshake phase of the secure connection.

Of course, you probably don't have a server certificate installed, so if you ran this program, an error would be generated indicating that the certificate doesn't exist.

To create a certificate for testing purposes, we'll do two things. First, we'll create a self-signed certificate which establishes you as your own Certification Authority. Then we'll use that to sign a server certificate which will be placed in your personal certificate store. Note that because you're functioning as your own CA, any other systems that would attempt to connect to your secure echo server would return an error (indicating that the certificate was not trusted) until your root certificate was installed on their system.

Included in Microsoft's Platform SDK is a utility called CreateCert which will allow you to easily create digital certificates, and we've included this utility with SocketWrench. To create the self-signed certificate, enter the following from the command line:

```
CreateCert "CN=TestCA" -k s
```

This will create a file called SelfSigned.cer which contains the self-signed root certificate with a name of "TestCA". The certificate will already be installed in your own personal certificate store, however you need to install it as a trusted root certificate. To do this, use the CertMgr utility and select the Import button. This will start the Certificate Import Wizard. Select the SelfSigned.cer file, and then choose the option to place the certificate in a specified store (do not have it automatically select the store). Press the Browse button and select Trusted Root Certification Authorities. A confirmation dialog will make sure that you want to install it; once complete, your new test root certificate has been installed.

Next, we need to create a server certificate. To do this, enter the following command at the command prompt:

```
CreateCert "CN=localhost" -is TestCA my u
```

This will create a file called Certificate.cer in the current directory, and will install your certificate in your personal store. Now, with your new server certificate, you should be able to connect to your secure echo server. It should function just as the standard, non-secure version except that the data that is being sent and received is encrypted. Remember, if you want a different system to connect to your server, you need to copy the SelfSigned.cer to that system and install it in the trusted root certificate store using CertMgr, otherwise the server certificate will be considered invalid. Note that to access the secure features of SocketWrench requires a Secure Edition development license.

Debugging Applications

One of the issues that every developer has to contend with are problems that arise in an application after it's been distributed to end-users. And errors related to Windows Sockets programming can be even more difficult to track down because there are so many variables involved (such as the platform, operating system version, system configuration, and so on). To address these difficult problems, the SocketWrench control has the built-in ability to log the Windows Sockets API function calls that are made. There are three properties related to function tracing: **Trace**, **TraceFile** and **TraceFlags**. Setting these properties allows your application to dynamically manage function tracing features available to the control. The **Trace** property is a boolean flag which simply enables or disables the function tracing feature.

The **TraceFile** property specifies the name of a trace log file in which each function and its parameters will be written. If this property is not explicitly set, then a file named CSTRACE.LOG will be created in the system's temporary directory (the directory specified by the TEMP environment variable). The **TraceFlags** property specifies what type of logging will be performed by the control, and may be set to one of four values: 0 (TRACE_ALL) in which all functions will be logged, 1 (TRACE_ERROR) in which only errors will be logged, 2 (TRACE_WARNING) in which case both warnings and errors will be written to the log file, and 4 (TRACE_HEXDUMP in which all functions will be logged, together with ASCII and hexadecimal displays of all data that is sent or received on sockets. By default, all functions calls are logged by the control (TRACE_ALL).

For the SocketWrench library there are two functions related to function tracing: **InetEnableTrace** and **InetDisableTrace**. The arguments to **InetEnableTrace** are equivalent to the **TraceFile** and **TraceFlags** properties of the controls, as described above. Calling **InetEnableTrace** is equivalent to setting Trace = True, and calling **InetDisableTrace** is equivalent to setting Trace = False.

The trace file has the following format:

```
VB6 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process that is being traced (in this case, it is Visual Basic 6.0). The second column identifies if the trace record is reporting information, a warning, or an error. What follows is the name of the function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record (the value is placed inside brackets).

When reading a trace log, there are two common things that you will see:

1. The error code 10035, which corresponds to the Winsock error WSAEWOULDBLOCK is a normal occurrence on connect calls, and should not be taken as a cause for concern by itself.
2. The normal return value for a select call is greater than zero, typically a value of one. A select call that returns zero usually indicates a timeout.

If parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address) it is recorded as a hexadecimal value preceded with "0x". A special type of pointer, called a null pointer, is recorded as NULL. Those functions which expect socket addresses are displayed in the following format: aa.bb.cc.dd:nnnn

The first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. Note that in the second line of the above example, the control is attempting to connect to a system with the IP address 192.0.0.1 on port 1234.

To enable tracing in your application, you also need to redistribute an extra file called WSATRACE.DLL. This library contains the actual function tracing code, and If WSATRACE.DLL is not included with your software, the application will proceed nonetheless. If you are using the SocketWrench control, it will reset the **Trace** property to False if the tracing library cannot be loaded. If you are using the SocketWrench library, the function **InetEnableTrace** will return FALSE if the tracing library cannot be loaded. Note that this DLL will only provide function tracing capability to the SocketWrench control and library; it is not a general purpose DLL for tracing Windows Sockets functions and will not log the functions made by any other application or component.

There are several ways that you could incorporate function tracing in your software. The simplest would be a menu item or a command line switch (like /DEBUG) in which the **Trace** property would be set to True. A more complex approach would be to include a dialog or property sheet which allows the user to specify the log file name and tracing options. When an end-user calls for technical support and is encountering a problem that you think may be network related, you can instruct them to enable the tracing feature and then email or fax you a copy of the log file. In turn, if it is a problem that you don't understand, you can send the log file to a support technician who can analyze the log and provide you with additional information about what may be going on inside your application.

Remember that if you do not use the tracing features at any time during the execution of your program, there is no additional performance penalty. If you do enable tracing at some point, the tracing library will be loaded and memory will be allocated by the logging functions. These functions open, append to the trace log, flush and then close the log file for each Windows Sockets function call that is made. This insures that the last function called is logged in case of a general protection fault or other abnormal termination of the program. However, because of the file I/O overhead, it's recommended that your program rename or remove the log file before beginning a new trace.

Putting It All Together

This guide has introduced 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.

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 nineteen 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 website at **http://www.catalyst.com/products/sockettools/index.html**.

Other Products from Catalyst

Catalyst is committed to providing the tools and component software that help developers meet the increasingly complex needs of their customers.

ActivePatch

ActivePatch is a software development kit that enables developers to create their own updates in the form of a patch, and integrate the patch application process directly into their own software. Unlike other products, ActivePatch does not simply create incremental updates and re-package the files. It analyzes each file at the byte level, and determines the best method for updating the target file on the user's system. It is designed to work on both text and binary files of any type, including executables, libraries, data files and documents. ActivePatch can be used to create a patch of a single file, or can be used to create an update for a complete product, modifying existing files, removing files that are no longer needed and creating the new files that have been added.

Catalyst File Transfer

The Catalyst File Transfer Control is an ActiveX control which enables developers to easily integrate file transfer functionality within their applications. The control implements the standard protocols for sending and receiving files over the Internet and corporate intranets, and can be used in a wide variety of programming languages which can use ActiveX components. The control itself is based on the core networking and message handling code in our popular SocketTools toolkit, providing all of the features and flexibility of those components packaged in an ActiveX control with no external dependencies on third-party libraries. Included with the package is a comprehensive on-line help file, technical reference and example programs.

Catalyst Internet Mail

Catalyst Internet Mail Control is an ActiveX control which provides Internet e-mail services to applications, with the ability to compose, send and retrieve messages from a mail server. All functionality is provided by the single control, so there is no need to use multiple controls or write additional code to interface different components. The Internet Mail control uses the standard protocols for sending and retrieving messages and is desgined to work with a wide variety of servers. The control itself is based on the core networking and message handling code in our popular SocketTools toolkit, providing all of the features and flexibility of those components packaged in an ActiveX control with no external dependencies on third-party libraries. Included with the package is a comprehensive on-line help file, technical reference and example programs.