

## Using Microsoft Winsock Control

For Microsoft Visual Basic 6.0

### Introduction

The Microsoft Winsock Control ("Winsock Control") is the simplest way to get your Microsoft Visual Basic 6.0 ("VB") connected to the TCP/IP or UDP protocol. The control simplifies the connection and data transfer process so that you can spend more time developing the actual content of the application.

This tutorial assumes that you have no knowledge of networking protocols. You should be able to use the intrinsic VB controls and have a working knowledge of the basic elements of the VB language specification. You will also need access to a TCP/IP network in order to build the programs that are outlined in this tutorial.

This tutorial works as follows:

- Code that you type will be displayed as such:  
`MsgBox "Hello! You are learning VB!", vbInformation or vbOkOnly, "Hi!"`
- New keywords are displayed in bold:  
`sckTCP.senddata`
- Information about menu structures will be in quotation marks and new levels will be indicated with a right angle bracket (sometimes known as a right chevron). You are expected to click on the final item in the menu structure (in the following example, you would click Exit):  
`"File > Exit"`
- Code blocks will be indented and will include line numbers. You should not type these line numbers as they may adversely affect the performance of the program:
 

```
1: Option Explicit
2: Private Sub Form_Load()
3:   MsgBox Hello! You are learning VB!", vbInformation or vbOkOnly,
   "Hi!"
4: End
5: End Sub
```

### Table of Contents

- 1.0 [Review of Controls](#)
  - Review of the different parts of a control
  - Review of how to add non-intrinsic controls to a project
- 2.0 [Connections](#)
  - Connecting two simple TCP/IP programs together
  - Disconnecting two simple TCP/IP programs
- 3.0 [Sending Data](#)
  - Sending string values
- 4.0 [The Error Object](#)
  - Detecting errors in a program
  - Recovering from errors

#### 1.0 — Review of Controls

One of VB's most powerful features is its full support for ActiveX controls. ActiveX is one of Microsoft's ("MS") many solutions to the increasing need for consistent user interface ("UI") elements and also the increasing demand for dynamic elements. VB comes with two types of ActiveX controls: Intrinsic and ActiveX control. Both types work exactly the same; their level of support and integration into the VB language varies. All of intrinsic controls are preloaded when VB loads and are usually shown in the toolbox. Intrinsic controls include:

- CheckBox Control
- ComboBox Control
- CommandButton Control
- Data Control
- Directory List Box Control
- Drive List Box Control
- File List Box Control
- Frame Control
- Function Control (Data Report Designer)
- Horizontal and Vertical Scroll Bars Control (two controls)
- Image Control (Data Report Designer)
- Label Control
- Label Control (Data Report Designer)
- Line Control
- Menu Control
- OptionButton Control
- PictureBox Control
- Shape Control
- Shape Control (Data Report Designer)

- TextBox Control
- TextBox Control (Data Report Designer)
- Timer Control

Some of the more popular ActiveX controls are the CommonControls, CommonDialogs, MS FlexGrid, and, of course, the Winsock Control! To access the controls, do the following:

1. Open a New EXE Project (as you would most likely normally do)
2. Goto "Project > Components". It will take a few seconds for the dialog box to load
3. Scroll down until you find the 'Microsoft Winsock Control 6.0'. It should be near the middle of the list, as the controls are listed in alphabetical order.
4. Place a check mark the CheckBox beside 'Microsoft Winsock Control 6.0'
5. Click OK

The process outlined above is known as **adding components**. You need to do this every time you start a new project involving the Winsock Control. You can also use this process to load other ActiveX controls, such as the CommonDialog control.

Each ActiveX control is made up of three important components: **Events**, **Properties** and **Methods**. If you program using the Object Oriented Paradigm, you may be familiar with two of these words. Understanding these terms will allow you to quickly learn any new control

Properties are the most basic function of an ActiveX Control ("control"). They are a semi-complicated version of a variable, but contain many advantages. As you know, variables can be either Public or Private. There are these two types because programmers want to ensure that the internal workings are not subject to a careless programmer two years down the road, but need a type of variable that can be accessed from outside their code, otherwise the set of code would never be accessed! However, there is still the problem that at certain times the programmer may want to have a line of code execute when the value changes, or prevent the user from the reading or changing the value altogether. To set the value of a property, you do the following:

```
Control.property = "Value to give the control" 'Property is assigned the
string
```

To get the value of the property, do the following:

```
Variable = Control.property 'Variable is assigned the value of the property
```

Methods are the most popular and common part of a control. The methods are the blocks of code that are called for the sole purpose of making something happen. Methods are usually passed values, called **arguments** to modify the way they work. As with properties, there are both Public and Private methods. Usually methods return a value to indicate the status of what happened during the execution. When you **invoke**, or call, a method you must decide whether you will be expecting a value to be returned. If a value is to be returned, then the arguments should be enclosed within parenthesis. If no value is to be returned, the arguments must not be enclosed within parenthesis. This is how to invoking a method:

```
Control.Method Argument1, Argument2, Argument3 'No value is returned
Variable = Control.Method(Arg1, Arg2, Arg3) 'value is assigned to Variable
```

Events are the final topic of interest. An event is **raised** by the control so that you know when an important 'event' happens. This means that your code can respond to various things the control does. In the VB IDE, you access the Events in the right most ComboBox in the code window, of the specified control in the left most ComboBox.

This section reviewed the basics of ActiveX controls. At this point you should be comfortable adding ActiveX controls that are not classified as VB's intrinsic controls. You should be comfortable with placing a control in the VB project and giving it a name. You should also be comfortable with the terminology of "event", "property" and "method".

Next section you will develop your first Winsock application!

## 2.0 — Connections

The Winsock control allows for connections to be made with incredible ease. Before discussing how to connect to a remote computer, it is necessary to plan out how your connection will work. The nature of the Winsock control lends itself to a **Client/Server** connection model.

A client/server connection model involves having two classes of computers. One class is the **server**, the other class is the **client**.

The server is a central computer that is responsible for linking the many computers together. Your server may or may not function as a player of the game. You should be very careful when deciding whether or not the server should be a player. On one hand, the user who has their computer set up to work as a server will probably want to also play in the game. If the user can not play from the server, they will have to run your program again and *connect to themselves* in order to play. On the other hand, if the server is also a player and your game experiences a difficulty the game will most likely end for all the players.


The client is one of the many computers that connect to the server. They will need to know the location and address of the server computer in order to play on that server. The client computer does not directly talk to the other client computers. It passes messages through the server which **echoes** said messages to the other clients.

NOTE: You may have heard of other connection models. One of the most popular ones is the Peer-to-Peer model. This method is not inherently supported by the Winsock control and is not discussed in this article. Some APIs have native support for these protocols, such as DirectPlay. More information on DirectPlay can be found at [MSDN](#).

Now that we understand how the computers will connect to each other, we can begin to learn some of the basic requirements to make a connection. Just like a building has unique forms of identification, the computer also has unique forms of identification. The computer's address is four bytes long and takes the form "xxx.xxx.xxx.xxx". This address, simply known as an **IP** address is the basis for most communication on a computer. A sample IP address is 192.168.0.100 . However, the computer also requires a second address, known as the **port**. A port is what the data goes through when it comes into the computer; it separates the data so that one program does not get another program's data. There are many thousands of ports available on the computer. Your program must just be sure to select a port that is not in use by a different program!

Our completed model can be drawn as follows:

It is important to

 Radial Diagram

note in this diagram that *many* computers are able to connect to the one server at the same address and on the same port. Windows will automatically ensure that the data goes into your program in the same order that it arrived at the computer.

This process may seem quite complicated: connecting computers through sockets and opening ports for access to other computers, however, the Winsock control handles all this for us! The Winsock control uses the following methods, properties and events to simplify the process for us:

- .Accept (method) — This method receives the request from a client computer to connect to the server. The server calls the method in order to connect the remote client
- .Connect (method) — This method sends a request to the server computer for a connection
- Connect (event) — This event is called when the connection has been established
- ConnectionRequest (event) — This event is called whenever a remote computer asks for a connection
- .Listen (method) — This method is called to allow remote clients to connect to the server
- .LocalIP (property) — This property returns the computer's IP address
- .LocalPort (property) — This property returns the port that will be used for incoming connections
- .RemoteIP (property) — This property returns the server's IP address to a client. The server CAN NOT use this property to determine what the client's IP address

**APPLICATION -> "Connecting Two Computers"**

This application will connect two computers together. We will need to actually make *two* applications: one will be the server and the other will be the client.

SERVER:

FORM:

The server will use a form that looks like the following:



This form contains a label with the caption "Server IP". When the program starts, it will also show the server's IP address. The textbox allows the user to type in the port that they wish to have computers connect to them on. There is a CommandButton that allows users to exit the program. Finally, the Winsock control is located just the left of the exit button. It is important to note that the Winsock control has no visual component; it can be placed anywhere on the form and will always work the same.

The following names were given to the controls:

wskServer -> Winsock Control

txtPort -> Textbox Control

lblIP -> Label Control that is currently set to "Server IP"

cmdExit -> CommandButton

CODE:

The code for this program is not overly complex. This first bit of code that we can write is the Exit button. In the Click event, add the following line of code.

```
Unload Me
```

This line of code ensures that the Form's Unload event is called. We have to add code into the Unload event that will 'clean up' the Winsock control. The Winsock control must close all of the ports that it 'opens' (for the clients to connect through) or else the program may not be able to use this port next time. This is sometimes known as a **memory leak**. Place the following code in the Form's Unload event:

```
wskServer.Close
```

This line of code tells the Winsock control to close the port that is currently open. The next piece of code that we'll add will display the server's IP address in the lblIP. The code to detect the IP will go in the Form's Load event:

```
lblIP.Caption = lblIP.Caption & " " & wskServer.LocalIP
```

The line of code asks the Winsock control for the Local IP address and then assigns that value into the label's caption property. Next, code will be written into the textbox's validation. This event is executed when the textbox loses focus. At this point we will disable the textbox so that the user can not change the port while we are waiting for the client to connect! The following code will be added:

```
If IsNumeric(txtPort.Text) then 'Make sure the user entered a number
    wskServer.LocalPort = CLng(txtPort.Text) 'Winsock needs a "long" port
    txtPort.Enabled = False 'No more changes!
```

```

wskServer.Close 'Prepare the control for use
wskServer.Listen 'Wait for a client to connect!!!
```

```
Me.Caption = "Winsock -- Server: Listening" 'Tell the user we're ready!
```

```
Cancel = False
```

```
Else
```

```
Cancel = True 'Don't let the focus leave the textbox
```

```
End if
```

Although in a real world situation this is definitely not the optimal situation, you will be able to see what happens when you use various ports. In a real world program, you will want to **hard code** the port into the program (this means that the user is not permitted to change the port because it is a physical part of the program). Right now the control has prepared itself for a connection from the client computer. All that remains to be programmed is the code that accepts the connection. This code will go in the Winsock control ConnectionRequest method:

```
1: Private Sub wskServer_ConnectionRequest(ByVal requestID as Long)
2:     wskServer.Close
3:     wskServer.Accept requestID
4:     DoEvents
5:     MsgBox "A client connected!"
6: End Sub
```

This sub does most of the work. It is passed the client's ID number (this is not their IP address). Line 2 closes the Winsock control. You must do this to ensure that nothing is sending data to that control. Line 3 accepts the client. Line 4 allows all the processes to update, this includes ensuring that the client computer receives the

automatic message that it has connected. Line 5 displays a message box so that the server knows that someone connected.

#### NOTES:

This program demonstrated the basic properties, methods and events in preparing a server application. You should be sure to remember that you must invoke the close method before trying to change ports, or accept a new user.

#### CLIENT:

##### FORM:

The client will use a form that looks like the following:



This program contains two CommandButtons. One of them will be used in conjunction with input boxes to determine the IP address to connect to and the port to connect through. The other CommandButton is used to exit the program and terminate any open connections. There is a Winsock control (hidden by the CommandButtons) that will be used to make the connection

The following names were given to the controls:

cmdConnect → The CommandButton with the caption "Connect"

cmdExit → The CommandButton with the caption "Exit"

wskClient → The Winsock control

#### CODE:

The exit button contains the following code:

```
Unload Me
```

This instructs the form to "self-destruct". It also calls the Form's Unload event where code will be placed to clean up our Winsock control. In the Form's unload event, the following code is written:

```
wskClient.Close
```

As in the previous program, this code prevents a memory leak and also ensures that the server receives a message indicating that the client disconnected. Finally, we will add the code that connects the client to the server:

```
1: Private Sub cmdConnect_Click()
2:   Dim lngPort as Long
3:   Dim strIP as String
4:
5:   strIP = InputBox$("Enter the IP address to connect to:")
6:   If strIP = "" Then Exit Sub
7:
8:   lngPort = CLng(InputBox$("Enter the port to connect through:")
9:
10:  wskClient.Close
11:  wskClient.Connect strIP, lngPort
12: End Sub
```

You now have enough code to connect two computers together. Lines five through eight get the required information. More aggressive error checking should be applied in a full-blown application, however the IP address **must** be stored as string and the Port **must** be stored as a "Long". You should close the Winsock control before connecting to ensure that there are no current connections. To determine when the client connects to the server, place a message box in the "wskClient\_Connected" event.

#### NOTES:

This program demonstrates the basic code required to create a client application. Obviously extensive error checking would need to be employed if an InputBox was to be used to get the information. One point of interest is the requirement of closing the control before you connect to another computer. The Winsock control has an "ether" state where the control is not open or closed, so it is important to explicitly close the connection!

That's it for this application! To run the application, you'll need to start the server on one computer. Set the port button and then hit the TAB key. This will cause the server to listen for connections. Once the form's caption displays "Listening", it's time to fire up the client on another computer. Click on the "Connect" button. Enter the IP address from the server computer. Next enter the port number that you typed into the server computer. If all goes well, the server computer will display a message box indicating that a client has connected. If you entered any code into the client application's "wskClient\_Connected" event, then you will receive notification from the client as well.

In this section you learned the basics of the Client/Server model. You learned how to prevent a Winsock memory leak and also how to connect a client to the server. These are the foundations for every other Winsock program, although you may employ a more elegant UI and a more proficient error debugging system.

Next section you will learn how to send data between two active connections. You should try designing an application that connects to a server in order to get comfortable with connections in Winsock. In future sections it will be assumed that you can make the connection, so applications will not have the part shown.

### 3.0 — Sending Data

This section will discuss how to send data between an open Winsock connection. With Winsock you must determine what type of data will be sent and how it will be received. If you are just sending short strings, you can safely use string variables, however if you are attempting to send files that contain binary data you will need to employ a more careful section to ensure against data getting mistaken for other data. It is beyond the scope of this tutorial to discuss how to send binary data safely, however a smart programmer should be able to figure out a system to send the binary data if necessary upon completing this tutorial.

The data that is sent and received by Winsock is sent through a **buffer**. A buffer holds excess data until it can be sent. The Winsock control automatically places the data on the buffer and retrieves data from it. In fact, the Winsock control will even tell you when new data arrives on the buffer!

The sending and receiving of data through a connection is not really a difficult feat, especially when you use the Winsock control. The Winsock control offers the following methods, properties and events for sending and receiving data:

- .BytesReceived (property) — Returns the number of bytes that have been retrieved through this connection
- .DataArrival (event) — Occurs when data is ready to be received by the Winsock control (in the buffer)
- .GetData (method) — Retrieves data from the Winsock control buffer and then removes it from the buffer
- .PeekData (method) — Retrieves data from the Winsock control buffer, but does not remove it from the buffer
- .SendComplete (event) — Occurs when all data has been sent
- .SendData (method) — Moves data onto the Winsock buffer and then sends it
- .SendProgress (event) — Occurs during process of sending data

Using those basic methods you can send and receive data quite easily.

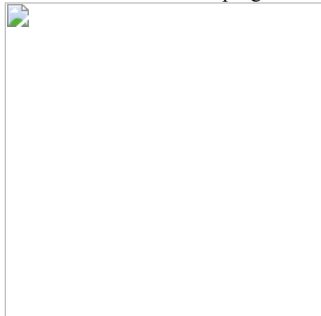
#### **APPLICATION -> “Sending Data Between Two Computers”**

This application will connect two computers together and then will transfer a message between the computers. Winsock doesn't care who the server is or client with regards to sending data, so only one application will be made; the user will specify if they would like to be the server or the client.

SERVER/CLIENT:

FORM:

This program will use a form that looks similar to the following:



The user will either click the Host or Connect button, at which point in time the Send button will become enabled. When a message arrives, it will be placed in the multi-line textbox. The following names were given to the controls:

cmdExit → CommandButton with the “Exit” caption  
 cmdSend → CommandButton with the “Send” caption. It is initially set to disabled  
 cmdHost → CommandButton with the “Host” caption.  
 cmdClient → CommandButton with the “Connect” caption  
 txtData → Textbox control  
 tcpWinsock → Winsock control

CODE:

Code should first be placed in the Exit button to avoid a memory leak and also terminate the application. Next, code should be placed within the Host button to “listen” for a connection and code should be placed in the Connect button to connect to the remote server. In the \_Connect and \_ConnectionRequest events of the Winsock control, code should be written to enable the send button. Keep in mind that if the user clicks the Host button, the Host and Connect buttons should be set to disabled, and the Send button shouldn't be enabled until the \_Connect and \_ConnectionRequest events have been successfully completed.

The next step is to program in the reception of new data. In the tcpWinsock\_DataArrival event, add the following code:

```
1: Dim strTemp as String
2:
3: tcpWinsock.GetData strTemp
4: txtData.Text = strTemp & vbCrLf & txtData.Text
```

This code calls the Winsock control's method to retrieve data and then places it in the temporary variable “strTemp”. At this point you could do some fancy string manipulation to extract message information, however this program doesn't. The string is simply added to the top of the textbox. The only thing remaining to be programmed is the code to send data. The following code should be placed within the Send CommandButton:

```
1: Private Sub cmdSend_Click()
2:   Dim strMessage as String
3:
```

```

4:    strMessage = InputBox$("Input the message to send") 'Get the
message
5:    If strMessage = "" then Exit Sub 'If the user clicked cancel, leave
sub
6:
7:    tcpWinsock.SendData strMessage 'Send the message
8: End Sub

```

The message is retrieved from the user by means of an inputbox (lines 4-5) and is then sent by the Winsock control on line 7.

#### NOTES:

That's all there is to sending data with the Winsock control. Click [here](#) to see the complete program listing if you do not feel comfortable with connections yet. Obviously some sort of system must be implemented to ensure that the user is even connected to someone before sending data, and that the data the Winsock control receives is legitimate!

You have now completed an application that can send data between two computers and then retrieve it! You will need to run this program twice, once as a server and then again as a client. You can click the Send button to send data to the other computer, after you connect.

In this section you learned how to send data using the Winsock control. You about buffers and how messages are stored on a buffer before being sent to the remote user, and how the message is stored on the buffer before being downloaded again by the Winsock control.

Next section you will learn about how to detect and prevent errors.

## 4.0 — The Error Object

Now that you have the skills to make a fully functional Winsock application, you need to know how to avoid errors. If you've tried to "enhance" any of the sample applications, you've undoubtedly come across errors. There are three parts to ensuring your program has excellent error checking: finding errors before they occur, placing code that occurs whenever an error occurs, reacting to the Winsock error event.

The most important thing to do in your code is to ensure that an error is highly unlikely to happen. While there are obviously many things that need checking before sending data, the most obvious one is, "Is there a connection"? In the previous application, it was nearly impossible to send a message without making a connection simply because the commandbutton was not enabled until a connection was made. However this is very tedious to enact on a much larger scale, especially because sometimes the program will be sending non-user drive events. The most useful property for helping you detect errors before they occur is the .State property. Along with the Winsock State Constants you can easily determine what 'state' the control is in.

Constant	Value	Description
<b>sckClosed</b>	0	Default. Closed. When the control is in this state, you can safely invoke the listen method or make changes to ports, and IPs
<b>sckOpen</b>	1	Open. This means that the port is open, however your program is not using the port. You must attempt to close the port before you can use it. You may want to switch to a different port.
<b>sckListening</b>	2	Listening. Your server is preparing to accept more connections. Your client should NEVER be in this state!!!
<b>sckConnectionPending</b>	3	Connection pending. Your client has sent a request for a connection, however the server has not yet authorized the connection. In a few seconds there will probably be a connection or you will be denied.
<b>sckResolvingHost</b>	4	Resolving host. The client and the server are "talking" about how the connection will work.
<b>sckHostResolved</b>	5	Host resolved. The connection is ready to go. However, you still cannot send data yet.
<b>sckConnecting</b>	6	Connecting. The client computer is in the process of connecting to the server computer.
<b>sckConnected</b>	7	Connected. Your client has connected to the server. At this point you can safely send and receive data.
<b>sckClosing</b>	8	Peer is closing the connection. This "ether" can cause many memory leaks. If the state is ever set to this, you should simply call .Close and then continue on.
<b>sckError</b>	9	Error. Something terrible has happened. Continue reading this section!!!

The state property can be used with a simple condition structure. For example if you are wanting to send data, have a function that checks the current to see if it is "sckConnected".

The next method of error checking involves using the Visual Basic error handling commands. While there is a detailed error "object" (similar to a control but with no visual interface), it is beyond the scope of this tutorial to delve into that object.



In order to use the error object, you need to place a command to “turn it on”. The command takes this form:

```
On [Local] Error {Resume Next|Goto <Label>}
```

If you choose the “Local” option then error handling will occur only within your current procedure or function. If you choose not to have this option, then all procedures and functions will *also* have this error checking (unless, of course, they have their own error handling commands — but don’t worry, once your procedure or function resumes, your error checking method will resume). If you select “Resume Next” then the program will continue on even if there is an error. It is assumed that you would simply put error handling code immediately after the line that is suspect. This method is preferred if you know that there is a specific line that may cause an error. The method of “Goto <Label>” allows you to place a label (any non-keyword followed by a colon (e.g. Failed: ) is considered a label, all labels must be within the procedure or function that you are currently in) that the code will jump to if an error occurs. This allows you to route all error to a common spot. An example may be:

```
1: On Local Error Goto Failed:
2:
3: Result = intNumberOne / intNumberTwo 'If user sets intNumberTwo to zero...
4:
5: Exit Function 'Leave the function so that error code doesn't run
6:
7: Failed:
8: Result = 0 'An error occurred. Handle it!
```

This code will watch out for all errors, however the one that is suspected to happen is division by zero. While this could be easily handled with a “Resume Next” option, it really doesn’t. If the user did try to divide by zero, the code would have jumped to line 7 (you could place code on this line, however it is usually considered ‘cleaner’ to not put code on this line). Finally, if for some reason you need to turn all error checking *off* you can do the following:

```
On Error Goto 0
```

The code tells the computer to not go anywhere on an error. This can be extremely useful during debugging because you want to see the error during the preliminary stages of program! On a final note, your code should be designed with the error checking in it from the start, not as an after thought. When you come back to your code you may have forgotten typical errors that occurred and will be left having to put in very generic code.

The final method of error handling is using the Winsock’s Error event. This event occurs whenever an error occurred. You get a chance to react to the Winsock error. Unfortunately most of the errors that get routed here will result in you having to gracefully shutdown the multi-player system. Whenever possible you should attempt to handle all errors before it becomes necessary for the Winsock Error event to become invoked. You can determine which Winsock error occurred by comparing the “Number” property to the following constants:

Constant	Value	Description
sckOutOfMemory	7	Out of memory
sckInvalidPropertyValue	380	The property value is invalid.
sckGetNotSupported	394	The property can't be read.
sckSetNotSupported	383	The property is read-only.
sckBadState	40006	Wrong protocol or connection state for the requested transaction or request.
sckInvalidArg	40014	The argument passed to a function was not in the correct format or in the specified range.
sckSuccess	40017	Successful.
sckUnsupported	40018	Unsupported variant type.
sckInvalidOp	40020	Invalid operation at current state
sckOutOfRange	40021	Argument is out of range.
sckWrongProtocol	40026	Wrong protocol for the requested transaction or request
sckOpCanceled	1004	The operation was canceled.
sckInvalidArgument	10014	The requested address is a broadcast address, but flag is not set.
sckWouldBlock	10035	Socket is non-blocking and the specified operation will block.
sckInProgress	10036	A blocking Winsock operation in progress.
sckAlreadyComplete	10037	The operation is completed. No blocking operation in progress
sckNotSocket	10038	The descriptor is not a socket.
sckMsgTooBig	10040	The datagram is too large to fit into the buffer and is truncated.
sckPortNotSupported	10043	The specified port is not supported.
sckAddressInUse	10048	Address in use.
sckAddressNotAvailable	10049	Address not available from the local machine.
sckNetworkSubsystemFailed	10050	Network subsystem failed.
sckNetworkUnreachable	10051	The network cannot be reached from this host at this time.

<b>sckNetReset</b>	10052	Connection has timed out when SO_KEEPALIVE is set.
<b>sckConnectAborted</b>	11053	Connection is aborted due to timeout or other failure.
<b>sckConnectionReset</b>	10054	The connection is reset by remote side.
<b>sckNoBufferSpace</b>	10055	No buffer space is available.
<b>sckAlreadyConnected</b>	10056	Socket is already connected.
<b>sckNotConnected</b>	10057	Socket is not connected.
<b>sckSocketShutdown</b>	10058	Socket has been shut down.
<b>sckTimeout</b>	10060	Socket has been shut down.
<b>sckConnectionRefused</b>	10061	Connection is forcefully rejected.
<b>sckNotInitialized</b>	10093	WinsockInit should be called first.
<b>sckHostNotFound</b>	11001	Authoritative answer: Host not found.
<b>sckHostNotFoundTryAgain</b>	11002	Non-Authoritative answer: Host not found.
<b>sckNonRecoverableError</b>	11003	Non-recoverable errors.
<b>sckNoData</b>	11004	Valid name, no data record of requested type.

There is one major problem that occurs when connecting that is extremely annoying to debug: Time-out errors. While these errors can occur frequently through out the program, during connections they can be gracefully handled. The following code demonstrates how a client can manually track down a time-out error during connection:

```
wskTCP.Connect IP, CLng(txtPort.Text) 'Attempt to connect

lTime = 0 'Initialize the counter

Do While (Not bReplied) And (lTime < 500000)
    DoEvents 'Continue to attempt to connect
    lTime = lTime + 1 'Increment counter
Loop
```

This code snippet uses a public variable (bReplied: Boolean variable) to determine when a connection is made. If the variable is not set to true before the lTime variable reaches 500 000, then it is assumed that a time-out error will occur. The socket can then be closed after a message box is shown.

That concludes this tutorial!!! You should now have the skills required to create complex Winsock applications. If you are interested in learning more about the Winsock control, check out the following websites:

- <http://www.developerfusion.com/show/78/>
- [http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/mswnsk98/dt\\_vbobjWinsockControl\\_C.asp](http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/mswnsk98/dt_vbobjWinsockControl_C.asp)

Of course, you can always e-mail me your questions: [kevin\\_0011@hotmail.com](mailto:kevin_0011@hotmail.com)

