Write a Control to Manage a POP3 Server

Click & Retrieve Source CODE!

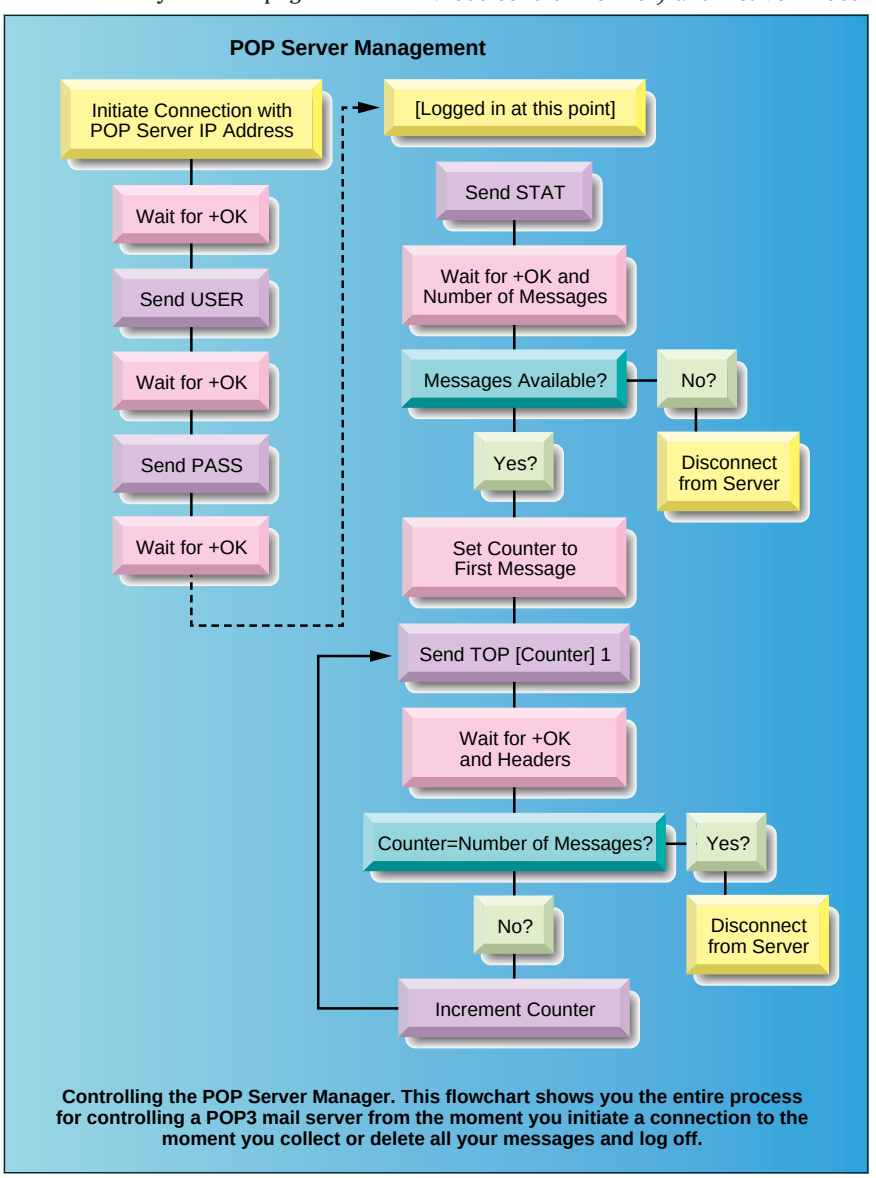
BY RON SCHWARZ

Use VB5 and the MS Winsock control to create custom controls that access the Internet.

VB5 provides a variety of Internet-oriented features. One of the most exciting is the ability to create Internet-aware ActiveX controls. These controls can operate in traditional applications, or within the context of a Web browser. They can also send and receive data over the Internet. Naturally, these features also apply to intranet apps. This article examines some of the ways to create and use Internet capabilities in the controls you create in VB5. It shows you how to create an Internet control that manages a Post Office Protocol version 3 (POP3) mail server to view messages without downloading them to your e-mail program. You can view the headers, selectively view message bodies, and delete individual messages on the server.

VB5 offers so much new material that it can be a bit daunting, even to experienced users. Rather than cover the essentials of control creation, this article focuses on implementing Internet access, and on showing you how to package controls for deployment over the Internet. These are two separate issues, and it's important to note that you can use the same controls in your standalone applications *and* in your Web pages. Web deployment, by the way, takes two forms: HTML (either plain vanilla HTML or layout control "forms") and ActiveX Docu-

Ron Schwarz is coauthoring a book with Ibrahim Malluf with the working title "Visual Basic 5 for the Enterprise," to be published by Addison-Wesley. He and Ibrahim also wrote Special Edition Using VBScript *(Que). Reach Ron at ron.schwarz@nethawk.com.*



Controlling the POP Server Manager. This flowchart shows you the entire process for controlling a POP3 mail server from the moment you initiate a connection to the moment you collect or delete all your messages and log off.

ments. For our purposes, you can think of ActiveX Documents as traditional applications because they use controls in essentially the same way.

The other issue of merit, besides Internet deployment, is Internet communications. This encompasses such things as live content—receiving information from a server without having to submit and refresh a Web page—and two-way communications. The AsyncRead method of ActiveX documents and controls facilitates live content. The Fetcher example in this article demonstrates this (see Figure 1). It implements a simple Web browser using nothing more than VB5 code and standard (non-OCX) controls.

Although the AsyncRead method is only available in ActiveX Document and Control contexts, this doesn't prevent you from using it in standalone apps. Simply create a control to provide its functionality. If you don't need or want to use the control outside of the project you're creating, you can simply include it in the project without compiling an OCX file. The Fetcher example does exactly this.

Although the control contains a bit of code that is extraneous to the AsyncRead method, it's easy to create with the ActiveX Control interface Wizard supplied with VB5. There are two main items of interest here: the actual AsyncRead call, which sends out the request for the URL you're getting, and the AsyncReadComplete event, which fires when the URL is received. Wrapping the AsyncRead method for the Fetcher example is not complicated:

```
Public Function FetchURL(URL As _
   String, hURL As String) As Long
   AsyncRead URL, _
      vbAsyncTypeByteArray, hURL
End Function
```

SETTING UP FETCHER

The FetchURL function is a wrapper that allows non-Control code to use the AsyncRead method. It accepts two strings from the calling code: URL and hURL. URL contains the address of the URL you want to fetch, and hURL contains a "handle" of sorts to the particular URL in question. You need to be able to tell which is which when they roll in because you can have multiple URL requests pending at the same time. Passing a unique string in the AsyncRead method's optional PropertyName property causes the string to show up in the PropertyType property of the AsyncProperty object that also contains the actual URL contents when the AsyncReadComplete event fires. By testing this value, you can tell which URL is which. The AsyncReadComplete event in the Fetcher example receives the URL, and notifies the container (the form that holds the control) by passing this information along to the DocReceived event. Fetcher then fires the DocReceived event. Yes, VB5 also allows you to create your own events.

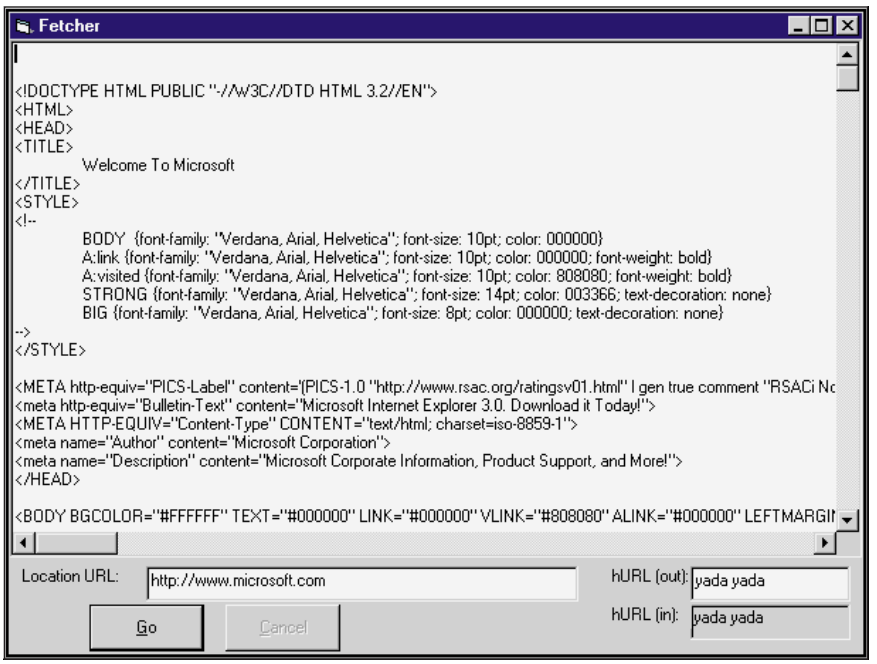


FIGURE 1 ***Playing Fetch.*** *The Fetcher example captures the HTML source code from a URL, using simple native VB code.*

```
Private Sub
UserControl_AsyncReadComplete _
   (AsyncProp As AsyncProperty)
On Error Resume Next
   RaiseEvent DocReceived(StrConv_
      (AsyncProp.Value, vbUnicode), _
      AsyncProp.PropertyName)
   If Err Then
      CancelURL AsyncProp.PropertyName
      RaiseEvent DocReceived_
      (Err.Description, "Failure")
   End If
End Sub
```

The AsyncReadComplete code includes a second call to raise the DocReceived event. This is for cases that generate an error, and it gives you an easy way to pass along error information. You can also use it to issue a CancelAsyncRead call to clean things up in case of error. You can use a wrapper procedure (the CancelURL method) to invoke it from outside the control as well.

If you've wondered about the purpose of the vbAsyncTypeByteArray constant in the AsyncRead call, it's because the data you're requesting in this case is a string of text. AsyncRead can handle three types of requests: files, byte arrays, and pictures. When you want to bring in a string, you need to request the URL as a byte array, and use the StrConv function to convert it to a string. The other constants are vbAsyncTypeFile, which receives a file, and vbAsyncTypePicture, which receives a Picture object.

When the data rolls in, it's contained, along with its type and "handle," in the AsyncProperty object. This object is returned as the only parameter of the AsyncReadComplete event. It has three properties: Value, which is a variant that contains the actual URL (a variant is necessary because the data can take one of several possible forms); PropertyName, which is a string that contains the "handle" you provide; and AsyncType, which is an integer matching one of the three URL type constants (vbAsyncTypeByteArray, vbAsyncTypeFile, or vbAsyncTypePicture).

The code for Fetcher is fairly straightforward, and it should help you if you're trying to implement remote retrieval of Web content. I'd be remiss if I didn't mention that although I'm using AsyncRead as a general purpose "fetcher," its main reason for existing is to provide asynchronous download of properties for controls you create. Let's say you've written a control that displays a chart representing the distillation of a large quantity of data. The AsyncRead method lets the control paint its container and bring in its contents as the rest of the Web page remains interactive. You can use this approach rather than make the control's properties download as part of the control. This allows you to keep from locking up the browser until the lengthy process is complete. Yes, it cheats a bit, but it sure proves handy.

INTERACTING WITH OTHER MACHINES

Retrieving Web data with the AsyncRead

method is easy and powerful, but it only addresses the need for information that is available from HTTP servers. There is a lot of information on the Internet that is *not* "on the Web." If you want to work with this information, you need to dig in and work with lower-level Winsock features. The Winsock control makes it fairly simple to connect to other computers for nearly anything. Of course, there are tradeoffs. You must work with data coming to you in bits and pieces, and you need to understand what to request and how to request it. The POP Manager example shows how to control a POP3 mail server to examine messages without downloading them to your e-mail program. You can view the headers, selectively view message bodies, and delete individual messages on the server (see Figure 2). This proves quite useful for cases where your e-mail client gets confused and refuses to delete messages it has already downloaded. Those "phantom" messages can eat up a lot of server space, and eventually cause you to hit your quota, causing incoming e-mail to bounce. It can also be handy if you're being mailbombed, or inundated with "spam" advertising that you don't want to download.

The POP Manager is a single ActiveX control that contains a handful of standard controls and the Winsock (MSWNSK.OCX) control. You can drop it on a VB form or a Web page, and have an instant POP server manager. If you place it on a Web page, everyone accessing it can use it to manage his or her own POP accounts.

Using host services over the Internet usually means studying one or more "RFC" documents. You can acquire an index and listing of these documents at http://andrew2.andrew.cmu.edu/rfc. RFC means "Request For Comments," which can be a bit misleading because it's really the end result of a series of discussions. By the time most people see an RFC, it's an established standard. RFC1939 is the RFC that defines the functioning of POP3 e-mail servers. A POP server is a program that stores incoming mail at your Internet provider's site until your e-mail program can download it. You can experiment with the commands and behavior documented in RFC1939 with any telnet program by telnetting to the server's IP address and the POP port (110), then entering the data the server expects. However, it's not productive to do this on a regular basis.

The POP protocol is fairly simple. Commands receive a response beginning with either "+OK" (if successful) or "-ERR" (if unsuccessful). Note that commands are not echoed back to the sender. Message bodies (or headers) are terminated with a line consisting of a dot (".") that is terminated by a CR/LF.

ISSUE COMMANDS TO A POP3 SERVER

The POP protocol contains only a few commands. The first two, USER and PASS, accept your ID and Password, and must be executed successfully before you can use any of the other commands. STAT returns the number of messages and their total size, LIST returns information on individual message sizes, RETR retrieves an entire message, TOP receives headers (and lets you specify a number of lines of body text to retrieve with the headers), DELE deletes a message, RSET resets deletions, and QUIT commits deletions and logs you off the server. There are other commands not included in this list, but these are the most common commands needed for accessing a POP3 server. Premier Level members of The Development Exchange (DevX) can pick up a copy of the RFC online. See the Code Online box at the end of this article for details.

Conversing with a POP server is a good tutorial for Internet communications because it's fairly simple, yet requires some

Speaking Out

"The point at which we stop ogling about how cool a technology is and just start using it in our everyday coding is when that technology becomes truly useful. VB5 allows us to develop apps and applets that use, in a stable way, 32-bit, object-oriented, multitiered, Internet-enabled technology."

Andrew J. Brust, president, Progressive Systems Consulting Inc.

handshaking and interaction between the client software and the server. You can talk to any type of host after you learn how to talk to a POP server, as long as you have access to its RFC or other documentation.

The actual conversation takes place on three levels. The low level manages the actual connection between the server and the client, the mid level handles the transport of raw data to and from the server, and the high level manages the commands to and replies from the server.

To initiate a connection with the POP server, use the Winsock control's Connect method. I named the Winsock control "tcpIn" because it uses TCP/IP to retrieve information. The Connect method expects an IP address (the name of the server) and a port number (110 in the case of POP mail). This code allows you to establish the lowest level connection between the two machines:

```
On Error Resume Next
tcpIn.Connect modPOPMgr.POPServer, 110
If Err Then
   FatalError = True
   Err = 0
   Exit Function
End If
On Error GoTo 0
```

It's necessary to trap for errors when connecting because you may find a site that's down or incorrectly spelled. The code to connect to the server passes the server's address in a property of the MODPOPMGR.BAS module. I added properties representing all necessary server and user information to that module, and they are persisted in the local registry. This means that you won't have to re-enter them every time you use the control.

Once you establish a successful connection with the POP server, you need to wait for a response. The lowest level of communication is the machine-to-machine link; the middle level transport sends the highest level data. In this case, it sends a confirmation reply beginning with "+OK" to the client. The WaitFor function in the POP Manager handles this. It's essentially a large DoEvents loop that repeatedly tests for one of several conditions, and allows the rest of the program to continue operation so incoming data can fire events. Calling the function is simple:

```
WaitFor STATUS, OutText
```

STATUS is one of two constants I use to track server activity. The other constant is DOT. It's important that you deal with these constants appropriately because the server can be in different states. For example, when you send a USER command, you test for a status reply beginning with either +OK or -ERR. When you receive a message body, however, you test for a string *ending* with a CR/LF.CR/LF sequence. In this case, you invoke the WaitFor loop procedure with the DOT parameter. This tells it to terminate when the client receives the end-of-message CR/LF.CR/LF sequence.

WRAP WINSOCK'S SENDDATA METHOD

After you receive the initial +OK from the server, you use the USER command to identify yourself to the server. The Send procedure wraps the Winsock control's SendData method. It does two things. First, it calls the ClearTests procedure, which clears the results of any previous WaitFor tests. Next, it sends the command to the

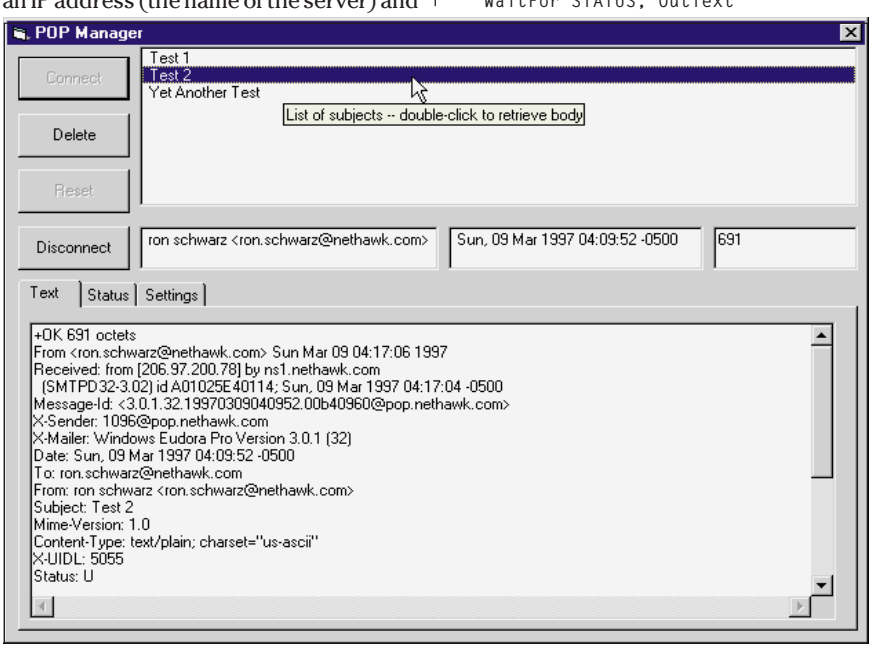


FIGURE 2 ***List, View, and Delete Messages.*** *The POP Manager Control lists all messages on your mail server, and lets you view and delete them directly from the server.*

VB5

```
'Get size and headers
'for each message
  Dim c As Long    'Message counter
  For c = 1 To Messages
    ReDim Preserve MailItem(c)
    'Get info for a message
    Send "TOP " & c & " 1"
    WaitFor DOT, OutText
    'Parse out headers
    Start = InStr(UCase$_
      (WholeThing), "+OK ") + 4
    Finish = InStr(Start, _
      WholeThing, " ")
    MailItem(c).Bytes = _
      Val(Mid$(WholeThing, _
      Start, Finish - Start))
    Start = InStr(LCase$_
      (WholeThing), "from:") + 5
    Finish = InStr(Start, _
      WholeThing, vbCrLf)
    MailItem(c).From = _
      Mid$(WholeThing, Start, _
      Finish - Start)
    Start = InStr(LCase$_
      (WholeThing), _
      "subject:") + 8
    Finish = InStr(Start, _
      WholeThing, vbCrLf)
    MailItem(c).Subject = _
      Mid$(WholeThing, _
      Start, Finish - Start)
    Start = InStr(LCase$_
      (WholeThing), "date:") + 5
    Finish = InStr(Start, _
      WholeThing, vbCrLf)
    MailItem(c).DateSent = _
      Mid$(WholeThing, _
      Start, Finish - Start)
    lstHeaders.AddItem _
      MailItem(c).Subject
  Next
```

LISTING 1 ***Parse the Headers.*** *If the parsing code finds any messages, the control uses the TOP command to get their headers.*

server. It also appends a CR/LF to the string, which is important. Failing to append this sequence is functionally equivalent to typing in the command, but never pressing the Enter key.

```
Private Sub Send(TextOut As String)
   ClearTests
   tcpIn.SendData TextOut & vbCrLf
End Sub
```

You need to send the PASS command with your password after receiving a +OK, then wait for another +OK. At this point, you're logged in to the POP server, and you can execute any of the commands that deal with mail management. The POP Manager example control uses the STAT command to find out if there are any messages. Messages on the server are represented by a +OK, followed by a space, the number of messages, another space, and the total size of the messages. You can parse the reply in this manner:

```
'  Parse returned string
   Start = InStr(WholeThing, " ") + 1
   Finish = InStr(Start, WholeThing, _
      " ")
   Messages = Val(Mid$(WholeThing, _
      Start, Finish - Start))
   If Messages = 0 Then
   'bail out if none
      GetHeaders = False
      cmdDC_Click
'disconnect
      Exit Function
   End If
```

This is pretty straightforward string handling code, so I won't delve into it here.

The Messages variable contains the number of messages on the server, if any. If the parsing code finds any, the control uses the TOP command to get their headers (see Listing 1).

Listing 1 uses standard VB string handling functions to pick out the size (in bytes), and the From, Subject, and Date headers for each message on the server, which I put into an array of UDT records. The UDT also contains items that track whether or not each message has its body loaded into memory, and whether or not it's been deleted. Finally, it stuffs the contents of the Subject header into a list box, which you can use at run time to navigate through the messages and download bodies.

If you look at the Send and WaitFor procedures, you might think you see the beginnings of a simple scripting language. This is intentional, because the multilayered/fragmented-response nature of Internet client/server communications is complicated enough without one. When you have a chance to play with the code, you should be able to port sections of it to other projects, using it for everything from FTP to telnet applications.

SWITCHING GEARS

Let's backtrack a bit, and take a look at some of the nitty-gritty details involved in pulling data off the server. Sending data is simple: just package your string and send it. Unfortunately, receiving data is more convoluted. The Winsock control has three events that you need to pay attention to when receiving data: the DataArrival, Close, and Error events. If you think it's simply a matter of picking up returned data from the DataArrival event, then dealing with errors as they occur—well, you're almost right. I wish you were right, but it's not that simple unless you're dealing with tiny bits of data.

The first complication is the GetData method. The DataArrival method doesn't actually return any data; it only lets you know that some data has arrived. You need to invoke the GetData method to retrieve the data from a buffer.

The second complication is the fact

that data arrives over the net in bits and pieces, and you're at the mercy of numerous factors over which you have zero control. So, the minuet begins. You send a request, then wait for the appropriate handshake—either a status token (+OK/-ERR) or a "dot line." While you wait, you're also ready—in the DataArrival event—to receive pieces of data, lace them together, and continue waiting (back in the WaitFor loop). Finally, exit the WaitFor loop after you receive all the data, and perform whatever parsing and other operations you require on the received data.

The Error event fires when the control reports certain types of errors. The POP Manager example traps them, and puts them in the Status box. This way, you have some idea of what happened. Errors that the Error event traps do *not* trigger a VB Error; however, errors that occur during the Connect invocation do.

The Close event fires when the low-level connection to the server is terminated. You need to keep track of this because it's bad form to end the program, or attempt a reconnection, while still connected. Extremely nasty things can happen, and you may end up crashing VB if you don't keep track of when the connection is open and closed.

You can find a working example of the POP Manager ActiveX control on the free, Registered Level of The Development Exchange. Premier Level members of The Development Exchange can also find examples that implement the POP manager control, as well as an example of the POP Manager that implements its functionality in a VB application *without* creating an ActiveX control. For details, see the Code Online box at the end of this article. One of the main differences between writing code in traditional VB forms and ActiveX controls is the fact that the controls don't have Load and Unload events. However, ActiveX controls do have Initialize and Terminate events, and also allow the use of Sub Main, which should give you ample opportunity to port your code over. I developed the POP Manager in a simple VB form (with a BAS module added), and afterward ported it to a control by the brute force method: cut-and-paste. I encountered one snag with the TabStrip control—I could not get it to work as part of a control when downloaded over the Internet. However, it works fine when you use the control in a standalone project.

PACKAGING FOR THE 'NET

Installation issues become more involved as VB becomes more powerful with each successive release. Packaging controls for Internet deployment brings a whole new set of requirements. Not only must the control be available, but constituent controls and run times must be downloaded as well, along with any dependencies.

Fortunately, Microsoft has beefed up the Setup Wizard considerably, so it's not the Herculean task it would be otherwise. The Setup Wizard takes care of creating the CAB and other files required, as well as marking your control safe for download and scripting when you deem it appropriate.

When you run the Wizard, you end up with an HTML file containing the "nugget," or a declaration for your control:

```
<OBJECT ID="POPMgr" WIDTH=642 HEIGHT=473
CLASSID="CLSID:56CA7589-9822-11D0-8DFB-000000000000"
CODEBASE="POPMgr.CAB#version=1,0,0,0">
</OBJECT>
```

Placing this declaration in your HTML creates an instance of your control at the place in the text stream where it's positioned.

The native-compiled version of the POP Manager control is *only* 38K, which I find amazing. Considering what it does, this is remarkable. The user downloads the run times and associated files only once, and they never have to be downloaded again unless Microsoft upgrades them. This means that your VB-compiled controls should load quickly.

After you experiment with the example code, you should be able to do nearly any type of Internet communications you require. You may want to extend the POP Manager by exposing some of its methods, events, and properties so you can control it externally. In fact, you can create a full-blown mail program with a little effort. You can even set the POP Manager's Visible property to False if you choose to build it into a more extensive system. You need to learn a few Simple Mail Transport Protocol (SMTP) commands from RFC822 to send outgoing mail, but you will find that to be a piece of cake after you become comfortable with POP. ☒

Code Online

You can find all the code published in this issue of* VBPJ *on The Development Exchange (DevX) at http://www.windx.com. All the listings and associated files essential to the articles are available for free to Registered members of DevX, in one ZIP file. This ZIP file is also posted in the Magazine Library of the* VBPJ *Forum on CompuServe. DevX Premier Club members ($20 for six months) can get each article's listings in a separate file, as well as additional code and utilities for selected articles, plus archives of all code ever published in* VBPJ *and* Microsoft Interactive Developer *magazines.

Write a Control to Manage a POP3 Server

Locator+ Codes

Listings ZIP file plus source and examples of Fetcher and the POP Manager control: VBPJ0597

✪ Listings for this article plus the files described above; the POP Manager EXE; the ActiveX deployment files for the POP Manager control, and the POP3 RFC file in HTML (subscriber Premier Level): RSZ0597P