DEV202

Getting Started with Office Automation

Tamar E. Granor, Ph.D.

Automation lets you drive one application from another. The Microsoft Office applications can be run from within Visual FoxPro to provide tools for reporting, presentations, graphing and more.

This session introduces the tools needed to get started with automation and to learn about automating the Office applications. It looks at ways to explore the Office object models, including the VBA Help files, the Macro Recorder and the Object Browser. It also examines the differences between VBA code and the Visual FoxPro code you write to automate the Office apps. Finally, it explores some of the Visual FoxPro code and tools that aid in writing automation code.

This session assumes familiarity with object-oriented programming. Please note that it will explore only very basic automation code. This session provides the foundation on which other sessions can build.

What is Automation?

Automation is the latest and greatest in the never-ending quest for applications that do one thing well and communicate with other applications that do something else well. It's the successor to a variety of technologies, including sharing data through common file formats, sharing data through the Windows Clipboard, and DDE.

Automation is one of the facets of OLE (object linking and embedding) and was introduced in OLE 2.0. With Automation, commands can be issued in one application and sent to another. They're written in standard object code, using the appropriate syntax for the host language (the one issuing them, which in our case is Visual FoxPro). Think of Automation as one application grabbing a megaphone and telling another application what to do. The number of applications that work with Automation, either as the application holding the megaphone or as the application listening on the other end, is increasing all the time. Automation is one of the technologies grouped under the COM umbrella.

Two applications are involved in any Automation session. The application that's in charge (issuing the orders) is called the *automation client* (or just *client*). The application being manipulated is the *automation server* (or just *server*). The client addresses an instance of the server as if it were any other object, reading and setting properties and calling methods. This simple technique means that a Visual FoxPro application (the client) can address anything from the Office applications to Windows' file system to Lotus Notes to all kinds of other things. In fact, VFP itself can be used as an automation server.

To get started, the client creates an instance of the server application. Internally, when a client attempts to instantiate a server, Windows goes off to the Registry and says "Help! Somebody wants to create such-and-such a server." The registry looks up the server by name, finds out what program it is and where that program is stored. Windows executes the program in question (assuming it finds such an entry in the registry and finds the specified program where it's

supposed to be). It then returns a reference to the newly executing program to the client, which hangs on to it, so it can find it again later.

Obviously, there are lots of places along the way where something can go wrong. The server name might not be found in the Registry, the program might not be found where it's supposed to be, there may not be sufficient system resources to start the program, and so on and so forth. If anything goes wrong, an error is raised and the server fails to start.

However, if all goes well, the client has a reference to the server and can start ordering the server around. It's like being in a restaurant once the waiter has introduced himself. You know his name, and you know what he looks like so you can call him over when you need him, and you can start telling him what to do. Except for one thing. You don't know what's on the menu. You can make some educated guesses based on what the place looks like. For an application, that corresponds to guessing based on what application it is and what you know it does. But to use the restaurant efficiently, you need the menu. To work with the application server efficiently, you need to know what it can do. The bulk of these notes looks at tools that help you find out what the server can do.

Introducing the Office Servers

The Office applications – Word, Excel, PowerPoint and Outlook - share a programming language, Visual Basic for Applications (VBA). For automation developers, having a shared programming language isn't as impressive as it sounds because, while each Office application uses the same syntax and commands, that's only helpful if you're actually writing in that programming language. For automation, you're not. You're writing in the language of the client application, the one controlling the automation process.

On the other hand, for a variety of reasons, it's very handy to be able to read VBA when you're writing automation code. Both the documentation for the servers and the macros you can record for most of them use VBA. (See below for more on each of these.) There are a couple of peculiarities about VBA from the FoxPro developer's perspective that make it hard to read for those unfamiliar with it. We'll look at each of these issues as we dig into the Office servers.

To use the Office servers, the key issue is finding out what methods and properties they have, and determining the parameters to pass to their methods. There are three main approaches. In most cases, you'll need to combine all three to get what you need. Beyond those, there are a couple of other ways to learn about automation and test out what you're learned.

Use the Help File

Each of the Office servers has a Help file that documents its members. The method of getting to that Help file varies with the application. In the big three (Word, Excel and PowerPoint), you can access it from the main Help menu for the application. On the Contents page, go down near the bottom. In Word and Excel 2000, look for Programming Information just above the bottom of the Contents list. In PowerPoint 2000, the entry to find is "Microsoft PowerPoint Visual Basic Reference," also near the bottom of the Contents list. In all three cases, you can open the specified item to see a list of automation topics. Outlook's Automation Help is well hidden. You have to open the Advanced Customization topic to find "Microsoft Outlook Visual Basic Reference."

Getting to the Help files is a little different in Office 97. You start out the same way in Word, Excel and PowerPoint – by choosing the Contents page from Help and scrolling down to the bottom. Then, in all three, look for "Microsoft X Visual Basic Reference," where X is the product you're using. When you choose that item, it opens to reveal "Visual Basic Reference". Choosing that item opens a new Help window containing just the Automation Help for that product. Getting to Visual Basic Help for Outlook 97 is extremely complex. The steps are spelled out in the main Help file – search for "Visual Basic". Understand, though, that VBA is not the primary programming language for Outlook 97 and that automating it with VBA is trickier is some ways than automating the other Office 97 products.

If you can't find the appropriate item in Help contents, it means you didn't install the VBA Help file with the application. To do so, you have to choose Custom installation. The standard installation doesn't include these files. (In Office 97, the standard installation was called "complete," but wasn't.) You can install the VBA Help files at any time by running Setup again. In Office 2000, you do so by choosing Visual Basic Help from Office Tools. (By default, the VBA Help files are set to install on first use. If you know you're planning to do automation work, it's easier to install them on your hard drive in the first place. The whole set of four (Word, Excel, PowerPoint and Outlook) for Office 2000 is under 5 MB.

By this point, you may have noticed that getting to the VBA portion of the Help can be something of a pain. In Office 2000, VBA Help is integrated into the regular Help and doesn't open a separate window, so working within it is difficult, too. In Office 97, once you get to VBA Help, it's way too easy to close it – just hit ESC and it's gone.

Fortunately, there's an easy solution. Although accessible from within the main Help file for its respective application, the VBA Help for each Office application actually lives in a separate file. In Office 97, the file name is in the form VBAapp8.HLP, where "app" is some form of the application name. For Office 2000, it's VBAapp9.CHM, because the Office 2000 applications use HTML Help; that's why you can't close Help with ESC anymore. Given all these difficulties of getting to VBA Help, it's best to create shortcuts on your desktop for each of the Help files you're likely to use. Then, they're only a double-click away.

Each of the VBA Help files includes a diagram of the object model for that product. The model is "live" – when you click on an item, you either move to the appropriate entry in Help. Click on a red triangle to move to another level in the diagram. Figure 1 shows the top level of the Word object model diagram.

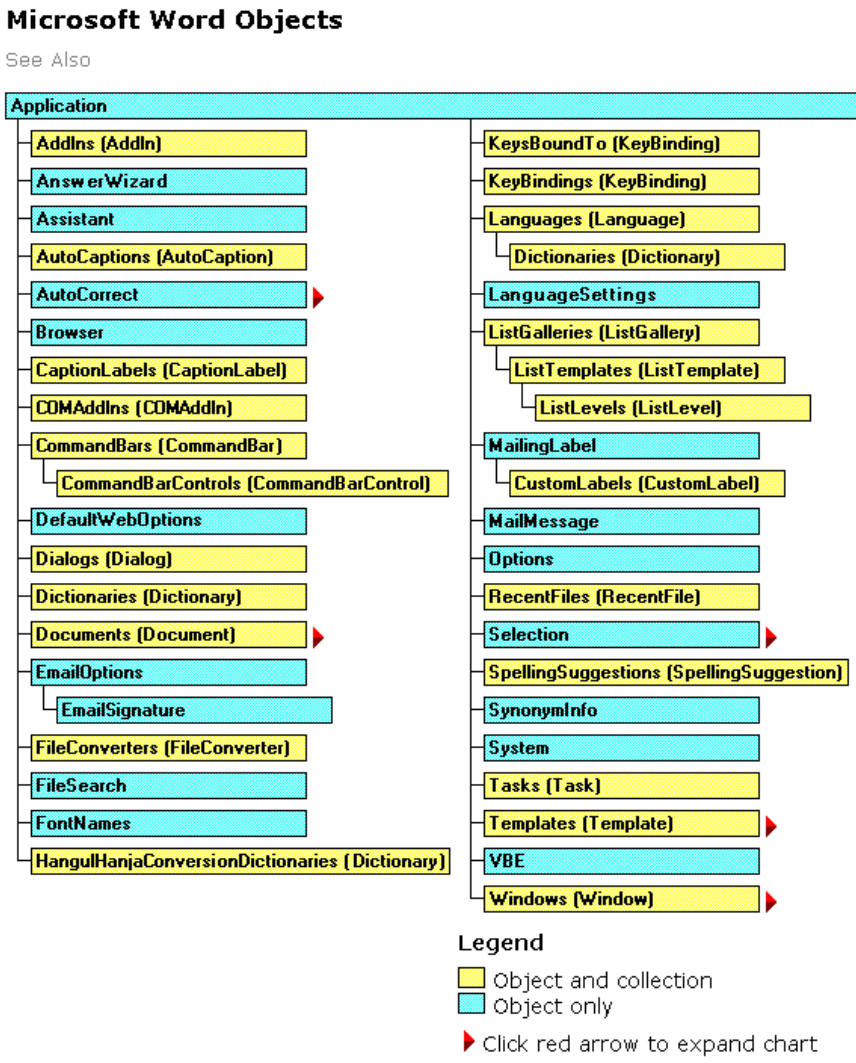

Figure 1. Object Model Diagram. The Help file for each Office product contains a diagram of the object model for that product. Clicking on one of the rectangles takes you to the Help entry for that object. Clicking an arrow takes you to another level in the object model diagram.

The Help entries for objects include small pieces of the object model diagram as well – clicking on these also jumps to the indicated entries. Figure 2 shows the entry for the Documents collection in the Word VBA Help file. The dotted box that says "Multiple Objects" has been clicked, bringing up a dialog box of objects contained in the Document object.

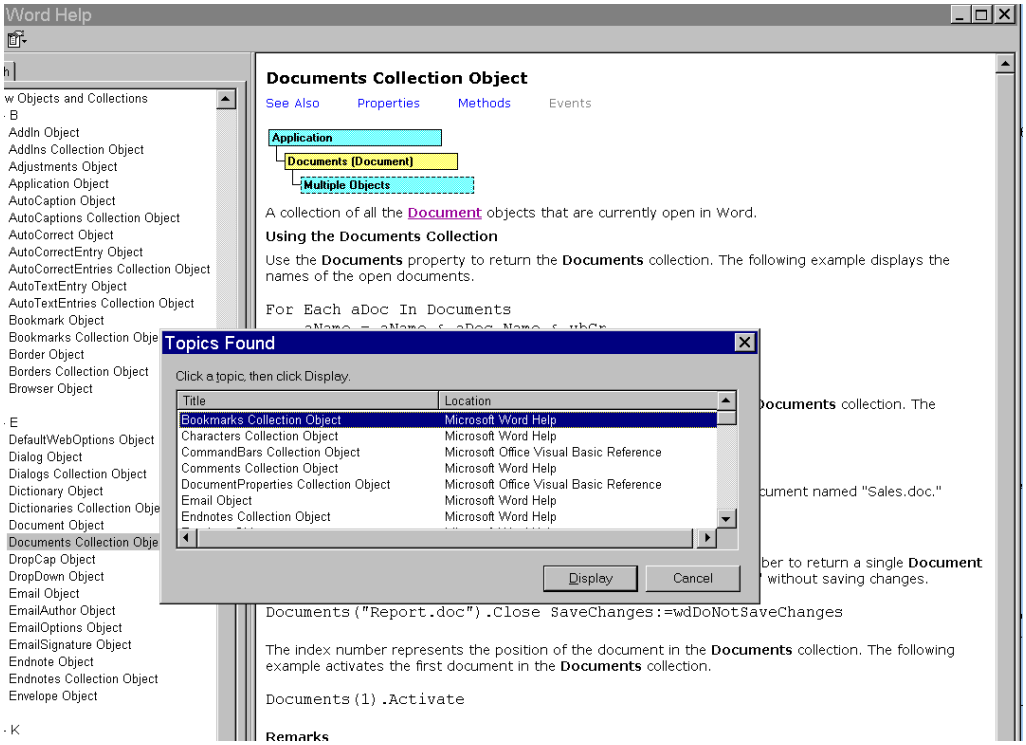

Figure 2. Live Help. Even within Help entries, the object model diagrams are live. In this case, clicking on the "Multiple Objects" rectangle calls up a dialog listing other objects contained in the Document object.

The VBA Help files are generally clear and correct. If you know what you're looking for, you can find it there. The real challenge, then, is to figure out what you're looking for.

Let the Server Write the Code

Word, Excel and PowerPoint have macro recorders that can turn user actions into VBA code. (While Outlook supports macros, unfortunately it doesn't include a macro recorder.)

So one way to figure out how to automate something is to record a macro to do it, then examine the macro.

From the menu, choose Tools|Macro|Record New Macro to start recording. Give the macro a name. Then interactively perform the operation you want to automate. When you're done, click on the Stop Recording button on the Macro toolbar (which appears automatically when you start recording).

To look at the recorded macro, choose Tools|Macro|Macros from the menu; that opens the Macros dialog shown in Figure 3. Highlight the macro you just created and choose Edit. This brings you into the Visual Basic Editor (VBE) – Figure 4 shows the highlighted macro from Figure 3 as it appears in the VBE. This particular macro was recorded in Word. (You can also get to the VBE by choose Tools|Macro|Visual Basic Editor or by pressing Alt-F11.)

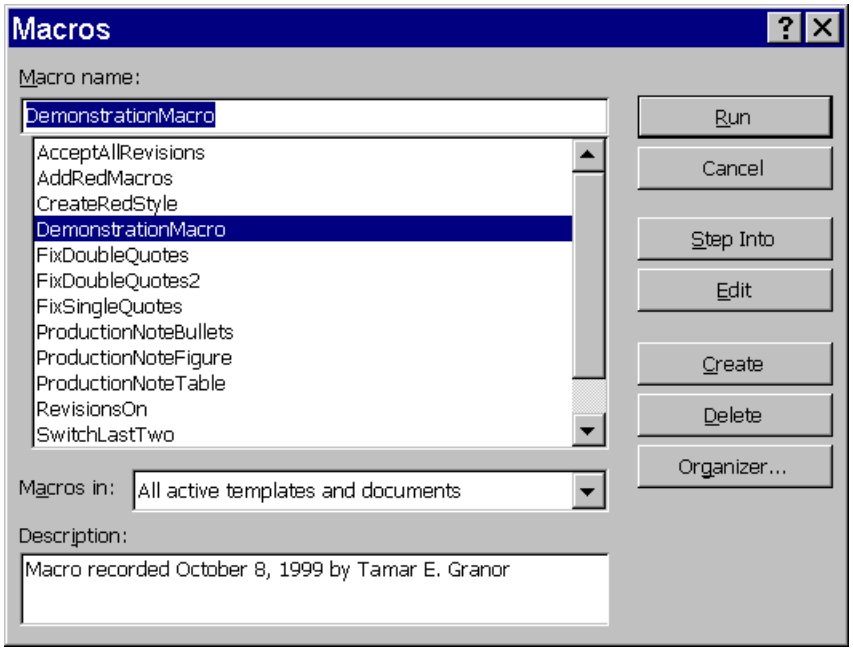

Figure 3. The Macro dialog. This dialog lists all the macros you've recorded or otherwise stored in an Office application. It's one entry point to the Visual Basic Editor.

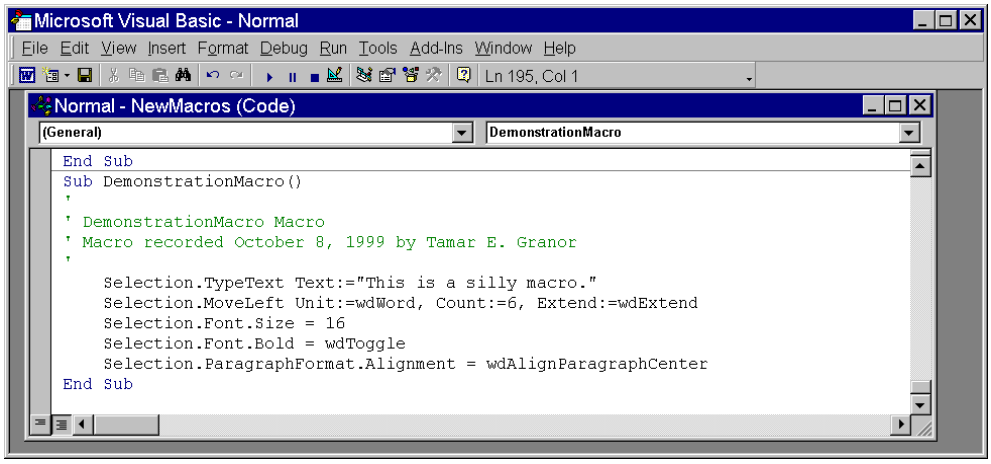

Figure 4. Viewing macros. Editing a macro takes you to the Visual Basic Editor. At first glance, the code may seem mysterious, but just a few tricks can decode it.

Converting a VBA macro to VFP code is harder than it should be for several reasons. The macro in Figure 4 demonstrates all of them. Consider this line of Word VBA code as you read the following sections. This line moves the insertion point (the vertical bar that determines where the next character is inserted) six words to the left, highlighting the words in the process:

```
Selection.MoveLeft Unit:=wdWord, Count:=6, Extend:=wdExtendDefault
```

Objects

First, unlike VFP, VBA makes some assumptions about what object you're talking to. In the line above, Selection translates to VFP as if it were This.Selection, which represents the current selection (highlighted area) of the Word instance.

In each Office application, certain objects are considered default objects in the VBA environment. Your code won't be treated so kindly – you need to be explicit about what object you're addressing. The code you send to the automation server must be addressed to the right object.

Named Parameters

VBA allows methods to use what are called *named parameters*. In the example code line, the method called is MoveLeft. Three parameters are passed, each in the form:

```
parameter name := parameter value
```

This syntax allows VBA programmers to include only the parameters they need and not worry about the order of the parameters. Since some VBA methods have a dozen or more parameters, this is a very handy option.

However, VFP doesn't support named parameters; you must specify parameters in the right order. Fortunately, the Help files show the parameters in their required order. (That wasn't true in versions of Office before Office 97; Help for many methods showed the parameters out of order and finding the correct order was extremely difficult.)

When translating from VBA to VFP, add parentheses around the list of parameters and delete the parameter names and ":=" symbols. Check Help (or the Object Browser, discussed below) to determine the correct order and number of parameters. Usually, the macro recorder puts the parameters in the correct order; however, some parameters may be omitted, as named parameters allow VBA developers to leave out any parameters that are to take the default value. Be sure to check Help for omitted values. Also check to ensure that the macro recorder really did put them in the right order; occasionally it doesn't.

Defined Constants

The first parameter in the example line shows the third problem that occurs in translating VBA to VFP. It specifies that a parameter called Unit should have the value wdWord. But what is wdWord? It's one of thousands of defined constants available in Word's version of VBA. (It turns out that wdWord is 2.)

The VBA Help files don't supply the values of defined constants. In fact, Help uses them exclusively and doesn't show their actual values anywhere (ditto for the macro recorder). (Outlook is the exception here. It has a Help topic entitled "Microsoft Outlook Constants" with a complete list.) To find out what wdWord and all the others stand for, use the Object Browser available through the Visual Basic Editor. (See the next section.)

If there was ever a reason to use header files, VBA constants is it. However, to build the header file, you need to find the values of the constants. Rick Strahl of West Wind Technologies has created a freeware tool that reads a COM type library, extracts the constants and creates a Visual FoxPro header file. The tool, called GetConstants, is included in the conference materials; the latest version can always be downloaded from Rick's website (www.west-wind.com). VFP 7 includes its own Object Browser, which also provides a way to create a Visual FoxPro header file. The VFP 7 Object Browser is discussed later in these notes.

I don't recommend using the complete header files created by GetConstants or the VFP 7 Object Browser as is in your automation work. The number of constants contained in them is mind-boggling. The smallest set is for Outlook – it contains 251 constant definitions. Word's file is

nearly ten times that size with over 2400 constants defined. Saving a form with no controls, but pointing to the full Word constant definition file as its Include file, took more than five seconds. (By contrast, on the same machine, saving a totally empty form was well under a second.)

However, having the complete set of constants at your disposal is very handy. You can cut and paste from them to create header files appropriate to the tasks you're doing. Rick's tool also makes it easy to keep your header files up to date as Microsoft adds new constants.

Using the Object Browser

One of the most powerful tools available for figuring out automation code is the Object Browser (Figure 5). It lets you drill into the various objects in the hierarchy to determine their properties and methods, see the parameters for methods, determine the value of constants, and more.

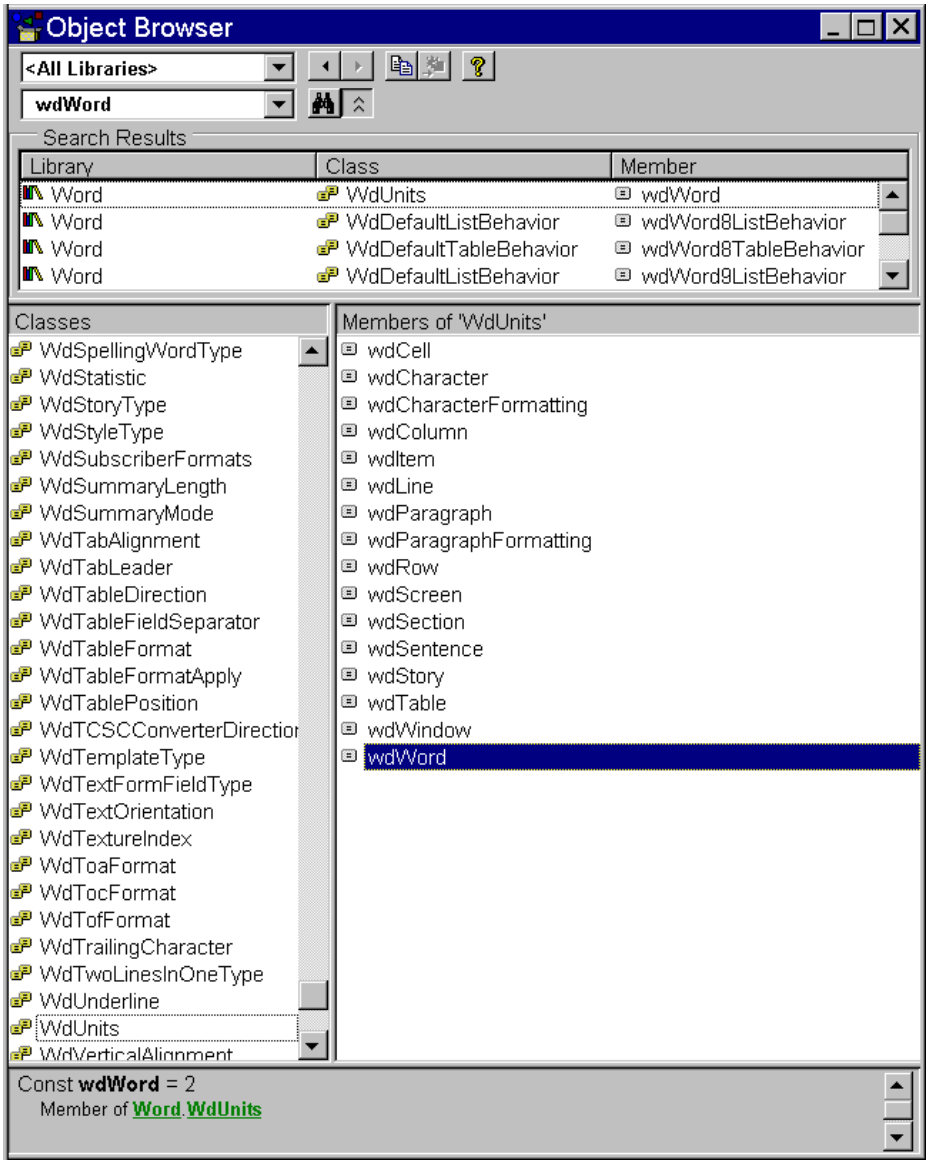

Figure 5. Object Browser. This powerful tool lets you drill down into objects, find out constant values and determine parameters. Here it shows that wdWord is a constant, is a member of a group of constants called WdUnits, and has a value of 2.

The easiest way to find out about a specific item is to type it into the search dropdown and press Enter or click the Find (binoculars) button. The middle pane fills with potential matches. Choose one to learn more about it in the main section of the Browser underneath. The left pane fills with the properties, methods, collections, and constants. The right pane describes what's available for the highlighted item in the left pane. In Figure 5, the Object Browser has been used to determine the value of the constant wdWord. In the bottom-most pane, you can see that it's a constant with a value of 2.

The Object Browser is also useful for moving around the object hierarchy to get a feel for what the various objects can do. Figure 6 shows the Object Browser with Excel's objects rather than Word's. The members of Excel's WorkBook object are shown in the right pane. The PrintOut method is highlighted, so the very bottom panel shows its (complex) calling syntax. The advantage of this approach over Help is that the Object Browser actually looks at the type library, so the list it shows is more likely to be correct than Help. Even better, the Object Browser and Help can work together. Press F1 in the Browser and Help opens to the highlighted item.

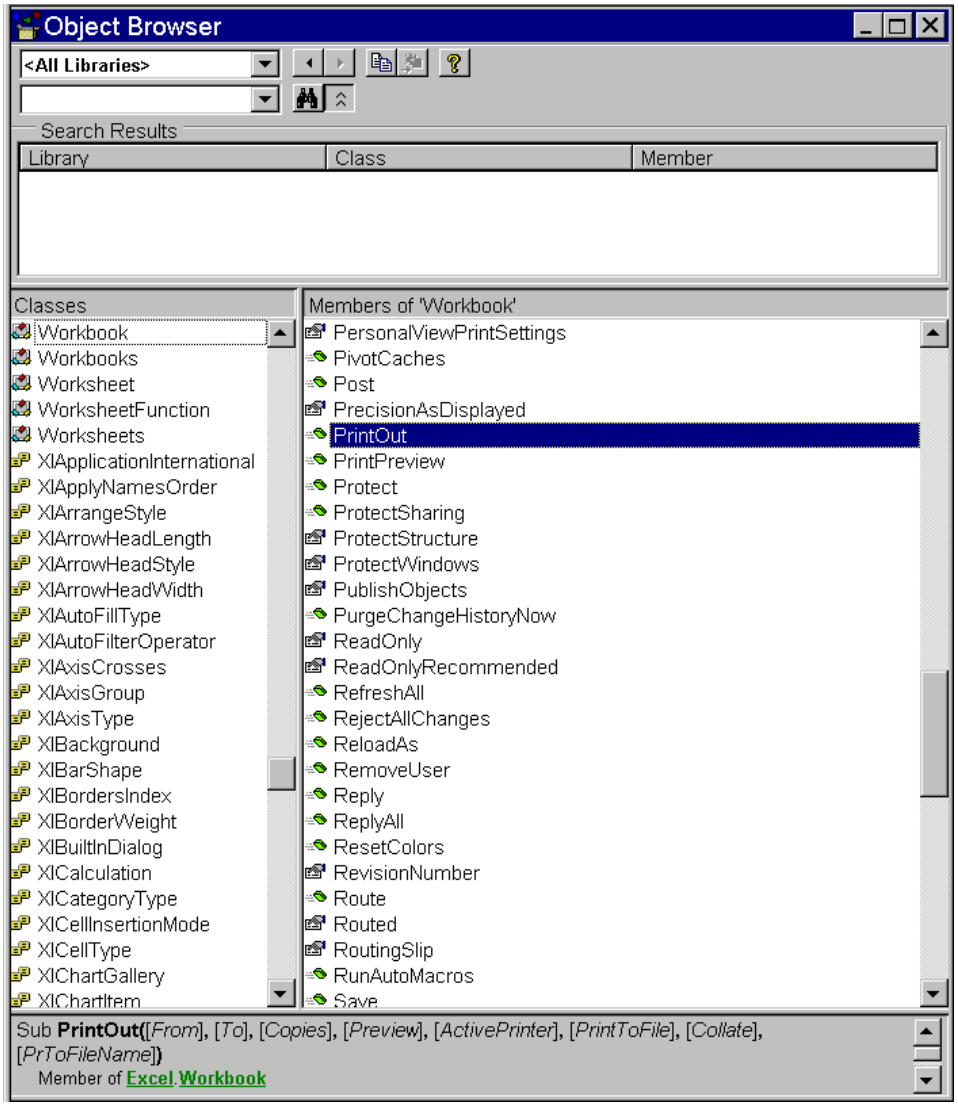

Figure 6. Using the Object Browser to determine parameters. When a method is highlighted, the bottom pane shows the calling syntax. Since the Browser gets its information right from the server's type library, it's unlikely to be wrong.

The Browser is also useful for exploring the object hierarchy itself. Figure 7 shows the PowerPoint version of the Object Browser. The Presentation object's members are shown in the right pane. The Slides property is highlighted. In the bottom pane, we learn that Slides is a reference to a Slides collection. Clicking on the underlined Slides takes us to the Slides collection, shown in Figure 8.

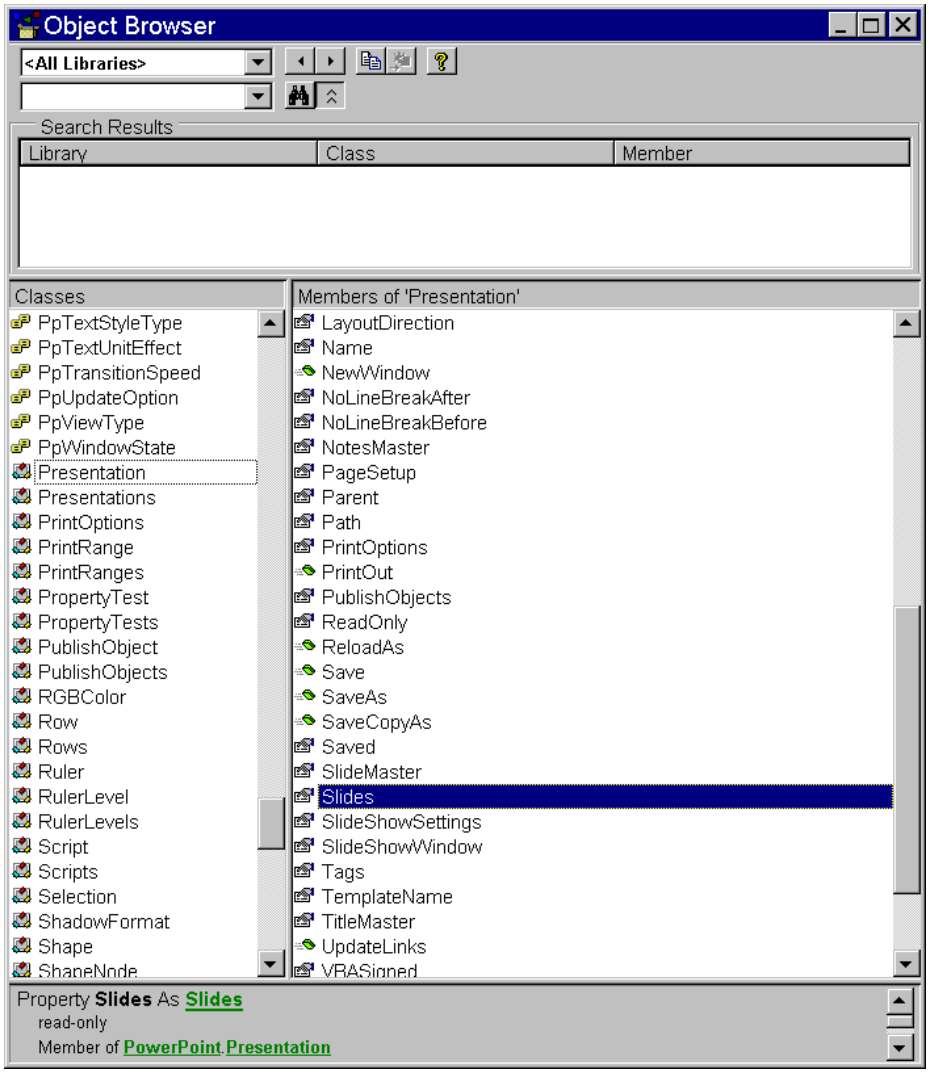

Figure 7. Exploring the Object Model. When an item is underlined in the bottom pane, you can click on it and change your focus in the Browser. Click on Slides, underlined here, to change to the display in Figure 8.

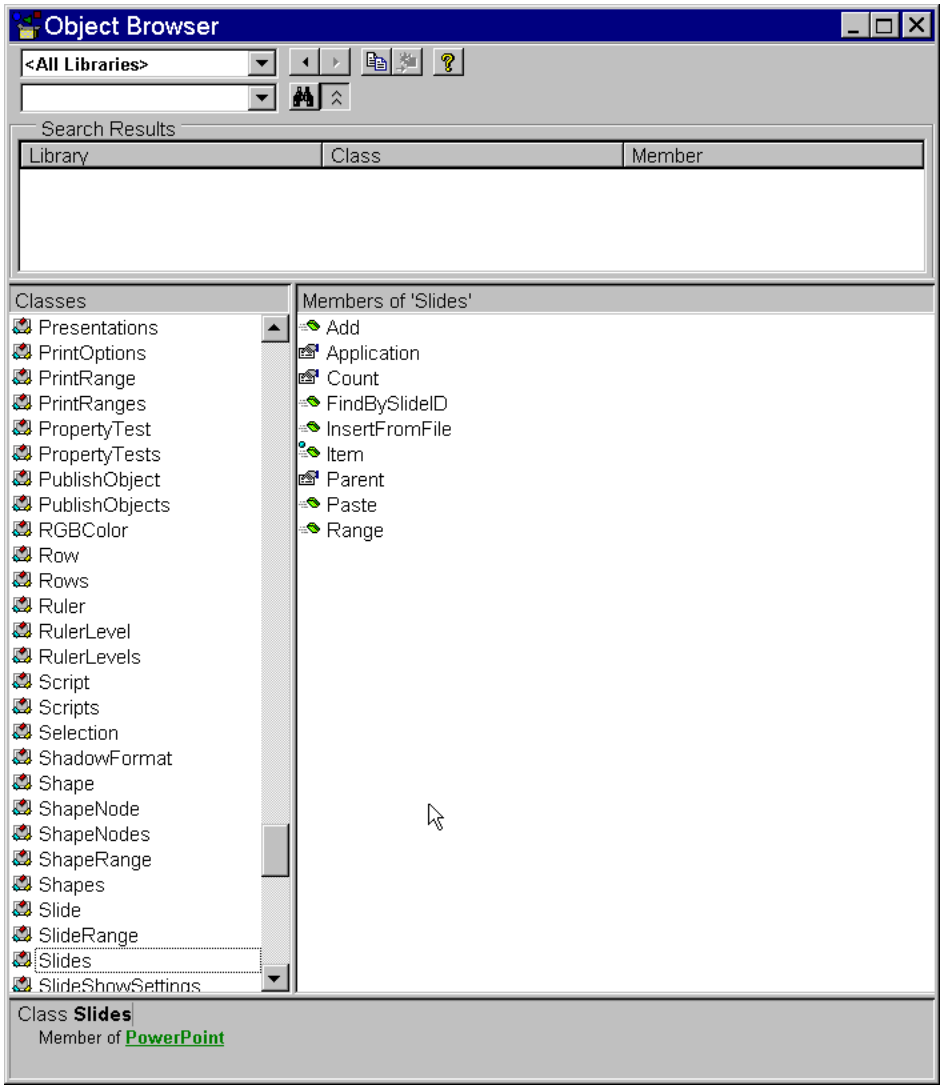

Figure 8. The Slides Collection. A click on the reference to the Slides collection in Figure 7 produces this display in the Object Browser. The Browser makes it easy to explore the relationships among objects in the hierarchy.

VFP 7's Object Browser

Visual FoxPro 7 introduces its own Object Browser. Like the one in the Visual Basic Editor, it allows you to explore Automation servers.

The VFP Object Browser has one trick up its sleeve than the VBE Browser doesn't. If you drag the constants for a server from the Browser into an editing window, VFP-style constant definitions are generated. Figure 9 shows the VFP Object Browser containing the Outlook server. A program window overlaps it; by dragging the highlighted word "Constants" and dropping it into the editing window, FoxPro definitions for Outlook's constants have been generated.

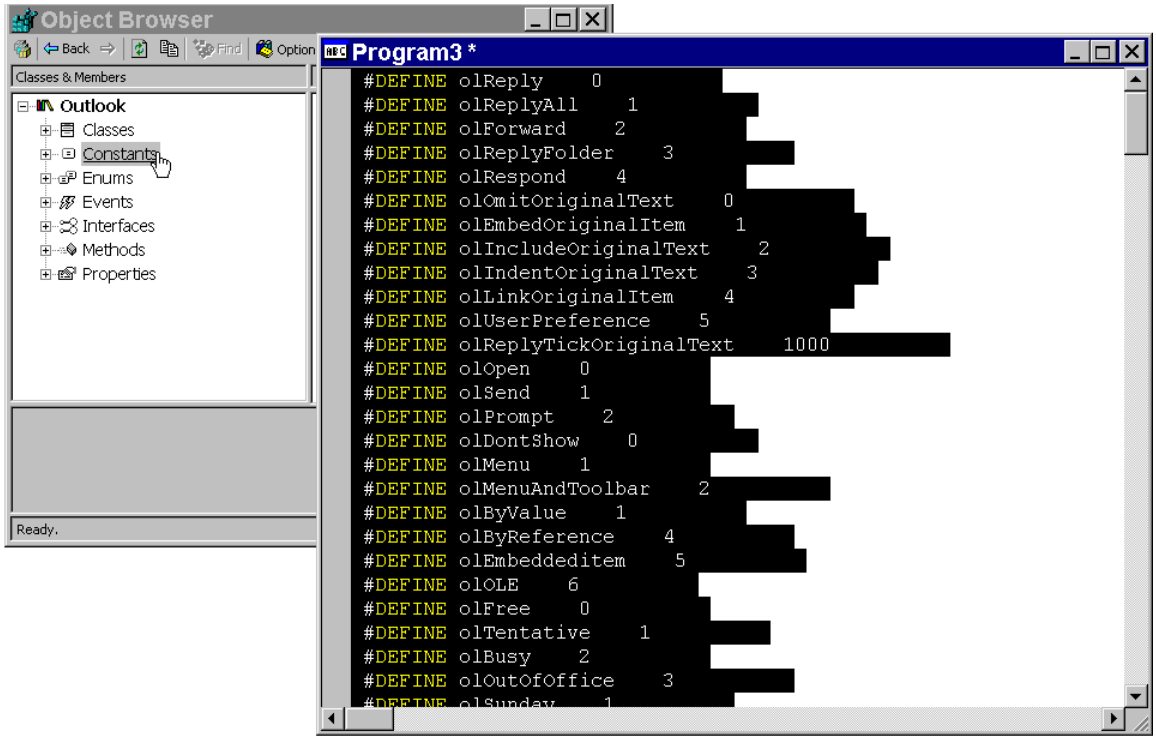

Figure 9. Creating Constant Definitions – VFP 7's new Object Browser makes it easy to generate constant definitions from an automation server. Just drag the constants into an editing window.

At Your Command

The Visual FoxPro Command Window is another powerful tool for learning about automation servers. Once you've read what Help has to say and looked it up in the Object Browser, sometimes you just need to try it. That's where the Command Window comes in.

Just as it does in every other aspect of working in VFP, the Command Window lets you try things and see what happens without the overhead of building entire applications or setting up complex scenarios. Create a reference to the appropriate server and try the sequence of commands one by one, observing the results as you go. You can query the value of a property with ? (assuming the value is printable) or execute a method.

Two changes in Visual FoxPro make working with a server interactively much easier in VFP 7 than in earlier versions. First, the addition of Intellisense means that all the properties and methods of the server are displayed as you work. For example, suppose you create an instance of Word in the Command Window, like this:

```
oWord = CREATEOBJECT("Word.Application")
```

When you then type oWord and follow it with a period, the list of properties and methods for the Word server pops up. You can choose the property or method you want to work with. Figure 10 shows Intellisense at work in the Command Window. As with VFP's native Intellisense, tooltips appear to guide you as you construct a command. Figure 11 shows the kind of help that appears once you choose a method.

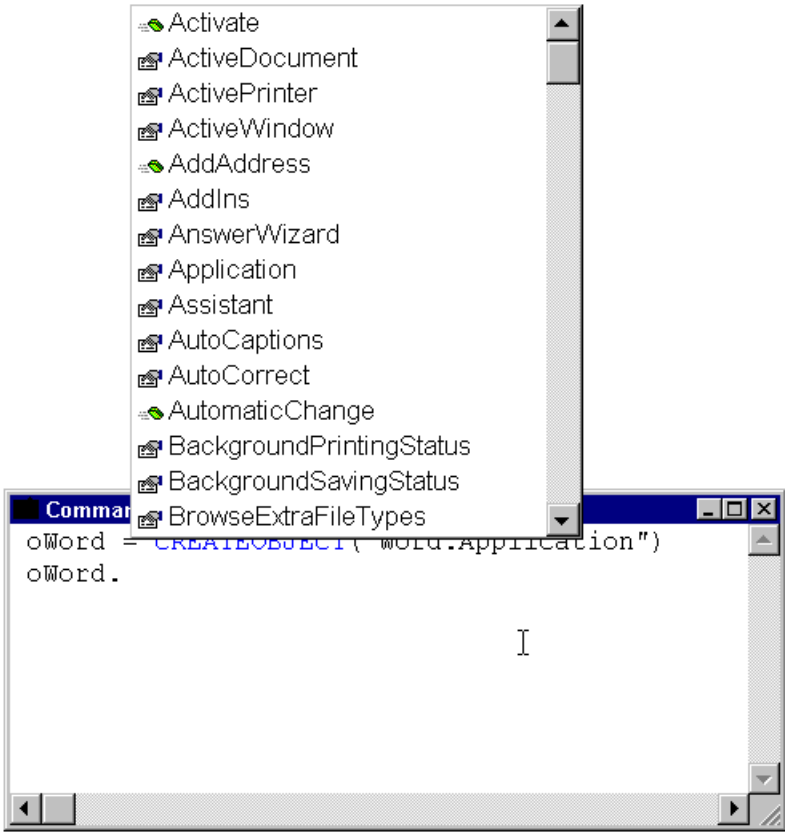

Figure 10. What can I do? – With Intellisense, you don't have to remember the names of properties and methods. They pop up as you write code.

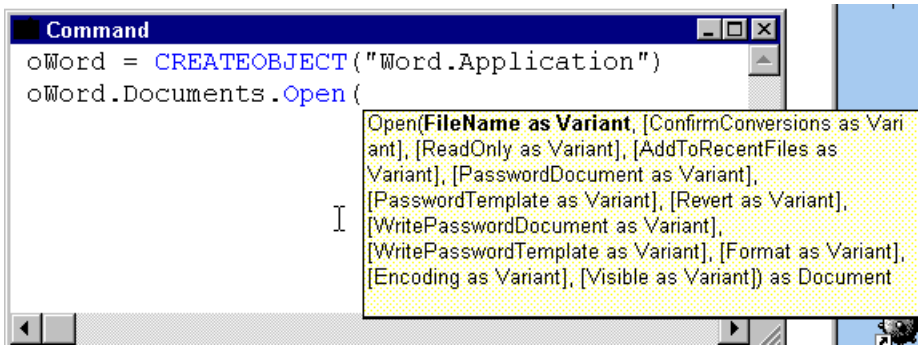

Figure 11. How do I do it? – Intellisense also provides tips for methods showing their parameters.

The second change in VFP 7 involves the use of the Debugger. Prior to this version, the VFP Debugger could be used only in a limited way with COM objects. The limit was that properties of COM objects were visible in the Debugger only after they had been accessed from VFP. You couldn't just drill down into COM objects in the Debugger the way you could into VFP's own objects.

In VFP 7, this is no longer true. COM objects, including Automation servers, are available in the Debugger, just like any other objects. So you can use the Debugger to explore the values of a

server's properties both when you're working interactively in the Command Window and when you're debugging an application.

You'll find it very useful to keep the Command Window visible as you write code. It can be very hard to get some commands right, particularly those with lots of specified named parameters (and lots more omitted parameters). Try them in the Command Window, and when they're finally correct, cut and paste the correct version into your code. One big drawback: the Command Window doesn't do #DEFINEs. Either set a variable in the Command Window, or use the corresponding values, then remember to change them to #DEFINEd constants when you paste the command into your code.

One really cool thing is that you can move back and forth between doing things interactively and doing them with automation. That is, when you have an instance of a server available from the Command Window and visible, you can switch over to the server application and work with it interactively, then come back to VFP and check the values of properties that were just set by your action or execute a method or set some other properties.

While trying to understand how a particular feature works, I often try something from the Command Window, then switch over to the server application to see the result, then hit Ctrl-Z (Undo) in the server to reverse that action before going back to VFP to try a different parameter, value or approach. Perhaps more than any other, this ability drives home the reality that Automation really is just one more way to do the same things a user can do interactively.

Using Visual FoxPro as an Automation Client

No matter what you're automating, much of what you do in Visual FoxPro is the same. This section looks at the things that stay the same across all servers and across the Office servers.

Opening Servers

The first step with any server is getting a reference. There are two possibilities for an automation server: you can create a new instance or you can get a reference to an existing server, if one exists. There are also two VFP functions that you can use, CreateObject() and GetObject(), but they don't map exactly to the two techniques.

CreateObject() always creates a new instance of the server and returns a reference to it. The syntax for creating automation objects is:

```
oServer = CreateObject( cServerClass )
```

cServerClass is the appropriate name for the main class of the automation server. For the Office applications, it's "<appname>.application" where <appname> is "Word" or "Excel" or whatever. For example, to open PowerPoint, issue this command:

```
oPowerPoint = CreateObject( "PowerPoint.Application" )
```

Note that creating automation objects is almost identical to creating native objects. The only difference is the class of the object created. This is polymorphism (one of the pillars of OOP) at work.

If the server is already open and you'd like to use the existing instance, you can use the GetObject() function instead. GetObject() takes two parameters. For this use, omit the first parameter and pass the server class as the second. Here's the syntax:

```
oServer = GetObject( , cServerClass )
```

If the server's already open, GetObject() finds it and returns a reference to that instance. However, if the server's not open, this version of the function generates an error.

For example, to attach to an open instance of Excel, use this command:

```
oExcel = GetObject( , "Excel.Application")
```

There's another way to start a server – by passing GetObject() the name of a file and letting it figure out which server to use:

```
oDocument = GetObject( cFileName )
```

VFP looks up the file's extension in the Registry to figure out what application it belongs to, opens that application, then opens the file; it works pretty much as if you'd double-clicked the file in Explorer, except that you get an object reference to the file as a "document," so that you can manipulate it as an object. This example opens the file "C:\Documents\MyFile.Doc" in Word and returns a reference to it:

```
oDocument = GetObject( "C:\Documents\MyFile.Doc" )
```

In some situations, the filename may not be sufficient to determine what application to open or what to make of the file. In that case, you can also pass the name of the class within the server. For example, if the file has a .TXT extension, but you want Word to open it as a document, pass "Word.Document" as the second parameter like this:

```
oDocument = GetObject( "Example.TXT", "Word.Document" )
```

However, you can't just pass the file type automatically. Most of the time, either VFP or the server application chokes when it receives the second parameter unnecessarily.

Displaying the Office Servers

When you instantiate an automation server, by default it's invisible. It doesn't show up on the task bar. In Windows NT, it's shows on the Processes page of Task Manager, but not on the Applications page. It does show in Windows 95/98's Close Program dialog. Keeping itself somewhat hidden is generally a good thing. Often, you're doing something behind the scenes and there's no reason for a user to see it happening. However, while debugging and in some other circumstances, you may want to make the automation process visible.

Why not just make the server application visible all the time? Speed. Not surprisingly, manipulating documents is faster when you can't see them. It's also tidier–while watching a spreadsheet or presentation get built is pretty cool the first few times, after a while, users are likely to get tired of watching.

This is one area (actually, one of many) where Outlook is the odd man out. The commands in this section apply to Word, Excel and PowerPoint (and, for that matter, to Visual FoxPro when it's used as a server), but not to Outlook. To make the automation server visible, set its Visible property to .T.; set Visible to .F. to turn it off. (Actually, in PowerPoint, you can set Visible to .T., but setting it to .F. generates an error message. Word and Excel give you complete control over their visibility.) Outlook doesn't have an analogous property.

The WindowState property of the Application object determines whether the application is minimized, maximized or "normal," meaning some user-determined size. Manipulating WindowState when the application is invisible generally makes it visible (at least to the extent of showing on the toolbar). Each of the applications has a set of constants for the three possible values of WindowState. For example, for Excel, you can make these definitions:

```
#DEFINE xlMaximized     -4137
```

```
#DEFINE xlMinimized     -4140
#DEFINE xlNormal        -4143
```

while Word needs the following:

```
#DEFINE wdWindowStateNormal     0
#DEFINE wdWindowStateMaximize   1
#DEFINE wdWindowStateMinimize   2
```

The Application object of all three applications has Left and Top properties that determine where it's located on the screen, as well as Height and Width. You can manipulate these unless the application is maximized. The following code opens Word, sets the application window to normal, positions it a little below the upper left corner of the screen, then makes it visible.

```
#DEFINE wdWindowStateNormal 0

LOCAL nScreenHeight, nScreenWidth
LOCAL nWindowHeight, nWindowWidth

* Compute sizes
nScreenHeight = SYSMETRIC(2)
nScreenWidth = SYSMETRIC(1)

* Make it two-thirds the size of the screen in each dimension.

nWindowHeight = INT(.67 * nScreenHeight)
nWindowWidth = INT(.67 * nScreenWidth)

oWord = CreateObject( "Word.Application" )

WITH oWord
  .WindowState = wdWindowStateNormal

  .Height = .PixelsToPoints( nWindowHeight, .T. )
  .Width = .PixelsToPoints( nWindowWidth, .F. )

  .Top = 10
  .Left = 10

  .Visible = .T.
ENDWITH

RETURN
```

As the example indicates, Height, Width, Top and Left are measured in points. Word provides a number of methods for converting other measurements into points (including, in Word 2000, the PixelsToPoints method used in the example). The other applications offer fewer such methods; neither Excel nor PowerPoint includes PixelsToPoints, in fact. The conversion factor between pixels and points is about .75.

Unless you're doing very precise work, you can probably live with that conversion factor, so in Excel and PowerPoint, just define your own conversion, like this:

```
#DEFINE autoPixelsToPoints .75
```

You can handle converting between inches and points the same way. There are 72 points to the inch. Define your own constant for that task, like this (the auto prefix is for "automation"):

```
#DEFINE autoInToPts 72
```

Even when you're automating Word, there's some argument that you're better off doing your own conversions in VFP than using Word's built-in methods. Each call to Word is expensive. My tests found that arithmetic in VFP was about 100 times faster than calling Word's conversion methods.

Are We There Yet?

The big three applications start with no document open and no document window when you call CreateObject(), but they behave differently when GetObject() is used to open a specific file.

In Word, once GetObject() is through, there's a document window containing the specified document. Just set Visible .T. for the Application object and the document window shows:

```
oDocument.Application.Visible = .T.  && this is needed for any of the apps
                                     && but change the variable appropriately
```

PowerPoint and Excel are different. Setting Visible .T. isn't enough. In PowerPoint, you need to call the NewWindow method to provide a document window for the specified presentation:

```
oPresentation.NewWindow()
```

Excel uses yet another approach to the same problem. There is a document window; you just can't see it. Call the Activate method for the first window in the Windows collection:

```
oWorkBook.Windows[1].Activate()
```

Working with Servers

A few Visual FoxPro language features and interface components make it easier to write and maintain Automation code. Here's a look at things to remember as you work in the world of Automation.

SET OLEOBJECT

This VFP command determines whether VFP searches the registry when using CreateObject() or GetObject(). When it is set to ON, VFP searches the registry; when it's set to OFF, that search is skipped. Why does this command exist? The last place VFP looks to find an object is in the registry. Before searching the registry, it loads OLE support, which takes up memory. If your application doesn't require OLE support, setting OLEOBJECT OFF provides a bit of a performance enhancement. However, if you're using Automation, you need OLE support, as well as the ability to find the server in the registry. If you attempt to instantiate an Automation server while SET OLEOBJECT is set to OFF, an error like "Class definition WORD.APPLICATION is not found" is generated.

Use WITH… ENDWITH

Much of the code you write to automate any server involves setting properties or calling methods. It's common to have long stretches of code that consists of not much more than references to properties and methods with perhaps a little arithmetic or logic thrown in. You can make that code far more readable (thus easier to debug and maintain) by using VFP's WITH…ENDWITH command.

A series of commands that all begin with something like oWord.ActiveDocument.Tables[3].Rows[7] just isn't going to lend itself to readability. Instead, surround the group like this:

```
WITH oWord.ActiveDocument.Tables[ 3 ].Rows[ 7 ]
  * put the commands here with a dot in front of each property or method
ENDWITH
```

In fact, you can nest WITH commands. As you walk down the object hierarchy, doing a few things at each level, set up a WITH statement for each level, something like this:

```
#DEFINE wdAlignRowCenter    1
#DEFINE wdRowHeightExactly  2

WITH oWord.ActiveDocument
  * Do some things to the document as a whole, like
  .Save    && save using current file name

  * Then move on to the table.
  WITH .Tables[ 3 ]
    * Now issue commands aimed at the table as a whole
    .AllowPageBreaks = .T.  && allow table to break across pages

    * Then, when you're ready to talk to the single row.
    WITH .Rows[ 7 ]
      * Now issue the commands for the one row
      .Alignment = wdAlignRowCenter
      .HeightRule = wdRowHeightExactly
      .Height = .5
    ENDWITH
  ENDWITH
ENDWITH
```

The nesting makes it clear to the reader which WITH each property belongs to. VFP, of course, has no difficulty figuring it out. There's an added bonus besides readability. Code like this runs faster – FoxPro doesn't have to sort through multiple levels of hierarchy to find out what object a given property or method belongs to. In testing, a fairly simple example that queried about a dozen properties at four levels below the application object was about twice as fast using nested WITHs than addressing each property directly.

Also, note that there's no rule that says that the property or method has to be the first thing on the code line. It often works out that way, but it's perfectly fine to use them elsewhere. For example, the following code is acceptable.

```
WITH oExcel
  nHeight = .Height
  nWidth = .Width
ENDWITH
```

In fact, as the example in *Displaying the Office Servers* above shows, you can call methods, perform calculations, and generally do anything you normally would inside a WITH … ENDWITH pair, as long as you make sure to include the dot before the property or method name.

Use variables for object references

Another way to make your code more readable and speed it up is to assign complex object references to local variables. Even if a WITH…ENDWITH pair isn't called for, you may be better off assigning something like oPowerPoint.ActivePresentation to a VFP variable with a name like oPresentation. The shorter name is easier to type and easier to read. As with the WITH statement, it gives VFP a direct route to the object you're interested in rather than asking it to climb down the object hierarchy.

The same example as for WITH (querying properties at various levels) was tested - setting a local variable was as fast or even a little faster than using WITH. I suspect the exact trade-off point varies depending on factors like available memory, the number of references inside the WITH/to the local variable, and so forth. There's no question, however, that either approach is significantly faster than writing out a long reference to an automation object. The more deeply nested the reference and the more times you need it, the more time you save.

When you use local variables, you may need to clean up afterwards. In some situations, these references can prevent the server from closing when you call the Quit method.

Loop with FOR EACH

The Office object models include lots of collections (the OOP version of arrays). When you need to process all the members of a collection, VFP's FOR EACH loop is your best bet. FOR EACH lets you go through a collection (or array) without using a counter or worrying about how many members there are.

The syntax of FOR EACH is:

```
FOR EACH oMember IN oCollection
  * issue commands for oMember
ENDFOR
```

For example, to display the name of every open document in Word, you can use this code:

```
FOR EACH oDocument IN oWord.Documents
  ? oDocument.Name
ENDFOR
```

Note, by the way, that using FOR EACH implies the use of a local variable as described in the last section – the object reference used as the loop variable.

Resources

There are a number of references available for the Office servers besides their respective Help files. Microsoft Press offers a *Visual Basic Programmer's Guide* for both Office 2000 and Office 97. Each is available both in book form and online. Since Microsoft is in the habit of rearranging its website regularly, the best way to find the online versions is to search www.microsoft.com for "Visual Basic Programmer's Guide".

The VBA Help files are also available in printed form. If you'd rather work with a paper copy, you can order them from MS Press, as well. Look for the Office Language Reference (or the Language Reference for the individual application you're interested in). The Language Reference guides are available on the Microsoft website, too, in case you find yourself stuck somewhere without the Help file.

Microsoft's website for Office development is msdn.microsoft.com/officedev/ (or, at least it was as of this writing). Check it out for official support, technical articles, bug fixes, and so forth, as well as pointers to other useful sites.

Once you get comfortable enough reading VBA code, the various Office and Visual Basic magazines and journals can be useful resources. Take a look at *Microsoft Office & Visual Basic for Applications Developers* (www.officevba.com) and Woody's Office Watch (www.woodyswatch.com) for starters. The major FoxPro magazines, *FoxPro Advisor* and *FoxTalk*, cover automation occasionally-the Office servers are the automation target only for some of those articles.

When you need help immediately, the place to get it is on-line. The thing that distinguishes one place from another is the quality of the help you get. For the most part, all the help that's available comes from volunteers who enjoy answering questions as a way of honing their own skills or to thank the people who helped them along the way. Many of the people who answer questions in

the places listed here are Microsoft Most Valuable Professionals, recipients of an annual award given to those who spend inordinate amounts of time helping others on-line.

FoxPro Help Sites

There's been a Fox presence on CompuServe for nearly as long as there have been Fox products. In December, 1999, the two FoxPro forums were merged with several other forums devoted to Microsoft software to form MSDevApps. Four of its sections are devoted specifically to FoxPro and Visual FoxPro, while a number of other sections address more general development issues and topics related to FoxPro development (like HTML Help). CompuServe forums are a pleasure to use because of their rich threaded messaging model – each original message is followed by the replies to that message, as well as the replies to the replies and so on. In addition to being threaded, it's easy to find replies to messages you posted, something that's hard to do in many web-based messaging systems, even those that are threaded. In addition, CompuServe does a tremendous job of keeping out spam – you won't find abusive messages or ads for get rich quick schemes or porno sites. As of April, 2000, CompuServe forums are open to the general public, so you can participate in them through go.compuserve.com/msdevapps?loc=us&access=public, even if you're not a CompuServe member.

Microsoft sponsors newsgroups for FoxPro and Visual FoxPro on its public server, msnews.microsoft.com. Look for groups with names beginning microsoft.public.fox. Although Microsoft tries to keep spam out, there's more of it here than in any of the other FoxPro sites.

One of the busiest places to talk FoxPro on the Internet is the Universal Thread at www.universalthread.com. Like CompuServe, it offers threaded messaging with a way to find replies to your own postings. It's privately owned, and offers free access to all messages with a limited user interface. A reasonable monthly fee gets you a Premier Membership that gives access to some more advanced tools for using it.

Unique in the FoxPro community is the FoxForum WIKI, fox.wikis.com. Rather than a discussion group or forum, it's an evolving knowledge base where anyone can post information on a topic and others can add to it or edit it. You can find entries there on everything from object-oriented frameworks to naming conventions and just about anything else to do with FoxPro and the Fox community you can think of. If it's not there, you can add it yourself.

Advisor Media sponsors forums for a variety of products, including FoxPro, on their website, www.advisor.com. Choose Advisor forums from the home page.

A fairly new addition is a discussion group for Visual FoxPro sponsored by Fawcette Technical Publications. Devx.com is well known in the developer community for their other discussion groups, so I expect this one to quickly become active. You'll find it at news.devx.com, though upi can also access it as a web forum.

In addition to all this, a number of companies and individuals working with Visual FoxPro provide useful information and downloads on their websites. You'll find links to them either as advertising at some of the sites listed above or in the messages themselves.

Office Help Sites

As with FoxPro, you can find help for the Office products on CompuServe. Also like FoxPro, the forums that support Office were consolidated in late 1999. The new home for Office is

MSOForum and it includes sections for Word, Excel, PowerPoint, Outlook, Access, and several other related products. Best of all, as with FoxPro, the people who hang out there are warm, friendly and helpful. The URL for accessing the Office forum from the web is go.compuserve.com/MSOfficeForum?loc=US&access=public

The Microsoft public newsgroups on msnews.microsoft.com include groups for all the Office products. They're spread out all over the server, but look for groups with the various product names in their names plus strings like "vba" or "programming" or "automation." Don't be too choosy, though – it appears that there's a single group for all PowerPoint questions called microsoft.public.powerpoint.

Woody Leonhard has been writing about Word and the other Office products practically since the dawn of time. In addition to Woody's Office Watch (mentioned above), his website, www.wopr.com, offers discussion groups for the Office products and more.

Many of the Microsoft MVPs have websites that provide tips, tricks, and code. Rather than trying to provide a comprehensive list here of the Office-related sites, I'm pointing you to a site that contains such a list and gets updated occasionally. Thanks to Karl Petersen for maintaining this list. Check out www.mvps.org/links.html for links to sites about not just Office, but pretty much any Microsoft product you can think of.

Ready to Go

Equipped with the tools, commands and resources described here, you can begin to automate the Office applications. The good news is that you don't have to learn everything about each application in order to use it productively. Start small by automating simple tasks and learn new aspects as you need them. Before you know it, you'll be writing complex automation code.

Acknowledgements

These notes are adapted from *Microsoft Office Automation with Visual FoxPro* by Tamar E. Granor and Della Martin, Hentzenwerke Press (2000). Thanks to my co-author, Della and to our technical editor, Ted Roche, for their contributions and to publisher, Whil Hentzen, for his permission to use this material.