DEV185

Driving Word and Excel from VFP

By Tamar E. Granor, Ph.D.

Visual FoxPro is a great database product, but it's not a word processor or a spreadsheet. Why try to fit a square peg in a round hole? If you need a word processor or spreadsheet, use one.

This session shows you how to create, access and manipulate Word documents and Excel spreadsheets from a Visual FoxPro application. Learn how VFP can use Word to create reports, send data to an Excel workbook, read Word and Excel documents, and much more. This session assumes some familiarity with Word and/or Excel and a basic understanding of object-oriented programming. They refer to Office 97, except where noted, but all examples should work as shown in Office 2000.

Is Everything a Nail?

When you're really familiar with a product, it's tempting to use it for all your needs. Many people have been doing that with FoxPro for years. Need a number cruncher? Write it in FoxPro. Need a word processor? Write it in FoxPro. Need a file manager? Write it in FoxPro. That we can do some many diverse things is a testament to FoxPro's strength and versatility. But using our hammer (FoxPro) to do it doesn't make all those tasks nails.

Both user's needs and applications' capabilities are growing. While it made sense to write a simple word processor for editing customer notes in FoxPro/DOS in 1994, it doesn't make sense to do so in Visual FoxPro 6.0 in 1999. There are too many other alternatives out there that can just be plugged into a VFP app.

Microsoft's COM (Component Object Model or Common Object Model) specifies ways for applications to work together. It's a successor to OLE (Object Linking and Embedding), which in turn was a successor to DDE (Dynamic Data Exchange). COM offers several approaches to communication between apps. One of the most powerful is *automation*, the ability for one application to order another around. (Other COM techniques include ActiveX controls and in-place activation.)

Automation Basics

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 VFP application (the client) can address anything from

the Office applications to Windows' file system to Lotus Notes and much more. In fact, VFP itself can be used as an automation server.

To start an automation session, the client creates or grabs a reference to an instance of the server. Creating an automation server instance isn't much different than creating any other object in VFP. You call CreateObject() and tell it what to create. For example, to create an instance of the Word automation server, issue:

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

Once this command finishes, oWord contains a reference to an instance of Word, which can be manipulated by calling methods or changing properties. (That's if all is well. If Word isn't installed or isn't properly registered, you get an error message.) By default, the Office automation servers are created invisible. To make the Word instance visible, issue:

```
oWord.Visible = .T.
```

To close the Word instance, use:

```
oWord.Quit()
```

Using Excel is almost identical. To create an Excel instance, issue:

```
oExcel = CreateObject("Excel.Application")
```

Except for the variable name used (and that's just my choice for readable code), the commands to make the Excel automation server visible and to close it are identical to those for Word:

```
oExcel.Visible = .T.
oExcel.Quit()
```

Of course, not all the properties and methods of Word and Excel are the same; after all, the two applications have different purposes and abilities. Other servers can also be instantiated with CreateObject(), but have different properties and methods.

It's also possible with some servers to latch onto an existing instance. The GetObject() function looks for an instance of the server and returns a reference to that object, if there is one. To get a reference to an open instance of Word, use:

```
oWord = GetObject(,"Word.Application")
```

Note that the first parameter is omitted. If the application you name isn't already open, you get an error; GetObject() doesn't open a new instance for you.

There's another way to use GetObject(), as well. You can pass a filename as the first parameter; this asks VFP to open the appropriate application for the file, then open the file. It's like double clicking on the file in Explorer. This approach returns a reference to the object inside the application that refers to the open file (a Document in Word, a WorkBook object in Excel).

But What's in There?

A Visible property to make the application show and a Quit method to close it are pretty obvious guesses for most automation servers (though not all have them or call them that). But they're not all that useful. How do you find out the properties and methods of the server you want to use? If you're lucky, the application's documentation tells you. If not, you're headed down a rough road.

For the Office applications, there are three main approaches to determining the properties and methods of the servers and how to use them. In most cases, you'll need to combine all three to get what you need.

Read the Fine Manual

Word and Excel (along with the other Office apps) each supply a Help file that documents its members. In each case, you'll find it from the main Help menu by selecting Contents and Index, then going down to the bottom. Near the last item, you should find "Microsoft Blah Visual Basic Reference," where "Blah" is the application you're looking at. Double-clicking that item expands it and makes available "Visual Basic Reference". Choose that one to open the relevant Help file. (For the Office 2000 applications, look for "Programming Information" near the bottom of the Contents list.)

If you didn't find the appropriate Visual Basic reference, it means you didn't install that Help file with the application. To do so, you have to choose Custom rather than Complete installation. ("Complete" seems to mean "typical" more than "everything.") You can install the file at any time by running Setup again.

Why "Visual Basic Reference?" The language used for writing code in the Office applications is a version of Visual Basic called "Visual Basic for Applications" or VBA. When you use automation with those apps, although you're writing VFP code, you're talking to the VBA parsing engine.

In addition, the Office 97/Visual Basic Programmer's Guide is available online. As of this writing, you could find it at http://www.microsoft.com/OfficeDev/Articles/OPG/.

Let Someone Else Write the Code

The macro recorders of Word and Excel turn user actions into VBA code. So one way to figure out how to automate something is to record a macro to do it, then examine the macro. (Choose Tools-Macro-Record New Macro to start recording.) This approach shows you the properties and methods involved as well as the parameters to pass to the methods.

Converting a VBA macro to VFP code is harder than it should be for several reasons. This line of Word VBA code, which moves the insertion point (the vertical bar that determines where the next character is inserted) one character to the right, demonstrates the problems:

```
Selection.MoveRight Unit:=wdCharacter, Count:=1
```

First, unlike VFP, VBA makes some assumptions about what object you're talking to. In the line above, Selection is really This.Selection (though VBA doesn't actually support the VFP keyword This), the current selection (highlighted area) of the Word instance.

Second, VBA allows methods to use what are called *named parameters*. In the code above, the method called is MoveRight. Two 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. However, VFP doesn't support named parameters; you must specify parameters in the right order. Fortunately, the Help file shows the parameters in their required order.

The first parameter in the example shows the third problem that occurs in translating VBA to VFP. It specifies that a parameter called Unit should have the value wdCharacter, but what's wdCharacter? It's one of dozens of defined constants available in Word's version of VBA. (It turns out that wdCharacter is 1.)

Word's VBA Help doesn't supply the values of defined constants. In fact, Help uses them exclusively and doesn't show their actual values anywhere. To find out what wdCharacter and all the others stand for, use the Object Browser available through the Visual Basic Editor.

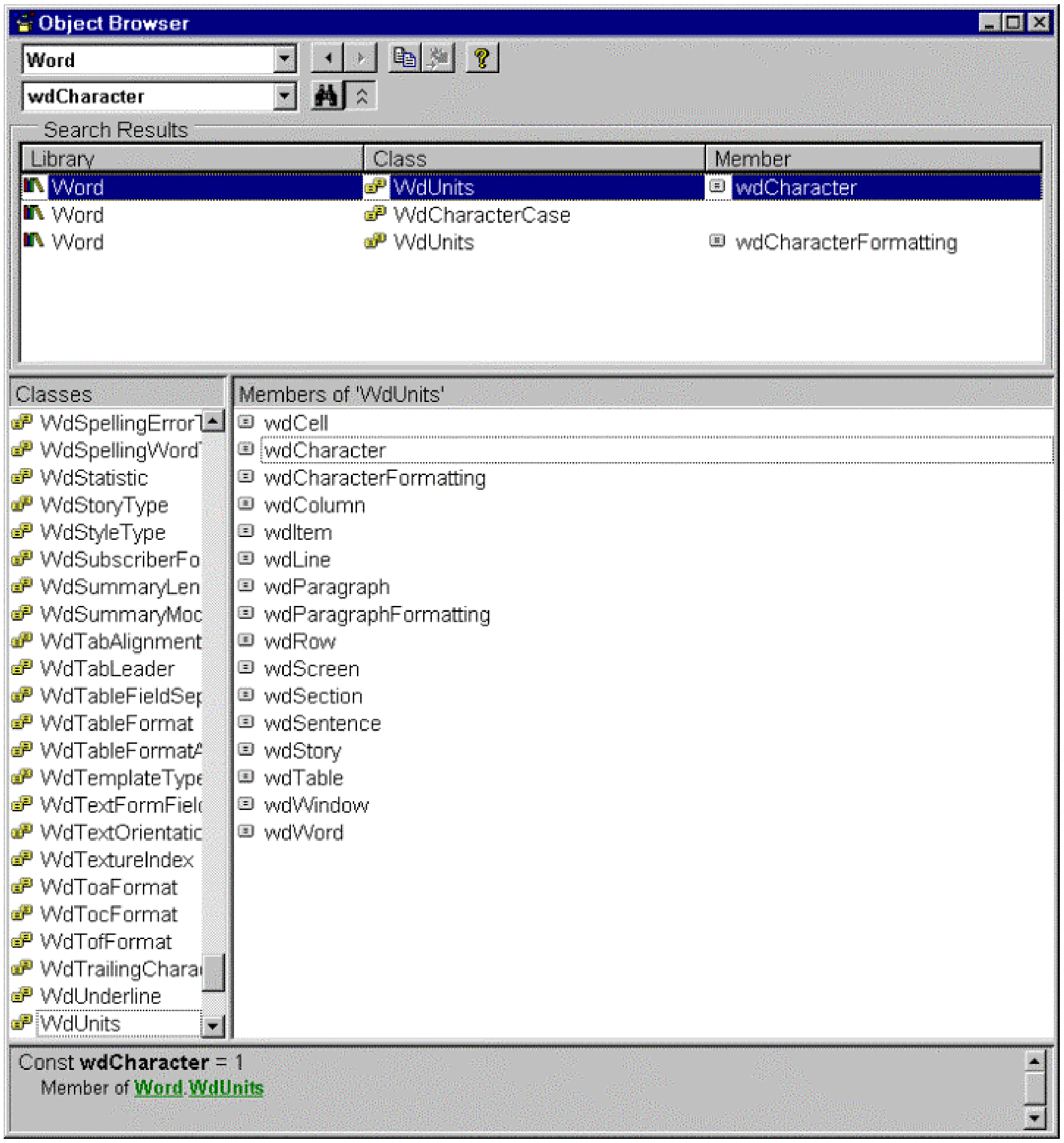
Before moving on to the Object Browser, there are a couple of thing worth noting about the Macro Recorder and the Visual Basic Editor. First, be aware that the Macro Recorder doesn't always produce the *best* code for a task. The code it produces gives you an idea of what can be done, but there may be better ways to accomplish the same task.

Second, the Visual Basic Editor has a feature called IntelliSense that makes writing code there easier. When you type an object name and a period (like "oRange."), an appropriate list of properties and methods appears. When you choose a method from the list (use Tab to choose without ending the line), a list of parameters for that method appears like a tooltip to guide you. As you enter parameters, your position in the parameter list is highlighted to keep you on track. Unfortunately, VFP doesn't support IntelliSense yet.

Take Me for a Browse

One of the most powerful tools available for figuring out automation code is the Object Browser (figure 1). 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.

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 possibly appropriate references – choose one to learn more about it. Figure 1 shows the Object Browser used to determine the value of the constant wdCharacter. At the very bottom, you can see that it's a constant with a value of 1.



Library	Class	Member
Word	WdUnits	wdCharacter
Word	WdCharacterCase	
Word	WdUnits	wdCharacterFormatting

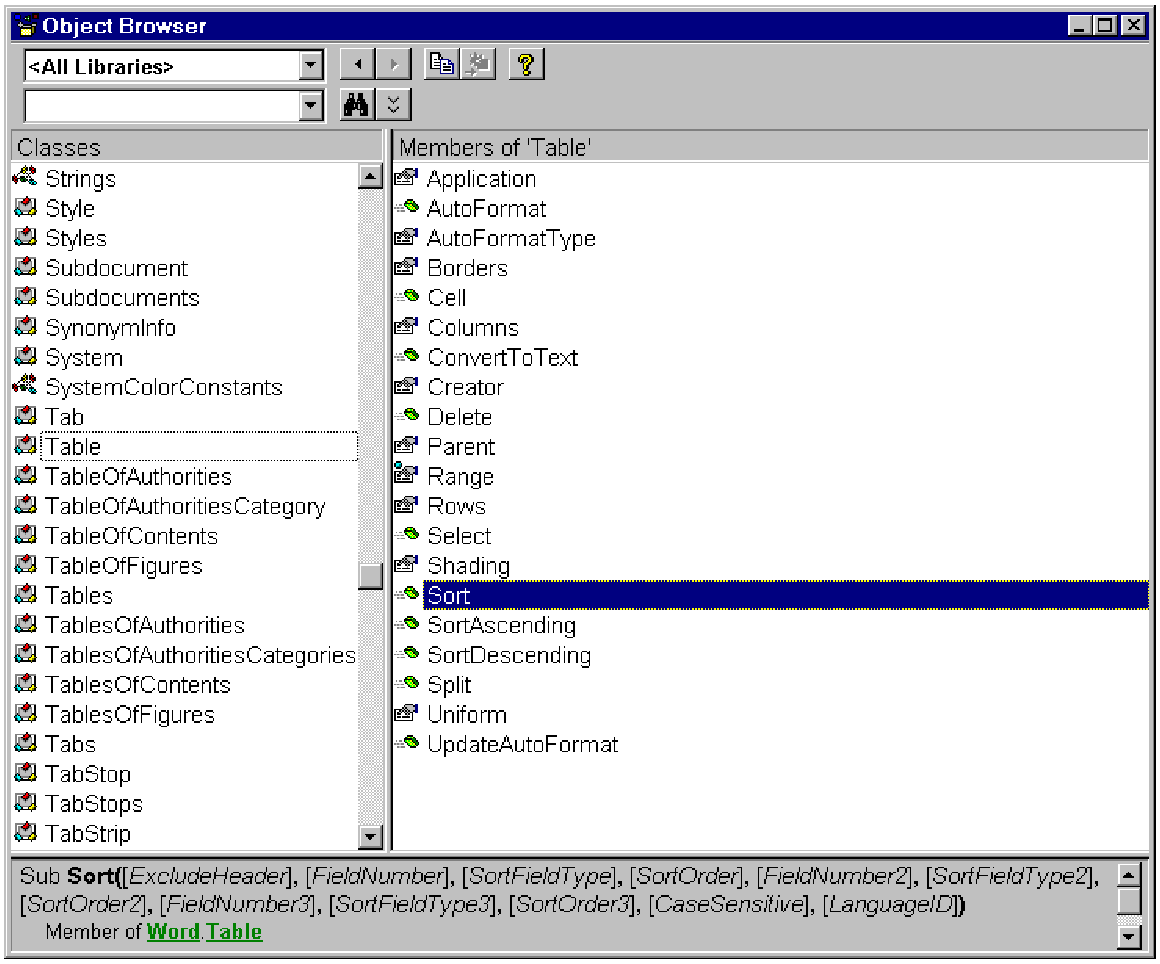
Classes: WdSpellingErrorT, WdSpellingWord, WdStatistic, WdStoryType, WdStyleType, WdSubscriberFo, WdSummaryLen, WdSummaryMoc, WdTabAlignment, WdTabLeader, WdTableFieldSep, WdTableFormat, WdTableFormatA, WdTemplateType, WdTextFormField, WdTextOrientatic, WdTextureIndex, WdToaFormat, WdTocFormat, WdTofFormat, WdTrailingChara, WdUnderline, WdUnits

Members of 'WdUnits': wdCell, wdCharacter, wdCharacterFormatting, wdColumn, wdItem, wdLine, wdParagraph, wdParagraphFormatting, wdRow, wdScreen, wdSection, wdSentence, wdStory, wdTable, wdWindow, wdWord

Const **wdCharacter** = 1
Member of **Word.WdUnits**

Figure 1 The Object Browser – Use this tool to find out constant values, properties and methods, and more. Here, it shows that wdCharacter is a constant with a value of 1 and is a member of a group of constants called WdUnits.

The Object Browser is also useful for moving around the object hierarchy to get a feel for what the various objects can do. In Figure 2, the Search Results pane has been closed and the members of Word's Table object are shown in the right pane. The Sort 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 server itself, so the list it shows must be correct while Help can contain errors. Even better, the Object Browser and Help can work together. Press F1 in the Browser and Help opens to the highlighted item.



Sub **Sort(**[*ExcludeHeader*], [*FieldNumber*], [*SortFieldType*], [*SortOrder*], [*FieldNumber2*], [*SortFieldType2*], [*SortOrder2*], [*FieldNumber3*], [*SortFieldType3*], [*SortOrder3*], [*CaseSensitive*], [*LanguageID*]**)**
Member of **Word**.**Table**

Figure 2 Members and Syntax in the Object Browser – You can examine objects and their properties and methods using the Browser.

With these tools in hand, you can start the process of writing code to automate Word and Excel.

Taking up a Collection

The object models of both Word and Excel (along with most COM servers) contain many collections. A *collection* is the OOP world's version of an array. It contains references to a number of objects, generally of the same type, and provides access to those objects. Visual FoxPro has a few native collections (Forms, Pages, etc.) as well as several COM collections (such as Projects).

Most collections (though not the native VFP collections) have a few methods and properties in common. The Count property typically tells you how many items are in the collection. Item, which is a method in some collections and a property in others (it's a property in the Office apps), provides access to the individual members of the collection.

Whether a method or a property, Item typically takes a single parameter or index, and returns a reference to that member of the collection. In many cases, you can specify the item you want by name rather than number. _VFP.Projects.Item["Tastrade.PJX"] provides a reference to the TasTrade project, if it's open, for example.

In addition, many collections let you omit the Item keyword and simply attach the parameter/index to the collection name. So we can write _VFP.Projects["Tastrade.PJX"] and still get a reference to the project.

What's the Word?

Automation to Word changed dramatically between Word 95 and Word 97. Word 95 had a very simple object model consisting primarily of a single object (Word.Basic). In Word 97 and later, the object model better reflects the internal structure of Word and its documents.

The key object in Word is Document, which represents a single, open document. The Word server has a Documents collection, containing references to all open documents. The server also has an ActiveDocument property that points to the currently active document.

The Document object has lots of properties and methods. Many of its properties are references to collections, such as Paragraphs, Tables and Sections. Each of those collections contains references to objects of the indicated type. Each object contains information about the appropriate piece of the document. For example, the Paragraph object has properties like KeepWithNext and Style.

The Word server object, called Application, has its own set of properties and methods, including a number of other collections. In addition to ActiveDocument and Visible, the Application object's properties include StartupPath, Version and WindowState. The Quit method used above has several optional parameters – the first indicates what to do if any open documents have been changed and not saved. In addition to Quit, the application object has methods for converting measurements from one set of units to another, checking grammar and spelling, and much more.

Word Visual Basic Help contains a diagram of Word's object model. The figure is "live" – when you click on an object, you're taken to the Help topic for that object. Figure 3 shows the portion of the object model diagram that describes the Document object.

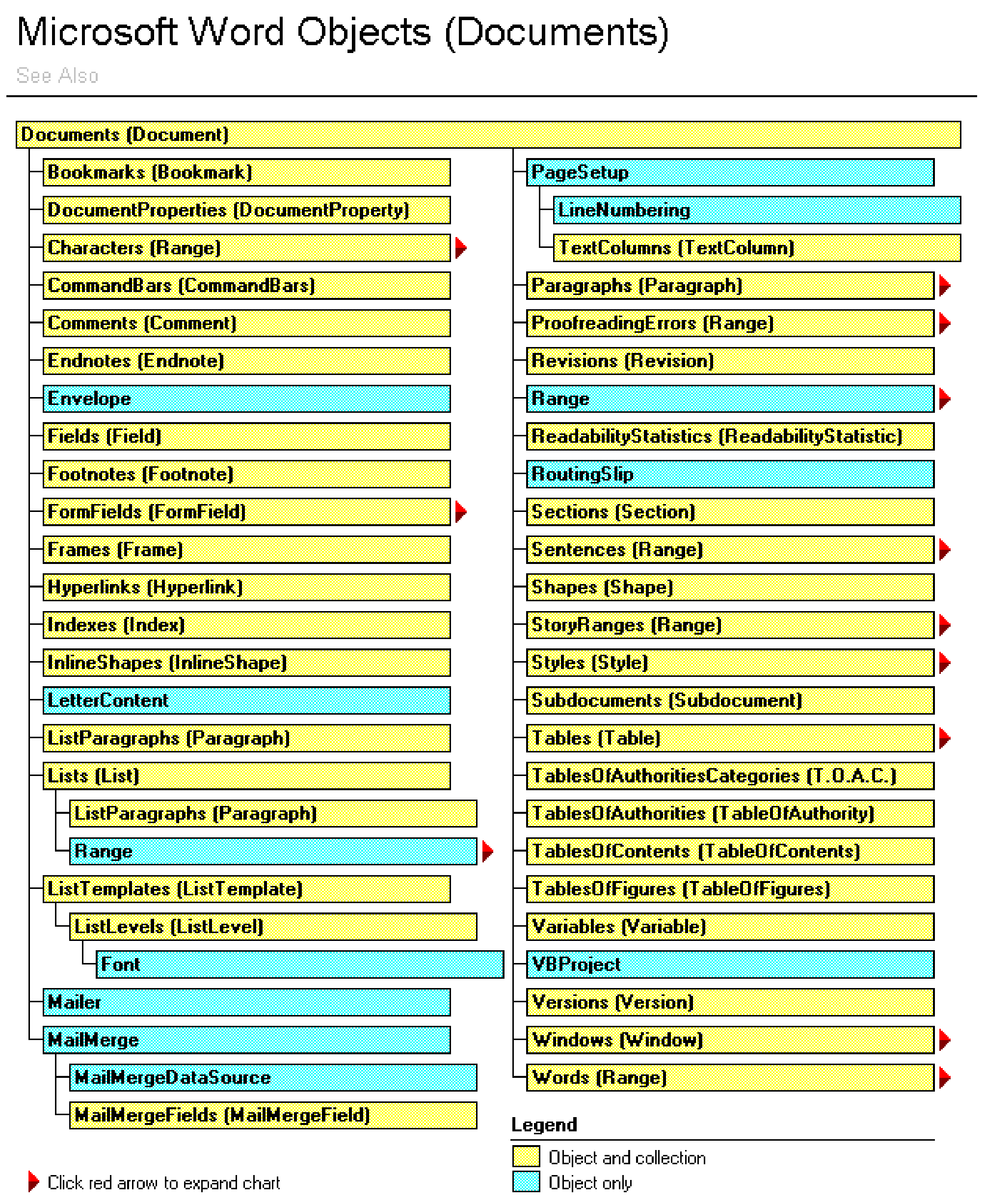


Figure 3 Word Object Model – The Help file offers a global view of Word's structure.

Managing Documents

The methods for creating, opening, saving and closing documents are fairly straightforward. The only complication is in what object provides which method. The methods for creating and opening a document belong to the Documents collection. The methods for saving and closing a document belong to the Document object. Although confusing at first glance, this actually makes sense since you don't have a Document object to work with at the time you create or open a document. But when it comes time to save or close it, a document reference is available. (Files in VFP's Project object work the same way. Add is a method of the Files collection, but Remove belongs to the File object.)

To open an existing document, use the Open method. Open has many parameters, but most of the time, the only one that matters is the first, the name and path of the file to open. The Word server doesn't recognize VFP's search path, so you usually need to provide the full path to the document, like this:

```
oDocument = oWord.Documents.Open("d:\writing\confs\la99\dev185.doc")
```

If it's successful, Open returns an object reference to the newly opened document. If the specified document doesn't exist, an error is generated and nothing is returned. (This example, like the others below, assumes you have an instance of Word running, with oWord holding a reference to it.)

To create a new document, use the Add method, which has only two, optional, parameters. The important one is the first, which indicates the path to the template on which the new document should be based. If it's omitted, the new document is based on the Normal template.

Templates in Word allow you to define the framework for documents and then create as many documents as you want based on that framework. A template can contain both text and formatting, including style definitions (see Working with Styles below), and can be quite simple or extremely complex. It can also contain *bookmarks*, named points in the document, which you can use to quickly access to a particular position.

Like Open, Add returns a reference to the newly created document. This line creates a new document based on the "Elegant Fax" template that comes with Word.

```
oDocument = oWord.Documents.Add( ;
   "E:\Msoffice\Templates\Letters & Faxes\Elegant Fax.DOT")
```

As with Open, the full path to the template is needed. To do so, though, you need to know where Word is installed or at least, where the templates are kept. Fortunately, Word provides some clues. The DefaultFilePath method of the Options object (which is a member of the Word Application object) can return the root directory for both the user's own templates and user's workgroup's templates. Using that information, we can search for the path to a specific template, given its name. The NewDocument method of the cusWord class on the conference CD accepts a template with or without a path and attempts to locate the template file before creating a new document.

When you're finished working with a document, two methods are available to save it. Save saves the document back to its current file; if it's never been saved, that pops up the Save As dialog box. SaveAs lets you specify the file name (and a lot of other stuff) without seeing a dialog, which is usually what you want with automation code.

If the currently active document has already been saved, this line resaves it without user intervention:

```
oWord.ActiveDocument.Save()
```

To save the document to a different file or to save a document for the first time without the user specifying a file name, call SaveAs and pass the file name, like this:

```
oWord.ActiveDocument.SaveAs("D:\Documents\ThisIsNew.DOC")
```

Be careful. When you pass a file name to SaveAs, it overwrites any existing file without prompting. (Of course, SaveAs, like Word's other methods, doesn't respect VFP's SET SAFETY setting, since it's not running inside VFP.)

Accessing Parts of a Document

Most of what you want to do with Word involves adding to, modifying or reading an existing document. There are a variety of ways to do these things, but the key to just about all of them is the Range object and, to a lesser extent, the Selection object.

The Selection object represents the currently highlighted (that is, selected) portion in a document. If nothing is highlighted, Selection refers to the insertion point. There's only one Selection object, accessed directly from the Word server. For example, to find out how many paragraphs are in the current selection, you can use this code:

```
nParagraphs = oWord.Selection.Paragraphs.Count
```

A Range object can represent any portion of a document. Multiple ranges can be defined and used at the same time. Many Word objects have a Range property that contains an object reference to a range for the original object. For example, to create a Range from the third paragraph of the active document, you can use:

```
oRange = oWord.ActiveDocument.Paragraphs[3].Range
```

Document has a Range method that lets you specify a range by character position. For example, to get a reference to a Range from the 100th to 250th characters in the active document, use:

```
oRange = oWord.ActiveDocument.Range(100,250)
```

Document's Content property contains a reference to a Range consisting of the entire main document (the body of the document without headers, footers, footnotes, and so on). So the next two commands are equivalent:

```
oRange = oWord.ActiveDocument.Range()
oRange = oWord.ActiveDocument.Content
```

It's easy to switch between Selection and Range objects. Like many other objects, Selection has a Range property. Range has a Select method that highlights the range's contents, turning it into the Selection. For example, to highlight the range from the previous example, use:

```
oRange.Select()
```

Selection and Range seem quite similar and, in many ways, they are, but there are differences. The biggest, of course, is that you can have multiple Ranges, but only one Selection. In addition, working with a Range is usually faster than working with a Selection. On the whole, Word VBA experts recommend using Range rather than Selection wherever possible. The main reason is that using Selection is essentially duplicating screen actions with code; Range lets you operate more directly. Word's Macro Recorder tends to use the Selection object; this is one thing to be aware of when converting Word macros to VFP code.

Manipulating Text

The Text property of Range and Selection contains whatever text is in the specified area. To import document contents, create an appropriate Range and read its Text property, like this:

```
oRange = oWord.ActiveDocument.Paragraphs[7].Range
cParagraph7 = oRange.Text
```

Text also lets you add or change the document contents. You can add text by assigning it to the Text property.

```
oRange.Text = "This is a new sentence."
```

You can also add the text to whatever's already there. Simple text manipulation does the trick.

```
oRange.Text = oRange.Text + "Add text at the end."
```

or

```
oRange.Text = "Add text at the beginning " + oRange.Text
```

Another possibility is to read text into VFP, manipulate it in some way and write back.

```
cMakeUpper = oRange.Text
cMakeUpper = UPPER(cMakeUpper)
oRange.Text = cMakeUpper
```

That example can be shortened to a single line, like this:

```
oRange.Text = UPPER(oRange.Text)
```

Using this approach, we can send the data from a record to a new, blank document:

```
USE _samples + "\TasTrade\Data\Customer"

LOCAL oDocument, oRange
oDocument = oWord.Documents.Add()  && Use the Normal template
oRange = oDocument.Range()

oRange.Text = Customer_ID + ": " + Company_Name
oRange.Text = oRange.Text + "Attn: " + TRIM(Contact_Name) + ;
              " - " + Contact_Title
oRange.Text = oRange.Text + Address
oRange.Text = oRange.Text + TRIM(City) + " " + TRIM(Region) + ;
              Postal_Code
oRange.Text = oRange.Text + UPPER(Country)
```

Because Word always keeps a paragraph marker (CHR(13)) at the end of the document, when executing this code, it adds a paragraph marker after each addition to oRange.Text. In other situations (including the examples below), you need to add the paragraph marker explicitly.

Of course, building a document by composing a single string doesn't take advantage of the special capabilities of Word. Range's InsertAfter and InsertBefore methods let you add text at the end or beginning, respectively, and expand the range to include the new text.

Here's an alternative, faster approach to the example:

```
#DEFINE CR CHR(13)

USE _samples + "\TasTrade\Data\Customer"

LOCAL oDocument, oRange
```

```
oDocument = oWord.Documents.Add()  && Use the Normal template
oRange = oDocument.Range()

oRange.InsertAfter(Customer_ID + ": " + Company_Name + CR)
oRange.InsertAfter("Attn: " + TRIM(Contact_Name) + ;
              " - " + Contact_Title + CR)
oRange.InsertAfter(Address + CR)
oRange.InsertAfter(TRIM(City) + " " + TRIM(Region) + ;
              Postal_Code + CR)
oRange.InsertAfter(UPPER(Country))
```

In my tests, the InsertAfter version was one-and-a-half times to twice as fast as the concatenation method.

Moving in a Range or Selection

Besides changing a range or selection's content, you may need to modify its extent. A number of methods change the area covered by a range or selection. One of the simplest is the Move method, which changes the boundaries of the range or selection.

Move accepts two parameters. The first indicates the unit of movement – you can move by characters, words, paragraphs, rows in a table, or the whole document. The second parameter tells how many of the specified units to move – a positive number indicates forward movement (toward the end of the document), while a negative number means backward movement (toward the beginning of the document).

In all cases, the range or selection is collapsed to a single point before moving. When moving forward, the range or selection is collapsed to its end point, then moved; when moving backward, it's collapsed to its beginning point before moving.

Constants from the wdUnits group are used for the units of movement. Table 1 shows the values for this group that can passed to the Move method. VFP constant definitions for these, as well as for other Word constants used in these notes, are in the file Word.H on the conference CD. The notes use the constants for readability.

Constant	Value	Description
wdCharacter	1	One character.
wdWord	2	One word.
wdSentence	3	One sentence.
wdParagraph	4	One paragraph.
wdSection	8	One section of a document. (Word allows you to divide documents into multiple sections with different formatting.)
wdStory	6	The entire length of whichever part of the document you're in. Word considers the main body of the document to be one "story," the header to be another "story," the footnotes to be a third and so forth.

wdCell	12	One cell in a table.
wdColumn	9	One column of a table.
wdRow	10	One row of a table.
wdTable	15	The entire space of a table.

Table 1 Word units – The constants in the wdUnits group represent portions of a document.

To create a range at the end of the document, you can:

```
oRange  = oWord.ActiveDocument.Range()
oRange.Move( wdStory, 1)
```

To create the same document based on the Customer table as in the examples above, use this code:

```
#DEFINE CR CHR(13)
#DEFINE wdStory 6

USE _samples + "\TasTrade\Data\Customer"

LOCAL oDocument, oRange
oDocument = oWord.Documents.Add()  && Use the Normal template
oRange = oDocument.Range()

oRange.Text = Customer_ID + ": " + Company_Name + CR
oRange.Move(wdStory)
oRange.Text = "Attn: " + TRIM(Contact_Name) + ;
              " - " + Contact_Title + CR
oRange.Move(wdStory)
oRange.Text = Address + CR
oRange.Move(wdStory)
oRange.Text = TRIM(City) + " " + TRIM(Region) + ;
              Postal_Code + CR
oRange.Move(wdStory)
oRange.Text = UPPER(Country)
```

In terms of speed, this version comes out in between the concatenation and insert versions.

The MoveStart and MoveEnd methods let you change one boundary without affecting the other. Collapse lets you reduce a range or selection to a single point. It takes one parameter, indicating the direction of the collapse. Passing the constant wdCollapseEnd, (with a value of 0) collapses the range or selection to its end point (the point closest to the end of the document). Passing wdCollapseStart (whose value is 1) reduces the range or selection to its starting point.

The example can be rewritten yet again to use Collapse to control the range:

```
#DEFINE CR CHR(13)
#DEFINE wdCollapseEnd 0

USE _samples + "\TasTrade\Data\Customer"

LOCAL oDocument, oRange
oDocument = oWord.Documents.Add()  && Use the Normal template
oRange = oDocument.Range()

oRange.Text = Customer_ID + ": " + Company_Name + CR
oRange.Collapse(wdCollapseEnd)
oRange.Text = "Attn: " + TRIM(Contact_Name) + ;
              " - " + Contact_Title + CR
oRange.Collapse(wdCollapseEnd)
oRange.Text = Address + CR
oRange.Collapse(wdCollapseEnd)
oRange.Text = TRIM(City) + " " + TRIM(Region) + ;
              Postal_Code + CR
oRange.Collapse(wdCollapseEnd)
oRange.Text = UPPER(Country)
```

Timing-wise, this version comes out about the same as the Move version.

A number of other methods allow fine-tuning of movement. They include MoveEnd, MoveStart, MoveLeft, MoveRight, EndOf and StartOf. Some methods apply only to the selection, not to ranges.

Before leaving the example, it's worth commenting that, for this particular task, the fastest approach of all is to concatenate all the strings in VFP, then send one string to the document:

```
#DEFINE CR CHR(13)

cText = Customer_ID + ": " + Company_Name + CR
cText = cText + "Attn: " + TRIM(Contact_Name) + ;
        " - " + Contact_Title + CR
cText = cText + Address + CR
cText = cText + TRIM(City) + " " + TRIM(Region) + ;
        Postal_Code + CR
cText = cText + UPPER(Country)

oRange.Text = ""
oRange.InsertAfter(cText)
```

Formatting

If all we could do was send text to Word and read the text already there, Automation would be useful, but not worth too much trouble. However, there's much more to automating Word than just sending and receiving text. One of the big benefits of using Word rather than VFP is the ability to apply complex formatting to documents.

Word allows documents to be formatted in a number of ways and the objects involved in formatting reflect that structure. For example, the Font object contains properties for the settings found in Word's Font dialog (Format-Font on the menu). The ParagraphFormat object controls the settings in the Paragraph dialog, such as indentation, spacing and alignment. Similarly, the settings from the Page Setup dialog are controlled by the PageSetup object. Style objects represent the individual styles available in the document. These four objects manage most of the frequently used settings. Other objects control other aspects of formatting.

Setting Fonts

Range and Selection (as well as some other objects) each have a Font property that points to a Font object. Changing the properties of the Font object modifies the font of the range or selection. For example, to change all the customer information in the example above to 12-point Arial, use these lines:

```
oRange = oDocument.Range()
oRange.Font.Name = "Arial"
oRange.Font.Size = 12
```

To simplify matters, just set the desired font first. Here's an updated version of the code that uses 12-point Arial from the start:

```
#DEFINE CR CHR(13)

USE _samples + "\TasTrade\Data\Customer"

LOCAL oDocument, oRange
oDocument = oWord.Documents.Add()  && Use the Normal template
oRange = oDocument.Range()

oRange.Font.Name = "Arial"
oRange.Font.Size = 12

oRange.InsertAfter(Customer_ID + ": " + Company_Name + CR)
oRange.InsertAfter("Attn: " + TRIM(Contact_Name) + ;
              " - " + Contact_Title + CR)
oRange.InsertAfter(Address + CR)
oRange.InsertAfter(TRIM(City) + " " + TRIM(Region) + ;
              Postal_Code + CR)
oRange.InsertAfter(UPPER(Country))
```

(My Word contacts point out that this isn't the best way to set the font for a whole document. It's better to use a template where the font of the Normal style has been set as needed. See Working with Styles.)

You can allow the user to choose the font by calling VFP's GetFont() function first. Here's a function that lets the user specify a font, prompting with the font currently in use. Then, it changes to the specified font:

```
* SetUserFont.PRG
* Let the user specify a font, then set
* a passed font object to use it.

#DEFINE TRUE -1
#DEFINE FALSE 0

LPARAMETERS oFont
   * oFont = Reference to a font object

LOCAL cName, nSize, cStyle
LOCAL cFontString, aFontInfo[3]

* Did we get a font object to work with?
IF VarType(oFont) <> "O"
   RETURN .F.
ENDIF

* Get current settings of font object.
WITH oFont
   cName = .Name
   nSize = .Size
   cStyle = ""
   IF .Bold = TRUE  && Can't use VFP .T. here
      cStyle = cStyle + "B"
   ENDIF
   IF .Italic = TRUE  && or here
      cStyle = cStyle + "I"
   ENDIF
ENDWITH
```

```
* Ask the user for a font
cFontString = GetFont(cName, nSize, cStyle)

IF EMPTY(cFontString)
   * User cancelled
   RETURN .F.
ELSE
   * Parse the chosen into its components
   cFontString = CHRTRAN(cFontString, ",", CHR(13))
   ALINES(aFontInfo,cFontString)

   * Apply them to the font object
   WITH oFont
      .Name = aFontInfo[1]
      .Size = VAL(aFontInfo[2])
      IF "B"$aFontInfo[3]
         .Bold = .T.  && .T. works here
      ENDIF
      IF "I"$aFontInfo[3]
         .Italic = .T.
      ENDIF
   ENDWITH
ENDIF
RETURN .T.
```

The example demonstrates another complication of working with VBA objects in VFP. Although logical properties (like Bold and Italic) can be set by assigning VFP's logical values .T. and .F., they can't be compared to logical values. Code like:

```
IF oFont.Bold
```

fails with the error "Operator/operand type mismatch". That's because Bold isn't just a logical setting with acceptable values of true and false; it accepts two other values, represented by the constants wdToggle (9999998) and wdUndefined (9999999). (wdUndefined indicates that the property is true for part of the range and false for the rest. wdToggle is used to reverse the current setting.)

When you assign logical values, Word translates them somewhere along the way, but for comparison, you have to use the numeric values. The example defines constants TRUE and FALSE to avoid coding the actual values.

To use this function, pass it a reference to a font object. For example:

```
SetUserFont( oRange.Font )
```

This call prompts the user to change the font for the range with VFP's GetFont dialog.

Formatting paragraphs

The ParagraphFormat object determines things like alignment of text, indentation (both amount and type), spacing of lines and paragraphs, handling of widow and orphans, and much more. Both Range and Selection have a ParagraphFormat object, accessed through the same-named Property. The Paragraph object has a Format property that accesses a ParagraphFormat object. In all cases, it's accessed through the Format property. So, to set a range for full justification, you'd use:

```
oRange.Format.Alignment = wdAlignParagraphJustify
```

wdAlignParagraphJustify is one of a set of alignment constants. (Its value is 3.)

You can generally omit the Format property in the command, so the following line has the same effect:

```
oRange.Alignment = wdAlignParagraphJustify
```

Working with Styles

While it's appropriate to adjust the formatting of a word, sentence or paragraph here or there, the most effective way to use Word is to take advantage of styles. A *style* in Word is a named format that you can apply to a portion of a document. (When you're working in Word, you can see the current style in the first dropdown on the Formatting toolbar.)

Word has two kinds of styles: paragraph styles and character styles. Character styles are used for fragments and control only a few settings, primarily font-related. Paragraph styles, as the name implies, apply to entire paragraphs and include a lot more options. Paragraph Styles can specify font and paragraph formatting, as well as tab settings and much more.

Using styles is much like using classes in VFP. They make it easy to enforce uniformity throughout and across documents and let you change the characteristics of sections of text with a single change. Word's styles offer some other benefits, as well. For example, each paragraph style knows the style for the paragraph to follow. So, a style normally used for a heading can be set to be followed by the style used for body text following that heading. With a little more work, styles can used to provide an outline of a document.

The Document object includes a Styles collection, containing Style objects for all the styles stored in the document. You can add your own styles using the Add method. The Paragraph object's Style property points to the Style object for that paragraph's style.

What all this means is that, rather than writing a lot of code to change fonts and sizes, and to set alignment and leading and other things like that, you can simply define a few custom styles or modify built-in styles, then apply them to your paragraphs as needed.

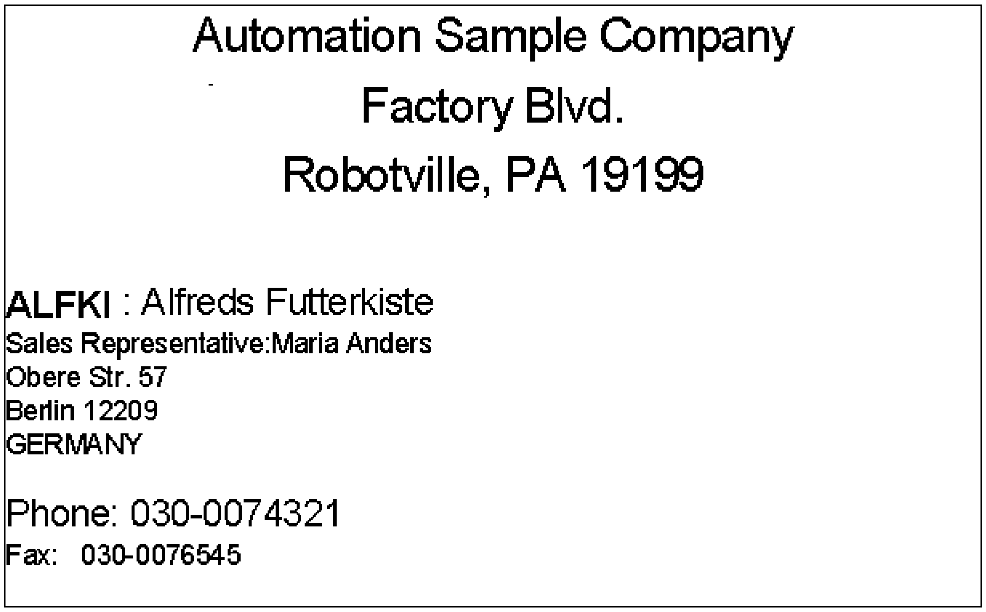
This example takes the simple customer address document from the previous examples and begins to create a document worthy of using Word. It creates several new styles to do the job. In practice, you could probably use the built-in Normal and Heading x (there are multiple heading levels) for this document. But the example shows how easy it is to create new styles.

```
* Create a formatted document by sending data from one record.
* Demonstrates Style objects, but it's more likely the needs here
* could be met by existing styles.

#INCLUDE "Word.H"
#DEFINE CR CHR(13)

USE _samples + "\TasTrade\Data\Customer"

LOCAL oDocument, oRange
LOCAL oBodyStyle, oMajorHeadingStyle, oMinorHeadingStyle

oDocument = oWord.Documents.Add()  && Use the Normal template
oRange = oDocument.Range()

* Set up styles. Base body style on Normal.
oBodyStyle = oDocument.Styles.Add( "Body", wdStyleTypeParagraph )
WITH oBodyStyle
```

```
   .BaseStyle = oDocument.Styles[ wdStyleNormal ]
   WITH .Font
      .Name = "Arial"
      .Size = 12
   ENDWITH

   WITH .ParagraphFormat
      * These are fairly normal defaults, so these lines
      * may not be necessary
      .Alignment = wdAlignParagraphLeft
      .SpaceAfter = 0
   ENDWITH
ENDWITH

* Major heading is big and centered.
oMajorHeadingStyle = oDocument.Styles.Add( "MajorHeading", wdStyleTypeParagraph)
WITH oMajorHeadingStyle
   .BaseStyle = oBodyStyle
   .Font.Size = 20

   WITH .ParagraphFormat
      .Alignment = wdAlignParagraphCenter
      .SpaceAfter = 6  && leave a line after
      .KeepWithNext = .T.  && include at least one line of next
                                          && paragraph before new page
      .KeepTogether = .T.  && keep the whole paragraph together
   ENDWITH
ENDWITH

* Minor heading is just big.
oMinorHeadingStyle = oDocument.Styles.Add("MinorHeading", wdStyleTypeParagraph )
WITH oMinorHeadingStyle
   .BaseStyle = oBodyStyle
   .Font.Size = 16
ENDWITH

* Now create customer report
* First, our company info centered at the top
oRange.Style = oMajorHeadingStyle
oRange.InsertAfter("Automation Sample Company" + CR)
oRange.InsertAfter("Factory Blvd." + CR)
oRange.InsertAfter("Robotville, PA 19199" + CR)

* Now leave some blank space, then put info about this customer
oRange.Collapse( wdCollapseEnd )
oRange.End = oRange.End + 1 && to allow assignment to font
oRange.Style = oBodyStyle
oRange.InsertAfter(CR + CR)

* Use minor heading for customer id and name
* Put customer id in bold
oRange.Collapse( wdCollapseEnd )
oRange.End = oRange.End + 1 && to allow assignment to font
oRange.Style = oMinorHeadingStyle
oRange.InsertAfter(Customer_ID + ": " + TRIM(Company_Name) + CR)
oRange.Words[1].Font.Bold = .t.

* Regular body style for address info
oRange.Collapse( wdCollapseEnd )
oRange.End = oRange.End + 1 && to allow assignment to font
oRange.Style = oBodyStyle
oRange.InsertAfter(TRIM(Contact_Title) + ":" + TRIM(Contact_Name) ;
              + CR)
oRange.InsertAfter(TRIM(Address) + CR)
oRange.InsertAfter(TRIM(City) + " " + TRIM(Region) + ;
              Postal_Code + CR)
```

```
oRange.InsertAfter(UPPER(TRIM(Country)) + CR )
* Extra line for spacing
oRange.InsertAfter( CR )

* Back to minor heading for phone number
oRange.Collapse( wdCollapseEnd )
oRange.End = oRange.End + 1 && to allow assignment to font
oRange.Style = oMinorHeadingStyle
oRange.InsertAfter( "Phone: " + TRIM(Phone) + CR)

* Fax number in regular body style
oRange.Collapse( wdCollapseEnd )
oRange.End = oRange.End + 1 && to allow assignment to font
oRange.Style = oBodyStyle
oRange.InsertAfter( "Fax:   " + TRIM(Fax) + CR )
```

Note the use of the Words collection to bold only the customer id rather than the whole line. Here's a look at the resulting document:



Automation Sample Company
Factory Blvd.
Robotville, PA 19199

ALFKI : Alfreds Futterkiste
Sales Representative:Maria Anders
Obere Str. 57
Berlin 12209
GERMANY

Phone: 030-0074321
Fax: 030-0076545

Figure 4 Formatted document – The Font and ParagraphFormat objects control lots of formatting options. Style objects let you use predefined combinations easily.

Printing

Producing attractive documents is a good start, but users usually want printed copies, too. The PrintOut method of the Document object automates printing. It accepts a huge array of parameters. (This is one situation where named parameters start to look pretty good.)

Fortunately, all parameters are optional and you won't need most of those parameters for normal processing. Unfortunately, the ones you're most likely to need are in the middle of the list rather than up front.

To print three copies of pages 4 to 10 of a document (referenced by oDocument), use:

```
oDocument.PrintOut( , , wdPrintFromTo, , "4", "10", , "3")
```

where wdPrintFromTo is, of course, a constant. Note that the numeric values for page numbers and number of copies are passed as strings.

To print to a file, you have to specify both the file name and a flag that you're printing to file:

```
oWord.ActiveDocument.PrintOut( , , , "fileoutput", , , , , , , .t.)
```

The resulting file is ready to print, complete with printer codes. Keep in mind that the file is stored in Word's current directory by default, so it's a good idea to provide the full path.

Be forewarned that printing to file sets the Print to File checkbox in the Print dialog and leaves it set. A subsequent print to the printer resets it.

Word can also automatically create and print an envelope document. To do so, use the document's Envelope object. This example assumes that oRange is a range containing the customer name and address in a mailing format. It bookmarks the address and asks Word to create the envelope.

```
oDocument.Bookmarks.Add("EnvelopeAddress", oRange)
oDocument.Envelope.PrintOut(.T., , ,.F., oWord.UserAddress)
```

A bookmark is a way of naming a range or location. To create one, call the Bookmarks collection's Add method. The call to PrintOut tells Word to use whatever is at the EnvelopeAddress bookmark for the address, and to include the user's stored address (from the Tools-Options dialog's User Information page) as the return address. Additional parameters let you set the type and size of the envelope as well as the printing orientation. Omitting those parameters uses the current settings.

Search and Replace

The ability to find a text string and replace it with another throughout a document without retyping was one of the "killer" items that led to early acceptance of word processors. Word's version is extremely powerful – it includes the ability to search for text alone, for text in a particular format, for formatting only, for certain styles, and so forth.

Searching is handled by the Find object. It has an assortment of properties, such as Text, MatchCase and MatchWholeWord, that determine what it's searching for. Some properties, like Font and ParagraphFormat, are references to other objects. The Format property determines whether formatting is considered in the search. Set it to .F. to search for the string specified by Text, regardless of format.

The Execute method actually performs the search. It accepts a number of parameters, most of which duplicate properties of the Find object. Omitting them uses the current settings of the Find object.

Find is a member of Range and Selection, but not of Document. The search begins at the beginning of the range or selection, unless Forward is .F., in which case it starts at the end. The Wrap property determines what happens when it reaches the end (or beginning, when searching backward) of the range or selection.

To search the whole document, create a range at the beginning of the document. When Execute finds a match, the range or selection moves to cover only the matching item.

For example, to search for the first occurrence of the string "Visual FoxPro", regardless of case, in the current document:

```
oRange = oWord.ActiveDocument.Range(0,0) && Start of document
WITH oRange.Find
   .Text ="Visual FoxPro"
   .MatchCase = .F.
   .Format = .F.
   lFound = .Execute()
ENDWITH
```

Things get a little more complex when you want to replace the found text. There are two ways to go about it. The last two parameters of Find's Execute method let you specify a replacement string and the number of replacements to perform (none, one or all). So, in the code above, we can replace every instance of "Visual FoxPro" with "Visual FoxPro!" by replacing the Execute line with:

```
lFound = .Execute( , , , , , , , , , "Visual FoxPro!", wdReplaceAll )
```

wdReplaceAll is an constant indicating that all instances should be replaced.

The alternative approach uses a Replacement object referenced through the Find object's Replacement property. Like Find, Replacement has Text, Font, ParagraphFormat and Style properties, among others. You can fill in Replacement's properties to specify exactly what should replace the found item.

```
WITH oRange.Find
   * what to look for
   .Text ="Visual FoxPro"
   .MatchCase = .F.
   .Format = .F.

   * what to replace it with
   .Replacement.Text = "Visual FoxPro!"
   .Replacement.Font.Bold = .T.
   .Replacement.ParagraphFormat.LeftIndent = 12

   * go!
   lFound = .Execute( , , , , , , , , , , wdReplaceAll )
ENDWITH
```

It's also possible to search for and replace only formatting. To change every occurrence of 12-point Arial to 16-point Times New Roman, use code like this:

```
oRange = oWord.ActiveDocument.Range(0,0)
WITH oRange.Find
   * make sure to clean up from last search
   .ClearFormatting

   * what to look for
   .Text =""
   .Format = .T.
   .Font.Name = "Arial"
   .Font.Size = 12

   * what to replace it with
   WITH .Replacement
      .ClearFormatting
      .Text = ""
      .Font.Name = "Times New Roman"
      .Font.Size = 16
   ENDWITH

   * go!
   lFound = .Execute( , , , , , , , , , , wdReplaceAll )
ENDWITH
```

With a little creativity, it's possible to find and replace pretty much anything you want in a document. You can also combine VFP's data handling strength with VBA for power searching. Imagine putting a collection of search and replacement strings in a table, then using automation to make all the changes without intervention. (I don't actually have to imagine this one. I've done it.)

Tables

Word's tables seem like a natural fit for representing VFP data. A table can be formatted as a whole, but individual cells can be separately formatted, too. Borders of tables and cells can be visible or invisible, and can take on a range of sizes and styles. Both columns and rows can be individually sized.

The object hierarchy for tables is a little confusing. Each document has a Tables collection, which in turn contains individual Table objects. The Table object contains Rows and Columns collections, which contain Row and Column objects respectively. Those objects each have a Cells collection that references the individual cells in the row or column, each represented by a Cell object. While the Table object doesn't have a Cells collection, the individual Cell objects can be accessed using the Cell method, which accepts row and column number parameters. Here are several ways to refer to the cell in the third row and fourth column of the first table in the active document:

```
oWord.ActiveDocument.Tables[1].Rows[3].Cells[4]
oWord.ActiveDocument.Tables[1].Columns[4].Cells[3]
oWord.ActiveDocument.Tables[1].Cell[3,4]
```

Table, Row and Cell all have Range properties, so that an entire table, row or a cell can be easily converted to a range. This means that the same techniques work for inserting text into a table as for other parts of a document. However, a Range created from a cell contains a special end-of-cell marker. To access only the text in a cell, move the end of the range back one character. Either of the following does the trick:

```
oRange.End = oRange.End - 1
oRange.MoveEnd( wdCharacter, -1 )
```

The program below opens TasTrade's Order History view and creates a Word table showing the order history for the chosen customer. It demonstrates a variety of features, including borders, shading and auto-sizing of columns.

```
* Create a Word table with order information for one customer
* Assumes: Word is already open and accessible through oWord.
*          The table is to be added at the end of the current document.
*          Customer table is open and positioned on desired customer

#INCLUDE "..\Word.H"

LOCAL oRange, oTable, nRecCount, nTotalOrders

* Open the Order History view, which contains
* a summary of orders for one customer.
USE "Order History" ALIAS OrderHistory

* Find out how many records.
nRecCount = _TALLY

* Add a table at the end of the document.
* Give it two more rows than records to handle headings and totals.
oRange = oWord.ActiveDocument.Range()
oRange.MoveEnd( wdStory )
oTable = oWord.ActiveDocument.Tables.Add( oRange, nRecCount + 2, 4)

* Set up a font for the table
oRange.Font.Name = "Arial"
oRange.Font.Size = 12

WITH oTable
   * Set up borders and shading.
   * First, remove all borders
   .Borders.InsideLineStyle = .F.
   .Borders.OutsideLineStyle = .F.

   * Add a double line before the totals
   .Rows[nRecCount + 2].Borders[ wdBorderTop ].LineStyle = ;
      wdLineStyleDouble

   * Shade first row for headings
   .Rows[1].Shading.Texture = 100

   * Put heading text in and set alignment
   .Cell[1,1].Range.ParagraphFormat.Alignment = wdAlignParagraphRight
   .Cell[1,1].Range.InsertAfter("Order Number")


   .Cell[1,2].Range.ParagraphFormat.Alignment = wdAlignParagraphLeft
   .Cell[1,2].Range.InsertAfter("Date")

   .Cell[1,3].Range.ParagraphFormat.Alignment = wdAlignParagraphRight
   .Cell[1,3].Range.InsertAfter("Total")

   .Cell[1,4].Range.ParagraphFormat.Alignment = wdAlignParagraphCenter
   .Cell[1,4].Range.InsertAfter("Paid?")

   * Add data and format
   * Compute total along the way
   nTotalOrders = 0
   FOR nRow = 1 TO nRecCount
      WITH .Rows[nRow + 1]
         * Right align first column
         .Cells[1].Range.ParagraphFormat.Alignment = ;
            wdAlignParagraphRight
         .Cells[1].Range.InsertAfter( Order_Id )

         .Cells[2].Range.InsertAfter( TRANSFORM(Order_Date, "@D") )

         * Right align third column
         .Cells[3].Range.ParagraphFormat.Alignment = ;
            wdAlignParagraphRight
         .Cells[3].Range.InsertAfter( TRANSFORM(Ord_Total, ;
            "$$$$$$$$$9.99") )

         * Center fourth column
         .Cells[4].Range.ParagraphFormat.Alignment = ;
            wdAlignParagraphCenter
         * Put an X in fourth column, if paid; blank otherwise
         IF Paid
            .Cells[4].Range.InsertAfter("X")
         ENDIF
      ENDWITH

      nTotalOrders = nTotalOrders + Ord_Total
      SKIP
   ENDFOR
```

```
   * Put total row in
   WITH .Rows[ nRecCount + 2]
      .Cells[1].Range.InsertAfter("Total")
      .Cells[3].Range.ParagraphFormat.Alignment = wdAlignParagraphRight
      .Cells[3].Range.InsertAfter(TRANSFORM(nTotalOrders, ;
         "$$$$$$$$$9.99"))
   ENDWITH

   * Size columns. For simplicity, let Word
   * do the work.
   .Columns.Autofit
ENDWITH
```

The results are shown in Figure 5.

Order Number	Date	Total	Paid?
1045	03/17/95	$541.16	
999	02/25/95	$637.71	X
907	01/19/95	$432.41	X
871	12/29/94	$164.04	X
793	11/16/94	$403.64	X
612	06/18/94	$807.85	X
375	10/29/93	$439.99	X
128	12/09/92	$169.94	X
Total		$3596.75	

Figure 5 Using tables for data – A customer's order history looks good when poured into a Word table.

An alternate version of the code creates a two-row table, inserts the headings, then formats the cells in the second row. The loop then inserts the data and adds a new row. Each new row picks up the formatting of the previous one, so the formats only have to be applied once. Here's the key portion, which is inside a WITH oTable … ENDWITH pair.

```
   * Format data cells
   .Cell[2,1].Range.ParagraphFormat.Alignment = wdAlignParagraphRight
   .Cell[2,3].Range.ParagraphFormat.Alignment = wdAlignParagraphRight
   .Cell[2,4].Range.ParagraphFormat.Alignment = wdAlignParagraphCenter

   * Add data and format
   * Compute total along the way
   nTotalOrders = 0
   FOR nRow = 1 TO nRecCount
      WITH .Rows[nRow + 1]
         .Cells[1].Range.InsertAfter( Order_Id )
         .Cells[2].Range.InsertAfter( TRANSFORM(Order_Date, "@D") )
         .Cells[3].Range.InsertAfter( TRANSFORM(Ord_Total, ;
            "$$$$$$$$$9.99") )
         * Put an X in fourth column, if paid; blank otherwise
         IF Paid
            .Cells[4].Range.InsertAfter("X")
         ENDIF
      ENDWITH

      * Add a new row
      .Rows.Add()

      * Running Total
      nTotalOrders = nTotalOrders + Ord_Total
      SKIP
   ENDFOR
```

Combine either version of the code with the code that produced Figure 4, and you have a reasonably attractive order history report for a customer. Wrap that in a loop with a few more commands (such as InsertBreak to add page breaks) and you can produce order histories for all customers or a selected set.

A few more words

The discussion here barely touches the surface of what you can with automation to Word. With perseverance and creativity, you can put the full power of Word to work in your applications.

Do Something EXCELlent

Working with Excel through automation has a lot in common with automating Word – chalk one up for polymorphism. Unfortunately, there's also a lot not in common since the two servers have different abilities and purposes.

The fundamental object in Excel is a Workbook. The Excel application object has a collection of Workbooks and an ActiveWorkBook property. Each Workbook has two main collections, Worksheets and Charts, which represent the pages of the workbook and the graphs it contains, respectively. Workbook has ActiveSheet and ActiveChart properties containing references to the current worksheet and chart objects. Excel also provides a shortcut by offering ActiveSheet, ActiveChart, and ActiveCell properties at the Application level. So, you can reference the current cell of the current worksheet with oExcel.ActiveCell.

Like the Word Application object, Excel's Application object has properties and methods that relate to Excel as whole. A number, like StartupPath, Version and WindowState, are the same, while others like Calculation, which determines when calculations occur, are specific to Excel.

Again, as with Word, the Excel Visual Basic Help file contains a live diagram of the object model.

Managing Worksheets

Polymorphism means that you already know how to open and close workbooks. The Open method of the Workbooks collection opens an existing workbook, given the filename:

```
oWorkBook = oExcel.Workbooks.Open("d:\writing\confs\la99\sample.xls")
```

All the Excel examples here assume that you've already created an Excel server object, accessed by oExcel.

To create a new workbook, use the Add method. As in Word, you can specify a template to begin with a particular kind of worksheet.

Both Open and Add return a reference to the new workbook.

Continuing down the polymorphic path, the Save and SaveAs methods of the WorkBook object let you store a workbook, either back to its source or to a new file. SaveAs allows an alternate format to be specified, along with many other options.

The Close method closes an individual workbook or all open workbooks.

Accessing Parts of a Worksheet

The analogies to Word continue somewhat when it comes to working with parts of a worksheet. Like Word, there's a Range object. There's also a way to select part of a worksheet and operate on it, but that approach is frowned on even more in Excel than in Word.

Creating and working with Excel ranges is different than working with Word ranges, however. A worksheet is composed of cells, which have addresses in the form X9, where X is one or two letters indicating the column and 9 is one or more digits indicating the row. The top left cell in a worksheet is A1. The 29th cell in the 32nd column is AF29. To specify a range, you provide the addresses for the range boundaries. For example, to access the data in cell C5, use:

```
?oExcel.ActiveSheet.Range("C5").Value
```

To create a range containing a rectangular group of cells from D12 to F19, use:

```
oRange = oExcel.ActiveSheet.Range("D12:F19")
```

Another way to access cells is using the Rows, Columns and Cells properties of Worksheet and Range. These properties takes appropriate index values and return a range containing the specified cells. To display the value of all cells in the third row of a range, you can write:

```
FOR nColumn = 1 TO oRange.Columns.Count
   ?oRange.Cells[3, nColumn].Value
ENDFOR
```

To see the contents of all cells in a range, use:

```
FOR nRow = 1 TO oRange.Rows.Count
   FOR nColumn = 1 TO oRange.Columns.Count
      ?oRange.Cells[nRow, nColumn].Value
   ENDFOR
ENDFOR
```

Note that the indexes for Cells list the row, then the column (just as arrays in VFP do), but the addresses of cells (in the "X9" format) list the column first.

Ranges can be specified without hard-coding the cell addresses. To create a range relative to another range, use the Offset property. This example creates a range 20 rows down and 30 rows to the left of oRange. The new range has the same size and shape:

```
oRange2 = oRange.Offset(20, 30)
```

Range don't have to consist of a single rectangle. Multiple groups can be listed when creating a range:

```
oRangeMixed = oExcel.ActiveSheet.Range("F21:F30, H21:H30")
```

The Union method combines several ranges into one. Here, the two ranges oRange and oRange2 are consolidated into a single range referenced by oBigRange:

```
oBigRange = oExcel.Union(oRange, oRange2)
```

Traversing a range with a loop like the one above doesn't work for a range composed of non-contiguous cells. The Areas collection has an entry for each rectangular portion (called an *area*) of the range, so to traverse all the cells in a range, whether or not it's rectangular, you can use code like this:

```
FOR nArea = 1 TO oRange.Areas.Count
   FOR nRow = 1 TO oRange.Areas[nArea].Rows.Count
      FOR nColumn = 1 TO oRange.Areas[nArea].Columns.Count
         ?oRange.Areas[nArea].Cells[nRow, nColumn].Value
      ENDFOR
   ENDFOR
ENDFOR
```

Manipulating Worksheet Contents

As the previous examples indicate, you can access the contents of a cell through its Value property. But, when working with spreadsheets, it's not always the value of a cell we're interested in. To access the formula contained in a cell, use the Formula property:

```
?oExcel.ActiveSheet.Range("C22").Formula
```

If a cell contains only a value, Formula returns the value as a string, while Value returns numbers as numbers. If there's a real formula in the cell, it's returned in the format you'd use to enter it in Excel, beginning with "=".

You can set values and formulas by assigning them to the appropriate cells:

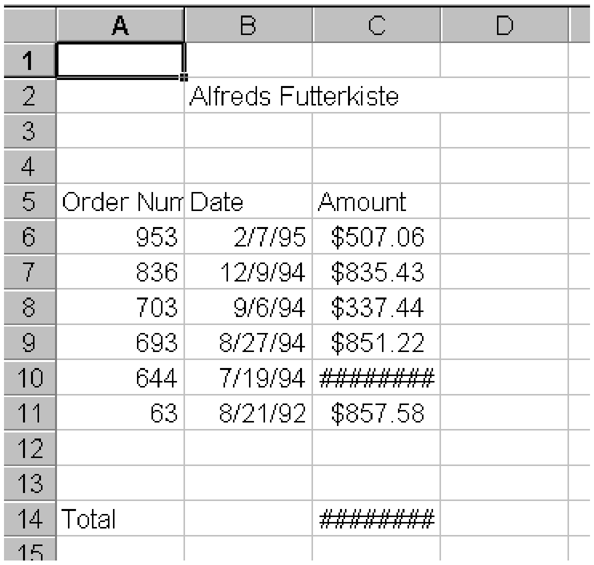
```
oExcel.ActiveSheet.Range("C13").Value = 100
oExcel.ActiveSheet.Range("C22").Formula = "=SUM(C5:C20)"
```

Returning to the TasTrade Order History view, we can send the information to Excel instead of Word so that the number-crunchers can work with it. While it's possible to copy VFP tables and views to XLS format using the COPY TO or EXPORT command, automation provides more flexibility in the process, including the ability to put data from more than one table into a worksheet.

```
* Put order information for one customer into an Excel worksheet.
* This results of this version are fairly unattractive.

* Assumes: Excel is already open and accessible through oExcel.
*          Customer table is open and positioned on desired customer

LOCAL oBook, oRange

* Open the Order History view, which contains
* a summary of orders for one customer.
IF NOT USED("OrderHistory")
   USE "Order History" AGAIN IN 0 ALIAS OrderHistory
ENDIF
SELECT OrderHistory

* Add a workbook, using default settings
oBook = oExcel.Workbooks.Add()

WITH oExcel.ActiveSheet
   * Put customer name at top
   .Range("B2").Value = Customer.Company_Name

   * Put column headings in Row 5
   .Range("A5").Value = "Order Number"
   .Range("B5").Value = "Date"
   .Range("C5").Value = "Amount"
```

```
   oRange = .Range("A6:C6")
ENDWITH

* Loop through orders and send data
SCAN
   WITH oRange
      .Columns[1].Value = Order_Id
      .Columns[2].Value = Order_Date
      .Columns[3].Value = Ord_Total
   ENDWITH

   * Move range down one row
   oRange = oRange.Offset(1,0)
ENDSCAN

* Now add total row
nLastRow = oRange.Row && Row property always give first row of range
                      && This range has only one row
nTotalRow = nLastRow + 2

WITH oExcel.ActiveSheet
   .Cells( nTotalRow, 1 ) = "Total"

   * Need to convert nLastRow to char to use in formula for sum
   oExcel.ActiveSheet.Cells( nTotalRow, 3 ).Formula = ;
      "=SUM( C6:C" + LTRIM(STR(nLastRow)) + " )"
ENDWITH

USE IN OrderHistory
```

Figure 6 shows the results. Clearly, some formatting is called for.



	A	B	C	D
1				
2		Alfreds Futterkiste		
3				
4				
5	Order Num	Date	Amount	
6	953	2/7/95	$507.06	
7	836	12/9/94	$835.43	
8	703	9/6/94	$337.44	
9	693	8/27/94	$851.22	
10	644	7/19/94	########	
11	63	8/21/92	$857.58	
12				
13				
14	Total		########	
15				

Figure 6 Creating a spreadsheet – It's easy to send both data and formulas from VFP to Excel and to extract them from Excel for use in VFP. Making it look good take a little more work.

Formatting Cells

Excel's Font object is much like Word's with properties for Name, Size, Bold, Italic and so forth. To set cell D32 to 14-point bold Times New Roman, use:

```
WITH oExcel.ActiveSheet.Range("D32").Font
   .Name = "Times New Roman"
   .Size = 14
   .Bold = .T.
ENDWITH
```

(It turns out that the two applications don't use the same font class, but for most purposes, they're indistinguishable.)

In line with its purpose, Excel offers more formatting options than Word does. The NumberFormat object lets you specify a format for the cell data, using a "picture" like VFP's InputMask. Although the property is called NumberFormat, it applies to dates and times as well, since Excel sees them as numbers. This code tells column B to use dd-MMM-yyyy format (like 01-Feb-1999):

```
oExcel.ActiveSheet.Columns(2).NumberFormat = "dd-MMM-yyyy"
```

A number of other properties, such as HorizontalAlignment, VerticalAlignment, WrapText, and Borders, control other aspects of a cell's appearance.

As in Word, it's better to work with Styles than to format individual cells with their properties. A few styles are built-in to handle common tasks, but to really make styles useful, you need to add a set of your own. This code adds a style that formats dates as above and right aligns them.

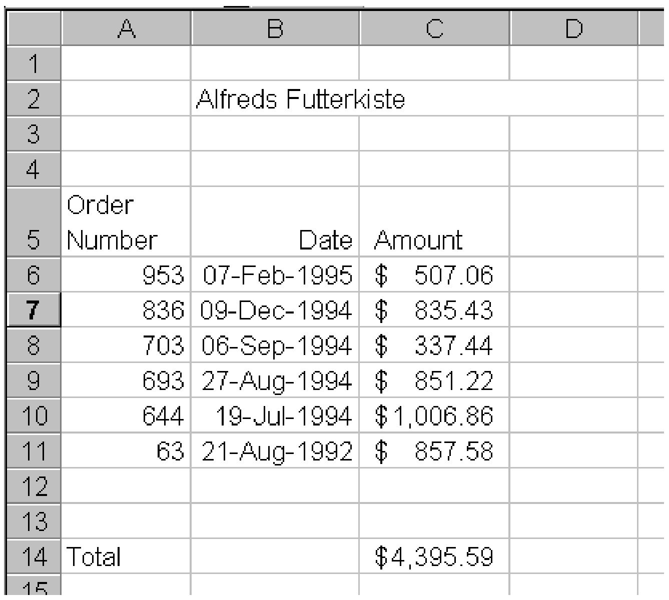
```
oStyle = oExcel.ActiveWorkbook.Styles.Add("DateDMY")
WITH oStyle
   .NumberFormat = "dd-MMM-yyyy"
   .HorizontalAlignment = xlHAlignRight
ENDWITH
```

Excel has its own set of constants like Word's. VFP #DEFINE's for Excel constants used in these examples can be found in Excel.H on the conference CD.

Here's an expanded version of the program above. This one handles formatting of the cells, as well. The results are shown in Figure 7.

```
* Put order information for one customer into an Excel worksheet.
* Includes formatting.

* Assumes: Excel is already open and accessible through oExcel.
*          Customer table is open and positioned on desired customer

#INCLUDE "Excel.H"

LOCAL oBook, oRange, lFound, oStyle
LOCAL nLastRow, nTotalRow

* Open the Order History view, which contains
* a summary of orders for one customer.
IF NOT USED("OrderHistory")
   USE "Order History" AGAIN IN 0 ALIAS OrderHistory
ENDIF
SELECT OrderHistory

* Add a workbook, using default settings
oBook = oExcel.Workbooks.Add()

WITH oExcel.ActiveSheet
   * Put customer name at top
   .Range("B2").Value = Customer.Company_Name
   * Put column headings in Row 5
   .Range("A5").Value = "Order Number"
   .Range("B5").Value = "Date"
   .Range("C5").Value = "Amount"

   oRange = .Range("A6:C6")
ENDWITH

* Loop through orders and send data
SCAN
   WITH oRange
      .Columns[1].Value = Order_Id
      .Columns[2].Value = Order_Date
      .Columns[3].Value = Ord_Total
   ENDWITH

   * Move range down one row
   oRange = oRange.Offset(1,0)
ENDSCAN

* Now add total row
nLastRow = oRange.Row && Row property always give first row of range
                      && This range has only one row
nTotalRow = nLastRow + 2

WITH oExcel.ActiveSheet
   .Cells( nTotalRow, 1 ) = "Total"

   * Need to convert nLastRow to char to use in formula for sum
   oExcel.ActiveSheet.Cells( nTotalRow, 3 ).Formula = ;
      "=SUM( C6:C" + LTRIM(STR(nLastRow)) + " )"
ENDWITH

* Format appropriately
WITH oExcel.ActiveSheet
   * Wrap heading for column 1
   .Range("A5").WrapText = .T.

   * Set up a style for dates in column 2
   * First make sure it doesn't already exist
   lFound = .F.
   FOR EACH oStyle IN oExcel.ActiveWorkbook.Styles
      IF UPPER(oStyle.Name) = "DateDMY"
         lFound = .T.
         EXIT
      ENDIF
   ENDFOR

   IF NOT lFound
      oStyle = oExcel.ActiveWorkbook.Styles.Add("DateDMY")
      WITH oStyle
         .NumberFormat = "dd-MMM-yyyy"
         .HorizontalAlignment = xlHAlignRight
      ENDWITH
   ENDIF

   .Columns(2).Style = "DateDMY"
   * Don't want this style for the company name
   * Use built-in normal style instead
   .Range("B2").Style = oExcel.ActiveWorkbook.Styles("Normal")

   * Set column 3 to built-in currency style
   .Columns(3).Style = "Currency"

ENDWITH
```

```
USE IN OrderHistory
```



	A	B	C	D
1				
2		Alfreds Futterkiste		
3				
4				
5	Order Number	Date	Amount	
6	953	07-Feb-1995	$ 507.06	
7	836	09-Dec-1994	$ 835.43	
8	703	06-Sep-1994	$ 337.44	
9	693	27-Aug-1994	$ 851.22	
10	644	19-Jul-1994	$1,006.86	
11	63	21-Aug-1992	$ 857.58	
12				
13				
14	Total		$4,395.59	

Figure 7 Formatted worksheet – With a little more effort, the sheet gets a lot easier to read.

Row and Column Sizes

In the example above, applying formatting changed column widths and row heights so that no manual control was needed. But sometimes you need to explicitly set the size of a row or column.

The RowHeight and Height properties control the depth of a row. For a single row, both indicate the height of that row in points.

For a range containing more than one row, they behave differently, though. Height indicates the total number of points for the whole range. RowHeight supplies the height of individual rows, but only if all are the same. RowHeight can be changed while Height is read-only. This line sets the height of each row in the range to 24 points:

```
oExcel.ActiveSheet.Range("A7:A42") = 24
```

ColumnWidth and Width, not surprisingly, control the width of columns. ColumnWidth measures in characters, using the font of the Normal style as its yardstick. Width measures in points.

Like RowHeight, ColumnWidth can't tell you about a group of columns of different widths. A line like:

```
?oExcel.ActiveSheet.Range("A1:C1").ColumnWidth
```

returns .NULL. if the columns have differing widths. Width returns the total width of the columns in the specified range. ColumnWidth can be set while Width is read-only.

Functions for everyone

One of Excel's strengths is the variety of calculations that are built-in. Spreadsheet designers can call on Excel to perform tasks like computing standard deviations, converting between degrees and radians, computing depreciation, and much more. There are literally dozens of functions available. Excel calls them *worksheet functions* and they're available through the WorksheetFunction property of the Application object.

Finding information about worksheet functions in Excel's VBA help is hard. The secret is that they're not actually documented there. To get a list of the functions and links to them, choose the topic "Using Microsoft Excel Worksheet Functions in Visual Basic" in the Contents pane. Alternatively, just look them up in the regular Excel help file.

The range of functionality is somewhat amazing. Here's a simple example. The CountIf function tells you how many cells in a range meet a specific condition. Returning to our Order History example, we can determine how many of the orders in the sheet are over $1000. We could put the function in a formula in the worksheet:

```
oExcel.ActiveSheet.Range("C16").Formula = ;
   '= CountIF( oExcel.ActiveSheet.Range("C6:C11"), ">1000")'
```

But the worksheet functions offer another possibility as well. We can call on them from VFP to get results without putting those results into a worksheet.

```
nOver1000 =  oExcel.WorksheetFunction.CountIF( ;
   oExcel.ActiveSheet.Range("C6:C11"), ;
   ">1000")
```

While this example could be done just as easily with native VFP code, many of Excel's functions offer calculations that would take considerable work in VFP. Consider creating an instance of Excel, dumping appropriate data into it and applying a function when you need complex statistical, financial, or mathematical functions.

Using Excel for graphing

Excel has a sophisticated engine for graphing worksheet data and, of course, it's available via automation. The Chart object is the key to adding graphs to a workbook.

Excel has two ways of managing Charts. A chart can have its own sheet (page) in a workbook or it can be embedded on another sheet. In the latter case, it's enclosed in a ChartObject object, which has a Chart property to provide access to the actual graph.

To add a Chart to a workbook, use the Add method of the Charts collection. Optional parameters let you indicate where in the workbook the new chart is added. Like other Add methods, this one returns a reference to the new object. To add a chart as a new sheet after the active worksheet, use:

```
oChart = oExcel.ActiveWorkbook.Charts.Add( , oExcel.ActiveSheet )
```

The ChartType property indicates the type of graph to create. Excel supports a wide variety from bar charts to scatter diagrams to pie graphs.

A lot of properties and methods control the appearance of a chart. The exact properties and methods used vary depending on the desired result. This example demonstrates a number of them. It creates two different charts based on the same data from TasTrade.

```
* Create graphs in Excel
* First, send data representing orders (count) by year
* Then, create a pie chart and a bar chart showing the data

* Assumes: Excel is open and referenced by oExcel
#INCLUDE "Excel.H"

LOCAL oBook, oSheet, nRow, nColumn
LOCAL cEndRow, oRange, oChart, oBarChart

* Generate data from Tastrade
OPEN DATA _SAMPLES+"TasTrade\Data\TasTrade"
SELECT YEAR(Order_Date) AS OrderYear, CNT(*) AS OrderCount;
   FROM Orders ;
   GROUP BY 1 ;
   INTO CURSOR OrdersByYear

* Create a workbook
oBook = oExcel.Workbooks.Add
oSheet = oBook.Sheets(1)

* Send the data
* Start in row 3 and column 1
nRow = 3
nColumn = 1

SCAN
   WITH oBook.ActiveSheet
      .Cells[ nRow, nColumn ].Value = OrderYear
      .Cells[ nRow, nColumn + 1].Value = OrderCount
   ENDWITH

   nRow = nRow + 1
ENDSCAN

* Now add a chart after the sheet
oChart = oBook.Charts.Add( , oBook.ActiveSheet )

* Set up this chart
WITH oChart
   * Name it
   .Name = "Pie"

   * Set the chart type
   .ChartType = xlPie

   * Specify the data

   * First, create a range containing the data to graph
   cEndRow = ALLTRIM(STR(nRow-1))
   oRange = oSheet.Range("B3:B" + cEndRow)

   * SetSourceData method specifies data to graph
   .SetSourceData( oRange )

   * Specify labels for the data ("category labels")
   .SeriesCollection(1).XValues = oSheet.Range("A3:A" + cEndRow)

   * Include a legend
   .HasLegend = .T.

   * Include a title and set it
   .HasTitle = .T.
   .ChartTitle.Text = "Number of Orders by Year"

   * Specify that data should be labelled with its percent of the whole
   .ApplyDataLabels ( xlDataLabelsShowPercent )

   * Format the overall graph area to have no border
   * and no background color
   WITH .PlotArea
      .Border.LineStyle = xlNone
      .Interior.ColorIndex = xlNone
   ENDWITH
ENDWITH

* Now create a bar chart (called a column chart by Excel)
* for the same data. Rather than starting over, use the Copy
* method to start with the pie chart.

oChart.Copy( , oChart )
* Get a reference to the new chart
oBarChart = oBook.Sheets(oChart.Index + 1)

WITH oBarChart

   * Name it
   .Name = "Bar"

   * Change the type to a vertical bar chart
   .ChartType = xlColumnClustered

   * No legend is needed in this case
   .HasLegend = .F.

   * Turn off tick marks on the X axis since
   * the graph shows discrete years
   .Axes( xlCategory ).MajorTickMark = xlTickMarkNone
ENDWITH
```

Figure 8 shows the pie chart, while figure 9 shows the bar chart.

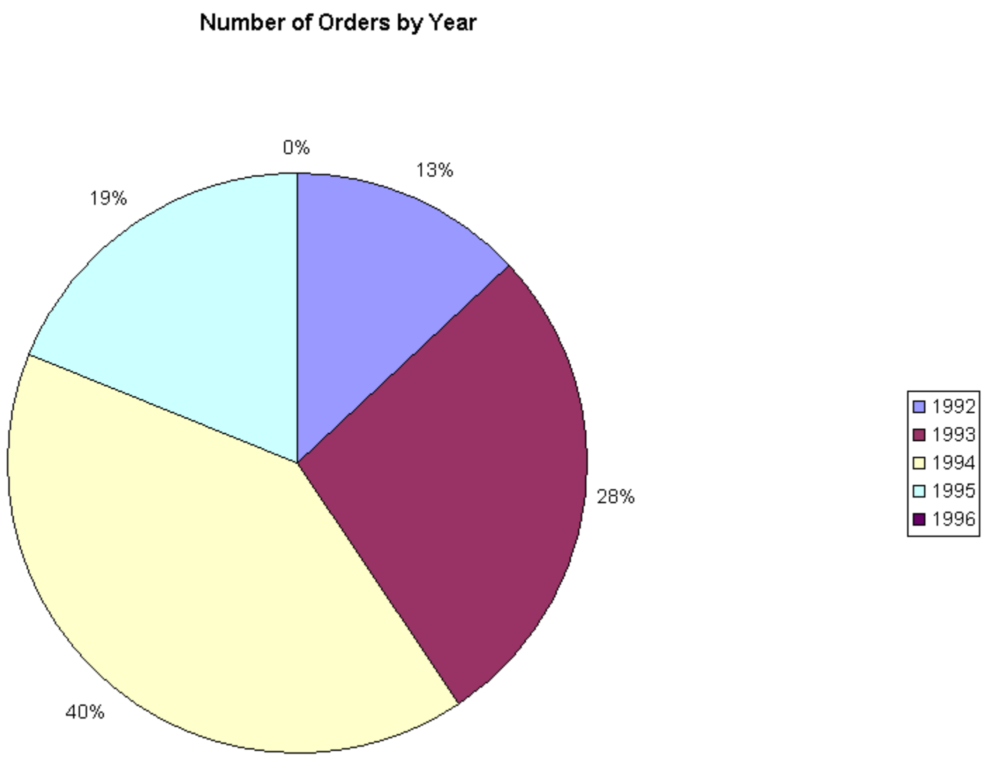


Figure 8 Baking a Pie Chart – Excel's graphing engine is open to automation.

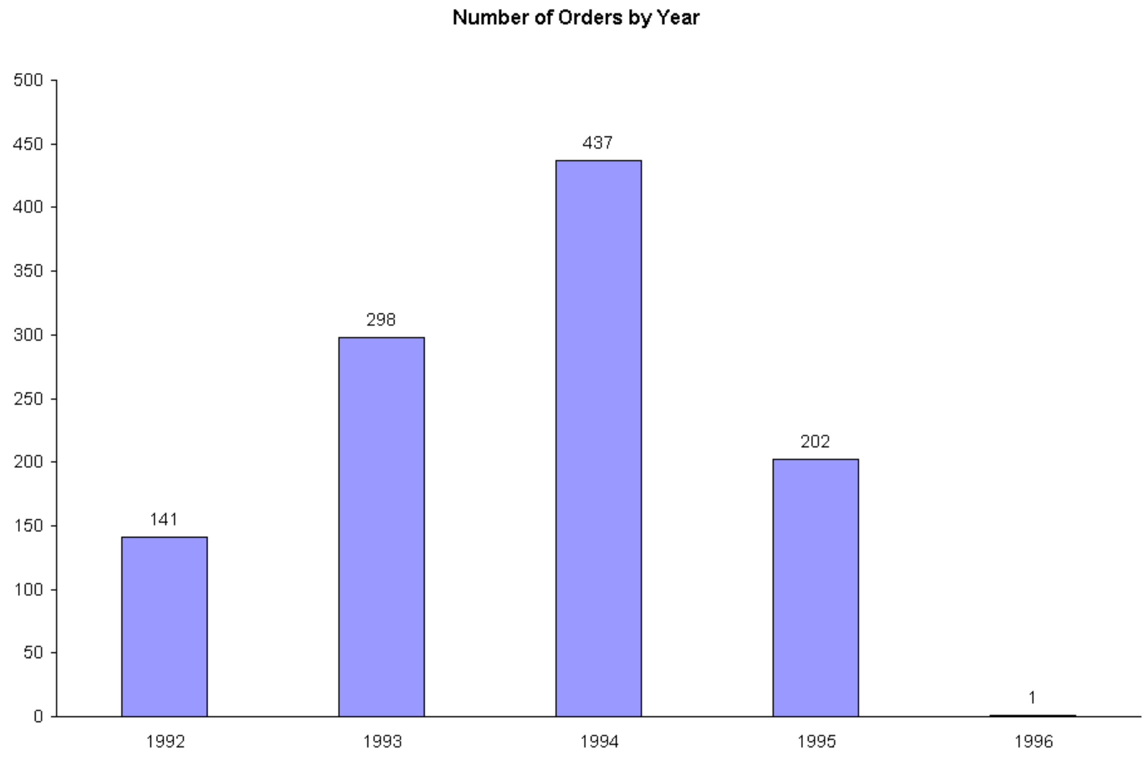


Figure 9 Bar Charts for Everyone – Only a few properties have to be changed to convert the pie chart in Figure 8 into a bar chart.

A few items in the code are worth noting. First, although there is a Charts collection that includes only charts, when a graph has a worksheet of its own, it's also included in the Sheets collection. Where this is really important is with the Chart's Index property. It indicates the position of the chart in the Sheets collection, not the Charts collection.

Second, unlike other methods that create objects, the Copy method doesn't return a reference to the new object. So, to get a reference to the new chart, the code combines the Index property and the Sheets collection.

Finally, there doesn't appear to be a rhyme or reason to which chart settings use properties and which use methods. If you don't find the setting you're looking for in the list of properties, check the methods.

Wrapping up Automation

The examples above demonstrate both the power and the frustrations of automation. You can do just about anything, but it requires you to know a lot about the server.

One way to simplify matters is to wrap the operations you use a lot into methods of a class you can call on as needed. The conference CD includes a class library called Automation that demonstrates some of the ways to wrap automation code.

The most error-prone and difficult step in the whole process is starting the automation server. The server might be missing or unregistered or Windows may be unable to start it for some reason. The cusOpenServer class handles these problems. Its OpenServer method opens the server, if possible, and returns a reference to it. If the server can't be opened, the method returns .F. The class can attempt to create the server on instantiation or wait until later.

The Init method checks whether the server is available. It uses the registry class that comes with VFP 6.

```
LPARAMETERS cServerName, lCreateAtInit

LOCAL oRegistry

IF VarType(cServerName) = "C"
   * Check to see if this is a valid server name
   oRegistry = NewObject("Registry",HOME()+"FFC\Registry")
   IF oRegistry.IsKey(cServerName)
      This.cServerName = cServerName
   ELSE
      RETURN .F.
   ENDIF
   RELEASE oRegistry
ELSE
   * No point in creating this object
   RETURN .F.
ENDIF

IF VarType(lCreateAtInit) = "L" AND lCreateAtInit
   This.OpenServer()
ENDIF

RETURN
```

OpenServer simply instantiates the object and saves the reference:

```
* Open the server specified by This.cServerName
* Store the new instance to This.oServer

IF VARTYPE(This.cServerName) = "C"
   * Should check here that the server name is valid on this machine
   This.oServer = CreateObject(This.cServerName)
ENDIF

RETURN
```

This code uses cusOpenServer to open a Word instance:

```
LOCAL oGetServ

oGetServ = NewObject("cusOpenServer", "Automation", "", ;
                     "Word.Application", .T.)
IF VarType(oGetServ) = "O"
   oWord = oGetServ.oServer
   IF VarType(oWord)<>"O"
      MessageBox("Can't create Word automation object")
   ENDIF
ENDIF
```

The next obvious candidates for wrapping are creating, opening, saving and closing documents. Other operations, such as moving data from a VFP cursor or table into a Word table or an Excel worksheet are good items to wrap, as well. The Automation class library includes classes that demonstrate some useful operations.

Once you start working with automation, the possibilities can be overwhelming. My own experiences include opening a series of worksheets to read specified data into a VFP table, converting a large number of Word documents from one template to another, performing a series of search-and-replace operations on a group of documents, cataloguing the figures in a group of documents, and a number of other tasks. (See www.advisor.com/wArticle.nsf/wArticles/FA9901.GRANT77 and www.advisor.com/wArticle.nsf/wArticles/FA9712w.GRANT62 for more about some of these adventures.)

Thanks to Word MVP Cindy Meister for reviewing the Word portion of these notes. Her recommendations improved them. Thanks also to Mac Rubel, off whom I bounced various ideas.