Session FT-

Automating Microsoft Word

Tamar E. Granor, Ph.D.

Visual FoxPro's Report Designer is fine in many cases, but it's not hard to come up with reports that go beyond its capabilities. Automation gives you the ability to use all of Microsoft Word as your report writer. This session shows you how to put Word to work for you. It starts with the basics of creating a Word automation object, and then explores the Word object model to create both simple and complex Word documents. Learn how to send VFP data to Word and format it.

This session assumes some familiarity with Word, an understanding of object-oriented programming, and knowledge of automation fundamentals.

Starting Word

To work with Word, you need an object reference to the Word automation server or to a specific object in Word. Two Visual FoxPro functions let you access Word. CreateObject() opens a new instance of Word and returns a reference to it, like this:

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

GetObject() is a little more complex. It takes either of two parameters, which reflect two approaches to providing a reference to Word. If you pass it the name of an existing document, it checks whether or not Word is already open. If not, it opens Word. Either way, it opens the specified document and returns a reference to the document:

```
oDocument = GetObject("d:\writing\confs\aut250.doc")
```

If you omit the first parameter to GetObject() and specify the Word automation server as the second, it looks for an open instance of Word and returns a reference to that instance:

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

If Word isn't open, an error message is generated.

Word's Object Model

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, as well as methods like Indent and Outdent.

The Word server object, called Application, has its own set of properties and methods, including a number of other collections. In addition to ActiveDocument, the Application object's properties include Visible, StartupPath, Version and WindowState. The Application object also has methods. The simplest is Quit, which shuts down the server. It has several optional parameters –

the first indicates what to do if any open documents have been changed and not saved. Other methods of the application object convert measurements from one set of units to another, check grammar and spelling, and much more.

Word Visual Basic Help (VBAWRD9.CHM for Word 2000, VBAWRD8.HLP for Word 97) 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 1 shows the portion of the object model diagram that describes the Document object.

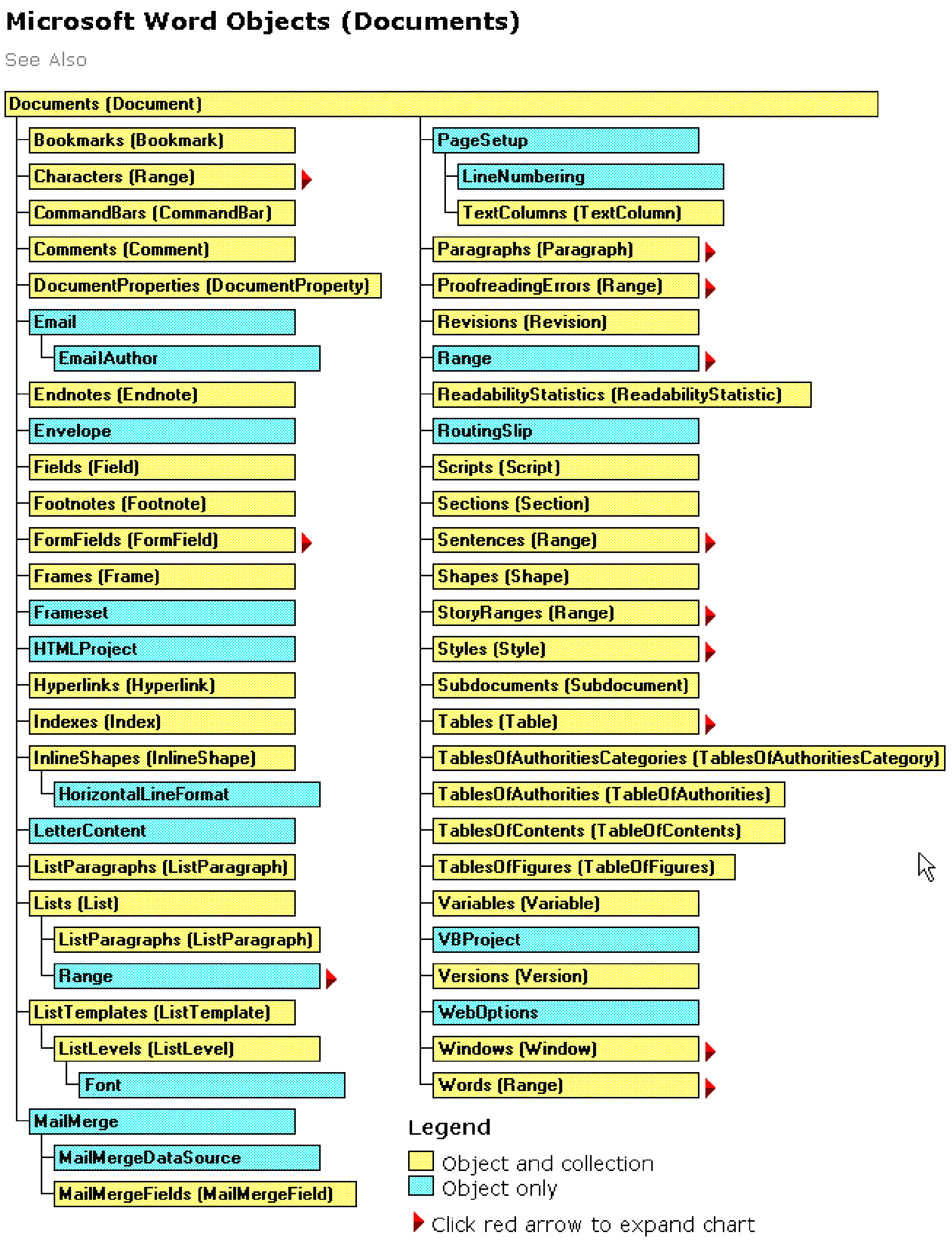


Figure 1. Word Object Model – The Word Visual Basic Help file offers a global view of Word's structure.

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's time to save or close it, a document reference is available.

To open an existing document, use the Open method of the Documents collection. 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\aut250.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. (Almost all the examples in this session assume 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. (The Normal template is the one used when you click on the New button on Word's Standard toolbar.)

Like Open, Add returns a reference to the newly created document. This line creates a new document based on a template called "OfficeFax".

```
oDocument = oWord.Documents.Add( ;
   "C:\WINNT\Profiles\Tamar\Application Data\Microsoft\OfficeFax.DOT")
```

As with the file name in Open, the full path to the template is needed.

The Document object has two methods for saving files. Save saves the document back to its current file; if it's never been saved, a Save As dialog box appears. The SaveAs method lets you specify the file name (and a lot of other stuff) without seeing a dialog, which is usually what you want in automation code.

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

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

To save the document to a different file or to save a document for the first time without displaying a dialog to the user, 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.)

You can check whether the document has been saved by testing its Saved property. If Saved is .T., the document is unchanged. This can be the case either because you've already saved the document and haven't changed it since or because it's a new document and it hasn't yet been modified.

The Name and FullName properties give you an alternative way to check whether a document has ever been saved. When you create a new document, Word assigns a name in the form "Document*n*" where *n* is a number. When you save the document, you can give it a more meaningful name, as well as specifying the file path. The Name property of the Document contains just the file stem with no path or extension. The FullName property contains the complete file name, including path and extension. However, before the file is saved for the first time, both Name and FullName contain the same string, the initial document name assigned by Word. You can use code like this to figure out whether to use Save or SaveAs:

```
WITH oWord.ActiveDocument
  IF .Name = .FullName
    * Prompt user to get a name,
    * then:
    .SaveAs( cFileName )
  ELSE
    .Save
  ENDIF
ENDWITH
```

To close an open document, call the document's Close method:

```
oDocument.Close()
```

or:

```
oWord.ActiveDocument.Close()
```

Accessing Parts of a Document

Most of what you want to do with Word involves adding to, modifying or reading a document, whether it's a new document you're building or an existing document you're modifying. 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 application object. 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. Ranges are not the same as the Selection area. You can define or modify Ranges without affecting the current Selection. You can even define multiple ranges for a document, whereas only one Selection object is available for each document. Ranges are very useful for repeatedly referencing specific portions of a document.

Ranges can be obtained in many ways. Many Word objects, like Sentence and Paragraph, have a Range property that contains an object reference to a Range object 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
```

The Document object has a Range method that lets you specify a range by character position. For example, to get a reference to a Range containing the 100th to 250th characters in the active document (probably not a particularly useful range), 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
```

Beware: for a large document, creating such a variable can take a significant amount of time.

It's easy to convert a Range object to a Selection, and vice-versa. Like many other objects, Selection has a Range property, which provides a Range object from the Selection. Similarly, the Range object 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 are in many ways, 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 bring document contents into FoxPro, 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 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:

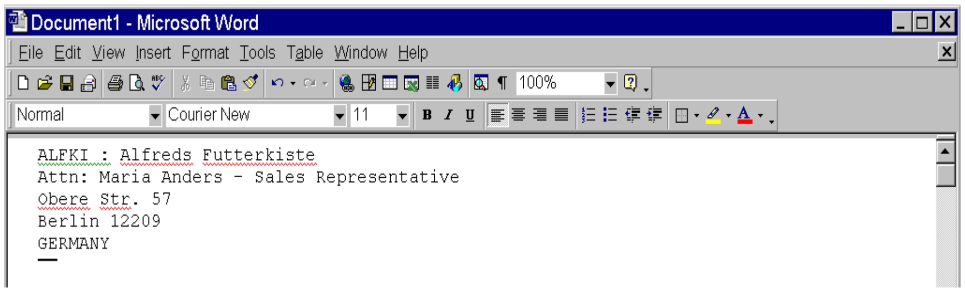
```
oRange.Text = UPPER(oRange.Text)
```

While you can send VFP data to Word by manipulating the Text property, that isn't really the best way to do it. There are a number of Word methods designed for adding text to a range. The method you'll use the most is InsertAfter – it adds the text you pass to the end of the range and enlarges the range to include the new text. (There's also a corresponding InsertBefore method that adds the text at the beginning of the range.) InsertParagraphAfter adds a paragraph mark at the end of the range and expands the range to include it.

The following code sends the data from a record to a new, blank document:

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

LOCAL oDocument, oRange
oDocument = oWord.Documents.Add()
oRange = oDocument.Range()

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

Figure 2 shows the resulting document.


Document1 - Microsoft Word
File Edit View Insert Format Tools Table Window Help
100%
Normal
Courier New
11

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

Figure 2. This return address can be sent to Word many different ways. In this case, with no special formatting involved, the fastest is to build the whole string in VFP, then send it to Word.

Several other methods help you navigate within the range or document, so that text can be added exactly where you want it. 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 reduced (or "collapsed," in Word VBA terms) to a single point before being moved. While collapsing a range or selection sounds dire, it's not. The text contained in the range/selection remains in the document—only the extent of the range or selection is changed. When moving forward, the range or selection is reduced to its end point, then moved; when moving backward, it's reduced to its beginning point before moving. You don't need to do anything special afterward. For the automation programmer, the key issue is to understand where in the range movement begins.

Constants from the wdUnits group are used for the units of movement. Table 1 shows the values for this group that can be passed to the Move method. (Keep in mind that Word's constants are not available in VFP. You have to #DEFINE them before using them.)

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

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.

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

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

The Move method provides another way to create the return address in Figure 2:

```
#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)
```

The Collapse method lets you explicitly 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. As with the Move method, Collapse doesn't remove content from the range. It changes the definition of the range to just a single point at what was previously either the end or beginning of the range.

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)
```

Finally, 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:

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

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

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) + CR

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

With VFP's speed at constructing strings, this version takes only one-third to one-quarter as long as the other approaches.

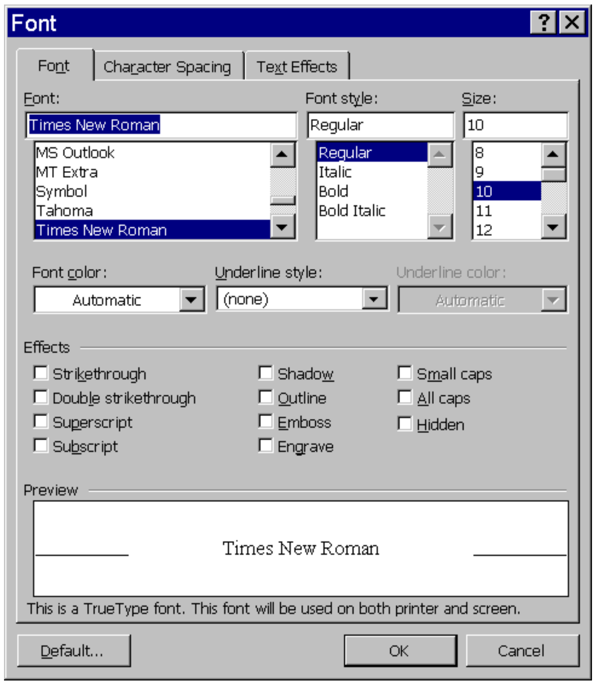
Formatting

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 available for formatting reflect the way Word structures its commands. 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 found 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 preformatted and user-defined styles available in the document. These four objects manage most of the frequently used settings. Other objects control other aspects of formatting.

Setting Fonts

Fonts in Word are controlled by the Font dialog on the Format menu (shown in Figure 3). That dialog controls the font name, size, style (such as bold, italic, underline, and so forth), color, and effects (like strikethrough, superscripts and subscripts, and much more). It also controls more esoteric options such as kerning, animation of or around text, the vertical position of text with respect to the baseline, spacing between characters, and more. The Font object manages these options.


Font

Font | Character Spacing | Text Effects

Font: Times New Roman
MS Outlook
MT Extra
Symbol
Tahoma
Times New Roman

Font style: Regular
Regular
Italic
Bold
Bold Italic

Size: 10
8
9
10
11
12

Font color: Automatic
Underline style: (none)
Underline color: Automatic

Effects
☐ Strikethrough
☐ Double strikethrough
☐ Superscript
☐ Subscript
☐ Shadow
☐ Outline
☐ Emboss
☐ Engrave
☐ Small caps
☐ All caps
☐ Hidden

Preview
Times New Roman

This is a TrueType font. This font will be used on both printer and screen.

Default... | OK | Cancel

Figure 3. Specifying Fonts. The Font dialog controls font, size, style, and color, as well as unusual options like kerning and spacing between characters. In Automation, all of these features are managed by the Font object.

Range, Selection and Style (discussed in *Working with Styles*) as well as many other objects each have a Font property that points to a Font object. Changing the properties of the Font object modifies the font of that portion of the document. For example, to change all the customer information in Figure 2 to 12-point Arial, you can use this code:

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

To simplify matters, just set the desired font before sending the text to the document. Add the two lines that set the font properties to the example program right after the line that sets oRange (as shown here) and the document will be created with 12-point Arial to begin with.

In fact, 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* below.

Table 2 lists Font properties you're likely to want to work with, along with Word constants for them, where appropriate.

Table 2. Font Properties. The Font object controls the appearance of the font from the font face to its size, style and much more. This table shows the more common properties. Check Help for more unusual settings.

Property	**Type**	**Description**
Name	Character	The name of the font.
Size	Numeric	The size of the font in points.
Bold	Numeric or Logical	Indicates whether the text is bold.
Italic	Numeric or Logical	Indicates whether the text is italic.

Underline	Numeric	The type of underline. wdUnderlineNone 0 wdUnderlineDouble 3 wdUnderlineSingle 1 wdUnderlineDotted 4 wdUnderlineWords 2 wdUnderlineThick 6
Superscript, Subscript	Numeric or Logical	Indicates whether the text is superscript or subscript.

It's possible for the text in a range (or whatever area the font object covers) to have more than one font setting. When that happens, the various numeric properties get the value wdUndefined (9999999). (That's also why properties that you'd expect to be logical are listed as numeric or logical.) Font.Name is the empty string in that situation. Although these 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:

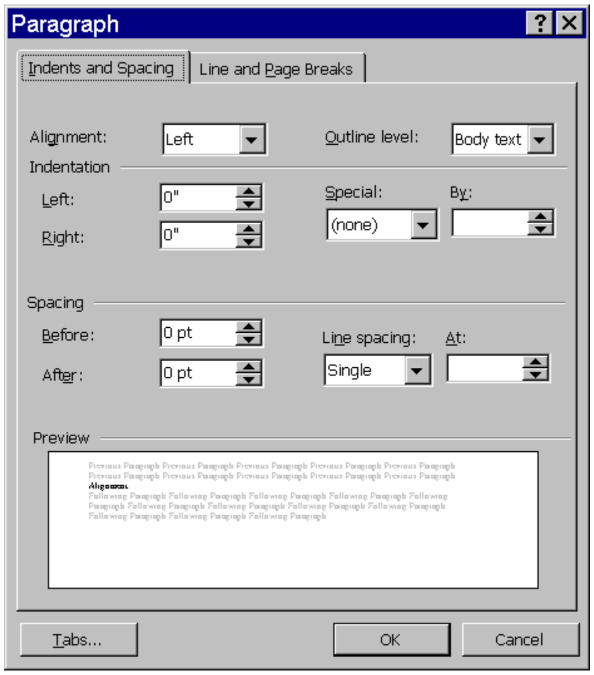
```
IF oFont.Bold
```

fails with the error "Operator/operand type mismatch". That's because of Bold's dual numeric/logical capabilities. When you assign logical values, Word translates them somewhere along the way, but for comparison, you have to use the numeric values.

Formatting paragraphs

Paragraphs are a key concept in Word. Much of Word's formatting can be thought of as being stored "in" the marker that follows each paragraph. That's why moving text sometimes changes its formatting. If you fail to take the paragraph marker along, the moved text picks up the formatting of the new location.

At the paragraph level, you can determine alignment of text (left, right, centered or full justification), various kinds of indentation (both amount and type), spacing of lines and paragraphs, handling of widows and orphans, and much more. Word allows regular indentation from both the left and right margins, as well as first line indents and hanging indents. Interactively, all of this is managed by the Paragraph dialog on the Format menu (shown in Figure 4). Behind the scenes, the ParagraphFormat object controls these settings.



Alignment: Left
Outline level: Body text
Left: 0"
Right: 0"
Special: (none)
By:
Before: 0 pt
After: 0 pt
Line spacing: Single
At:

Figure 4.The Paragraph Format dialog. This page lets you indicate alignment, indentation and spacing. The second page handles widow and orphan control and automatic hyphenation. The ParagraphFormat object controls these settings for automation.

Range, Selection and Style, among others, have a ParagraphFormat object, accessed through the same-named Property. The Paragraph object has a Format property that accesses a ParagraphFormat object.

Table 3 shows commonly used properties of the ParagraphFormat object and frequently used constant values for them. Like many of Word's objects, ParagraphFormat has only a few methods; none are likely to be useful in most automation code.

Table 3. ParagraphFormat Properties. Paragraphs are a key entity in a document. See Help for more esoteric settings.

Property	Type	Description
Alignment	Numeric	The alignment of text in the paragraph. wdAlignParagraphLeft 0 wdAlignParagraphRight 2 wdAlignParagraphCenter 1 wdAlignParagraphJustify 3
LeftIndent	Numeric	The indentation of the left edge of this paragraph from the left margin in points.
RightIndent	Numeric	The indentation of the right edge of this paragraph from the right margin in points.
FirstLineIndent	Numeric	The indentation of the first line of the paragraph. This property determines whether the paragraph has the first line indented or "outdented" (providing a hanging indent). Set a positive value to indent the first line, 0 to keep the first line flush with the rest of the paragraph or a negative value for a hanging indent. Note that, with a hanging indent, the first line doesn't move to the left; subsequent lines move to the right.
SpaceBefore, SpaceAfter	Numeric	The amount of white space (known as *leading)* before and after the paragraph, in points.
LineSpacingRule	Numeric	The kind of line spacing in effect. This setting can entirely determine the line spacing or it can set the stage for LineSpacing. wdLineSpaceSingle 0 wdLineSpaceAtLeast 3 wdLineSpaceDouble 2 wdLineSpace1pt5 1 wdLineSpaceExactly 4 wdLineSpaceMultiple 5
LineSpacing	Numeric	When LineSpacingRule is wdLineSpaceAtLeast, wdLineSpaceExactly, or wdLineSpaceMultiple, the actual line spacing in points.
WidowControl	Numeric or Logical	Indicates whether the first and last lines of the paragraph are kept on the same page as the rest of the paragraph.
KeepTogether	Numeric or Logical	Indicates whether the entire paragraph is kept on a single page.
KeepWithNext	Numeric or Logical	Indicates whether the paragraph is kept on the same page with the paragraph that follows it.
Hyphenation	Numeric or Logical	Indicates whether the paragraph is hyphenated automatically

This example sets the first paragraph in the range oRange to have a 0.5" first line indent, widow and orphan control, double spacing, and full justification:

```
#DEFINE wdAlignParagraphJustify 3
#DEFINE wdLineSpaceDouble 2

WITH oRange.Paragraphs[1].Format
  .FirstLineIndent = oWord.InchesToPoints( .5 )
  .WidowControl = .T.
  .Alignment = wdAlignParagraphJustify
  .LineSpacingRule = wdLineSpaceDouble
```

```
ENDWITH
```

This example triple spaces a range:

```
#DEFINE wdLineSpaceMultiple 5

WITH oRange.ParagraphFormat
  .LineSpacingRule = wdLineSpaceMultiple
  .LineSpacing = oWord.LinesToPoints( 3 )
ENDWITH
```

Working with Styles

While it's appropriate to manually 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, which are named formats that you can apply to a portion of a document. When you're working interactively in Word, you can see the style for the insertion point or currently highlighted text 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. In the Style dialog available from Word’s Format menu (Figure 5), Paragraph styles are preceded with a paragraph marker while character styles begin with an underlined "a".

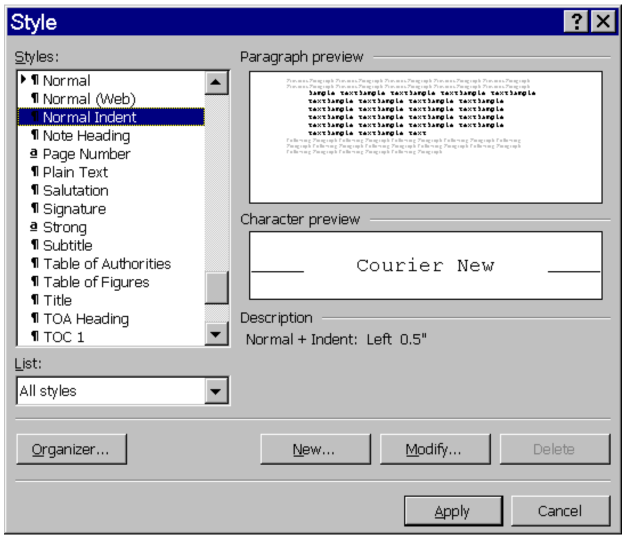


Figure 5. Creating and choosing styles. Word's Style dialog lists the available styles. Paragraph styles are preceded by a paragraph symbol, while an underlined "a" precedes Character styles. Note in the Description that the style is described in terms of another – its "base style".

Using styles is much like using classes in an object-oriented language. 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 sets the default style for the paragraph to follow.

The Document object includes a Styles collection, containing one Style object for each of the styles stored in the document. You can add your own styles using the Style collection’s Add method. Various objects' Style properties point to Style objects.

What 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 documents as needed. For example, this code modifies the Normal style, which is always available, to use 16-point centered Garamond Italic:

```
#DEFINE wdStyleNormal -1
#DEFINE wdAlignParagraphCenter 1

WITH oWord.ActiveDocument.Styles[ wdStyleNormal ]
  WITH .Font
    .Name = "Garamond"
    .Size = 16
    .Italic = .T.
  ENDWITH
  .ParagraphFormat.Alignment = wdAlignParagraphCenter
ENDWITH
```

To apply an existing style to a portion of a document, set the Style property for the Range or Paragraph to a built-in style using a constant, or to the name of a custom style. Table 4 lists the constants for some of the more commonly used built-in styles. This example applies the Heading 1 style to the range referenced by oRange:

```
#DEFINE wdStyleHeading1 -2
oRange.Style = oWord.ActiveDocument.Styles[ wdStyleHeading1 ]
```

Table 4. Built-in Styles. Word has over 100 built-in styles, each referenced by a defined constant. The table shows just a few of the most common. Use the Object Browser to find the rest.

Constant	Value	Constant	Value
wdStyleNormal	-1	wdStyleHeading1	-2
wdStyleBodyText	-67	wdStyleHeading2	-3
wdStyleDefaultParagraphFont	-66	wdStyleHeading3	-4

Creating Custom Styles

In addition to modifying the built-in styles, you can create your own custom styles. To add a new style, use the Add method of the Styles collection. Add takes two parameters, the name of the new style, and the Word constant indicating whether it’s a paragraph style (wdStyleTypeParagraph = 1) or character style (wdStyleTypeCharacter = 2).

Every style is based on an existing style. By default, new paragraph styles are based on the Normal style and new character styles are based on the Default Character Font style. The BaseStyle property indicates which style another style inherits from, however.

Whatever style BaseStyle points to, all other changes to the style's properties use the BaseStyle as their point of reference. In the Style dialog (Figure 5), the style's characteristics are described as "<The base style>"+"<various other characteristics>". The Description property contains the same information. So, much like classes in OOP, changes to the base style change styles based on it.

Table 5 lists key properties of Style, along with significant constant values.

Table 5. Style Counts. Styles are Word's version of OOP. They offer a way to provide uniform formatting within and across documents.

Property	Type	Description
BaseStyle	Character,	The name, constant value or pointer to the style on which this

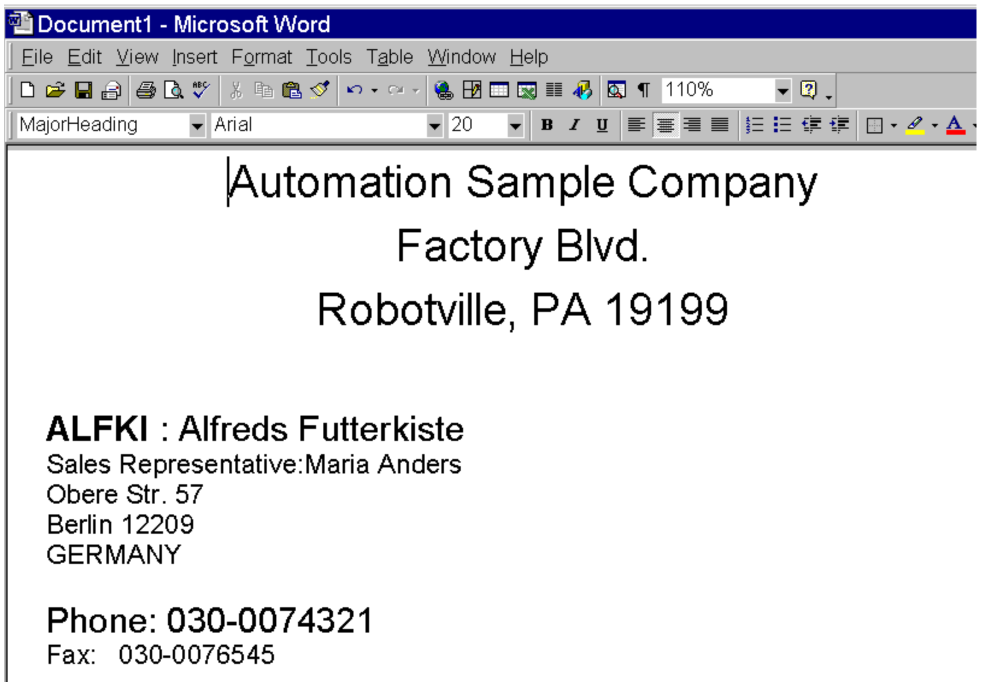
	Numeric or Object	style is based. See Table 4 and Help or the Object Browser for constant values for built-in styles.
Type	Numeric	Indicates whether this is a paragraph or character style. wdStyleTypeParagraph 1 wdStyleTypeCharacter 2
Builtin	Logical	Indicates whether this is a built-in style.
Description	Character	The description of the style (as shown in the Style dialog).
Font	Object	Pointer to a Font object for the style.
ParagraphFormat	Object	Pointer to a ParagraphFormat object for the style.
Borders	Object	Pointer to a Borders collection for the style.
Shading	Object	Pointer to a Shading object for the style.
NextParagraphStyle	Character, Numeric or Object	The name, constant value or pointer to the style for the paragraph to follow this paragraph, for paragraph styles.

This example takes the simple customer address document in Figure 2 and begins to create a document worthy of Word. It creates several new styles to do the job. In practice, you could use the built-in Normal and Heading X (there are multiple heading levels) styles for this document, redefining them as needed. But the example shows how easy it is to create new styles. (You'll find this program as Styles.Prg in the materials for this session.)

```
* Styles.PRG
* © 2000, Tamar E. Granor and Della Martin
* From:  Microsoft Office Automation with Visual FoxPro
* Hentzenwerke Publishing. www.hentzenwerke.com

* 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.

#DEFINE CR CHR(13)
#DEFINE wdStyleTypeParagraph 1
#DEFINE wdStyleNormal -1
#DEFINE wdAlignParagraphLeft 0
#DEFINE wdAlignParagraphCenter 1
#DEFINE wdCollapseEnd 0

USE _Samples + "TasTrade\Data\Customer"

LOCAL oWord, oDocument, oRange
LOCAL oBodyStyle, oMajorHeadingStyle, oMinorHeadingStyle

oWord = CreateObject("Word.Application")
oWord.Visible = .T.
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
   * This line is overkill since it's the default
   .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.InsertParagraphAfter( )
oRange.InsertParagraphAfter( )

* 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.InsertParagraphAfter( )

* 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. Figure 6 shows the resulting document in Word.


Document1 - Microsoft Word
File Edit View Insert Format Tools Table Window Help
110%
MajorHeading
Arial
20

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 6. Using Styles. Rather than formatting every item independently, styles let you define and name sets of formatting characteristics, then apply them uniformly within and across documents. Styles can be considered OOP for formatting.

It's worth noting that the code in the example is divided almost exactly evenly between formatting the styles and putting the data in the document. But far more lines would have been needed to perform the same formatting without using styles. Furthermore, consider what would be needed to add more information for the same customer. Once the styles are defined, they can be used over and over. With 42 lines of code, we have three styles that can be applied wherever we need them, no matter how many more lines of code we write to send text to the document.

Output

Producing attractive documents is a good start, but users usually want output of some sort, too. Word has a number of methods for producing output. In the interactive product, they're collected on the File menu in the Print Preview and Print menu options.

Printing

The PrintOut method of the Document object automates printing. It accepts a huge number of parameters. 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. The key parameters for most situations are shown here. (Incredibly, there are still eight more parameters after these.)

```
oDocument.PrintOut( lBackground, , nPrintRangeType, cFileName, cFrom, cTo, ,
                    cCopies, cPageRange, , lPrintToFile)
```

lBackground	Logical	Indicates whether or not to print in the background, continuing with other code while printing.
nPrintRangeType	Numeric	Indicates which part of the document to print. See below.
cFileName	Character	If printing to file, the filename, including path, of the destination file.
cFrom, cTo	Character	If nPrintRangeType is wdPrintFromTo, the beginning and end of the print range. See below.
cCopies	Character	The number of copies to print.
cPageRange	Character	If nPrintRangeType is wdPrintRangeOfPages, the print range. See below.
lPrintToFile.	Logical	Indicates whether to print to a file.

The syntax reflects two different approaches to specifying the pages to be printed. The nPrintRangeType parameter determines which, if either, is used. The valid constant values are wdPrintAllDocument (0), wdPrintSelection (1), wdPrintCurrentPage (2), wdPrintFromTo (3), and wdPrintRangeOfPages (4). When wdPrintFromTo is passed, pass values for cFrom and cTo as the fifth and sixth parameters, respectively. Note that, for reasons known only to Microsoft, they're passed as characters. For example, to print three copies of pages 4 to 10 of a document referenced by oDocument, use:

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

The wdPrintRangeOfPages choice lets you specify a single string and has more flexibility. In that case, pass a value for cPageRange as the eighth parameter - it can include multiple comma-separated values; each may include a range. For example, "3, 4-6, 12".

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

```
oDocument.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. Omitting that parameter in a subsequent call to PrintOut defaults to .T.; you have to explicitly pass .F. to print to the printer.

Print Preview

Perhaps all your users want is to see how the document will look when printed. That's easy. Just call the Document object’s PrintPreview method. That switches Word to PrintPreview mode.

Of course, that's only useful if Word is visible. If Word is hidden, it doesn't do any good. Making Word visible is as easy as setting the Application’s Visible property to .T. Better yet, if you've been doing everything in the background and now you're ready to show the user what you've been up to, call PrintPreview, make Word visible, then call Word's Activate method. That'll bring Word to the front. Try it like this:

```
oDocument.PrintPreview  && Get the document ready for the user to see
oWord.Visible = .t.     && Show word
oWord.Activate()        && Bring it to the front
```

Regardless of whether you're keeping Word hidden most of the time or showing it all along, when you're done with Print Preview, you turn it off by calling the ClosePrintPreview method of the Document object.

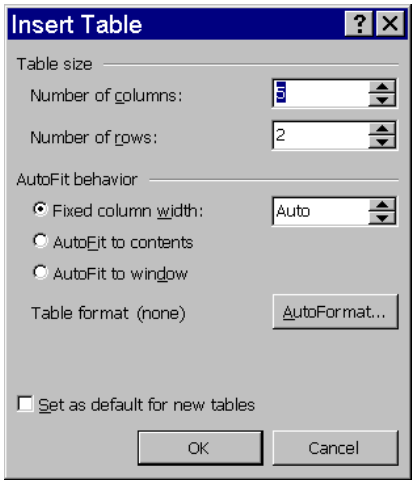
Using Word's Tables

Word's tables seem like a natural fit for representing Visual FoxPro data. A row can represent a record, with each column representing a field. What Word buys you is the ability to format the data and the table itself in sophisticated ways, well beyond the capabilities of FoxPro's Report Designer, as well as letting users manipulate the results or produce output in alternative formats. A table can be formatted as a whole, or individual cells can be separately formatted. Borders of tables and cells can be visible or invisible, and can take on a range of sizes and styles. (The details of borders and shading are beyond the scope of this session – see Word VBA Help for information on the Border and Shading objects.) 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 both 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 Table's 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]
```

Interactively, you can add a table by choosing Table | Insert | Table from the menu, which brings up the dialog in Figure 7.



Number of columns: 5
Number of rows: 2
Fixed column width: Auto
Table format (none)

Figure 7. Adding a table. To add a table interactively, you specify the number of columns and rows and, if you want, whether the columns should resize automatically to fit their contents. You can also specify a predefined format for the table.

With Automation, since Tables is a collection, it's not surprising that the way to add a table to a document is to call the Add method of that collection. It takes several parameters:

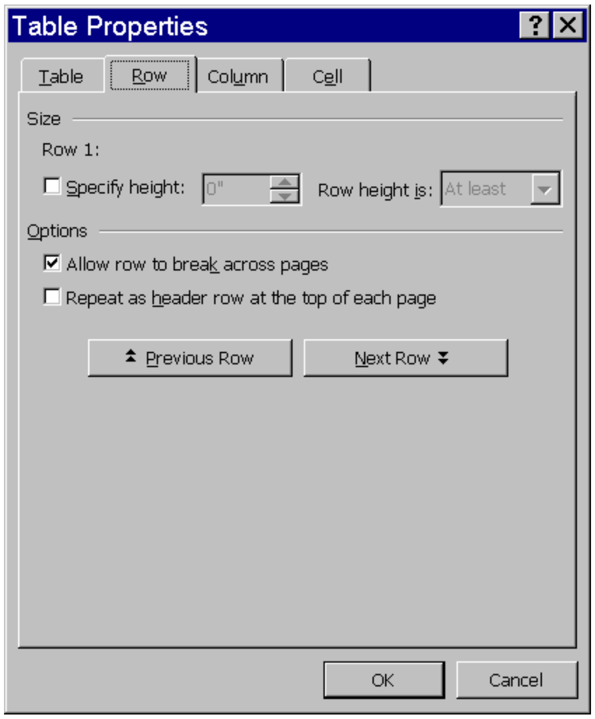
```
oDocument.Tables.Add( oRange, nRows, nColumns, nTableBehavior, nAutoFit )
```

oRange	Object	Reference to a range indicating where to insert the new table.
nRows	Numeric	The number of rows in the new table.
nColumns	Numeric	The number of columns in the new table.
nTableBehavior	Numeric	A constant indicating whether the table automatically resizes to fit its contents. wdWord8TableBehavior 0 Don't resize automatically. wdWord9TableBehavior 1 Resize automatically.
nAutoFit	Numeric	If nTableBehavior is wdWord9TableBehavior, indicates which algorithm is used to resize the cells. wdAutoFitFixed 0 Fixed column width wdAutoFitContent 1 Size cells to content wdAutoFitWindow 2 Size table to full width of window

The last two parameters are optional. If you omit them, you get the older (Word 97) behavior of a table that doesn't resize as you fill it. However, you can change that behavior. The AllowAutoFit property and AutoFitBehavior method control this resizing capability.

Even if you're not allowing automatic resizing of columns, the AutoFit method of the Column object lets you resize individual columns based on their content. Rather than having cells change size as data is entered, you apply the changes once you've put something into the table. Column's SetWidth method lets you set a column to a specific width, in points.

Once you've added a table interactively, the Table Properties dialog on the Table menu allows you to adjust various characteristics of the table as a whole and of the individual rows, columns and cells. Figure 8 shows the Row page of that dialog. With Automation, you use various properties of the Table, Row, Column and Cell objects and the Rows, Columns and Cells collections to make these adjustments.



Size
Row 1:
☐ Specify height: 0"   Row height is: At least
Options
☑ Allow row to break across pages
☐ Repeat as header row at the top of each page

Figure 8. Formatting a table. The Table Properties dialog lets you resize rows and columns, as well as specify other characteristics of tables and their components.

Table 6 shows the most commonly used properties of Table.

Table 6. Defining Tables. These properties of the Table object are the ones you're most likely to work with.

Property	Type	Description
Rows	Object	Pointer to the Rows collection for the table.
Columns	Object	Pointer to the Columns collection for the table.
Uniform	Logical	Indicates whether every row has the same number of columns. Read-only.
Borders	Object	Pointer to the Borders collection for the table.
Shading	Object	Pointer to the Shading object for the table.
AllowAutoFit	Logical	Indicates whether columns are automatically resized as data is added to the table. Corresponds to the nTableBehavior parameter of the Tables.Add method.
AllowPageBreaks	Logical	Indicates whether the table can be split over multiple pages in the document.
AutoFormatType	Numeric	A constant indicating which, if any, of a set of predefined formats has been applied to the table. AutoFormats are applied with the AutoFormat method. Here are a few: wdTableFormatNone 0 wdTableFormatSimple1 1 wdTableFormatClassic1 4 wdTableFormatColorful1 8 wdTableFormatContemporary 35 wdTableFormatElegant 36 wdTableFormatGrid1 16
Spacing	Numeric	Indicates the space between cells, in points. This is space that actually separates the cells, not the boundary between the cell border and the text.

Row and Column, not surprisingly, have a number of properties in common, including Cells to point to a Cells collection and Shading to reference a Shading object. Row also has a Borders property that references a Borders collection, though Column does not. Both objects have logical IsFirst and IsLast properties that, as their names suggest, indicate whether the particular row or column is the first or last in the collection.

At this point, the two objects part company, though there are still similarities. The size of a Row is determined by HeightRule and Height, as indicated in Table 7. Column width also uses two properties, PreferredWidth and PreferredWidthType, shown in Table 8.

Row has one other size-related property, SpaceBetweenColumns. It indicates the distance between the cell boundaries and the text. The value of the property is half what you set in Word itself because that one is measured from the text in one cell to the text in the next cell.

Row's AllowBreakAcrossRows property determines what happens when the contents of a row don't fit on the current page. If it's .T., the row can be split over two pages; if .F., a page break occurs before the row.

When a table is split over multiple pages, rows whose HeadingFormat property is set to .T. are repeated.

One big difference between Row and Column is that a Row can be a Range while a Column cannot.

Table 7. Determining Row Size. Two Row properties combine to let you indicate the height of the row.

Property	Type	Description
HeightRule	Numeric	Indicates the logic used to determine the height of this row. Use one of the following constants: wdRowHeightAuto 0 wdRowHeightAtLeast 1 wdRowHeightExactly 2
Height	Numeric	The height for the row, if HeightRule is wdRowHeightExactly. The minimum height for the row, if HeightRule is wdRowHeightAtLeast. Ignored (and uninformative when queried) if HeightRule is wdRowHeightAuto; in that case, the row height is based on the row's contents.

Table 8. Specifying Column Width. These two properties combine to determine the width of a column.

Property	Type	Description
PreferredWidth	Numeric	Desired width for this column, either in points or as a percentage of the overall window width. Interpretation is determined by PreferredWidthType.
PreferredWidthType	Numeric	Indicates whether PreferredWidth is measured in points or percent, or is ignored. wdPreferredWidthAuto 0 Size column by contents. wdPreferredWidthPoints 1 Size column in points wdPreferredWidthPercent 2 Size column as percent of total window.

Cell shares a number of properties of Table, Row and Column, including Borders, Shading, HeightRule, Height, PreferredWidth and PreferredWidthType. Table 9 shows some other properties unique to Cell.

Table 9. Cell holdings. At the bottom of the table hierarchy, cells have quite a few properties. Here are some you're likely to deal with.

Property	Type	Description
Width	Numeric	The width of the cell in points.
WordWrap	Logical	Indicates whether the text is wrapped into multiple lines and the cell height is increased to fit the entire contents.
FitText	Logical	Indicates whether the display size of the text (but not the actual font size) is reduced in order to make the entire contents of the cell fit onto a single line.
VerticalAlignment	Numeric	Indicates the vertical position of the text in the cell. WdAlignVerticalTop 0 WdAlignVerticalCenter 1 WdAlignVerticalBottom 3

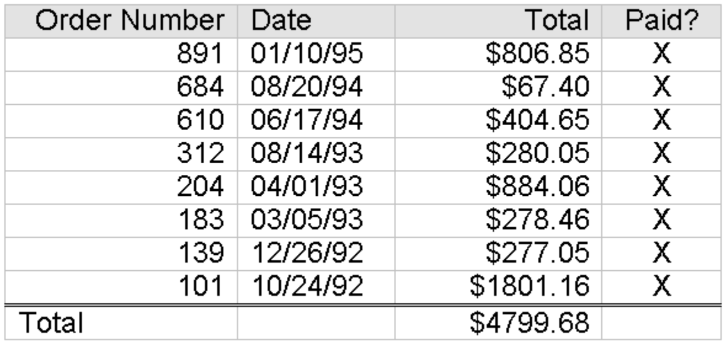
Table, Row and Cell all have Range properties, so that an entire table, a row or a cell can be easily converted to a range. This means that the same techniques work for inserting text into a table as for other parts of a document. However, a Range created from a cell contains a special end-of-cell marker. 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 in Listing 1 opens TasTrade's Order History view and creates a Word table showing the order history for the current customer. It demonstrates a variety of features, including borders, shading and auto-sizing of columns. It's OrderTblFormat.PRG in the materials for this session. The results are shown in Figure 9.

Listing 1. Creating a table. This program generates a table containing a customer's order history.

```
* OrderTblFormat.PRG
* © 2000, Tamar E. Granor and Della Martin
* From:  Microsoft Office Automation with Visual FoxPro
* Hentzenwerke Publishing. www.hentzenwerke.com

* Create a Word table with order information for one customer
* Set up the table with two rows, formatting the second row for
* the data. Then add rows as needed for each record.

#DEFINE wdStory                    6
#DEFINE wdCollapseEnd              0
#DEFINE CR                         CHR(13)
#DEFINE wdBorderTop                -1
#DEFINE wdLineStyleDouble          7
#DEFINE wdAlignParagraphLeft       0
#DEFINE wdAlignParagraphCenter     1
#DEFINE wdAlignParagraphRight      2

LOCAL oRange, oTable, nRecCount, nTotalOrders
LOCAL nRow

LOCAL oWord
oWord = CreateObject("Word.Application")
oWord.Documents.Add()

OPEN DATABASE (_SAMPLES + "\Tastrade\Data\Tastrade")
USE CUSTOMER
GO INT(RECCOUNT() * RAND(-1)) + 1 && pick a customer at random

* Open the Order History view, which contains
* a summary of orders for one customer.
SELECT 0
USE "Order History" ALIAS OrderHistory

* Find out how many records.
nRecCount = _TALLY

oRange = oWord.ActiveDocument.Range()

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

* Move to the end of the document
* Leave two empty lines
oRange.MoveEnd( wdStory )
oRange.Collapse( wdCollapseEnd )
oRange.InsertAfter( CR + CR )
oRange.Collapse( wdCollapseEnd )

* Add a table with two rows
oTable = oWord.ActiveDocument.Tables.Add( oRange, 2, 4)

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

   * 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?")

   * 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

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

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

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



Order Number	Date	Total	Paid?
891	01/10/95	$806.85	X
684	08/20/94	$67.40	X
610	06/17/94	$404.65	X
312	08/14/93	$280.05	X
204	04/01/93	$884.06	X
183	03/05/93	$278.46	X
139	12/26/92	$277.05	X
101	10/24/92	$1801.16	X
Total		$4799.68	

Figure 9. Using tables for data – A customer's order history looks good when poured into a Word table.

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. After all the data has been added, the AutoFit method of the Columns collection is called to resize the columns based on the data.

You can combine the code (removing the part that chooses a customer record) with Styles.Prg (you can see the result in Figure 6), 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.

Irregular Tables

Tables don't have to consist of simple grids. Not every row has to have the same number of columns. The Merge and Split methods of Cell and Cells let you combine and take apart groups of cells to create irregular tables. The Uniform property of Table indicates whether a table is regular or not; be sure to check it before using nested FOR EACH loops to try to process every row and column in a table.

Merge works two ways. You can either call it with a range of cells to have them merged or you can call it for one cell and pass it another to have those two merged. Here are the two syntax formats:

```
oCells.Merge()
oFirstCell.Merge( oSecondCell )
```

For example, to combine the second and third cells in row 1 of table oTable, you can use this code (all of the following assume that oTable is a reference to the table you're working with):

```
oTable.Cell(1,2).Merge( oTable.Cell(1,3) )
```

To change the fourth row of a table into a single cell, use code like this:

```
oRange = oTable.Rows[4].Range()
oRange.Cells.Merge()
```

Figure 10 shows a table (that started out with five rows and seven columns) after making those two changes.

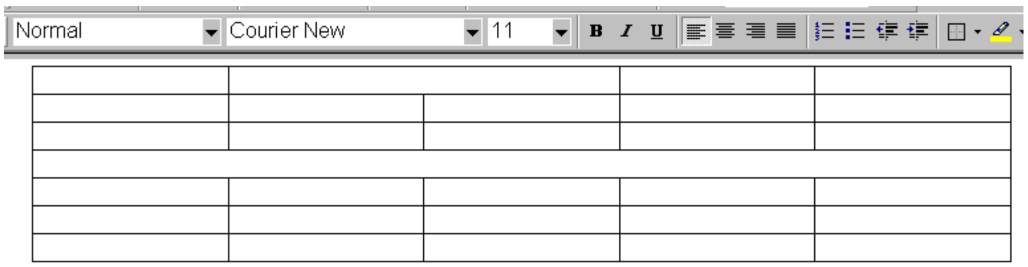


Figure 10. Irregular Table. Tables don't have to have the same number of columns in each row. The Merge and Split methods let you create irregular tables.

Split takes a cell or collection of cells and divides it up into one or more cells. It can optionally merge the cells before splitting them. Again, there are two different forms for the method, depending whether you call it from a single cell or from a collection of cells:

```
oCell.Split( nRows, nColumns)
oCells.Split( nRows, nColumns [, lMergeFirst ] )
```

For example, to divide the first cell in row 3 into two cells in the same row, use this command:

```
oTable.Cell(3,1).Split(1,2)
```

To take the cells in the second row of a table, combine them, then split them into three, resulting in just three cells in that row:

```
oTable.Rows[2].Cells.Split( 1, 3, .t.)
```

If you omit the third parameter from that call (.t.), each cell in the row would be split into three. If you pass something other than 1 as the first parameter, the single row would become multiple rows in the table.

Using Merge and Split, you can create extremely complex tables. While this provides for an attractive way to display data, keep in mind that it does make it harder to process the document. Simple FOR EACH loops through the Rows and Columns collections don't work when Uniform is .F.

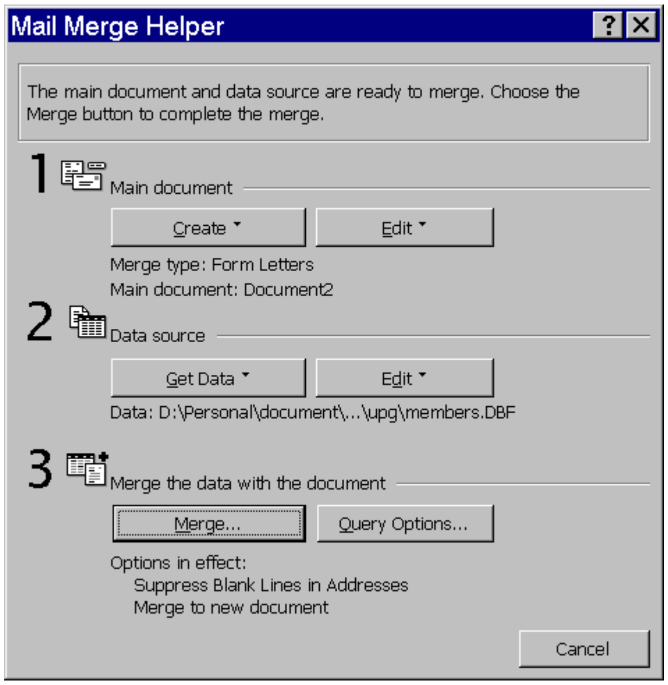
Merging documents with data

Mail merge is one of the killer functions that put word processors on everybody's desks. The ability to write a document once, then merge it with data repeatedly to produce personalized documents made sense to businesses, academics, and home users, too.

Word offers a variety of ways to merge data and documents. The best known is the most complex, using the built-in mail merge ability. However, that approach has some risks. Combining the built-in mail merge capabilities with some manual labor is better suited to automation.

Word's mail merge structure

Mail merge has been included in Word for many versions. Interactively, there's a wizard called the Mail Merge Helper (Figure 11) that guides users through the process. Even with this tool, though, setting up a new mail merge document, connecting it to data, and generating results is not simple. I've spent many hours on the phone walking relatives and friends through the process.


Mail Merge Helper

The main document and data source are ready to merge. Choose the Merge button to complete the merge.

1 Main document

Merge type: Form Letters
Main document: Document2

2 Data source

Data: D:\Personal\document\...\upg\members.DBF

3 Merge the data with the document

Options in effect:
Suppress Blank Lines in Addresses
Merge to new document

Figure 11. Mail Merge. This dialog walks interactive users through the steps needed to use Word's built-in mail merge facility.

There are a number of objects behind the Mail Merge Helper. Help lists no fewer than eight different objects whose names begin with "MailMerge." But it's not the complexity of the object model that leads to a search for alternatives; it's the fragility of the connection between the document and the data.

When you indicate that a Word document is a mail merge document, you specify the data source for the merge. It can be another Word document, an Excel worksheet, a text file, or come from any of a variety of databases files, including FoxPro, of course. For most data sources, Word uses ODBC to read the data.

If the data file is deleted or moved or something happens to the ODBC driver, the merge stops working. Many people using FoxPro 2.x tables in mail merge had a nasty surprise when they installed Service Pack 3 for VFP 6 (or anything else that installed ODBC 4.0) because it disabled the ODBC driver for FoxPro 2.x tables and failed to automatically substitute the VFP driver for it. Mail merges that had been working for years failed.

The need to deal with ODBC drivers and connections makes this approach harder and, especially, riskier than it needs to be. Unless you're dealing with extremely large tables, there are other, simpler, ways to solve this problem. If you are dealing with large tables, plan to control the ODBC aspects yourself rather than relying on what's on the user's machine. (You can set up ODBC connections programmatically, but that's beyond the scope of this session.)

If you need to merge only small amounts of data, I recommend avoiding the ODBC aspects while taking advantage of Word's sophisticated mail merge capabilities. The way to do that is to create the data source for a mail merge on the fly and attaching it to the document just long enough to merge it. This strategy is appropriate when the amount of data to be merged is small to moderate, but may need to be reconsidered for extremely large data sets.

There are several ways to create the data source on the fly. The one shown here, creating an Excel worksheet, works for data sets up to over 16,000 records and is quite fast. For tiny data sets, another approach (not shown in this session) is to send the data directly from FoxPro into Word.

The documents involved in mail merge

Once you take ODBC out of the picture, mail merge involves two or three documents. The first is what Word calls the *main document*. That's the document that contains the text of the letter, envelope, labels or whatever it is you're trying to create. Most of this document looks like any other document. The exception is that, in a few places, it contains *fields*, special markers that indicate that something is to be substituted at that location. Word actually offers a wide range of fields, including such things as the date, time and page number. For mail merge, we're specifically interested in fields of the type MergeField.

The second document in a mail merge is the *data source*. This is the document that contains the data to be substituted into the fields. When you use a native Word data source, it contains an ordinary Word table with one column for each MergeField. In the strategy described here, however, we'll create an Excel worksheet to use as the data source.

The third document is optional. It's the *result* created by merging the main document with the data source. I prefer to merge to a new document rather than directly to the printer, but there may be situations where you choose to skip this step.

The objects involved in mail merge

The main object in mail merge is, in fact, called MailMerge – it's accessed through the MailMerge property of Document. MailMerge's Fields property references a MailMergeFields collection, made up of MailMergeField objects – these objects represent the mail merge fields in the main document. When the document is attached to a data source, the DataSource property of MailMerge accesses a MailMergeDataSource object. Its DataFields property references a collection of MailMergeDataField objects that provide information about the fields in the data source. MailMergeDataSource also has a FieldNames property that references a MailMergeFieldNames collection with information about the field names for the data.

If this seems like a lot of objects, that's because it is, but in the strategy described here, you'll need to work directly with only the MailMerge and MailMergeFields objects.

Creating a main document

The first step is creating a main document. There are several ways to do this, not all involving Automation. Your users may simply create a main document using the Mail Merge Helper. The problem with that approach is that such documents will have data sources attached, but there are some solutions (discussed below).

You can build main documents with Automation just like other documents. In fact, you can also use a hybrid approach, initially setting up the document with Automation, then allowing a user to edit it manually.

To add a mail merge field to a document, use the Add method of the MailMergeFields collection. It calls for two parameters, as follows:

```
oDocument.MailMerge.Fields.Add( oRange, cField )
```

oRange	Reference to a range where the mail merge field is to be added.
cField	The mail merge field to be inserted.

Attaching a data source

One thing that makes the Mail Merge Helper so helpful is that it provides a list of the fields in the data source and lets you choose from that list as you create the main document. Figure 12 shows part of the Mail Merge toolbar with the Insert Merge Field dropdown open, showing a list of the fields from TasTrade's Employee table.

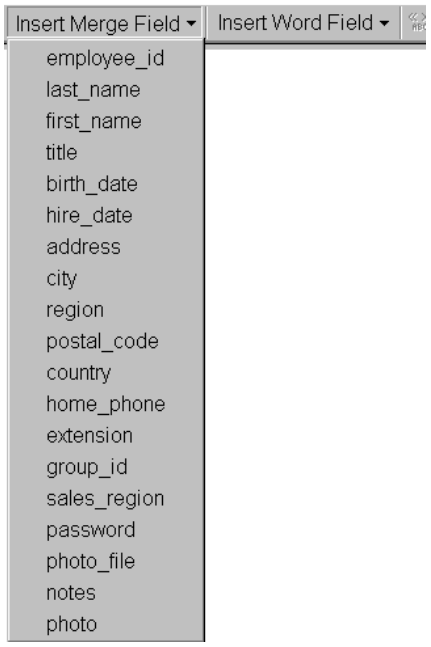


Figure 12. Adding fields interactively. When a main document is attached to a data source, you can add fields by choosing them from the Mail Merge toolbar.

Using header sources for field lists

If we want users to be able to create and edit main documents, we need a way to provide them with a list of fields, even though we don't want to create permanent connections between main documents and data sources. Several methods of the MailMerge object let us set up field lists.

There are two kinds of connections a main document can have to data. It can be connected to an actual data source that contains records available for merging. However, a main document can instead be connected to a *header source*, a document that provides only field names for the merge, but no actual merge data.

The advantage of a header source is that it's small and easy to create. We can use a header source to provide users with a list of fields while creating or editing the main document, but wait to create the complete data source until the user is ready for the actual merge. We can also create the header source and hide it from the user, when that's an appropriate strategy (such as where users are known to delete files they shouldn't.)

Listing 2 is a program that attaches a header source to a main document, based on the field list in a table or view. The key to the whole process is calling the CreateHeaderSource method of

MailMerge – the rest is just typical FoxPro string manipulation. (This program is AltCreateHeaderSource.PRG in the materials for this session.) You might call this routine like this:

```
DO CreateHeaderSource WITH oDocument, _SAMPLES+"TasTrade\Data\Employee", ;
                           "C:\Temp\EmployeeHeader.DOC"
```

Listing 2. Creating a header source. This program generates a header source on the fly from any table or view and attaches it to a document.

```
* AltCreateHeaderSource.PRG
* © 2000, Tamar E. Granor and Della Martin
* Derived From:  Microsoft Office Automation with Visual FoxPro
* Hentzenwerke Publishing. www.hentzenwerke.com

* Create a header source for the current document
* based on a table or view
* Assumes:
*      Word is open.

LPARAMETERS oDocument, cCursor, cDocument
   * oDocument = the document for which a header source is to be created.
   * cCursor = the filename, including path, of the table or view
   * cDocument = the filename, including path, where the
   *                         header source document is to be stored.

* Check parameters
IF PCOUNT()<3
   MESSAGEBOX("Must provide table/view name and document name")
   RETURN .F.
ENDIF

IF VarType(oDocument) <> "O"
   MESSAGEBOX("No document specified")
   RETURN .F.
ENDIF

IF VarType(cCursor) <> "C" OR EMPTY(cCursor)
   MESSAGEBOX("Table/View name must be character")
   RETURN .F.
ENDIF

IF VarType(cDocument) <> "C" OR EMPTY(cDocument)
   MESSAGEBOX("Document name must be character")
   RETURN .F.
ENDIF

LOCAL nFieldCount, cFieldList, aFieldList[1], nField

* Open the table/view
USE (cCursor) AGAIN IN 0 ALIAS MergeCursor
SELECT MergeCursor

* Get a list of fields
nFieldCount = AFIELDS( aFieldList, "MergeCursor" )
* Go through the list, creating a comma-separated string
cFieldList = ""
FOR nField = 1 TO nFieldCount
   IF NOT INLIST( aFieldList[ nField, 2], "G", "M" )
      * Can't use General or Memo fields
      cFieldList = cFieldList + aFieldList[ nField, 1] + ","
   ENDIF
ENDFOR
```

```
cFieldList = LEFT( cFieldList, LEN(cFieldList) - 1 )

* Attach the header to the document
oDocument.MailMerge.CreateHeaderSource( cDocument, , , cFieldList )

USE IN MergeCursor

RETURN
```

The resulting header file is a Word document consisting of a one-row table, with each column containing a fieldname.

When you open a main document interactively and the header source or data source is missing, Word insists that you either find the missing source or take action. In Word 2000, when the same thing happens with Automation, Word simply opens the file and detaches the header source or data source itself. (Unfortunately, in Word 97, when you open a main document with automation and the data source is missing, Word insists on your finding the missing data source, though it's surprisingly inventive if you point to the wrong file.)

It's easy to change the header source or data source attached to a main document. The OpenHeaderSource method of MailMerge attaches an existing header source to a main document. OpenDataSource attaches an existing data source to a main document. Both take long lists of parameters, but in each case, only the first is required – it's the name of the header/data source file, including path. Here's an example:

```
oDocument.MailMerge.OpenHeaderSource( "C:\Temp\EmployeeList.DOC" )
```

Using a data source at merge time

The header source allows your users to create their own main documents using a list of merge fields. Header sources contain no data. You also need the ability to create and attach a complete data source on the fly. The CreateDataSource method lets you build a new data source. Listing 3, included as Alt2CreateDataSource.PRG in the session materials, creates and attaches a data source to a document. It accepts the same three parameters as AltCreateHeaderSource in Listing 2. For speed, rather than creating Word document and putting all the data in the table, this routine copies the data to an Excel worksheet and attaches that as the data source. This approach is limited to 16,383 records, the maximum number that VFP can copy to an Excel worksheet.

Listing 3. Build a better data source. This program creates a data source on the fly. Rather than dealing with ODBC, send just the records and fields you need to a Excel worksheet, which is then attached to the main document, when you're actually ready to do a mail merge.

```
* Alt2CreateDataSource.PRG
* © 2000, Tamar E. Granor and Della Martin
* Derived From:  Microsoft Office Automation with Visual FoxPro
* Hentzenwerke Publishing. www.hentzenwerke.com

* Create a data source for the current document
* based on a table or view
* Assumes:
*      Word is open.
*      Excel is installed.

#DEFINE wdSeparateByTabs 1
#DEFINE wdFormatDocument 0

LPARAMETERS oDocument, cCursor, cDocument
   * oDocument = the document for which a header source is to be created.
   * cCursor = the filename, including path, of the table or view.
   *                  Data should already be filtered and sorted.
   * cDocument = the filename, including path, where the
   *                        data source document is to be stored.

* Check parameters
IF PCOUNT()<3
   MESSAGEBOX("Must provide table/view name and document name")
   RETURN .F.
ENDIF

IF VarType(oDocument) <> "O"
   MESSAGEBOX("No document specified")
   RETURN .F.
ENDIF

IF VarType(cCursor) <> "C" OR EMPTY(cCursor)
   MESSAGEBOX("Table/View name must be character")
   RETURN .F.
ENDIF

IF VarType(cDocument) <> "C" OR EMPTY(cDocument)
   MESSAGEBOX("Document name must be character")
   RETURN .F.
ENDIF

LOCAL nFieldCount, cFieldList, aFieldList[1], nField
LOCAL oWord, oRange, oSourceDoc, oRow, oTable

* Get a reference to Word
oWord = oDocument.Application

* Turn off pagination on the fly
oWord.Options.Pagination = .F.

* Open the table/view
USE (cCursor) AGAIN IN 0 ALIAS MergeCursor
SELECT MergeCursor

cTempFile = ADDBS(SYS(2023)) + SYS(3) + ".XLS"
COPY TO (cTempFile) TYPE XL5

oDocument.MailMerge.OpenDataSource( cTempfile, 0, .F.,;
                                    ,.t.,,,,,,,"Entire Spreadsheet" )

USE IN MergeCursor


RETURN
```

Performing the mail merge

Once you've jumped through all the hoops to get the data there, actually performing the mail merge is the easy part. Just call the MailMerge object's Execute method and -poof!- the main document and the data source are merged to a new document. This is all it takes:

```
oDocument.MailMerge.Execute()
```

Of course, you probably want to exercise more control than that over the merge. Various properties of the MailMerge object let you set things up before you call Execute. The two you're most likely to deal with are Destination and SuppressBlankLines. SuppressBlankLines is a logical that indicates whether lines in the document that are totally empty should be eliminated. The default is .T.

Destination determines where the merge results are sent. The default is wdSendToNewDocument (0). Other choices are wdSendToPrinter (1), wdSendToEmail (2) and wdSendToFax (3). There are several properties, all of which begin with "Mail", dedicated to particulars of the case where results are sent to email.

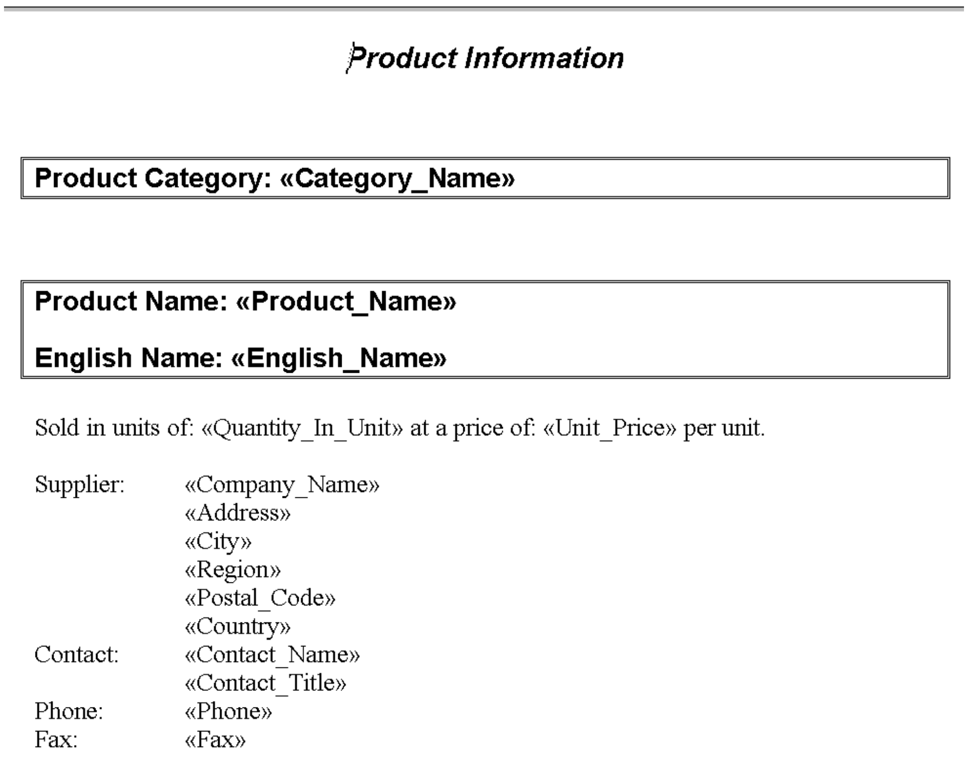
Putting mail merge together

Since mail merge is a two-step process, this final example has two parts. Listing 4 shows a program (AltBuildMerge.PRG in the session materials) that creates a template for product information sheets for Tasmanian Traders. The template is a mail merge main document attached to a header source only. The program runs a query that collects the data needed (in a real application, you'd probably have a view for this data), then calls on Alt2CreateHeaderSource.PRG (Listing 2 above) to attach the header. It then populates and saves the template. Figure 13 shows the completed template.

Listing 4. Creating a mail merge template. This program generates both a header source and a main document, in this case, a template for a main document.

```
* AltBuildMerge.PRG
* © 2000, Tamar E. Granor and Della Martin
* Derived From:  Microsoft Office Automation with Visual FoxPro
* Hentzenwerke Publishing. www.hentzenwerke.com

* Create a main document for product sheets.
* The document is created as a template so that it can be
* then be used with File-New.

#DEFINE CR CHR(13)
#DEFINE TAB CHR(9)
#DEFINE wdHeaderFooterPrimary 1
#DEFINE wdGray25 16
#DEFINE wdAlignParagraphCenter 1
#DEFINE wdCollapseEnd 0
#DEFINE wdParagraph 4
#DEFINE wdWord 2
#DEFINE wdLineStyleDouble 7
#DEFINE wdUserTemplatesPath 2
#DEFINE wdGoToBookmark -1

LOCAL oWord, oDocument, oRange, oBorderRange, cTemplatePath

* Open Word and create a new template document
oWord = CreateObject("Word.Application")
oDocument = oWord.Documents.Add(, .T.)

* Create a cursor of all products
OPEN DATABASE _SAMPLES + "TasTrade\Data\TasTrade"
SELECT product_name, english_name, category_name, ;
           quantity_in_unit, unit_price, ;
           company_name, contact_name, contact_title, ;
           address, city, region, postal_code, country, ;
           phone, fax ;
   FROM products ;
      JOIN supplier ;
         ON products.supplier_id = supplier.supplier_id ;
      JOIN category ;
         ON products.category_id = category.category_id ;
   ORDER BY Category_Name, Product_Name ;
   INTO CURSOR ProductList
```

```
* Attach a header source to the template document
DO AltCreateHeaderSource WITH oDocument, DBF("ProductList"), ;
   AddBs(SYS(2023))+"ProdHeader.DOC"

USE IN ProductList

* Now set up the product sheet
* First, assign a font for Normal
WITH oDocument.Styles["Normal"].Font
   .Name = "Times New Roman"
   .Size = 12
ENDWITH

* Add a header
WITH oDocument.Sections[1].Headers[ wdHeaderFooterPrimary ]
   oRange = .Range()
   WITH oRange
      .Text = "Tasmanian Traders"
      .Style = oDocument.Styles["Heading 1"]
      .ParagraphFormat.Alignment = wdAlignParagraphCenter
      .Shading.BackgroundPatternColorIndex  = wdGray25
   ENDWITH
ENDWITH

* Page heading
oRange = oDocument.Range(0,0)
WITH oRange
   .Style = oDocument.Styles[ "Heading 2" ]
   .ParagraphFormat.Alignment = wdAlignParagraphCenter
   .InsertAfter("Product Information" + CR + CR )
   .Collapse( wdCollapseEnd )

   * First, add fixed text and set up bookmarks where we want
   * the merge fields to go.
   * Add Product Category
   .Style = oDocument.Styles[ "Heading 3"]
   .InsertAfter( "Product Category: "  )
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add( "ProductCategory", oRange )
   .InsertAfter( CR )
   .Expand( wdParagraph )
   .Borders.OutsideLineStyle = wdLineStyleDouble
   .Collapse( wdCollapseEnd )
   .InsertAfter( CR )

   * Add Product Name
   .InsertAfter( "Product Name: ")
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("ProductName", oRange )
   .Collapse( wdCollapseEnd )
   .InsertAfter( CR )
   oBorderRange = oRange.Paragraphs[1].Range()
   .InsertAfter( "English Name: ")
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("EnglishName", oRange)
   .InsertAfter( CR )
   .Collapse( wdCollapseEnd )
   oBorderRange.MoveEnd( wdParagraph, 1)
   oBorderRange.Borders.OutsideLineStyle = wdLineStyleDouble

   * Now units and price
   .Style = oDocument.Styles[ "Normal" ]
   .InsertAfter( CR + "Sold in units of: " )
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("Quantity", oRange )
   .InsertAfter( " at a price of: ")
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("UnitPrice", oRange)
   .InsertAfter( " per unit." + CR + CR )
   .Collapse( wdCollapseEnd )

   * Now supplier information
   * To make things line up, we'll need a tab, so set  it up.
   WITH oDocument.Paragraphs.TabStops
      .ClearAll()
      .Add( oWord.InchesToPoints( 1 ) )
   ENDWITH

   .InsertAfter( "Supplier: " + TAB )
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("CompanyName", oRange)
   .InsertAfter( CR + TAB)
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("Address", oRange)
   .InsertAfter( CR + TAB )
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("City", oRange)
   .InsertAfter( CR + TAB )
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("Region", oRange)
   .InsertAfter( CR + TAB )
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("PostalCode", oRange)
   .InsertAfter( CR + TAB )
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("Country", oRange)
   .InsertAfter( CR )
   .InsertAfter( "Contact: " + TAB )
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("ContactName", oRange)
   .InsertAfter( CR + TAB)
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("ContactTitle", oRange)
   .InsertAfter( CR )
   .InsertAfter( "Phone: " + TAB )
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("Phone", oRange )
   .InsertAfter( CR )
   .InsertAfter( "Fax: " + TAB )
   .Collapse( wdCollapseEnd )
   oDocument.Bookmarks.Add("Fax", oRange)
   .InsertAfter( CR )

* Now insert a mail merge field at each bookmark

   oRange = oDocument.Bookmarks["ProductCategory"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "Category_Name")

   oRange = oDocument.Bookmarks["ProductName"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "Product_Name")

   oRange = oDocument.Bookmarks["EnglishName"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "English_Name")

   oRange = oDocument.Bookmarks["Quantity"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "Quantity_In_Unit")

   oRange = oDocument.Bookmarks["UnitPrice"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "Unit_Price")

   oRange = oDocument.Bookmarks["CompanyName"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "Company_Name")

   oRange = oDocument.Bookmarks["Address"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "Address")

   oRange = oDocument.Bookmarks["City"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "City")

   oRange = oDocument.Bookmarks["Region"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "Region")

   oRange = oDocument.Bookmarks["PostalCode"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "Postal_Code")

   oRange = oDocument.Bookmarks["Country"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "Country")

   oRange = oDocument.Bookmarks["ContactName"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "Contact_Name")

   oRange = oDocument.Bookmarks["ContactTitle"].Range()
   oDocument.MailMerge.Fields.Add( oRange, "Contact_Title")

   oRange = oDocument.Bookmarks["Phone"].Range()
   oDocument.MailMerge.Fields.Add( oRange,"Phone")

   oRange = oDocument.Bookmarks["Fax"].Range()
   oDocument.MailMerge.Fields.Add( oRange,"Fax")

ENDWITH

cTemplatePath = oWord.Options.DefaultFilePath( wdUserTemplatesPath )
oDocument.SaveAs( AddBs(cTemplatePath) + "ProdInfo")

oWord.Quit()

RETURN
```


Product Information

Product Category: «Category_Name»

Product Name: «Product_Name»

English Name: «English_Name»

Sold in units of: «Quantity_In_Unit» at a price of: «Unit_Price» per unit.

Supplier:	«Company_Name»
	«Address»
	«City»
	«Region»
	«Postal_Code»
	«Country»
Contact:	«Contact_Name»
	«Contact_Title»
Phone:	«Phone»
Fax:	«Fax»

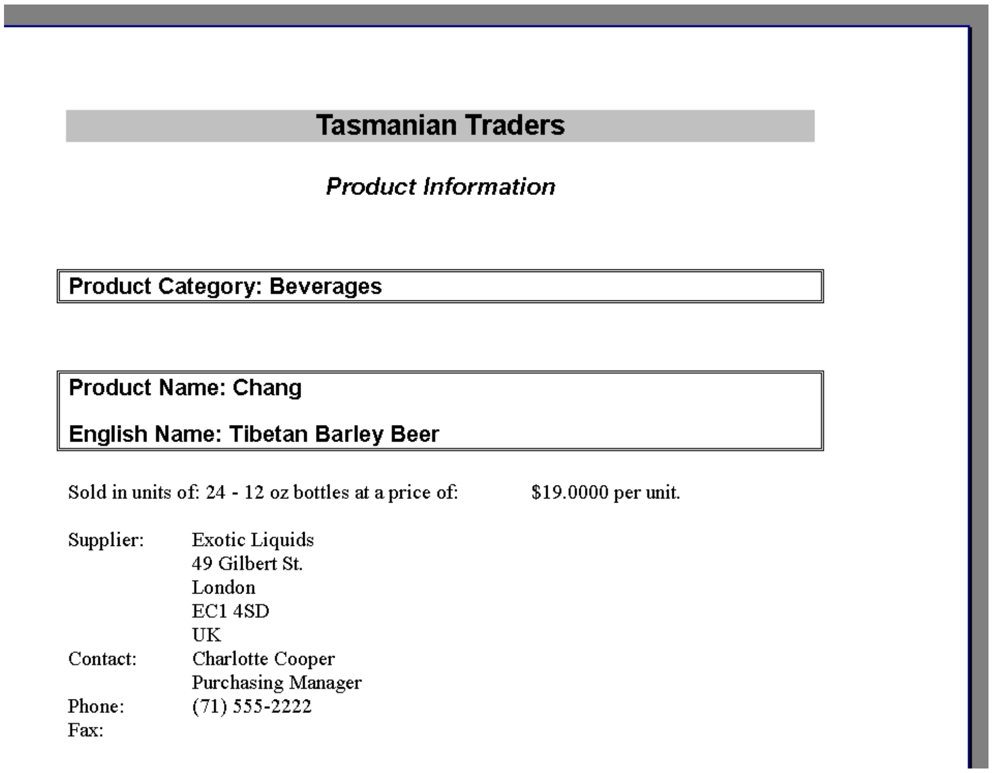
Figure 13. Creating mail merge documents. This template was created by Listing 4. It has a header source and is based on a query from the TasTrade database.

The second part of the process is to create an actual data source when you're ready to perform the mail merge. Listing 5 shows the code (Alt2DoMerge.PRG in the session materials) that creates the new document from the template, calls on Alt2CreateDataSource.PRG (Listing 3 above), then performs the merge and shows the result. Figure 14 shows part of the result.

Listing 5. Performing a merge. This code uses the template created by Listing 4 to generate a new document, creates a data source, and executes the merge.

```
* Alt2DoMerge.PRG
* © 2000, Tamar E. Granor and Della Martin
* Derived From:  Microsoft Office Automation with Visual FoxPro
* Hentzenwerke Publishing. www.hentzenwerke.com

* Create the Product Information sheets based on the
* template, using mail merge
#DEFINE wdUserTemplatesPath 2
#DEFINE wdWindowStateMaximize 1

LOCAL cTemplatePath, oDocument, oMergedDocument

* Create an instance of Word.
* Make it public for demonstration purposes.
RELEASE ALL LIKE o*
PUBLIC oWord
oWord = CreateObject("Word.Application")

* Make Word visible.
* oWord.Visible = .t.

* Create a new document based on the template
```

```
cTemplatePath = oWord.Options.DefaultFilePath( wdUserTemplatesPath )
oDocument = oWord.Documents.Add( ADDBS(cTemplatePath) + "ProdInfo" )

* Run the query to create a cursor of all products
* Create a cursor of all products
OPEN DATABASE _SAMPLES + "TasTrade\Data\TasTrade"
SELECT product_name, english_name, category_name, ;
           quantity_in_unit, TRANSFORM(unit_price, "@$") AS unit_price, ;
           company_name, contact_name, contact_title, ;
           address, city, region, postal_code, country, ;
           phone, fax ;
   FROM products ;
      JOIN supplier ;
         ON products.supplier_id = supplier.supplier_id ;
      JOIN category ;
         ON products.category_id = category.category_id ;
   ORDER BY Category_Name, Product_Name ;
   INTO CURSOR ProductList

* Now create and attach a data source
DO Alt2CreateDataSource WITH oDocument, ;
   DBF("ProductList"), AddBs(SYS(2023)) + "ProdData"

USE IN ProductList

* Now make Word visible
oWord.Visible = .T.

* Perform the mail merge
oDocument.MailMerge.Execute()

oMergedDocument = oWord.ActiveDocument

WITH oMergedDocument
   IF .ActiveWindow.WindowState <> wdWindowStateMaximize
      * Move it to make it visible - for some reason, it comes up
      * way off screen
      .ActiveWindow.Left = 0
      .ActiveWindow.Top = 0
   ENDIF

   * Preview it
   .PrintPreview()
ENDWITH

RETURN
```



Tasmanian Traders

Product Information

Product Category: Beverages

Product Name: Chang

English Name: Tibetan Barley Beer

Sold in units of: 24 - 12 oz bottles at a price of: $19.0000 per unit.

Supplier:	Exotic Liquids 49 Gilbert St. London EC1 4SD UK
Contact:	Charlotte Cooper Purchasing Manager
Phone:	(71) 555-2222
Fax:	

Figure 14. Mail merge results. The product information sheet created by the programs in Listings 4 and 6. There's one sheet for each product.

Summing Up

This session covers just a small fraction of what Microsoft Word can be called on to do through Automation. With very few exceptions, if you can do it interactively in Word, you can automate it. I hope it's given you some ideas as to how to put Word to work for you.

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 allowing me to use this material here.