

Chapter 6

Advanced Word

Word can produce some very complicated documents. Several features help bring order to longer documents: outlines, tables of contents, and indexes. Our look at Word finishes with the prototypical blend of word processing and databases: mail merge.

Several of Word's more complex abilities can provide the finishing touches to documents you create via Automation. This chapter looks at outlines, tables of contents, and indexes. In addition, it explores perhaps the most sought after of Word's capabilities from the database point of view: mail merge.

Organizing a document using styles

Both outlines and tables of contents are based on styles. In each case, you associate a style with each heading level, and Word does the rest. (In fact, Word can use this approach for multi-level lists, as well. See "Organizing text with lists" in Chapter 4.) The first step in creating either an outline or a table of contents is to use styles for each heading level in your document. You can use the built-in styles or create your own set of styles. As an extra benefit, this makes it easy to ensure that headings at the same level look the same throughout.

Word recognizes nine heading levels. Not coincidentally, the Normal template includes nine styles, named "Heading 1" through "Heading 9," that are linked to those nine heading levels. The easiest way to prepare for outlines and tables of contents is to use those built-in heading styles in your own documents. **Figure 1** shows them in outline view.

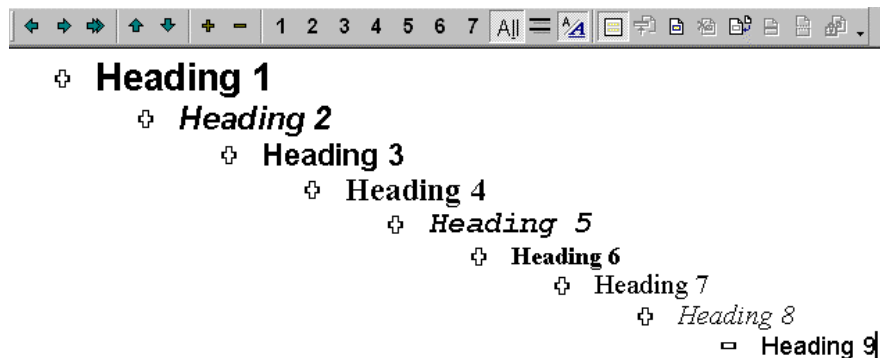


Figure 1. Built-in heading styles. You can use these styles for your headings to simplify creation of outlines and tables of contents, or you can set up your own.

You can use other styles, if you prefer. Set the heading level for a style through the OutlineLevel property of the Style object's ParagraphFormat object. The OutlineLevel property uses a set of constants with names like wdOutlineLevel1—the value of each is its outline level.



To demonstrate this use of styles, **Listing 1** is a program that creates a report of TasTrade sales organized by country. Within each country, customers are listed alphabetically. The report's title uses the Heading 1 style. Country names use Heading 2, and company names use Heading 3. The sales by year for each company (rounded to the nearest dollar) make up the body of the report and use the Body Text style. This program is included as MultiLevel.PRG in the Developer Download files available at www.hentzenwerke.com.

Listing 1. Using heading styles. This program creates a document that makes both outlining and table of contents creation simple.

```
* Create a multi-level document using built-in
* heading styles.

#DEFINE wdCollapseEnd 0
#DEFINE CR CHR(13)

LOCAL oDocument, oRange
LOCAL cCountry, cCompany, nYear

* Create an instance of Word.
* Make it public for demonstration purposes and so
* that you can use oWord in the following examples.
RELEASE ALL LIKE o*
PUBLIC oWord
oWord = CreateObject("Word.Application")
* Make Word visible.
oWord.Visible = .T.
oDocument = oWord.Documents.Add()

OPEN DATA _SAMPLES + "TasTrade\Data\TasTrade"

* Collect customer/order information. Result contains one record per
* company per year, order by country, then company,
* then year in descending order
SELECT Company_Name, Country, YEAR(Order_Date) AS Order_Year, ;
       ROUND(SUM(Order_Line_Items.Unit_Price*Order_Line_Items.Quantity - ;
       (0.01 * Orders.Discount * Order_Line_Items.Unit_Price * ;
       Order_Line_Items.Quantity)) + Orders.Freight, 0) AS OrderTotal ;
FROM Customer ;
   JOIN Orders ;
       ON Customer.Customer_Id = Orders.Customer_Id ;
   JOIN Order_Line_Items ;
       ON Orders.Order_Id = Order_Line_Items.Order_Id ;
GROUP BY 3, 2, 1 ;
ORDER BY 2, 1, 3 DESC ;
INTO CURSOR CompanyInfo

* Send data to document, as follows:
* Country is Heading 2
* Company is Heading 3
* Total with some text is Body Text

oRange = oDocument.Range()
WITH oRange
```

```

        .Style = oDocument.Styles[ "Heading 1" ]
        .InsertAfter("Tasmanian Traders Sales by Country and Company" + CR)
        .Collapse( wdCollapseEnd )
    ENDWITH

    cCountry = ""
    cCompany = ""
    nYear = 0

    SCAN
        WITH oRange
            .Collapse( wdCollapseEnd )
            IF NOT (Country == cCountry)
                .Style = oDocument.Styles[ "Heading 2" ]
                .InsertAfter( Country + CR )
                .Collapse( wdCollapseEnd )
                cCountry = Country
                cCompany = ""
                nYear = 0
            ENDIF

            IF NOT (Company_Name == cCompany)
                .Style = oDocument.Styles[ "Heading 3" ]
                .InsertAfter( Company_Name + CR )
                .Collapse( wdCollapseEnd )
                cCompany = Company_Name
                nYear = 0
            ENDIF

            .Style = oDocument.Styles[ "Body Text" ]
            .InsertAfter( "Total Sales for " + TRANSFORM(Order_Year,"9999") + ;
                " = " + TRANSFORM(OrderTotal,"@B, $$ 99,999,999") + CR)
            .Collapse( wdCollapseEnd )
        ENDWITH
    ENDSCAN

```

Figure 2 shows a portion of the resulting document.

Working with outlines

Outlines in Word are more a state of mind than a separate entity. They allow you to hide the details of a document while still showing its structure. Because different portions of an outline can be expanded different amounts, Word's outlines also provide you with the effect of a drill-down report.

Once you use appropriate styles to create the headings in a document, you've outlined the document. To see the outline, all you have to do is switch to Outline View by choosing View|Outline from the menu. As Figure 1 shows, Outline View includes a special toolbar for working with outlines. It allows interactive users to expand and collapse the outline, to determine which heading levels are visible, to decide whether to show all body text or just the first line, and to manipulate items in the outline, changing their level. When a document is displayed in Outline View, printing it prints the displayed outline, not the entire document. We suppose this is what you should get from a WYSIWYG word processor, but it's something of a surprise, since other views don't affect printing.

Tasmanian Traders Sales by Country and Company

Argentina

Cactus Comidas para llevar

Total Sales for 1995 = \$1,012 |

Total Sales for 1994 = \$699

Total Sales for 1992 = \$17,738

Océano Atlántico Ltda.

Total Sales for 1995 = \$2,948

Total Sales for 1994 = \$112

Total Sales for 1993 = \$2,133

Total Sales for 1992 = \$367

Rancho grande

Total Sales for 1995 = \$697

Total Sales for 1994 = \$1,986

Austria

Ernst Handel

Total Sales for 1995 = \$31,221

Total Sales for 1994 = \$47,060

Total Sales for 1993 = \$24,476

Total Sales for 1992 = \$5,372

Piccolo und mehr

Total Sales for 1995 = \$3,345

Total Sales for 1994 = \$8,816

Total Sales for 1993 = \$12,072

Figure 2. Multi-level document. The headings in this document, created by the program in Listing 1, use the built-in heading styles.

As Views are a visual issue and Automation is usually performed behind the scenes, what does all this mean for Automation? You probably won't need to display an outline in an Automation setting, but you may need to print it, requiring a switch to Outline View. Fortunately, this switch to Outline View and subsequent outline manipulation can be done even when Word is hidden.

The key object for working with outlines is the View object, accessible through the View property of the Window object. The Window object is accessed using the ActiveWindow property of Document. So, to set the ActiveDocument to OutlineView, use code like:

```
#DEFINE wdOutlineView 2  
oWord.ActiveDocument.ActiveWindow.View.Type = wdOutlineView
```

Figure 3 shows the results of issuing this code against the multi-level document from Figure 2. Other values for View's Type property include wdNormalView (1) and wdPrintPreview (4).



Figure 3. Document outline. If a document's headings use the right styles, creating an outline is as easy as switching to Outline View.

Once you're in Outline View, you can expand and collapse the outline, either as a whole or parts of it, using View's ExpandOutline and CollapseOutline methods. Each accepts a single, optional, parameter—the range to be expanded or collapsed. If the parameter is omitted, the current selection/insertion point determines what is expanded or collapsed. For example, to collapse the detail for the vendor "Cactus Comidas para llevar" in Figure 3, issue this code:

```
oRange = oWord.ActiveDocument.Paragraphs[3].Range()
oWord.ActiveDocument.ActiveWindow.View.CollapseOutline( oRange )
```



ExpandOutline has a bug (acknowledged by Microsoft). When you pass it a range consisting of a heading paragraph that has body text both above and below it, it expands that paragraph, showing the body text below it, but it also shows the body text above it. This expansion error causes problems when you attempt to collapse this part of the outline. In order to do so, you have to collapse a much larger portion of the outline than you should.

The ShowHeading method lets you determine how many levels of headings are displayed overall. You pass it a number—all headings up to and including the specified level are displayed, while all headings below that level, plus body text, are hidden. For example, to display headings up to level 3, use this code:

```
oWord.ActiveDocument.ActiveWindow.View.ShowHeading( 3 )
```

If you want to display all headings, call the ShowAllHeadings method:

```
oWord.ActiveDocument.ActiveWindow.View.ShowAllHeadings()
```

To show only the first line of body text, set the ShowFirstLineOnly property to .T.; this is a good way to get a summary of the document without showing just the headings. (In the example shown in Figure 3, changing this property doesn't make a difference because each paragraph has only a single line.) If you're showing all levels, it gives you the first line of each paragraph. Unfortunately, this property works only on screen; it doesn't affect printed output.

Notwithstanding the inability to use ShowFirstLineOnly, once you've arranged the outline the way you want it, call Document's PrintOut method (described in the "Printing" section in Chapter 4) to print your outline.

Creating a table of contents

When you have a document with headings that use heading level styles, creating a table of contents is a piece of cake. One call to the Add method of the document's TablesOfContents collection and you're done. All you need to know is where to put the table of contents (a range) and which heading levels you want to include. This code adds a table of contents right at the beginning of the document. It includes headings through level 3.

```
#DEFINE wdNormalView 1
oWord.ActiveDocument.ActiveWindow.View.Type = wdNormalView
oRange = oWord.ActiveDocument.Range(0,0)
oWord.ActiveDocument.TablesOfContents.Add( oRange, .T., 1, 3 )
```

Figure 4 shows one page of the results for the document in Figure 2.

TablesOfContents is a collection of TableOfContents objects. (Note the plural "Tables" in the collection name.) Each TableOfContents object represents one table of contents in the document. Why would you have more than one? Because you might have a separate table of contents for each chapter or each major section in a document.

QUICK-Stop.....	9
Toms Spezialitäten.....	9
Ireland.....	9
Hungry Owl All-Night Grocers.....	9
Italy.....	9
Franchi S.p.A.....	9
Magazzini Alimentari Riuniti.....	9
Reggiani Caseifici.....	9
Mexico.....	10
Ana Trujillo Emparedados y helados.....	10
Antonio Moreno Taqueria.....	10
Centro comercial Moctezuma.....	10
Pericles Comidas clásicas.....	10
Tortuga Restaurante.....	10
Norway.....	10
Santé Gourmet.....	10
Poland.....	10
Wolski Zajazd.....	10
Portugal.....	11
Furia Bacalhau e Frutos do Mar.....	11
Princesa Isabel Vinhos.....	11
Spain.....	11
Bóldo Comidas preparadas.....	11
FISSA Fabrica Inter. Salchichas S.A.....	11
Galería del gastrónomo.....	11
Godos Cocina Típica.....	11
Romero y tomillo.....	11
Sveden.....	12
Berglunds snabbköp.....	12
Folk och få HB.....	12
Switzerland.....	12
Chop-suey Chinese.....	12
Richter Supermarkt.....	12
UK.....	12
Around the Horn.....	12
B's Beverages.....	12
Consolidated Holdings.....	13
Eastern Connection.....	13
Island Trading.....	13
North/South.....	13
Seven Seas Imports.....	13
USA.....	13
Great Lakes Food Market.....	13
Hungry Coyote Import Store.....	13
Lazy K Kountry Store.....	13
Let's Stop N Shop.....	13
Lonesome Pine Restaurant.....	14
Old World Delicatessen.....	14
Rattlesnake Canyon Grocery.....	14
Save-a-lot Markets.....	14
Split Rail Beer & Ale.....	14
The Big Cheese.....	14

Figure 4. Table of contents. Using built-in heading styles makes it a breeze to create a table of contents.

The Add method takes quite a few parameters. Here's the syntax, listing the ones you're most likely to use:

```
oDocument.TablesOfContents.Add( oRange, lUseHeadingStyles, nFirstHeadingLevel,
                                nLastHeadingLevel, lUseFields, ,
                                lRightAlignPageNums, lIncludePageNums,
                                cAddedStyles )
```

oRange	Object	Range where the table of contents should be placed.
lUseHeadingStyles	Logical	Indicates whether the table of contents should be created based on built-in heading styles. Defaults to .T.
nFirstHeadingLevel	Numeric	The lowest numbered heading style to be included in the table of contents. Defaults to 1.
nLastHeadingLevel	Numeric	The highest numbered heading style to be included in the table of contents. Defaults to 9.
lUseFields	Logical	Indicates whether items marked as table of contents fields are included in the table of contents. See Help for more information. Defaults to .F.
lRightAlignPageNums	Logical	Indicates whether page numbers are right-aligned on the page. Defaults to .T.
lIncludePageNums	Logical	Indicates whether page numbers are included in the table of contents. Defaults to .T.
cAddedStyles	Character	A comma-separated list of styles other than the built-in heading styles that should also be used in creating the table of contents.

Once you create the table of contents, you can set the character that fills the space between the entries and the page numbers on each line by using the `TabLeader` property of the `TableOfContents` object (not the collection). **Table 1** shows constant values for `TabLeader`.

Table 1. *Filling the space. The `TabLeader` property determines how the space between the headings and the page numbers in a table of contents is filled.*

Constant	Value	Constant	Value
<code>wdTabLeaderSpaces</code>	0	<code>wdTabLeaderLines</code>	3
<code>wdTabLeaderDots</code>	1	<code>wdTabLeaderHeavy</code>	4
<code>wdTabLeaderDashes</code>	2	<code>wdTabLeaderMiddleDot</code>	5

Several other properties of `TableOfContents` let you change parameters set by `Add`, such as `IncludePageNumbers`, `RightAlignPageNumbers`, `LowerHeadingLevel` (corresponds to `nFirstHeadingLevel`) and `UpperHeadingLevel` (corresponds to `nLastHeadingLevel`).

As you may have guessed from our comments on the `lUseFields` parameter, there is another approach to creating a table of contents. It involves marking each item you want listed with a code. For details, see the topic “Use field codes for indexes, tables of contents, or other tables” in the regular Word Help.

Creating indexes

Long documents are far more useful when they’re indexed. Unfortunately, the best way to produce a high-quality index is still the old-fashioned way—by having a person do it. That’s because no automated technique can read the contents of a manuscript and decide whether a particular use of a word or phrase deserves to be mentioned in the index. In fact, this problem is the reason that so many search engines produce irrelevant results so much of the time.

Nonetheless, Word offers a couple of ways to create indexes. Only one of the techniques is really suited to Automation, however, as the other involves a great deal of human intervention.

The easily automated technique uses a concordance table listing all the words to be indexed and the index entries for them.

Deciding what to index

The concordance table is a separate document containing a standard Word table with two columns. You can create it interactively or through Automation. **Table 2** is part of a concordance table for Chapter 3 of this book, “Visual FoxPro as an Automation Client” (though not the one actually used to index the book).

Table 2. *Creating an index. One way to index a document is by using a table that lists the terms to be indexed together with the index entries.*

1426	error:1426
1429	error:1429
Activate	Activate
Application	Application
client	client
collection	collection
collections	collection
Command Window	Command Window
CreateObject()	CreateObject()
CreateObjectEx()	CreateObjectEx()
DCOM	DCOM
debugger	debugger
debugging	debugging

The table shows several items worth noting. First, if a word appears in several forms in the document, you need to list all those forms in the table. (Notice “collection” and “collections” in the first column, both being indexed to “collection.”)

Second, you can handle multiple levels in the index within the table by separating them with a colon in the second column. Note the entries for 1426 and 1429. In the index, these will both be part of a general entry for “error,” but sub-divided into “1426” and “1429.”

Third, most rows in the table have the same thing in the two columns. If you’re creating the table interactively, this is a great place to take advantage of a Word feature that’s usually an annoyance. In Word, the Edit|Repeat Typing menu option (the shortcut is Ctrl-Y) is almost always available. When you’re creating a concordance table, you can use it to cut your typing almost in half. Type the string into the first column, then hit Ctrl-Y to repeat it in the second column.

Finally, Word is unfortunately not smart enough to notice when one index phrase is contained in another. For example, Table 2 contains the word “client.” If we added “client application” to the table, when the index is created, a phrase like “Using VFP as our client application” would be indexed twice—once for “client” and once for “client application.” Beware of this issue as you decide what to put in the table.

Marking items for the index

Once you have the concordance table, the next step is to get the relevant items in the document marked. This is the part where the concordance table gets put to work and actually saves time. Interactively, choose **Insert|Index and Tables** from the menu. This displays the dialog shown in **Figure 5**. Choose **AutoMark** and select your concordance file.

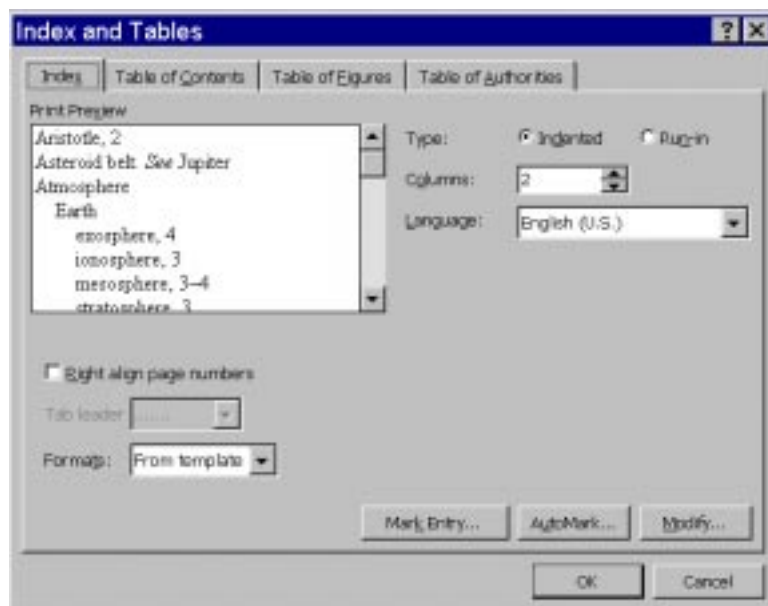


Figure 5. Creating an index. To mark items in a document based on the concordance table, choose **AutoMark** from this dialog.

Word goes through the document and adds special fields to the document everywhere it finds an item from the left column of the concordance table.

To automate this process, you use the `AutoMarkEntries` method of the document's `Indexes` collection, like this:

```
oDocument.Indexes.AutoMarkEntries( "d:\autovfp\chapter3\wordlist.DOC" )
```

Generating the index

After the items are marked, creating the index itself is simple. Interactively, you position the cursor where the index is to be placed, then bring up the **Index and Tables** dialog (shown in **Figure 5**) again, choose a format from the **Formats** dropdown, check **Right align page numbers**, if you want, and choose **OK**. **Figure 6** shows part of the index generated for Chapter 3, "Visual FoxPro as an Automation Client," based on the concordance in **Table 2**.

Activate	4	object reference	2, 6, 7
Application	1, 2, 3, 4	Office	1, 2, 6, 8, 9
collection	4, 6, 7	Outlook	2
Command Window	7	parameter	1, 2, 8
CreateObject()	1, 4, 5	PixelsToPoints	3
DCOM	2	polymorphism	1
debugging	2	PowerPoint	1, 2, 4
Developer Download files	3	registry	5
error 1426	8	server	1, 2, 5, 6, 8, 9
error 1429	8	SET OLEOBJECT	4, 5

Figure 6. Generated index. Applying the concordance table in Table 2 to Chapter 3 results in this index.

To produce the index with Automation, you use the Add method of Indexes, passing it the range where the index is to be placed and several other parameters. This example creates the same index as in Figure 6, positioning it at the end of the document.

```
#DEFINE wdCollapseEnd 0
#DEFINE wdHeadingSeparatorNone 0
#DEFINE wdIndexIndent 0
#DEFINE wdTabLeaderDots 1

oRange = oDocument.Range()
oRange.Collapse( wdCollapseEnd )

oIndex = oDocument.Indexes.Add( oRange, wdHeadingSeparatorNone, .T., ;
                                wdIndexIndent, 2, .F.)

* Change leader to dots
oIndex.TabLeader = wdTabLeaderDots
oIndex.Update()
```

The Add method has a number of parameters. In the example, following the range, we specify that no punctuation is needed to separate entries for each letter of the alphabet in the index. Other choices include wdHeadingSeparatorBlankLine (1) and wdHeadingSeparatorLetter (2). The Index object's HeadingSeparator property matches up to this parameter.

The third parameter corresponds to the RightAlignPageNumbers property of the Index object and determines whether the page numbers immediately follow the entry or are right-justified.

The fourth parameter determines how the index handles sub-entries. The default value, wdIndexIndent (0), lists each sub-entry indented on a separate line beneath the main heading. The alternative option, wdIndexRunIn (1), separates the sub-entries with semi-colons on the same line with the main heading. Don't use it when you're right-aligning the page numbers—it looks terrible in that case.

The fifth parameter specifies the number of columns in the index, two in the example. It corresponds to the NumberOfColumns property of the Index object.

The final parameter shown in the example indicates that accented letters should be combined with the basic letters. Pass .T. to separate accented letters into separate index listings. This parameter corresponds to Index's AccentedLetters property.

Add has several additional parameters, but they're fairly obscure. See Help for details.

Formatting indexes

As with tables of contents, once you've added an index, you can use its properties to change its appearance. The Index object has a number of properties that affect its look—a number of them correspond to parameters of the Indexes collection's Add method, as noted previously. In addition, as with TableOfContents, the TabLeader property lets you vary the character that appears between the entry and the page number(s). The constants are the same as for TableOfContents—see Table 1.

The Update method (used in the example in the previous section), as its name suggests, updates the index. It's useful both when the marked index entries change and when you've changed formatting.

Merging documents with data

Mail merge is another of the killer functions that put word processors on everybody's desk. 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, it's also possible to merge documents by using Find and Replace and by combining the built-in mail merge capabilities with some manual labor. We'll take a look at each of these approaches here and tell you why we think the third approach is the best 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 (see **Figure 7**) 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. We've both spent many hours on the phone walking relatives and friends through the process.

Behind the Mail Merge Helper, there are a number of objects. 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 us to recommend alternative approaches; 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, a text file, or any of a variety of database 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 got 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.

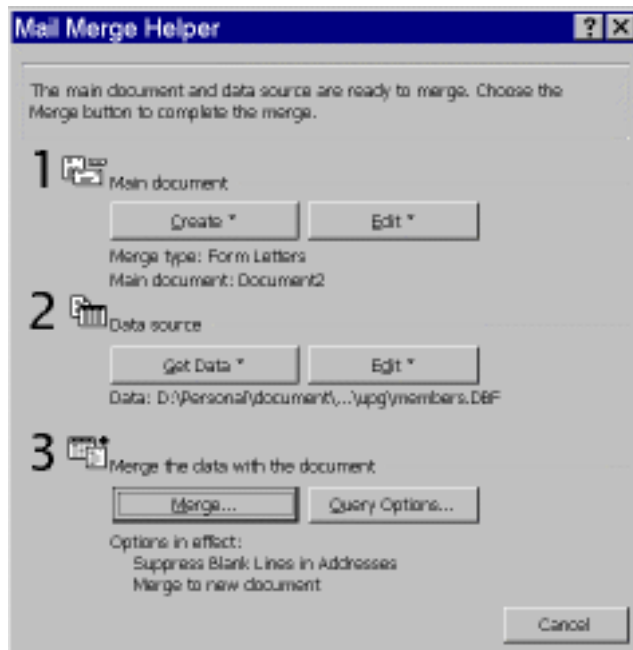


Figure 7. Mail merge. This dialog walks interactive users through the steps needed to use Word's built-in mail merge facility.

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. (To learn about managing ODBC and connections, check out the book's sister volume in the Essentials series, *Client-Server Applications with Visual FoxPro 6.0 and SQL Server 7.0*.) The key thing you'll need to do is pass the connection string to the `OpenDataSource` method of the `MailMerge` object (discussed later in this chapter).

Substituting data with Find and Replace

One alternative to Word's mail merge is to use the Find and Replace objects. Since these were discussed in detail in Chapter 5 (see "Search and replace"), we'll talk here only about the specific issues involved in using them for merging documents and data.

The basic idea is to create a document that contains the desired result except for a set of strings that are to be replaced by data from your database. Alternatively, you can use a template from which you create a document, then replace the strings.

The main issue is making sure that you only replace what you mean to replace. When you're working interactively, this isn't a problem because you can see each match before you agree to replace it. With Automation, however, you need a better solution. One possibility is to enclose the strings with a special character or characters. For example, you might surround the strings to be replaced with angle brackets (like "<name>") or exclamation points ("!name!"). Then, the code to replace the string would look something like this:

```
#DEFINE wdReplaceAll 2
oRange = oDocument.Range(0,0)
WITH oRange.Find
    .Text = "!Name!"
    .Replacement.Text = TRIM(Employee.First_Name) ;
                        - " " + TRIM(Employee.Last_Name)
    .Execute( , , , , , , , , , wdReplaceAll)
ENDWITH
```

The biggest problem with this approach is trying to find an appropriate special character to delimit your strings. You really need to be sure it's something that won't appear elsewhere in the document. While it's unlikely that a special character like an exclamation point would surround text, there's always the off chance that a user might choose to do something strange.

You can use a table to drive this approach. Consider putting the strings and formatting to be found and substituted into a table, and having a single method to perform the search and replace. Nonetheless, while searching for and replacing delimited strings is a viable solution to merging data into documents, we still think there's a better way.

Drop back 10 yards and punt

So what's a body to do? There's Word with a perfectly good mail merge engine, but the need to use ODBC to access FoxPro tables is a serious impediment. On the other hand, the rest of the mail merge facilities are really useful. The solution is to avoid the ODBC aspects while taking advantage of everything else Word has to offer.

The way to do that is to create the data source for a mail merge on the fly, sending FoxPro data to Word, 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 it may need to be reconsidered for extremely large data sets. (However, realize that no matter what approach you use, the data has to be sent to Word somehow, so this method may be as good as any other and it does afford you more control over the process and less likelihood of trouble caused by end users than the traditional approach.)

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. It contains an ordinary Word table with one column for each MergeField. In the strategy we describe, we'll build this document on the fly.

The third document is optional. It's the *result* created by merging the main document with the data source. We 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, of course, is that such documents will have data sources attached, but there are some solutions (discussed in the next section, "Attaching a data source").

Users can also create main documents manually by inserting mail merge fields by choosing Insert|Field from the menu. The dialog shown in **Figure 8** appears. To add a mail merge field, choose Mail Merge in the left column and MergeField on the right, then type the field name in the Field codes box, as shown in the figure.

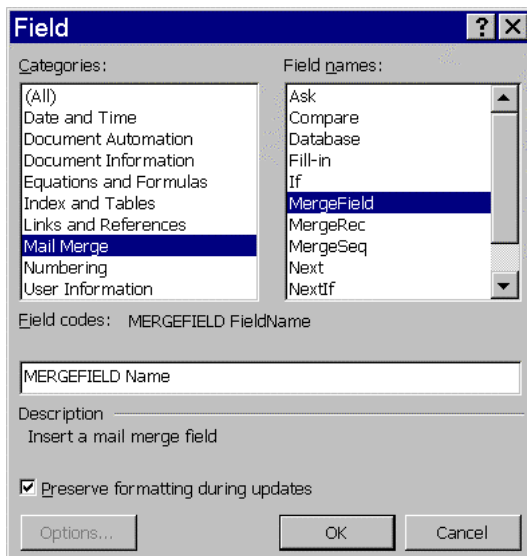


Figure 8. Inserting mail merge fields. Users can build mail merge documents manually rather than using the Mail Merge Helper.

Of course, 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.

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

Attaching a data source

One of the things 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 9 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 9. 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.

In fact, 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. (That might be appropriate where users are known to delete files when 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. (It's CreateHeaderSource.PRG in the Developer Download files available at www.hentzenwerke.com.) The key to the whole process is the call to the CreateHeaderSource method of MailMerge—the rest is just typical FoxPro string manipulation. You might call this routine like this:

```
DO CreateHeaderSource WITH oDocument, _SAMPLES+"TaTrade\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.*

```
* CreateHeaderSource.PRG
* 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 aFieldList[ nField, 2] <> "G"
    * Can't use General 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 simply a one-row Word table, 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.

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 the 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 need the ability to create and attach a complete data source on the fly. The `CreateDataSource` method lets you build a new data source. As with `CreateHeaderSource`, you build a Word table, then attach it to the main document. Most of the work is pretty straightforward. **Listing 3**, included as `CreateDataSource.PRG` in the Developer Download files available at www.hentzenwerke.com, creates and attaches a data source to a document. It accepts the same three parameters as `CreateHeaderSource` in Listing 2. Because the slowest part of the process is sending the actual data from VFP to Word, be sure to create a cursor or view that contains only the data you need for the mail merge before you call `CreateDataSource`. Do your filtering of both fields and

records on the VFP side. The EditDataSource method opens the DataSource associated with a main document. If it's already open, it activates it.

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 Word data source when you're actually ready to do a mail merge.

```
* CreateDataSource.PRG
* Create a data 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.
    *     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

* 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 aFieldList[ nField, 2 ] <> "G"
```

```
        * Can't use General fields
        cFieldList = cFieldList + aFieldList[ nField, 1] + ","
    ENDIF
ENDFOR
cFieldList = LEFT( cFieldList, LEN(cFieldList) - 1 )

WITH oDocument.MailMerge
    * Attach the data to the document
    .CreateDataSource( cDocument, , , cFieldList )
    .EditDataSource
    oSourceDoc = oWord.ActiveDocument
    oTable = oSourceDoc.Tables[1]
    oRow = oTable.Rows[1]

    * Now open the data source and put the data into the document
    SCAN
        WITH oRow
            FOR nField = 1 TO nFieldCount
                DO CASE
                    CASE TYPE( FIELDS( nField )) = "C"
                        * Get rid of trailing blanks
                        .Cells[ nField ].Range.InsertAfter( ;
                            TRIM( EVAL(FIELDS( nField ))) )
                    CASE TYPE( FIELDS( nField )) = "G"
                        * Do nothing
                    OTHERWISE
                        * Just send it as is
                        .Cells[ nField ].Range.InsertAfter( EVAL(FIELDS( nField )))
                ENDCASE
            ENDFOR
        ENDWITH
        oRow = oTable.Rows.Add()
    ENDSCAN
    oRow.Delete()
    oSourceDoc.Save()
ENDWITH

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), our preference. Our 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 e-mail.

Determining the document type

The `MailMerge` object has a couple of other important properties. `State` lets you check, for any document, what role it plays in the mail merge process. **Table 3** shows the possible values for `State`.

Table 3. What kind of a document am I? `MailMerge`'s `State` property tells you what kind of document you're dealing with.

Constant	Value	Constant	Value
<code>wdNormalDocument</code>	0	<code>wdMainAndHeader</code>	3
<code>wdMainDocumentOnly</code>	1	<code>wdMainAndSourceAndHeader</code>	4
<code>wdMainAndDataSource</code>	2	<code>wdDataSource</code>	5


`MainDocumentType` tells what kind of main document you have. **Table 4** shows the values this property can take.

Table 4. Merge document types. The `MainDocumentType` property classifies documents into the various kinds of merge documents you can create.

Constant	Value	Constant	Value
<code>wdNotAMergeDocument</code>	-1	<code>wdEnvelopes</code>	2
<code>wdFormLetters</code>	0	<code>wdCatalog</code>	3
<code>wdMailingLabels</code>	1		

Rescuing abandoned mail merge documents

You may be faced with main documents that have been detached from their data sources and for which you don't have access to the appropriate tables. It turns out that it's quite easy to go through these documents and build a header source or a data source for them, as well.

 **Listing 4** shows `ExtractDataSource.PRG` (also included in the Developer Download files available at www.hentzenwerke.com), which traverses the `MailMergeFields` collection of the document to create a list of fields, then uses that list to create a data source (albeit an empty one).

Listing 4. Restoring data sources. When main documents get detached from their data sources, this program can create an empty data source that allows the document to be edited.

```
* ExtractDataSource.PRG
* Process an existing document and create a data source document
* based on the fields used on the document.
```

```
LPARAMETERS oDocument, cSourceName
```

```
* oDocument = the existing mail merge document
* cSourceName = the filename, including path, for the data source

#DEFINE wdNotAMergeDocument -1

* Should check parameters here. Omitted for space reasons

WITH oDocument
  IF .MailMerge.MainDocumentType = wdNotAMergeDocument OR ;
    INLIST(.MailMerge.State, 0, 1)
    * Need to create a data source
    * Go through fields and create a list
    LOCAL oField, cCode, cField, cFieldList, cHeaderName

    cFieldList = ""
    FOR EACH oField IN .MailMerge.Fields
      cCode = oField.Code.Text
      * Parse out extraneous information
      cField = ALLTRIM( STRTRAN(STRTRAN(cCode, ;
        "MERGEFIELD", ""), "\* MERGEFORMAT", "") )
      IF NOT cField+"," $ cFieldList
        cFieldList = cFieldList + cField + ","
      ENDIF
    ENDFOR

    IF LEN(cFieldList) > 1
      cFieldList = LEFT(cFieldList, LEN(cFieldList)-1) + ""
    ENDIF

    * Now create a data source document.
    .MailMerge.CreateDataSource( cSourceName, , , cFieldList )
  ENDIF
ENDWITH

RETURN
```

Putting it all together



To demonstrate this chapter's main lesson, we have a two-part process. **Listing 5** shows a program (WordSample3Pt1.PRG in the Developer Download files available at www.hentzenwerke.com) 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 CreateHeaderSource.PRG (see Listing 2) to attach the header. It then populates and saves the template. **Figure 10** shows the completed template.

Listing 5. *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.*

```
* Create a main document for product sheets.
* The document is created as a template so that it can
* 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")
oWord.Visible = .T.
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 CreateHeaderSource 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" )

RETURN
```

Product Information

Product Category: «Category_Name»

Product Name: «Product_Name»

English Name: «English_Name»

Sold in units of: «Quantity_In_Units» 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 10. *Creating mail merge documents. This template was created by Listing 5. 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 6** shows the code (WordSample3Pt2.PRG in the Developer Download files available at www.hentzenwerke.com) that creates the new document from the template, calls on CreateDataSource.PRG (see Listing 3), then performs the merge and shows the result. **Figure 11** shows part of the result.

Listing 6. *Performing a merge. This code uses the template created by Listing 5 to generate a new document, creates a data source, and executes the merge.*

```
* 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 CreateDataSource WITH oDocument, DBF("ProductList"), ;
                    AddBs(SYS(2023)) + "ProdData"

USE IN ProductList

* 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
```

The screenshot displays a mail merge result for a product information sheet. At the top, there is a grey header bar with the text "Tasmanian Traders". Below this is a section titled "Product Information". The first field is "Product Category: Beverages". The second field is "Product Name: Chang" and "English Name: Tibetan Barley Beer". Below these fields, there is a line of text: "Sold in units of: 24 - 12 oz bottles at a price of: \$19.0000 per unit." The bottom section contains contact information: "Supplier: Exotic Liquids, 49 Gilbert St, London, EC1 4SD, UK" and "Contact: Charlotte Cooper, Purchasing Manager". The phone number is "(71) 555-2222" and the fax number is blank.

Figure 11. Mail merge results. This is the product information sheet created by the programs in Listings 5 and 6. There's one sheet for each product.

We'd like to say, "That's all, folks!" But the truth is that, even with three full chapters devoted to it, there's far more to Word than we've been able to cover here. So, if there's something you want to do in a document and we haven't shown you the way, just dig in and try it. Remember the keys to figuring it out. First, try to do it interactively. Try recording a macro. Use the Word VBA Help file and the Object Browser to find out what objects, methods, and properties are involved. Word's object model is extremely rich. If you can imagine it, it can probably be done.