Chapter 15

Programming in Word and Excel

G O A L S

After working through this chapter, you will be able to:

Recognize some of the differences between WordBasic and Visual Basic.

Record macros in WordBasic and add programming statements to turn them into applications.

Record macros in Excel and add Visual Basic for Applications (VBA) procedures to build a worksheet application.

OVERVIEW

Microsoft Office is completely programmable. Every part and feature can be programmed with languages that look very much like Visual Basic.

You may already have some experience with applications programming in Office because when you record a macro, that produces code. In addition to the macro code, you can add program statements such as ***For-Next*** *and* ***Do-While*** *loops to automate repetitive processes. Every time you find yourself cutting and pasting over and over again, or reformatting file after file, you are performing tasks that probably could be automated by writing a program. Using programs to link different parts of Microsoft Office together allows you to build custom tailored applications for your specific needs.*

The difficult part of programming Office is interacting with objects in the different applications. Using macros to learn how to interact with the application is the key to learning how to program the application. Recording a macro and then examining the code produced, teaches you how to program exactly what you want.

The first section in this chapter presents an overview of programming with Microsoft Office as well as examining some of the differences between Visual Basic and Office's versions of Basic. The second section guides you through recording macros in Word and Excel, examining the code produced, and then enhancing the macro code. The third section guides you through a Word application to learn how to automate the process of converting file formats. Finally, the fourth section shows you how to program an application using macros and programming techniques to automate a car dealership's customer and vehicle database.

Application Programming Overview

Microsoft Office Professional is comprised of four applications, or programs: Word, Excel, PowerPoint, and Access. Software, like most things, grows in fits and starts. For example, most software, including each of the Microsoft Office applications, evolves and changes from one revision to the next. Usually, revisions to a program range from fixing bugs to the addition of new features. At other times, a revision can mean a complete overhaul. Of the four applications in Microsoft Office, Word and Excel are quite mature, PowerPoint is newer, and Access, the database program, has been extensively revised and updated. Because of these different evolutionary stages, there are three versions of Basic to be used with the three main applications of Office.

Section 1 presents an overview of programming in Word and a little about Excel. You will learn the major differences between Visual Basic and Office's versions of Basic; a few reasons for programming Word and Excel; and you will record a Word macro and study its code.

WordBasic Programming

Because Word is the oldest application in Microsoft Office, WordBasic, its version of Basic, is the least sophisticated. With the release of Office 97, the old WordBasic language is being replaced with Visual Basic for Applications (VBA), a language much more like Visual Basic. However, thousands of companies still use Word with WordBasic, so the emphasis here will be on programming in WordBasic.

Why would you want to program your word processor? Word is a very rich program already. It has hundreds of features such as templates, wizards, and mail merge to help you create documents and perform tasks productively. However, even with all these tools there will be times when Word alone can't perform the tasks you need it to do. For example, suppose your company is switching from a different word processing program to Word, and you need to convert all of the company's documents to Word format. Also, newer versions of Word are integrated with the Internet. So a world of incompatible documents may be waiting to be processed. With a knowledge of WordBasic, you can be ready to do it.

If you're using Word now to create documents, you may be wondering where to put the program code. The answer is that WordBasic code is added to Word macros. When you record a macro, the keystrokes are converted into WordBasic commands and saved in a file. Originally, a macro was just a way to prerecord keystrokes, and that's

still part of what a macro is. However, today, you can extend that code by adding WordBasic commands.

Next, you will record a macro and look at the code it produces. Later in this chapter you will learn how to add your own code to macros.

Recording the email Macro

The world has not yet decided how to spell e-mail. Various authorities use the hyphenated spelling; others omit it. As an introduction to WordBasic, you will record a macro to convert E-mail or e-mail to email for a single document.

1. Start Word and open a document. It doesn't matter if the document contains the word e-mail or not. You cannot record or run a macro unless a document is open.
2. Click on the Tools menu and select Macro. The Macro dialog box appears.

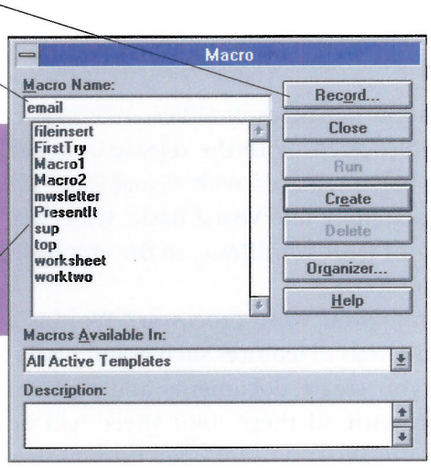


Figure 15-1
The Macro dialog box

3. Enter **email** as the name of the macro as shown in Figure 15-1.
4. Click on Record. The Record Macro dialog box appears. This dialog box lets you insert the macro command you are recording into a menu or toolbar, or assign keyboard strokes to run it. You won't do that now, but remember this for later.
5. Click OK in the Record Macro dialog box to begin recording your keystrokes. You are returned to your document, and the Macro toolbar appears as shown in Figure 15-3. It contains buttons to Stop and Pause recording.

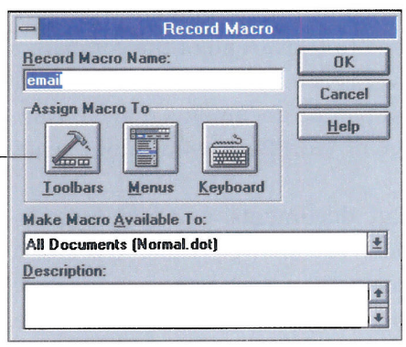


Figure 15-2
The Record Macro dialog box

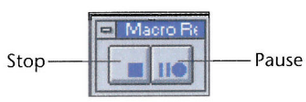
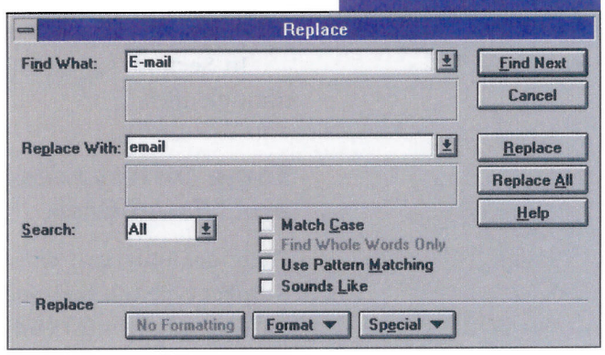


Figure 15-3
The Macro toolbar

6 Press Ctrl+Home to move the cursor to the top of the document. Be sure to do this even if you are already at the top of the document.

7 Click on Edit and select Replace. The Replace dialog box appears as shown in Figure 15-4.

Figure 15-4
The Replace dialog box



8 Enter **E-mail** in the Find What box and **email** in the Replace With box. The options Match Case, Find Whole Words Only, Use Pattern Matching, and Sounds Like, should not be checked. If they are checked, you must uncheck them.

9 Press the Replace All button.

10 Word searches the document for the word *E-mail* or *e-mail* and when either is found, Word replaces it with the word *email,* and reports the result in a message box. Click OK and Close.

11 Click the Stop button on the Macro toolbar to stop recording.

Once the macro is recorded, it can be executed from the Macro dialog box, which is displayed by first selecting the Tools menu, then Macro.

Next, you will examine the code generated by the macro.

1 Click Tools and select Macro. Enter the name **email** in the Macro Name box or click on *email* in the macro list. Do not double-click on the name because that will execute the macro.

2 Click on Edit. A window appears with the code that was generated when you recorded the email macro. The code you see should be similar to that shown below:

```
Sub MAIN
StartOfDocument
EditReplace .Find = "E-mail", .Replace = "email", .Direction = 0,
.MatchCase = 0, .WholeWord = 0, .PatternMatch = 0, .SoundsLike =
0, .ReplaceAll, .Format = 0, .Wrap = 1
End Sub
```

Here, the only lines of code that look like Visual Basic language are the first and last lines. The rest of the commands generated by the macro

are commands that execute Word actions. **StartOfDocument** is the command generated by Ctrl+Home, the command that moved the cursor to the top of the document.

EditReplace is equivalent to clicking the Edit menu and selecting Replace. **.Find**, **.Replace**, and the rest, correspond to actions chosen in the Replace dialog box. They are *named parameters*. WordBasic uses named parameters because most commands have many, many parameters.

In Section 3 you will learn how to adapt this macro to work with multiple files.

Some Differences Between WordBasic and Visual Basic

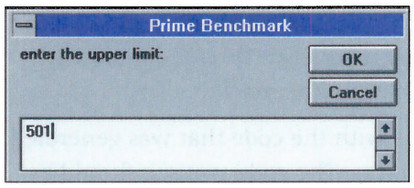
There are only two data types in WordBasic: Number and String. The Number type can represent either whole or decimal numbers. Variables of the String type must end in a dollar sign. The data types of the variables are implicit in the variables' names and are not stated in the program.

True and *False* are not predefined constants in WordBasic. A program that uses *True* and *False* must declare and initialize these variables.

WordBasic's **InputBox$** function is much the same as it is in Visual Basic, but it requires the dollar sign to indicate that the function returns a String value. In Visual Basic, omitting the dollar sign means the function returns a Variant type—a type that doesn't exist in WordBasic.

The **InputBox$** function in WordBasic opens a dialog box that looks quite different from the Input dialog box generated by Visual Basic.

Figure 15-5
The WordBasic InputBox



The **For-Next** and **Assignment** statements are the same as Visual Basic's; however, WordBasic uses a more primitive form of the indefinite loop:

While condition is *True*
... body of loop
Wend

This is similar to the Visual Basic command for the **Do-While** loop, but the syntax is not as versatile as it is in Visual Basic. Visual Basic (VB) allows the following variations of the Do-Loop:

Do
... body
Loop While condition is *True*

This version tests the condition at the end of the loop, thus ensuring that the body of the loop will execute at least once.

Do
... body
Loop Until condition is *True*

In this variation, the loop continues while the condition is false, in other words it continues until the condition is *True*.

Finally,

Do Until condition is *True*
... body
Loop

This version tests the condition at the top of the loop and continues executing the loop until the condition is *True*.

In contrast, WordBasic's **While-Wend** statement tests the condition at the top of the loop and continues while the condition is *True*.

The **If-Then-Else-End If** is the same in WordBasic as it is in Visual Basic.

The **Chr$()** function, which converts an ASCII code to a character, requires the dollar sign.

To join two strings, WordBasic uses the plus sign. The ampersand used in Visual Basic doesn't work in WordBasic.

The **Insert** command inserts text into a document. There will be more about this in the next section.

You *cannot* cut and paste a Visual Basic program into WordBasic, but you *can* cut and paste a WordBasic program into Visual Basic with only minor modifications.

All in all, there are more similarities than there are differences, but those differences can cause long frustrating minutes of debugging when you are writing programs.

Suggestions for How to Learn More

The best way to learn more about WordBasic is to experiment with it. Record a macro. Edit its code. We will do that later in this chapter. Put the cursor on a menu command and press F1, and WordBasic Help will provide an article on that command. Reading the article will probably raise further questions and provide links to other articles. Follow the links. Many of the articles have sample code. Study the code. Copy the

code and paste it into your experimental macros. This kind of interactive learning is the best way to learn about WordBasic.

In addition, Office comes with extensive printed documentation that explains each command. Microsoft will provide a Developer's Handbook. Books are available in bookstores, and sample code is available on the Internet.

There are many sources of information. Find an experienced programmer and ask questions. Post questions on an electronic bulletin board or the Usenet. Posted questions usually receive helpful answers.

VBA for Excel Programming

The Basic built into Excel is called Visual Basic for Applications (VBA). It is much more like Visual Basic than WordBasic. In fact, it has some of the most up-to-date features found in recent versions of Visual Basic. Because of the strong similarity between Visual Basic and VBA, writing code in VBA is much easier than it is in WordBasic. In addition, VBA's programming environment (the way you enter programs) is much more like the Visual Basic you already know.

You can use VBA for simple tasks such as formatting worksheets or changing the report heading in many worksheets. However, an entire complex financial analysis program can also be written with VBA. For example, you could write a VBA program that imports data from an Access database, and using Excel, rework that data into tabular or chart form. Then it can be formatted as a report or exported into a Word document.

Once you start programming in Office, you'll see that everything you do gives you ideas for new programs. In Sections 2 and 4 of this chapter you will find out how to record Excel macros, analyze their code, and extend them with VBA statements into true applications.

QUESTIONS

1. Give two reasons why programs are revised.
2. What is a macro?
3. In the email macro, you typed E-mail in the Find What box and email in the Replace With box. Why doesn't the case of the letter *e* matter in the search and replace operation that is performed?
4. Edit the email macro and delete all the named parameters except **.Find** and **.Replace**. Run the macro. What happens?
5. Why are named parameters used in WordBasic?

6. What are the two WordBasic data types? Are two types enough? How do Visual Basic's many data types make programming easier?
7. The earliest versions of Basic did not have any kind of indefinite loop. **While-Wend**, although an older statement than **Do-While-Loop**, did not appear in the original versions of Basic. Look up the **While-Wend** statement in Visual Basic Help. Can it be used in Visual Basic programs? Write a summary of the **While-Wend** syntax.
8. In Visual Basic, the **Chr()** function returns a Variant type. Because the Variant type is compatible with the String type, there is no problem when you use the result of the function in the place of String. Why won't **Chr()** work in WordBasic?
9. What symbol do you use in WordBasic to join two strings? Does it work in Visual Basic?
10. Describe two ways you can find out more about WordBasic.
11. Describe a situation in which you would use VBA to program an Excel application.

Programming Macros

In Section 2 you will record another macro in Word, examine the code produced, and add a shortcut button to a toolbar to run the macro. Then you will record an Excel macro, examine its code, and add Visual Basic statements to enhance its functionality.

As you saw earlier, recording a macro translates the keystrokes and mouse clicks into commands. In Word, the commands are in WordBasic. In Excel, the commands are in VBA. In Access, the commands are translated into Access Basic.

By studying the commands created and run by macros, you are studying the language that underlies each of the applications: Word, Excel, and Access.

Programming Word Macros

The macro you will record does not have a valid function. The keystrokes have been chosen just to learn how typical Word formatting commands are translated into WordBasic, a technique you will use frequently.

When you have completed all the steps, your document should look like Figure 15-6 (although your default fonts may be different):

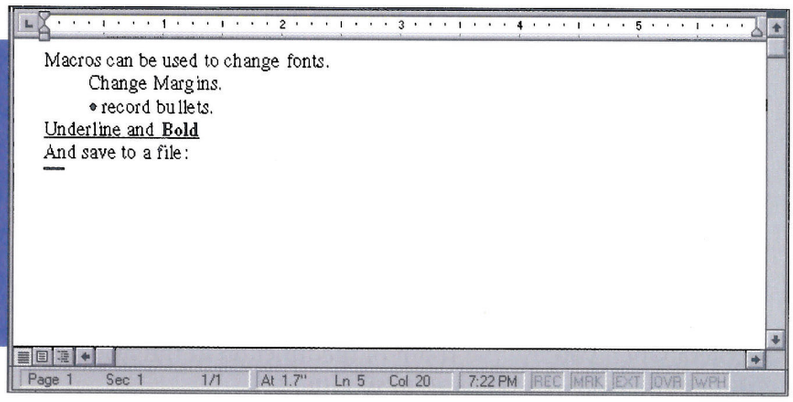


Figure 15-6
The resulting document

Now you will record the macro and examine the code generated.

1. Start Word, and, if necessary, click on the New document button in the toolbar or pull down the File menu and click on New. Accept the document based on the Normal template.
2. Click on Tools and select Macro to open the Macro dialog box.
3. Enter **FirstTry** as the name of the macro.
4. Click on Record to open the Record Macro dialog box.

   The Record Macro dialog box allows you to insert your new macro into a menu or toolbar, or define keystrokes that will execute your command when pressed.
5. Click on the Menus icon to add your new macro to a menu. The Customize dialog box appears with the Menus tab chosen as shown in Figure 15-7.

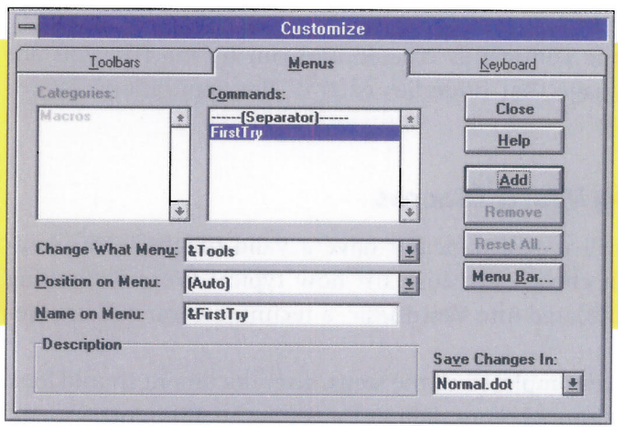


Figure 15-7
The Customize dialog box

As a default, the command is added to the Tools menu, below a separator bar, with the macro's name as its menu entry.

6 Accept these defaults by clicking on Add and then Close. You are returned to your original document with the Macro toolbar present, and each keystroke you make will be recorded.

7 Click on the New document button in the toolbar or pull down the File menu and click on New. Accept the document based on the Normal template.

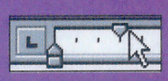
8 Click on the Font window in the toolbar and choose Times New Roman even if it is the default. If this font is not installed in your system, pick another font.

9 Set the Font Size to 12.

10 Enter the text **Macros can be used to change fonts**. Press Enter at the end of the line.

11 On the left edge of the ruler, pull the top indentation marker to the third unit from the left edge. (Click View and select Ruler if needed.) This changes the first line indentation of a paragraph as shown in Figure 15-8.

Figure 15-8
Indentation marker on ruler



12 Enter the text **Change Margins**. Press Enter at the end of the line.

13 Click the Bullet icon on the toolbar and press the spacebar. Enter the text: **record bullets**. Press Enter, then click the Bullet icon to turn bullets off.

14 Press the Underline button to turn on underlining and enter the following text: **Underline and** followed by a space.

15 Press the Bold button to turn on bold face and enter the word **Bold**.

16 Press the Bold button to turn bold face off. Press the Underline button to turn underlining off. Press Enter.

17 Enter the text **And Save to a file:**

18 Click File and select Save As. Enter the file and pathname **c:\temp\firsttry.doc** and click on OK or Save. Your new document is saved. This does not save the macro itself—the macro is saved as a part of the Normal template.

19 Click the Stop button on the Macro toolbar.

20 Close the document.

Before examining the code generated by the macro, execute the macro.

1. Click the Tools menu and select FirstTry, the macro you just recorded.
2. A new document is created and saved that should resemble Figure 15-6.

That's it. You've recorded and run your macro from a menu. Now let's look at the code.

1. Click the Tools menu and select Macro. The Macro dialog box opens again.
2. Click on the name FirstTry or enter it into the Macro Name box.
3. Choose Edit.
4. Your code should be similar to the following:

```
Sub MAIN
FileNewDefault
Font "Times New Roman"
FontSize 12
Insert "Macros can be used to change fonts."
InsertPara
FormatParagraph .FirstIndent = "0.38" + Chr$(34)
Insert "Change Margins"
InsertPara
FormatBulletDefault
Insert " record bullets"
InsertPara
FormatBulletDefault
Underline
Insert "Underline and "
Bold
Insert "Bold"
Bold
Underline
InsertPara
Insert "And Save to a file:"
FileSaveAs .Name = "C:\TEMP\FIRSTTRY.DOC", .Format = 0,
.LockAnnot = 0, .Password = "", .AddToMru = 1, .WritePassword =
"", .RecommendReadOnly = 0, .EmbedFonts = 0,
.NativePictureFormat = 0, .FormsData = 0
End Sub
```

All of these WordBasic commands have something to do with manipulating documents. Many of the commands in the code above are

self-explanatory, like: **Font "Times New Roman"** or **FontSize 12**. Some of the commands toggle features on and off like **FormatBullet-Default**, **Underline** and **Bold**. The first toggle turns the feature on, the second turns it off.

The **Insert** command inserts the text that follows into the document at the current cursor position. The text may be enclosed in double quotes as shown or may come from a string function or a variable. Strings can be joined together with a plus sign. Text longer than a single line wraps around to the next line. The **InsertPara** command appears wherever you press the Enter key when recording the macro.

Some commands, like **FormatParagraph** and **FileSaveAs**, have several optional subparts called arguments. Each argument, also called a *named parameter*, starts with a period.

As you would expect with any Microsoft product, the WordBasic Help system holds a large amount of information about each command. Many commands even have samples that can be copied and pasted into your own code. The only obstacle to using WordBasic Help is that you may not have installed it when you first installed Microsoft Office. One of the options when installing Office is to omit WordBasic Help. Luckily, however, it's easy to install just the Help system part of Office from the original disks.

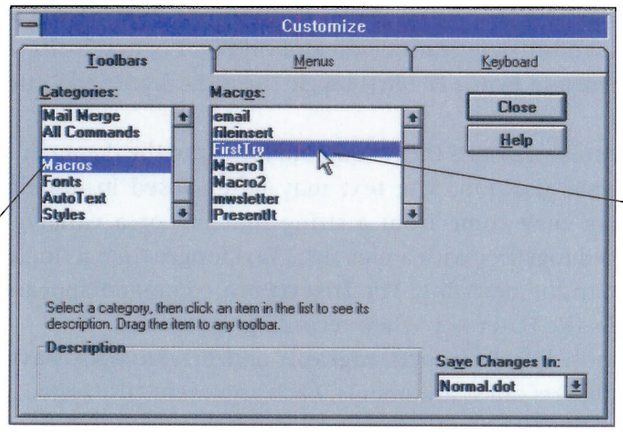
Is there anyone on Earth who knows all these commands? Maybe, but the usual way to learn commands that you don't know is to record them in a separate macro, and then cut and paste them into the macro on which you're working.

Adding *FirstTry* to a Toolbar

In the exercise above you learned how to add a macro to a menu, now you will add a shortcut button for the FirstTry macro on the Formatting toolbar.

1. Start Word. Open a document or click on the New document icon.
2. Click Tools and select Customize. The Customize dialog box appears.
3. Choose the Toolbars tab as shown in Figure 15-9.
4. In the Categories list, scroll down and select Macros.
5. In the Macros list, click and hold on FirstTry. When you click and hold on a macro name in this list, the mouse cursor becomes a square the size of a toolbar button. Drag the mouse to the Formatting toolbar and release the mouse button to the right of the Font Size control. A blank button appears in the toolbar.

Figure 15-9
The Customize dialog box with the Toolbars tab selected



6 The Custom Button dialog box appears, giving you a selection of buttons with icons or a Text Button. Click on Assign. The macro becomes a part of your toolbar with the name of the macro on the Text Button. With this button you can execute FirstTry with a single mouse click.

If you don't want the FirstTry macro as a permanent part of your toolbar, click and drag it off the bar. If you have closed the Customize dialog box, you will have to reopen it to remove the button.

Programming Excel Macros

In WordBasic, the canvas you use to record macros is the document. In Excel, you draw directly onto a worksheet to record macros. The resulting VBA programs are stored in module sheets that become part of the workbook. The module sheet can be moved from workbook to workbook as needed.

VBA allows you to vary the "look and feel" of your programs with many of the same features you used in Visual Basic: input boxes, text boxes, labels, command buttons, list boxes, and so on. These can help turn an Excel worksheet into something that looks and works like a custom application, and is easy for even a nontechnical user to run.

Now, you will record an Excel macro and examine the code generated. This macro builds a worksheet to determine the cost of an auto loan. Later you will enhance the code by adding input boxes.

1 Start Excel.

2 Click Tools and select Record Macro (in Win95, click Record New Macro). The Record New Macro dialog box appears as shown in Figure 15-10. Enter the name **AutoWorksheet** and click OK to

begin recording your keystrokes. You are returned to the worksheet, and the Macro Stop button appears.

3 Move to or click on each cell (including A1) listed below and enter the text, formulas, and further actions shown. Figure 15-11 shows the completed worksheet.

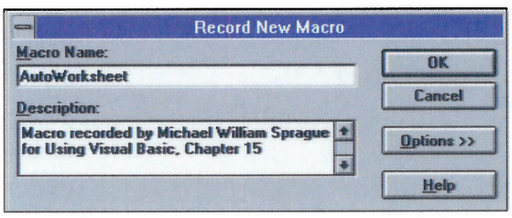


Figure 15-10
Record New Macro dialog box

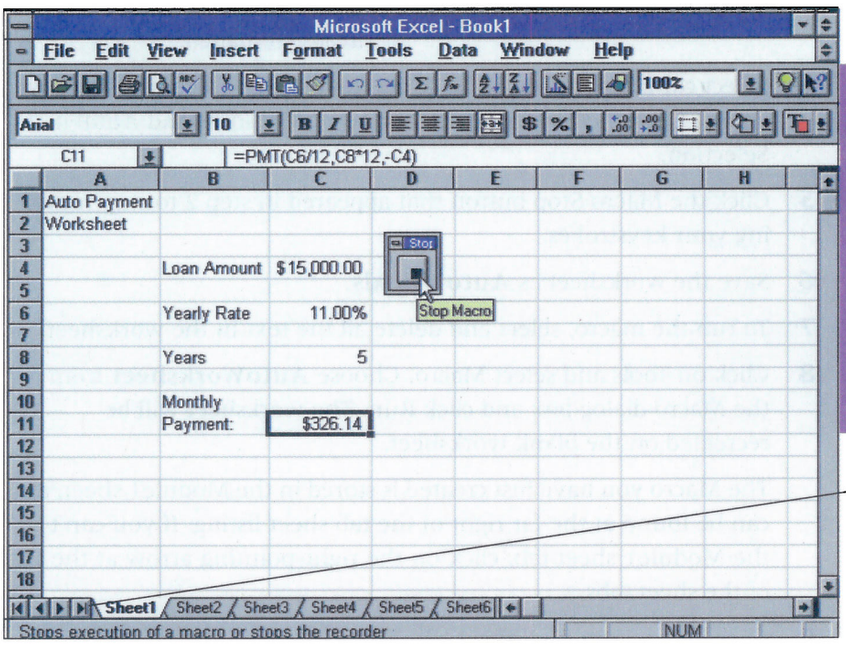


Figure 15-11
The completed worksheet

Cell	*Text*	*Further Action*
A1	Auto Payment	
A2	Worksheet	
B4	Loan Amount	
B6	Yearly Rate	
B8	Years	
B10	Monthly	
B11	Payment:	

Cell	*Text*	*Further Action*
C4	15000	Select cell C4 and click the currency style button on the toolbar.
C6	0.11	Select cell C6 and click the percent style button on the toolbar.
		Click the Increase Decimal button on the toolbar twice.
C8	5	
C11	=PMT(C6/12,C8*12,-C4)	Select C11 and click the currency style button even though it is already formatted.

4. Select columns A through C, using the mouse to select the column letters. Click the Format menu and select Columns and AutoFit Selection.

5. Click the Macro Stop button that appeared in step 2 to stop recording your keystrokes.

6. Save the worksheet as **AutoWrk.xls**.

7. To run the macro, select and delete all the text in the worksheet.

8. Click on Tools and select Macro. Choose **AutoWorksheet** from the Macro dialog box and click Run. The worksheet will be recreated on the blank worksheet.

   The Macro you have just created is stored in the Module1 sheet, and can be found at the far right of the tab sheet listing. If you can't see the Module1 sheet tab, click on the right-pointing arrow at the left of the sheet tabs.

9. Click on the Module1 sheet tab. Your code should be similar to the code below. If you altered the order of keystrokes in the steps above, the commands will be in a different sequence, but that probably won't affect the results.

```
Sub AutoWorksheet()
    Range("A1").Select
    ActiveCell.FormulaR1C1 = "Auto Payment"
    Range("A2").Select
    ActiveCell.FormulaR1C1 = "Worksheet"
    Range("B4").Select
```

```
    ActiveCell.FormulaR1C1 = "Loan Amount"
    Range("B6").Select
    ActiveCell.FormulaR1C1 = "Yearly Rate"
    Range("B8").Select
    ActiveCell.FormulaR1C1 = "Years"
    Range("B10").Select
    ActiveCell.FormulaR1C1 = "Monthly"
    Range("B11").Select
    ActiveCell.FormulaR1C1 = "Payment:"
    Range("C4").Select
    ActiveCell.FormulaR1C1 = "15000"
    Range("C4").Select
    Selection.Style = "Currency"
    Range("C6").Select
    ActiveCell.FormulaR1C1 = "0.11"
    Range("C6").Select
    Selection.Style = "Percent"
    Selection.NumberFormat = "0.0%"
    Selection.NumberFormat = "0.00%"
    Range("C8").Select
    ActiveCell.FormulaR1C1 = "5"
    Range("C11").Select
    ActiveCell.FormulaR1C1 = "=PMT(R[-5]C/12,R[-3]C*12,-R[-7]C)"
    Range("C11").Select
    Selection.Style = "Currency"
    Columns("A:C").Select
    Selection.Columns.AutoFit
    Range("C11").Select
End Sub
```

Selecting a cell by clicking, generates the command **Range("A1")**. The string included inside the quotes is the name of the cell or a range of cells. Once a cell is selected, text can be entered. Labels, numbers, and formulas are all considered text when entered. When calculations are made, numbers are interpreted as values and formulas as mathematical instructions, but when the values are entered, they are all considered text.

The command generated when inserting a label is similar to:

```
ActiveCell.FormulaR1C1 = "Auto Payment"
```

ActiveCell is the name of the currently selected cell. **FormulaR1C1** is the property that reads or sets the text contained in the cell. **R1C1** refers to the naming method used. **R2C12** is interpreted as row 2 and column 12 in this scheme. The label itself is enclosed inside double quotes.

The line:

```
Selection.Style = "Currency"
```

sets the style of the currently selected cell to the Currency format. The styles available are quite similar to the format strings available in the **Format$()** function.

Most of the other lines are self-explanatory. One that isn't is:

```
ActiveCell.FormulaR1C1 = "=PMT(R[-5]C/12,R[-3]C*12,-R[-7]C)"
```

This line enters the payment function, which is a built-in function that calculates the payment required to pay off a loan, given the parameters of interest rate per period, the number of periods, and the value of the loan. The formula looks a little like the one entered in step 3 above: **"=PMT(C6/12,C8*12,-C4)"**. The formula you entered contained references to cells like C6, C8, and C4. The formula generated for the macro uses relative addressing of cells. The cell containing the yearly interest rate, C6, is 5 cells above the cell where the formula is being entered. It is referred to as R[-5]C, meaning, the current row minus 5, same column. Each of the cell references in the formula is named using this scheme.

The loan amount is entered as a negative number. This returns a positive value for the monthly payment.

Enhancing the Auto Loan Macro

Once the macro is recorded, Visual Basic statements can be added to enhance its functionality. In the following steps, you'll add commands to query the user for the loan information using input boxes, and a command button to execute the macro.

1. Open **AutoWrk.xls**, and select Module1 (if it is not already open).
2. Move the cursor just above the lines:

```
Range("C4").Select
ActiveCell.FormulaR1C1 = "15000"
```

This is the code that sets the initial value of the auto loan.

3. Insert the following lines:

```
Dim Principal
Principal = InputBox("Enter the Loan Amount:", "Auto Loan")
```

and change **ActiveCell.FormulaR1C1 = "15000"** to

```
ActiveCell.FormulaR1C1 = Principal
```

You might be tempted to declare *Principal* as a Currency type, but this would require a **Val()** function to convert the text returned by the **InputBox** function to a value before it could be assigned to *Principal*. It is not necessary to do this since the value is converted back to a string when it is assigned to the cell. It is better to enter the value as a string and assign it directly to the cell.

4. Return to the worksheet Sheet1, delete the contents and run the revised macro. As the macro is building the worksheet, it will display an input box and ask you to enter the loan amount. Enter any number, and the macro will finish running.
5. Add code necessary to prompt the user to enter the yearly interest rate as a decimal and the number of years. Run the macro to make sure it works properly.

Now, you will add a command button to the worksheet to run the macro.

1. Select Sheet1. Click on the Drawing tool icon.
2. The Drawing toolbar appears. Click the Create Button icon on the toolbar.
3. Using the click and drag technique, draw a command button on the worksheet.
4. Once the button is drawn, the Assign Macro dialog box appears. This dialog box lets you associate a macro with a command button. Choose AutoWorksheet and click OK.
5. Click on the button text, Button1, and edit it. Replace the text with **Enter Data**.
6. Click the worksheet to leave the edit mode.
7. Click the command button and enter the loan amount, yearly rate, and number of years.
8. Save the program.

Command buttons and input boxes make the user interface clear. This is one way to turn a worksheet into an application.

QUESTIONS

1. In what way is an Office macro more than just a record of keystrokes?
2. What are three of the ways a macro may be executed?
3. What does the Customize dialog box allow you to do?

4. What is a bulleted list?
5. Where are Word macros saved?
6. The commands **FormatBulletDefault**, **Underline**, and **Bold**, are called toggles. What does that mean? And how does a toggle switch work?
7. What command is recorded in a Word macro when the Enter key is pressed?
8. Describe how you would find the WordBasic commands that perform an action such as setting the line spacing in a document to 2.
9. Where are Excel macros saved?
10. How might you use a command button and an **InputBox$** in an Excel program?
11. While recording a macro in Excel, what VBA command is generated when the cell "C3" is clicked?
12. What is an **ActiveCell**?
13. How does the **R1C1** naming convention work? How is it different from the **A1** naming convention?
14. What does this cell reference mean: **R[-5]C**?
15. What VBA statement would query the user to enter his/her first name and assign the result to *Fname*?

3

Section

Programming in WordBasic

Computer software is continually updated. Occasionally a piece of software is discontinued altogether. If this happens to your word processing program, you could be stuck with hundreds of documents saved in an old file format. Or you could find that you want to exchange documents with someone (or some company) that uses a different word processing program. In either of these situations, you need to convert many documents from one file format to another. You can automate this tedious process using WordBasic.

In Section 3 you'll develop a macro to convert any number of files from one file format to another. The program will be developed in incremental steps, each step adding another feature or level of complexity. When you're done, you'll have a useful file format conversion utility, and you'll know a lot more about WordBasic.

Converting from Write to Word

As the first step, you'll record a macro to convert a document created in Write for Windows 3.1 to the Word file format. To do this, the document conversion utilities must be installed (these are usually installed as a normal part of Office installation). Just opening the Write file in Word calls the conversion utility. If the conversion utilities are not installed, you will have to install them.

If you are running Windows 3.1, you will find Write in the Accessories program group. Use Write to create several test files and put them in a temporary directory. Then follow these directions to create a simple macro to open the file in the Write file format and save it in the Word file format.

1. Open Word.
2. If necessary, click on the New document icon in the toolbar, or click File, New, and create a new document using the Normal template.
3. Double-click the dimmed REC button in Word's status bar to open the Record Macro dialog box as shown in Figure 15-12. Enter **WriteToWord** as the Record Macro Name, and click OK.
4. The Macro toolbar appears indicating that your keystrokes will be recorded.
5. Click File menu and select Open. Find and select the .WRI file you want to convert; in this example, c:\windows\readme.wri. The file is opened. As the file is being loaded, a message in the status bar across the bottom of the screen announces that readme.wri is being converted.
6. The file appears in a window in Word.
7. Click File and select Save As. The Save As dialog box appears. Click the Save File as Type down arrow and a drop-down list appears at the bottom of the dialog box, giving you a choice of file formats in which to save, as shown in Figure 15-13. The Word file format is usually the first in the list. Choose the Word file format, and observe that the extension of the file automatically changes from .WRI to .DOC. Click OK to save the file.
8. The file is saved. Close the window displaying the file.
9. Click on the Stop button to stop recording the macro.

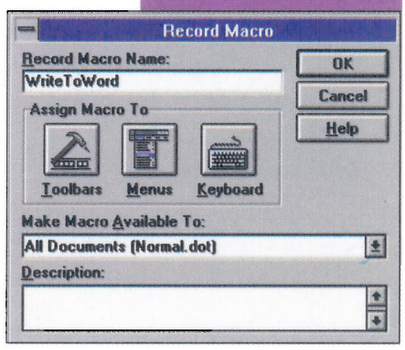


Figure 15-12
The Record Macro dialog box

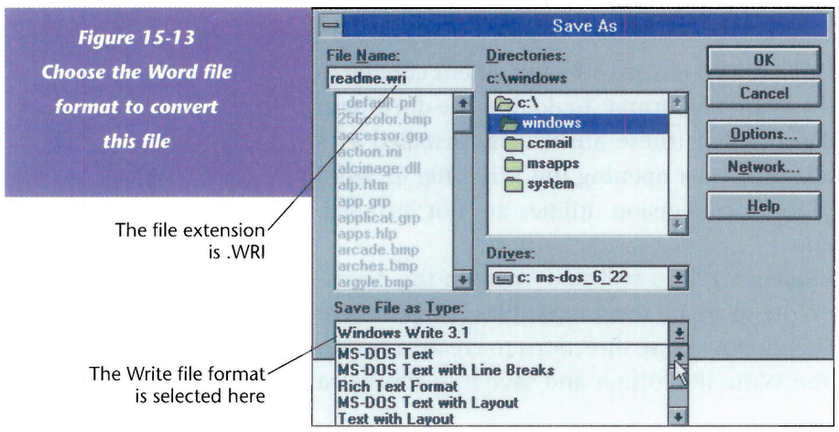


Figure 15-13
Choose the Word file format to convert this file

10 Click Tools and select Macro. Choose WriteToWord, the name of the macro just recorded. Click on Edit. Your code should be similar to the following:

```
Sub MAIN
ChDefaultDir "C:\WINDOWS\", 0
FileOpen .Name = "README.WRI", .ConfirmConversions = 0, .ReadOnly = 0,
.AddToMru = 0, .PasswordDoc = "", .PasswordDot = "", .Revert = 0,
.WritePasswordDoc = "", .WritePasswordDot = ""
FileSaveAs .Name = "README.DOC", .Format = 0, .LockAnnot = 0, .Password =
"", .AddToMru = 1, .WritePassword = "", .RecommendReadOnly = 0,
.EmbedFonts = 0, .NativePictureFormat = 0, .FormsData = 0
FileClose
End Sub
```

The macro will be discussed in four parts: (1) setting the default directory; (2) opening (and automatically converting) the file; (3) saving the converted file; and (4) closing the file and its window.

The line:

```
ChDefaultDir "C:\WINDOWS\", 0
```

changes the default directory to c:\windows\. In this macro, the directory is hard-coded to the location of your test file. However, you could replace the specific string, c:\windows\, with a variable whose value is loaded from an input box or dialog box.

The second part of the macro performs a number of different jobs.

```
FileOpen .Name = "README.WRI", .ConfirmConversions = 0, .ReadOnly = 0,
.AddToMru = 0, .PasswordDoc = "", .PasswordDot = "", .Revert = 0,
.WritePasswordDoc = "", .WritePasswordDot = ""
```

The **FileOpen** statement is self-explanatory. The **.Name** property provides the name of the file to open. Later you will replace this specific string with a string variable that allows you to process many files at the same time.

The **.ConfirmConversions** property is set to 0. This prevents Word from asking the user to confirm the conversion of the incoming file to Word file format. **.ReadOnly** is set to 1 if the file is not to be edited. In this case the file can be edited, so **.ReadOnly** is set to 0. In general, if a property is set to 0, it is disabled. If a property is set to 1, it is enabled. If the **.AddToMru** property is set to 1, it adds the incoming file to the Most Recently Used list of files at the bottom of the File menu. The **.Revert** property determines what happens if the file to be opened is already open. The rest of the properties deal with passwords associated with the document and with the document template. In this example, these properties are not set.

In the third part of the macro, the newly converted file is saved.

```
FileSaveAs .Name = "README.DOC", .Format = 0, .LockAnnot = 0, .Password =
"", .AddToMru = 1, .WritePassword = "", .RecommendReadOnly = 0,
.EmbedFonts = 0, .NativePictureFormat = 0, .FormsData = 0
```

The only properties that concern you are the **.Name** and **.Format** properties. The **.Name** property provides the name of the file to be saved. Notice that the file's extension has been changed. When you chose to save the file as a Word document, the extension was automatically changed to .DOC.

The **.Format** property is used to specify the file type of the document being saved. By changing the value assigned to this property, you can change the file type of the saved document.

Lastly, **FileClose** closes the file window, but it unnecessarily resaves the open document. Saving is part of the **FileClose** command. It should be replaced with **DocClose 2**. This command closes the document window without resaving the document.

Extending the Macro's Usefulness

As it stands, this macro is fairly useless. There is no good reason to record a macro to convert a single file to a different file format unless that particular file is constantly being imported (from a disk or a network) and needs to be converted each time.

So the next step is to extend your macro to read a number of files in a particular file format from one or more directories. Before you can convert the files, you have to find the files. To do this, you'll use the

FileFind statement. Pretend you do not know this and follow along to prepare for extending your macro.

1. Look in WordBasic Help for **FileOpen**. You know about this statement because it was generated by the macro you recorded. There are two ways to find an article about a particular statement: Move the cursor to the word for which you need help and press the F1 function key, or start WordBasic Help and search for **FileOpen**. At the end of many help articles there are references and links to related commands. At the end of **FileOpen**, there is a link to **FileFind**.

2. Click on the **FileFind** link to open its article. When you get there, you will find not only a description of all the **FileFind** options (there are many), but you will also find a link to some sample code. Click the link to open the sample code window.

   The importance of these snippets of code cannot be overemphasized. These samples are the way you get things done in WordBasic. Microsoft's programmers have used WordBasic for years in every kind of way. Many of their programs are found in the code samples attached to help topics.
   Figure 15-14 shows part of the code sample found in the **FileFind** article. It shows how to use **FileFind** to find all the files with a particular extension, and how to load the file names into an array of strings.

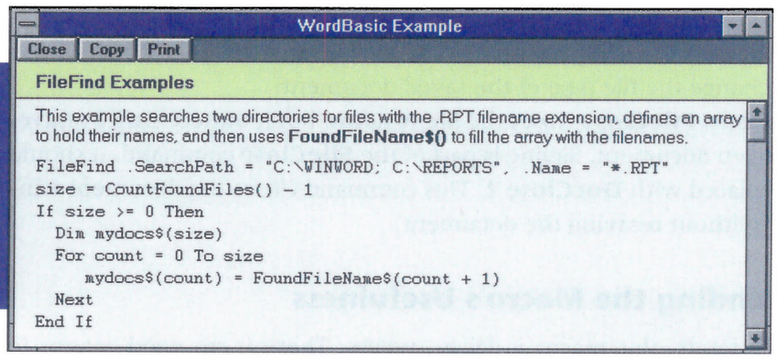


Figure 15-14
The FileFind code sample

   When examining the sample, you'll see a number of other WordBasic statements that you'll want to look at in WordBasic Help.

3. Click on the Copy button in the WordBasic Example window. Another window (in a different font) appears. This window allows you to select and copy code. Copy the following portion of code.

```
FileFind .SearchPath = "C:\WINWORD; C:\REPORTS", .Name = "*.RPT"
size = CountFoundFiles() - 1
If size >= 0 Then
   Dim mydocs$(size)
   For count = 0 To size
      mydocs$(count) = FoundFileName$(count + 1)
   Next
End If
```

4 Paste it into the macro just below the first line.

The **FileFind** code sample shows how to use the **FileFind** command along with other commands to create a list of files that meet the search criteria. In the search shown, files with the .RPT extension in either of two paths are found. Note in the **.SearchPath** parameter that pathnames are surrounded by quotes. If two or more paths are to be searched, their pathnames must be separated by semicolons.

As always, any string in quotes can be replaced with a string variable. This allows the user to set the pathname interactively through an input box or a dialog box.

The **.Name** parameter lists the file name in quotes. The asterisk stands in for any file name. This search string finds files with the .RPT extension and any first name.

The context makes it clear that **CountFoundFiles** is the number of files found in the search. What is not clear is that when the search is complete, an internal list of files will have been assembled. This list can be accessed using **FoundFileName$**.

CountFoundFiles minus one is used to initialize the *size* variable. The **FoundFileName$** list assembled by WordBasic is numbered from 1 to **CountFoundFiles**. The sample code numbers and stores the files with subscripts that start at 0. Thus, **size = CountFoundFiles - 1**. If no files are found, *size* will be -1. If one file is found, *size* will be 0.

The **Dim mydocs$(size)** statement reserves space for the list of file names. The loop that follows transfers file names from the internal list, **FoundFileName$()**, to **mydocs$()**. The subscript for the former is one more than the subscript for the latter.

Now you will modify this code to apply to the conversion process:

1 Change the **FileFind** statement as follows:

```
FileFind .SearchPath = "C:\temp", .Name = "*.wri"
```

NOTE:

If you don't have WordBasic Help installed, it is worth the time it takes to install it. But to keep going right now, type the code shown above in the WriteToWord macro right after the first line.

This uses a single search path, "C:\temp", and selects all files with the .WRI extension. Be sure before you begin that you have a few .WRI files in the c:\temp directory.

2 To see the file selection code actually work, insert the following statements after the line: **mydocs$(count) = FoundFileName$(count + 1).**

```
Insert mydocs$(count)
InsertPara
```

This code displays each file name as it is transferred to **mydocs$()**.

3 End the macro editing session by clicking File, Close. Answer Yes to the Save query. The macro is not finished, but it is time to run it to see how the file selection code works.

4 Run the macro from a blank document. Each file with a .WRI extension should be displayed in the document.

5 As the macro continues to run, it opens the file that has already been converted to Word format and converts it again. The macro may be interrupted with the Esc key at any time. After the macro has displayed the file names, stop execution by pressing Esc.

Converting Multiple Files

Now you'll wrap some code around the rest of the macro to make it open, convert, and save each of the files found in the file search.

1 Click Tools, select Macro, choose WriteToWord, and click on Edit.

When you finish, the macro will have two sections: the first part finds all the file names with the .WRI extension. The second part opens a particular file, converts it to the Word file format, and saves it as a .DOC file. The modifications in this section are made to the second part of the macro.

2 Insert the following line between the first and second section of the macro following the **End If** and preceding the **ChDefaultDir** command.

```
For count = 0 To size
```

This sets up a loop to process each file in the file list constructed in the first part of the macro.

3 Change the directory in the **ChDefaultDir** command to match the directory you've chosen.

```
ChDefaultDir "C:\temp\", 0
```

Word keeps track of a number of different default directories such as document, picture, user templates, and so forth. The 0 following the pathname specifies the default directory for documents.

4 Change the **.Name** parameter of the **FileOpen** command:

```
FileOpen .Name = mydocs$(count), ...
```

This replaces the file name recorded by the original macro with a reference to the list of file names contained in **mydocs$()**. The variable *count* controls access to this list. The rest of the **FileOpen** command remains the same.

5 If you want to change the destination path of the converted files, this is the time to do it. For instance, if you want the newly converted .DOC files to go into the (preexisting) directory c:\windoc\, insert the following line immediately preceding the **FileSaveAs** command:

```
ChDefaultDir "C:\windoc\", 0
```

6 The next change is more complex. You need to save the file with the same first name, but with a new extension, the .DOC extension. To do this you'll use the **FileNameInfo$** function. This function takes the full pathname of the file saved in **mydocs$()** and strips away all but the first name of the file. Other options are available. Replace the **.Name** parameter of the **FileSaveAs** statement with the following:

```
FileSaveAs .Name = FileNameInfo$(FileName$(), 4) + ".DOC", ...
```

The **4** following **FileName$()**, specifies that just the first name of the file is to be returned. **FileName$()** returns the path and file name of the active document. The active document is the one just opened (and converted) in the **FileOpen** statement above. The extension .DOC is joined to the first name of the file. The rest of the path is supplied by the **ChDefaultDir** command. The rest of the **FileSaveAs** command remains unchanged.

7 Remember to change **FileClose** to **DocClose 2**.

8 To finish, put the command **Next count** just above the **End Sub** statement. WordBasic allows you to just put in **Next** without reference to the control variable, *Count*. The entire macro appears below:

```
Sub MAIN
' Part One -- Find files with a .wri extension and save in mydocs$().
FileFind .SearchPath = "C:\temp", .Name = "*.wri"
size = CountFoundFiles() - 1
If size >= 0 Then
   Dim mydocs$(size)
   For count = 0 To size
      mydocs$(count) = FoundFileName$(count + 1)
      Insert mydocs$(count)
      InsertPara
   Next
End If
' Part Two -- Open, convert, and save files in Word format.
For count = 0 To size
   ChDefaultDir "C:\temp\", 0
   FileOpen .Name = mydocs$(count), .ConfirmConversions = 0, .ReadOnly =
0, .AddToMru = 0, .PasswordDoc = "", .PasswordDot = "", .Revert = 0,
.WritePasswordDoc = "", .WritePasswordDot = ""
   FileSaveAs .Name = FileNameInfo$(FileName$(), 4) + ".DOC", .Format =
0, .LockAnnot = 0, .Password = "", .AddToMru = 1, .WritePassword = "",
.RecommendReadOnly = 0, .EmbedFonts = 0, .NativePictureFormat = 0,
.FormsData = 0
   DocClose 2
Next count
End Sub
```

9 Click File, select Close, and choose Yes to save the macro to the Normal template.

10 Run the macro from a blank document. Depending on the speed of your computer and the size of the files opened and closed, you should see the first screenful of each document as it is opened, converted, and closed. When the macro is finished, the .WRI files are unchanged, and a set of matching .DOC files has been created.

This exercise has shown you how to generate a list of files using the **FileFind** command and how to use the list to process a number of files. The same programming techniques can be added to other macros.

QUESTIONS

1. Why might you need to convert a document from one file format to another?

2. Describe two ways to begin recording a Word macro.
3. How would you replace a hard-coded file name in a macro?
4. In the **.AddToMru** property, what does **Mru** stand for?
5. What is the difference between the **FileClose** command and the **DocClose** command?
6. How do you use Word to convert a document in the Write file format to a document in the Word file format?
7. What character is used to separate multiple pathnames in the **.SearchPath** parameter of the **FileFind** statement?
8. What does the **ChDefaultDir** command do? What is the meaning of the number that follows the parameter string?
9. Look up **FileNameInfo$** in WordBasic Help. Describe three of the options available in this function.
10. What does the **.Format** parameter of the **FileSaveAs** do? What does a value of 0 mean?

Programming in Excel

Your next program is a multipart project in Excel. You'll find that the programming statements and environment are much more like what you're used to in Visual Basic, but there's still plenty to learn about interacting with worksheets.

The workbook you create is a database for a car dealership. One sheet contains information on prospective customers: names, income, family members, and so on. Another sheet contains information on used autos in the lot. A final worksheet is a form letter to send to a customer about suitable cars on the lot.

Your program calculates a typical monthly payment for each vehicle and a reasonable payment affordable by the customer. The data is compared and a list of cars that meet the customer's needs is generated. The list is printed along with a letter urging the customer to come in and look over the stock.

Building the Worksheets

Parts of the worksheets are entered by hand and parts are calculated by the program.

1 Start by opening Excel and entering the following customer information on Sheet2. For now, reproduce the table as shown, paying careful attention to row and column numbers. The first line, including the column heads, are part of row 1. The last name, *Sprague*, starts in row 2 and so on. The first column starts in column A.

	Columns					
Row	A	B	C	D	E	F
1	Last Name	First Name	Title	Income	Family Members	First or Second
2	Sprague	Joel	Mr. & Mrs.	32000	4	1
3	Blaine	Briana	Ms.	28500	1	1
4	Laurent	Erin	Miss	12500	1	1
5	Sue	Cathy	Ms.	62000	2	1
6	Weston	Byron	Mr.	41000	1	1
7	Dorant	Jack	Mr. & Mrs.	49500	5	2

2 Format the Income as currency.

3 Select Sheet3 and enter the following information about the used cars on the lot. The labels *Yearly Rate:* and *Years:* are in the top row (row 1). Row 2 is blank. The label, *Make*, is in row 3, column A. The label, *Dodge*, in row 4, column A.

	Columns							
Row	A	B	C	D	E	F	G	H
1		Yearly Rate:	8%		Years:	4		
2								
3	Make	Model	Year	List Price	Wholesale Price	Passengers	Body Style	Color
4	Dodge	Caravan	1991	7500	5200	7	van	white
5	Ford	F150	1992	4500	3800	2	pickup	red
6	Mazda	626	1994	8200	6800	5	sedan	red
7	Renault	Encore	1984	2400	1800	4	sedan	silver
8	Ford	Thunderbird	1989	7200	4800	4	sedan	tan
9	Ford	Mustang	1995	16450	12500	4	sedan	white
10	Olds	Roadmaster	1994	14000	11500	7	wagon	brown
11	Olds	Park Avenue	1996	28000	24000	6	sedan	pale green

4 Save the worksheet with the name **AutoDlr.xls**.

At the top of the worksheet are fields for the yearly interest rate and the number of years to pay off the loan. These figures are used to calculate the monthly payment for each vehicle. If the interest rate or number of years is changed, then the monthly payments also change.

5 Select Sheet1 and create the form letter.

6 Enter the following in row 3, starting in the first column. Put each word in an adjacent column. *Dear* should be in A3, *Laurent* should be in D3.

Dear Miss Erin Laurent,

These, except for *Dear*, are just placeholders. Each will be replaced with specific data for each customer.

7 The next element in Sheet1 is a textbox. Add a textbox to the worksheet by clicking the Textbox button on the Standard toolbar. Draw the box on the screen by clicking in the left corner and dragging to the right corner. The textbox should span the sheet from about column A through column H, and from about row 4 to row 11.

Once the textbox is drawn on the screen, add text by clicking in the box. While the I-beam cursor is visible, a click on the right mouse button allows you to change the font. When the arrow cursor is pointing to the edge of the box, a click of the right mouse button brings up a dialog box that allows you to format the textbox. Figure 15-15 shows the Patterns folder with the borders of the textbox turned off.

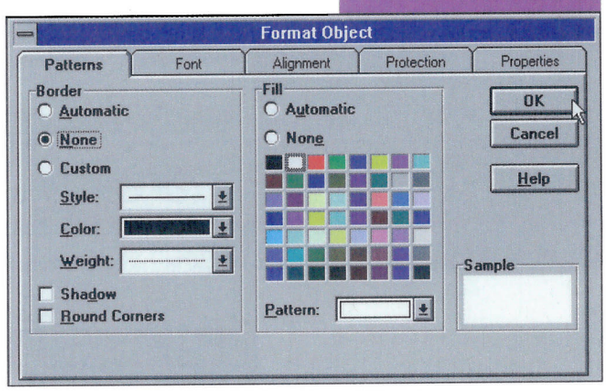


Figure 15-15
The Format dialog box for the textbox object

8 Enter the following paragraph into the Text Box.

```
We would like to draw your attention to the following automobile(s). We've matched your needs against our stock and we know you'll find something you like and can afford in the following list. Please come to the showroom and talk to one of our friendly salespersons to find out how you can drive one of these suburban driven used cars home tonight!
```

9 Below the textbox, starting in column D, row 12, enter the following three rows of information as shown in Figure 15-16. The actual entries below the column headings are cleared and replaced before the worksheet is printed.

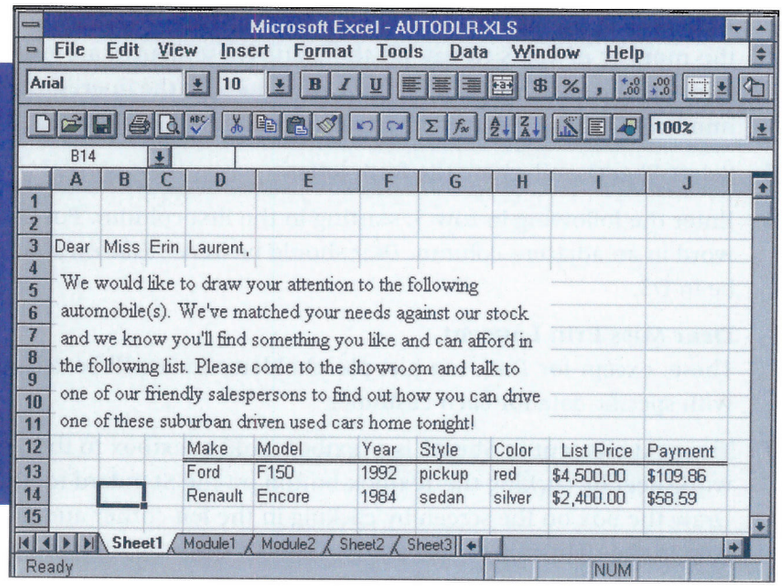


Figure 15-16
The form letter

10 Set up a header and footer for the printed document by clicking on File and selecting Page Setup. Select Header/Footer and enter a header and footer similar to the ones visible in Figure 15-17. Separating the parts of the header with commas left-justifies, centers, and right-justifies three short messages across the top of the page, as shown in Figure 15-17.

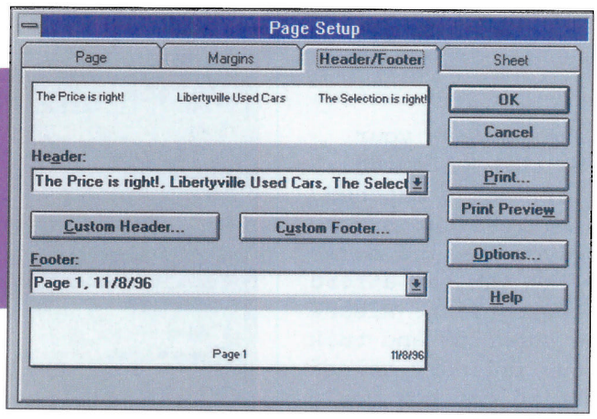


Figure 15-17
The Page Setup dialog box

Generating Possible Payments

The first program calculates the approximate monthly payment that the prospective customer can afford and the typical monthly payment for each car. An additional column is added to each of the tables you entered above for those values.

There are several ways to access the cells of the worksheet, two of these ways are represented in the code below. Of the two, the latter method, using the Cells method, is the preferred technique.

Recall that when a macro is recorded in Excel, a module sheet is created where the code is saved. Modules are part of the workbook for which they are created. A module can also be added using the Insert menu item.

Follow these instructions to write code that adds columns for the payments:

1. Click Insert, select Macro, and choose Module, to add a module to the worksheet.

2. Add the following code to the module. Parts of the code are commented out. The two lines that are commented out below "Next Row" call a macro that you haven't entered yet. The three lines below "For Row = 4 To 11" show the alternate way to address a cell. Later, you'll delete the comment markers from the calls to the **borderline()** macro.

```
Sub GenPossible()
Sheets("Sheet2").Select
Dim Row As Integer
Dim CellAddr As String
For Row = 2 To 7
   CellAddr = "g" & LTrim$(Str$(Row))
   Range(CellAddr).Select
   ActiveCell.FormulaR1C1 = "=(RC[-3]*.72)/12/3/2"
   Selection.Style = "Currency"
Next Row
'borderline ("G1")
' process payments
Sheets("Sheet3").Select
For Row = 4 To 11
   'CellAddr = "i" & LTrim$(Str$(Row))
   'Range(CellAddr).Select
   'ActiveCell.FormulaR1C1 = "=pmt(Yearly_Rate/12,Years*12,-rc[-5])"
   Cells(Row, 9).Formula = "=pmt(Yearly_Rate/12,Years*12,-rc[-5])"
   Selection.Style = "Currency"
Next Row
'borderline ("I3")
End Sub
```

3. Select Sheet2 from the tabs along the left-hand bottom part of the screen.
4. Run the macro by clicking Tools, Macro, the name of the macro GenPossible, and clicking Run.
5. Watch as the macro creates a new column for each of the tables.
6. Save the worksheet, the module will automatically be saved as part of the worksheet.

The **Sheets("Sheet2").Select** command makes sure the macro is working with the right worksheet. Later in the program the Sheets command is executed with "Sheet3" to switch to the worksheet with the list of used cars.

The first section of the program addresses cells with the **A1** addressing method, which assigns a letter to each column and a number to each row. The second section uses the **R1C1** method, which assigns numbers to both rows and columns. In the "A1" method, a string representing the cell's address is built:

```
CellAddr = "g" & LTrim$(Str$(Row))
```

The **LTrim$()** function trims blank characters from the front of a string. This is useful when building a pathname or cell address from a value by joining strings together. This address is used to select the cell with the **Range** command:

```
Range(CellAddr).Select
```

Once the cell is selected, it is filled with an estimate of the family's affordable monthly payment. The algorithm uses the gross yearly income, subtracts 28% for taxes, divides by 12 to scale the amount to the monthly income, divides by 3, using the assumption that housing should cost one third of the family income, and finally divides by 2, assuming a car payment should be no more than half a house payment. This value is assigned to the selected cell:

```
ActiveCell.FormulaR1C1 = "=(RC[-3]*.72)/12/3/2"
```

The formula uses relative addressing. **RC[-3]** refers to the cell three columns to the left of the selected cell. In Sheet2, this is the family's yearly income.

Finally, the entry is formatted in the currency style:

```
Selection.Style = "Currency"
```

The **borderline("G1")** command, which is commented out following the loop, calls a macro recorded separately to insert a column heading.

1 Add the following macro definition to the current module, create a new module, or record a macro with the name *borderline* to insert the column heading Payments. If you record a macro, modify your macro to match the code below:

```
Sub borderline(CellAddr As String)
   Range(CellAddr).Select
   Selection.Borders(xlLeft).LineStyle = xlNone
   Selection.Borders(xlRight).LineStyle = xlNone
   Selection.Borders(xlTop).LineStyle = xlNone
   With Selection.Borders(xlBottom)
      .LineStyle = xlDouble
      .ColorIndex = xlAutomatic
   End With
   Selection.BorderAround LineStyle:=xlNone
   ActiveCell.FormulaR1C1 = "Payment:"
   Range(CellAddr).Select
   With Selection
      .HorizontalAlignment = xlGeneral
      .VerticalAlignment = xlCenter
      .WrapText = False
      .Orientation = xlHorizontal
   End With
End Sub
```

This routine was recorded as a macro and modified to apply to the cell whose address is sent as a parameter.

2 Return to the **GenPossible** procedure and remove the comment marking in front of both occurrences of **borderline()**.

3 Return to Sheet2 and delete the Payments column. Return to Sheet3 and delete the Payments column.

4 Run GenPossible.

5 Save the worksheet.

Creating the Main Routine

To run the main routine, the user selects the last name of the customer to whom he or she wants to send a letter, and then executes the macro. **Main()** starts by saving the original location of the name selected. This

is necessary because various cells on various sheets are selected in turn—the location of the original cell is lost. **ActiveCell** returns the location of the currently selected cell. It is important that the last name of the customer be selected when the macro is run. Other cells are accessed with relative addressing in relation to the cell containing the last name.

1 Enter this code:

```
Sub Main()
   ' -- Save the original selected cell - it's the
   '    customer's last name.
    Dim NameCell
    Set NameCell = ActiveCell
```

2 The next bit of code calls the **GenPossible** routine to generate the Payment columns.

```
   ' -- Generate the possible monthly payments for
   '    the customers and for each vehicle.
    GenPossible
```

3 The next section of code declares variables for the customer information. The location of the customer's last name was saved in *NameCell* above. The commented out **MsgBox** statement was used to debug the code as it was being developed. The **Offset** method is used to address cells relative to the position of a given cell. The parameters indicate a row and column offset. For instance, **NameCell.Offset(0,1)** is the cell to the right of *NameCell*. The 0 indicates to leave the current row number unchanged, while the 1 indicates to add 1 to the current column number.

```
   Dim LastName As String, FirstName As String
   Dim Title As String, Members As Integer
   Dim CarNumber As Integer, Payment As Currency
  ' -- Transfer data from worksheet to variables.
   LastName = NameCell
   'MsgBox "Last name is: " & LastName
   FirstName = NameCell.Offset(0, 1)
   Title = NameCell.Offset(0, 2)
   Members = NameCell.Offset(0, 4)
   CarNumber = NameCell.Offset(0, 5)
   Payment = NameCell.Offset(0, 6)
```

4 Enter the next section of code. It selects Sheet1 and sets the salutation of the letter. The last line uses the **AutoFit** method to size the columns to fit the names and salutation. The textbox, located at the top of the worksheet, is not affected.

```
Sheets("Sheet1").Select
Range("b3").Select
ActiveCell.FormulaR1C1 = Title
Range("C3").Select
ActiveCell.FormulaR1C1 = FirstName
Range("D3").Select
ActiveCell.FormulaR1C1 = LastName & ","
Columns("a:d").Select
Selection.Columns.AutoFit
```

5 Enter the next section of code. It begins by declaring variables to store the information about each vehicle. The **Sheet1Row** variable is used to select the row on sheet1 on which to print vehicle information. Every time a line of information is inserted in Sheet1, the **Sheet1Row** variable is incremented. The monthly payment for the car, *MonthlyPay*, is read and compared to the amount of monthly payment the family can afford. If the car is big enough for the family (**Passengers >= Members**) the information for the vehicle is written to Sheet1.

```
' -- Find appropriate vehicles and list on Sheet1.
Dim Make As String, Model As String, Year As Integer
Dim ListPrice As Currency, Passengers As Integer
Dim BodyStyle As String, BodyColor As String
   Dim MonthlyPay As Currency
Dim Sheet1Row As Integer
Sheet1Row = 13
ClearTable
For Row = 4 To 11
   Sheets("Sheet3").Select
   MonthlyPay = Cells(Row, 9)
   Passengers = Cells(Row, 6)
   If MonthlyPay <= Payment And CarNumber = 1 And Passengers >= Members Then
      MsgBox Str$(Passengers) & " " & Str$(Members)
      Make = Cells(Row, 1)
      Model = Cells(Row, 2)
      Year = Cells(Row, 3)
      ListPrice = Cells(Row, 4)
```

```
            BodyStyle = Cells(Row, 7)
            BodyColor = Cells(Row, 8)
            'MsgBox BodyStyle
            Sheets("Sheet1").Select
            Cells(Sheet1Row, 4) = Make
            Cells(Sheet1Row, 5) = Model
            Cells(Sheet1Row, 6) = Year
            Cells(Sheet1Row, 7) = BodyStyle
            Cells(Sheet1Row, 8) = BodyColor
            Cells(Sheet1Row, 9) = ListPrice
            Cells(Sheet1Row, 10) = MonthlyPay
            Sheet1Row = Sheet1Row + 1
        End If
    Next Row
End Sub
```

6 The call to the ClearTable macro is not valid until you enter the following code. Its purpose is to clear the table of vehicles so the next letter can start over.

```
Sub ClearTable()
    Sheets("Sheet1").Select
    Range("D13:J31").Select
    Selection.ClearContents
End Sub
```

7 The last step is to put a Command button on the worksheet to run the macro. It should go on Sheet2 so the user can select the last name and click the macro. Activate the Drawing toolbar and click on the Command button icon.

8 Draw a Command button on the worksheet. The Assign Macro dialog box appears. Associate the button with Main.

9 Change Button1 to **Build Letter** by clicking on the button's text and typing. Click outside the button to end the editing session. Sheet2 should look like Figure 15-18.

10 Select a last name and click on the Command button.

11 Check the worksheet in Print Preview mode. Make any changes you need to polish the look of the display.

12 Print the worksheet.

13 Save the worksheet.

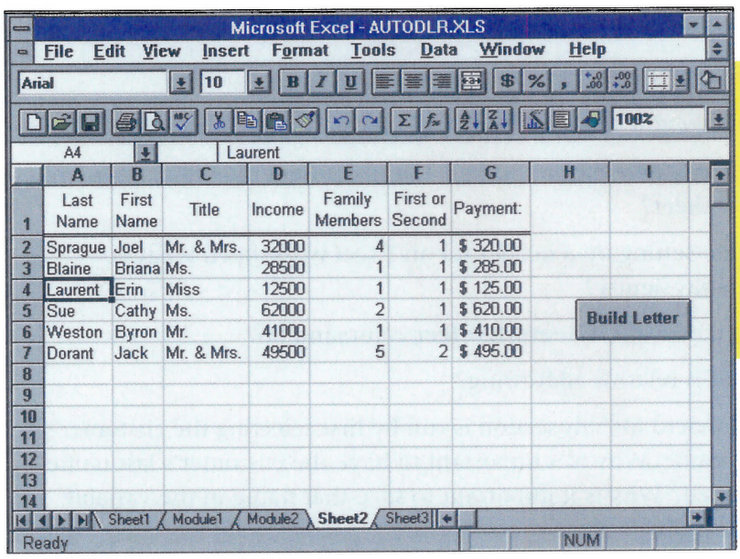


Figure 15-18
Sheet2 with the Command button

There is still a long way to go to make this into a really sharp application.

- Nothing has been done with the information about whether the customer is looking for a second car or a first car.
- Adding customer or vehicle information could be automated.
- Different letters can be defined.

This program, however, does illustrate getting information into and out of a multiple worksheet workbook. Excel is an application rich in features, each of which is available to the VBA programmer. Volumes could be written about those features, but now you have enough information to start writing your own applications.

QUESTIONS

1. How is a textbox added to a worksheet?
2. How do you change the font and formatting of a textbox?
3. In Excel, where can you find a button to put a textbox on a worksheet?
4. When setting up a header for an Excel worksheet, what do commas signify?
5. What does the **Sheets().Select** command do?
6. What is relative addressing?
7. The macro for this section is run by first selecting the customer's last name. Why is it important to have the customer's last name selected? Why is it important to save that name in the variable *NameCell*?
8. How does the Offset method work? Give an example.
9. How are the widths of the columns of worksheet 1 adjusted to accommodate the customer's name?

Summary

Software is revised to add new features, fix bugs, or to completely change the way the program works.

WordBasic is used when you want to automate actions for many files or to customize a document.

When a Word macro is recorded, WordBasic commands are generated and saved in the default template, Normal.

Once a macro is recorded, it can be added to a menu or a toolbar.

The **Do While** and **Do Until** loops are the versatile indefinite loop structures of Visual Basic and VBA. Unfortunately, they are not available in WordBasic. The **While <condition is true> ... Wend** loop is the indefinite loop structure of WordBasic, a throwback to earlier versions of Basic.

There are two data types in WordBasic, String and Numeric. String variables are suffixed with a dollar sign. Numeric variables represent either whole numbers or decimals.

The **InputBox$** and **MsgBox$** functions work the same in WordBasic as they do in Visual Basic, as do the **For-Next**, the assignment statement, and the **If-Then-Else-End If**.

WordBasic uses the plus sign to join strings.

The **Insert** command inserts text into a document. The **InsertPara** command is equivalent to pressing the Enter key.

Visual Basic for Applications, or VBA, is the future of programming for Office. It is the language used to program Excel. It will be the language for all of Office and will be indistinguishable from Visual Basic.

A macro is a way to record a sequence of keystrokes. Using Basic commands, macros can be enhanced to become entire application programs.

Macros are used to explore Basic code generated by events and commands within an application. To find out what code is generated when a file is saved, start recording a macro, save the file, and stop recording. Edit the macro to see what code has been produced. Every event and command has corresponding Basic code.

Macros in Word are saved in templates. Macros saved in the Normal template are global, available to every document created with that template.

In Excel, macros and Basic code are entered into modules. Each module is saved along with its workbook.

There are three ways to access cells in an Excel worksheet: **A1**, **R1C1**, and relative addressing. The **A1** method is most familiar to Excel users. It assigns each column a letter (starting with *A*) and each row a number. The **R1C1** method of addressing cells is used with some worksheets and facilitates automating repetitive actions with loops. Relative addressing is used when accessing a number of cells with a fixed relation to a known cell.

When designing documents where the alignment of elements is critical, use a nonproportionally spaced font, such as Courier New.

"Incremental programming" is a programming technique that starts with a minimal running program, and, as sections of code are added, makes sure the program still runs. This isolates problem areas in the code.

Named Arguments are parameters sent to WordBasic commands.

The **LTrim$()** function trims blank characters from the front of a string. This is useful when building a pathname or cell address from a value by joining strings together.

ActiveCell returns the location of the cell that's currently selected.

Application programs in both WordBasic and Excel are a combination of recorded macros and the programming that connect the pieces of the application together.

1. Favorite Fonts

Record a series of macros for your favorite fonts and font sizes. Install the macros as buttons on the toolbar.

2. Inserting a File

Record a macro that inserts a .DOC file into another .DOC file. Edit the macro and insert a statement with an **InputBox$** function to collect the name of the inserted file from the user.

3. Spellcheck From Top

Record a macro that spell-checks a document starting from the top of the document. Put the macro into the Edit menu.

4. Modified Auto Loan

Modify the Auto Loan Worksheet to display the total amount paid back (the monthly payment times the number of pay periods), and the total finance charge (the total amount paid back minus the loan amount).

5. Generalized WriteToWord

Rewrite the WriteToWord macro to use InputBoxes to collect the path names for both the original files and the converted files.

6. Generalized Email

Add code to the email macro to let the user enter pathnames and extensions for source files and destination files. When the macro is run, it should prompt the user to enter a path and extension for the source files and a path for the destination files.

7. File Printer

Write a macro to print all the .DOC files in a directory entered by the user. Use an **InputBox$** to enter the user's directory.

8. Home Budget

Write an Excel application that helps the user build a home budget worksheet. The program should prompt the user to enter dollar amounts for set categories as well as allow the user to add his or her own. The worksheet should not only list each line item and total them, but also display summary values. For instance, include: all housing expenditures, all charitable contributions, all utilities, and all credit cards.

The program should guide the user through the construction of the worksheet and format a copy for a printed report.