Chapter 1

Introducing Visual Basic for Applications

In addition to Excel's extensive list of worksheet functions and array of calculation tools for scientific and engineering calculations, Excel contains a programming language that allows users to create procedures, sometimes referred to as macros, that can perform even more advanced calculations or that can automate repetitive calculations.

Excel's first programming language, Excel 4 Macro Language (XLM) was introduced with version 4 of Excel. It was a rather cumbersome language, but it did provide most of the capabilities of a programming language, such as looping, branching and so on. This first programming language was quickly superseded by Excel's current programming language, Visual Basic for Applications, introduced with version 5 of Excel. Visual Basic for Applications, or VBA, is a "dialect" of Microsoft's Visual Basic programming language, a dialect that has keywords to allow the programmer to work with Excel's workbooks, worksheets, cells, charts, etc. At the same time, Microsoft introduced a version of Visual Basic for Word; it was called WordBasic and had keywords for characters, paragraphs, line breaks, etc. But even at the beginning, Microsoft's stated intention was to have one version of Visual Basic that could work with all its applications: Excel, Word, Access and PowerPoint. Each version of Microsoft Office has moved closer to this goal.

COPYRIGHTED MATERIAL

The Visual Basic Editor

To create VBA code, or to examine existing code, you will need to use the Visual Basic Editor. To access the Visual Basic Editor, choose **Macro** from the **Tools** menu and then **Visual Basic Editor** from the submenu.

The Visual Basic Editor screen usually contains three important windows: the Project Explorer window, the Properties window and the Code window, as shown in Figure 1-1. (What you see may not look exactly like this.)

The Code window displays the active module sheet; each module sheet can contain one or several VBA procedures. If the workbook you are using does not

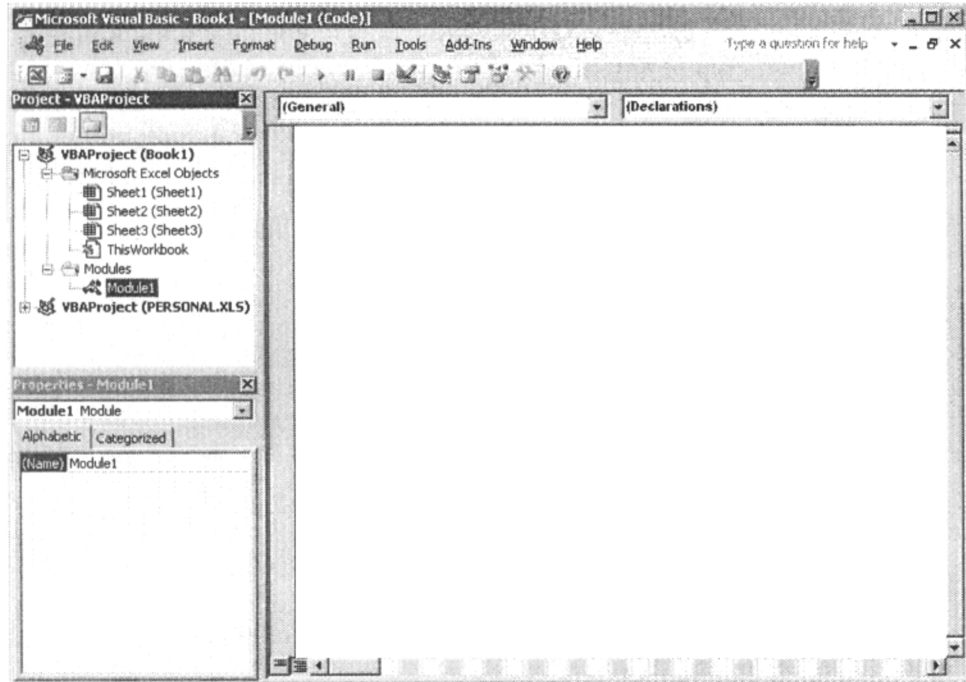
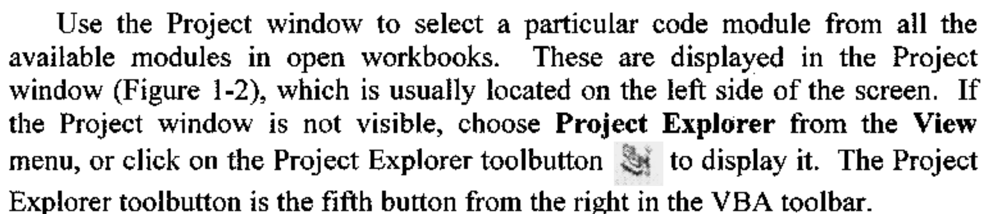


Figure 1-1. The Visual Basic Editor window.

contain any module sheets, the Code window will be empty. To insert a module sheet, choose **Module** from the **Insert** menu. A folder icon labeled Modules will be inserted; if you click on this icon, the module sheet Module1 will bedisplayed. Excel gives these module sheets the default names Module1, Module2 and so on.

Use the Project window to select a particular code module from all the available modules in open workbooks. These are displayed in the Project window (Figure 1-2), which is usually located on the left side of the screen. If the Project window is not visible, choose **Project Explorer** from the **View** menu, or click on the Project Explorer toolbutton  to display it. The Project Explorer toolbutton is the fifth button from the right in the VBA toolbar.

In the Project Explorer window you will see a hierarchy tree with a node for each open workbook. In the example illustrated in Figure 1-2, a new workbook, Book1, has been opened. The node for Book1 has a node (a folder icon) labeled Microsoft Excel Objects; click on the folder icon to display the nodes it contains—an icon for each sheet in the workbook and an additional one labeled ThisWorkbook. If you double-click on any one of these nodes you will display the code sheet for it. These code sheets are for special types of procedures called automatic procedures or event-handler procedures, which are not covered in this

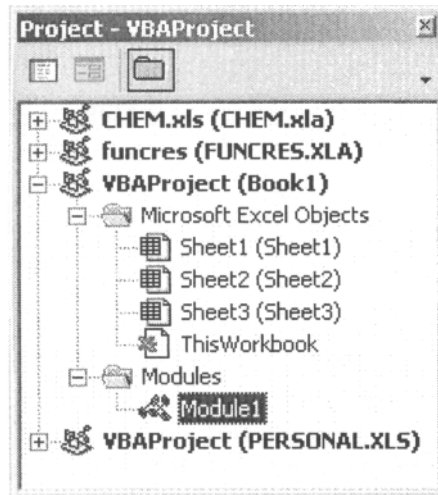


Figure 1-2. The VBE Project Explorer window.

book. Do not use any of these sheets to create the VBA procedures described in this book. The hierarchy tree in Figure 1-2 also shows a Modules folder, containing one module sheet, Module1.

The Properties window will be discussed later. Right now, you can press the Close button to get rid of it if you wish.

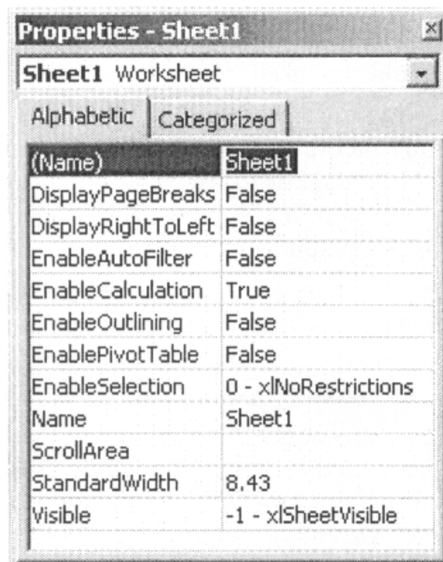


Figure 1-3. The Properties window.

Visual Basic Procedures

VBA macros are usually referred to as *procedures.* They are written or recorded on a *module* sheet. A single module sheet can contain many procedures.

There Are Two Kinds of Macros

There are two different kinds of procedures: **Sub** procedures, called command macros in the older XLM macro language, and **Function** procedures, called function macros in the XLM macro language and often referred to as custom functions or user-defined functions.

Although these procedures can use many of the same set of VBA commands, they are distinctly different. **Sub** procedures can automate any Excel action. For example, a **Sub** procedure might be used to create a report by opening a new worksheet, copying selected ranges of cells from other worksheets and pasting them into the new worksheet, formatting the data in the new worksheet, providing headings, and printing the new worksheet. **Sub** procedures are usually "run" by selecting **Macro** from the **Tools** menu. They can also be run by means of an assigned shortcut key, by being called from another procedure, or in several other ways.

Function procedures augment Excel's library of built-in functions by adding user-defined functions. A custom or user-defined function is used in a worksheet in the same way as a built-in function like, for example, Excel's SQRT function. It is entered in a formula in a worksheet cell, performs a calculation, and returns a result to the cell in which it is located. For example, a custom function named FtoC could be used to convert Fahrenheit temperatures to Celsius.

Custom functions can't incorporate any of VBA's "action" commands. No experienced user of Excel would try to use the SQRT function in a worksheet cell to calculate the square root of a number and also open a new workbook and insert the result there; custom functions are no different.

However, both kinds of macro can incorporate decision-making, branching, looping, subroutines and many other aspects of programming languages.

The Structure of a Sub Procedure

The structure of a **Sub** procedure is shown in Figure 1-4. The procedure begins with the keyword **Sub** and ends with **End Sub**. It has a ProcedureName, a unique identifier that you assign to it. The name should indicate the purpose of the function. The name can be long, since after you type it once you will probably not have to type it again. A **Sub** procedure has the possibility of using one or more arguments, Argument1, etc, but for now we will not create **Sub**

procedures with arguments. Empty parentheses are still required even if a **Sub** procedure uses no arguments.

```
Sub ProcedureName(Argument1, ...)
    VBA statements
End Sub
```

Figure 1-4. Structure of a **Sub** procedure.

The Structure of a Function Procedure

The structure of a **Function** procedure is shown in Figure 1-5. The procedure begins with the keyword **Function** and ends with **End Function**. It has a FunctionName, a unique identifier that you assign to it. The name should be long enough to indicate the purpose of the function, but not too long, since you will probably be typing it in your worksheet formulas. A **Function** procedure usually takes one or more arguments; the names of the arguments should also be descriptive. Empty parentheses are required even if a **Function** procedure takes no arguments.

```
Function FunctionName(Argument1, ...)
    VBA statements
    FunctionName = result
End Function
```

Figure 1-5. Structure of a user-defined function.

The function's *return statement* directs the procedure to return the result to the caller (usually the cell in which the function was entered). The return statement consists of an assignment statement in which the name of the function is equated to a value, for example,

```
FunctionName = result
```

Using the Recorder to Create a Sub Procedure

Excel provides the Recorder, a useful tool for creating command macros. When you choose **Macro** from the **Tools** menu and **Record New Macro...** from the submenu, all subsequent menu and keyboard actions will be recorded until you press the Stop Macro button or choose **Stop Recording** from the **Macro** submenu. The Recorder is convenient for creating simple macros that involve only the use of menu or keyboard commands, but you can't use it to incorporate logic, branching or looping.

The Recorder creates Visual Basic commands. You don't have to know anything about Visual Basic to record a command macro in Visual Basic. This provides a good way to gain some familiarity with Visual Basic.

To illustrate the use of the Recorder, let's record the action of applying scientific number formatting to a number in a cell. First, select a cell in a worksheet and enter a number. Now choose **Macro** from the **Tools** menu, then **Record New Macro...** from the submenu. The Record Macro dialog box (Figure 1-6) will be displayed.

The Record Macro dialog box displays the default name that Excel has assigned to this macro: Macro1, Macro2, etc. Change the name in the Macro Name box to ScientificFormat (no spaces are allowed in a name). The "Store Macro In" box should display This Workbook (the default location); if not, choose This Workbook. Enter "e" in the box for the shortcut key, then press OK.

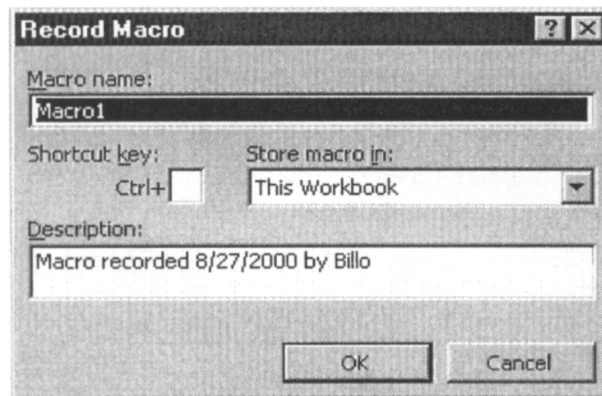


Figure 1-6. The Record Macro dialog box.

The Stop Recording toolbar will appear (Figure 1-7), indicating that a macro is being recorded. If the Stop Recording toolbar doesn't appear, you can always stop recording by using the **Tools** menu (in the Macro submenu the **Record New Macro...** command will be replaced by **Stop Recording**).

Figure 1-7. The Stop Recording toolbar.

Now choose **Cells...** from the **Format** menu, choose the Number tab and choose Scientific number format, then press OK. Finally, press the Stop Recording button.

To examine the macro code that you have just recorded, choose **Macro** from the **Tools** menu and **Visual Basic Editor** from the submenu. Click on the node for the module in the active workbook. This will display the code module sheet containing the Visual Basic code. The macro should look like the example shown in Figure 1-8.

```
Sub ScientificFormat()
'
' ScientificFormat Macro
' Macro recorded 6/22/2004 by Boston College
'
' Keyboard Shortcut: Ctrl+e
'
    Selection.NumberFormat = "0.00E+00"
End Sub
```

Figure 1-8. Macro for scientific number-formatting, recorded in VBA.

This macro consists of a single line of VBA code. You'll learn about Visual Basic code in the chapters that follow.

To run the macro, enter a number in a cell, select the cell, then choose **Macro** from the **Tools** menu, choose **Macros...** from the submenu, select the ScientificFormat macro from the Macro Name list box, and press Run. Or you can simply press the shortcut key combination that you designated when you recorded the macro (CONTROL+e in the example above). The number should be displayed in the cell in scientific format.

The Personal Macro Workbook

The Record Macro dialog box allows you to choose where the recorded macro will be stored. There are three possibilities in the "Store Macro In" list box: This Workbook, New Workbook and Personal Macro Workbook. The Personal Macro Workbook (PERSONAL.XLS in Excel for Windows, or Personal Macro Workbook in Excel for the Macintosh) is a workbook that is automatically opened when you start Excel. Since only macros in open workbooks are available for use, the Personal Macro Workbook is the ideal location for macros that you want to have available all the time.

Normally the Personal Macro Workbook is hidden (choose **Unhide...** from the **Window** menu to view it). If you don't yet have a Personal Macro Workbook, you can create one by recording a macro as described earlier, choosing Personal Macro Workbook from the "Store Macro In" list box.

As you begin to create more advanced Sub procedures, you'll find that the Recorder is a useful tool to create fragments of macro code for incorporation into your procedure. Instead of poring through a VBA reference, or searching through the On-Line VBA Help, looking for the correct command syntax, simply turn on the Recorder, perform the action, and look at the code produced. You may find that the Recorder doesn't always produce exactly what you want, or perhaps the most elegant code, but it is almost always useful.

Note that, since the Recorder only records actions, and **Function** procedures can't perform actions, the Recorder won't be useful for creating **Function** procedures.

Running a Sub Procedure

In the preceding example, the macro was run by using a shortcut key. There are a number of other ways to run a macro. One way is to use the Macro dialog box. Again, enter a number in a cell, select the cell, then choose **Macro** from the **Tools** menu and **Macros...** from the submenu. The Macro dialog box will be displayed (Figure 1-9). This dialog box lists all macros in open workbooks (right now we only have one macro available). To run the macro, select it from the list, then press the Run button.

Assigning a Shortcut Key to a Sub Procedure

If you didn't assign a shortcut key to the macro when you recorded it, but would like to do so "after the fact," choose **Macro** from the **Tools** menu and **Macros...** from the submenu. Highlight the name of the macro in the Macro Name list box, and press the Options... button. You can now enter a letter for the shortcut key: CONTROL+<key> or SHIFT+CONTROL+<key> in Excel for

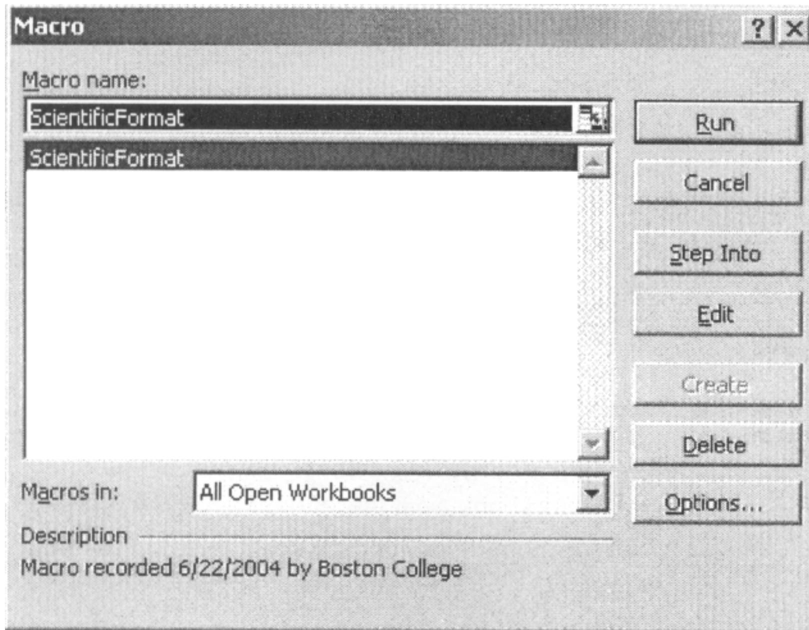


Figure 1-9. The Macro dialog box.

Windows, OPTION+COMMAND+<key> or SHIFT+OPTION+COMMAND+<key> in Excel for the Macintosh.

Entering VBA Code

Of course, most of the VBA code you create will not be recorded, but instead entered at the keyboard. As you type your VBA code, the Visual Basic Editor checks each line for syntax errors. A line that contains one or more errors will be displayed in red, the default color for errors. Variables usually appear in black. Other colors are also used; comments (see later) are usually green and some VBA keywords (**Function**, **Range**, etc.) usually appear in blue. (These default colors can be changed if you wish.)

If you type a long line of code, it will not automatically wrap to the next line but will simply disappear off the screen. You need to insert a *line-continuation character* (the underscore character, but you must type a space followed by the underscore character followed by ENTER) to cause a line break in a line of VBA code, as in the following example:

```
Worksheets("Sheet1").Range("A2:B7").Copy _
    (Worksheets("Sheet2").Range("C2"))
```

The line-continuation character can't be used within a string, i.e., within quotes.

I recommend that you type the module-level declaration **Option Explicit** at the top of each module sheet, before any procedures. **Option Explicit** forces you to declare all variables using **Dim** statements; undeclared variables produce an error at compile time.

When you type VBA code in a module, it's good programming practice to use TAB to indent related lines for easier reading, as shown in the following procedure.

```
Sub Initialize()
For J = 1 To N
    P(J) = 0
Next J
End Sub
```

Figure 1-10. A simple VBA **Sub** procedure.

In order to produce a more compact display of a procedure, several lines of code can be combined in one line by separating them with colons. For example, the procedure in Figure 1-10 can be replaced by the more compact one in Figure 1-11 or even by the one in Figure 1-12.

```
Sub Initialize()
For J = 1 To N: P(J) = 0: Next J
End Sub
```

Figure 1-11. A **Sub** procedure with several statements combined.

```
Sub Initialize(): For J = 1 To N: P(J) = 0: Next J: End Sub
```

Figure 1-12. A **Sub** procedure in one line.

Creating a Simple Custom Function

As a simple first example of a **Function** procedure, we'll create a custom function to convert temperatures in degrees Fahrenheit to degrees Celsius.

Function procedures can't be recorded; you must type them on a module sheet. You can have several macros on the same module sheet, so if you recorded the ScientificFormat macro earlier in this chapter, you can type this custom function procedure on the same module sheet. If you do not have a module sheet available, insert one by choosing **Module** from the **Insert** menu.

Type the macro as shown in Figure 1-13. DegF is the argument passed by the function from the worksheet to the module (the Fahrenheit temperature); the single line of VBA code evaluates the Celsius temperature and returns the result to the *caller* (in this case, the worksheet cell in which the function is entered).

```
Function FtoC(DegF)
  FtoC = (DegF - 32) * 5 / 9
End Function
```

Figure 1-13. Fahrenheit to Celsius custom function.

A note about naming functions and arguments: function names should be short, since you will be typing them in Excel formulas (that's why Excel's square-root worksheet function is SQRT) but long enough to convey information about what the function does. In contrast, command macro names can be long, since command macros are run by choosing the name of the macro from the list of macros in the Macro Run dialog box, for example.

Argument names can be long, since you don't type them. Longer names can convey more information, and thus provide a bit of self-documentation. (If you look at the arguments used in Excel's worksheet functions, you'll see that single letters are usually not used as argument names.)

Using a Function Macro

A custom function is used in a worksheet formula in exactly the same way as any of Excel's built-in functions. The workbook containing the custom function must be open.

Figure 1-14 shows how the FtoC custom function is used. Cell A2 contains 212, the argument that the custom function will use. Cell B2 contains the formula with the custom function. You can enter the function in cell B2 by

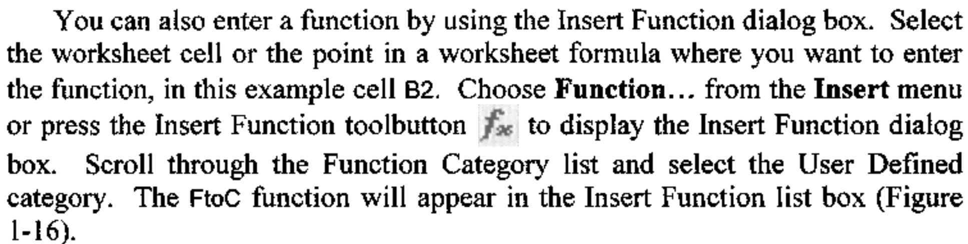
typing it (Figure 1-14). When you press enter, the result calculated by the function appears in the cell (Figure 1-15).

	A	B
1	T, °F	T, °C
2	212	=FtoC(A2)

Figure 1-14. Entering the custom function.

	A	B
1	T, °F	T, °C
2	212	100

Figure 1-15. The function result.



You can also enter a function by using the Insert Function dialog box. Select the worksheet cell or the point in a worksheet formula where you want to enter the function, in this example cell B2. Choose **Function...** from the **Insert** menu or press the Insert Function toolbutton *fx* to display the Insert Function dialog box. Scroll through the Function Category list and select the User Defined category. The FtoC function will appear in the Insert Function list box (Figure 1-16).

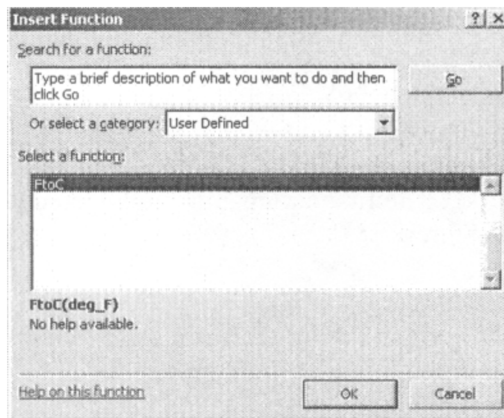


Figure 1-16. The Paste Function dialog box.

When you press OK, the Function Arguments dialog box (Figure 1-17) will be displayed. Enter the argument, or click on the cell containing the argument to enter the reference (cell A2 in Figure 1-14), then press the OK button.

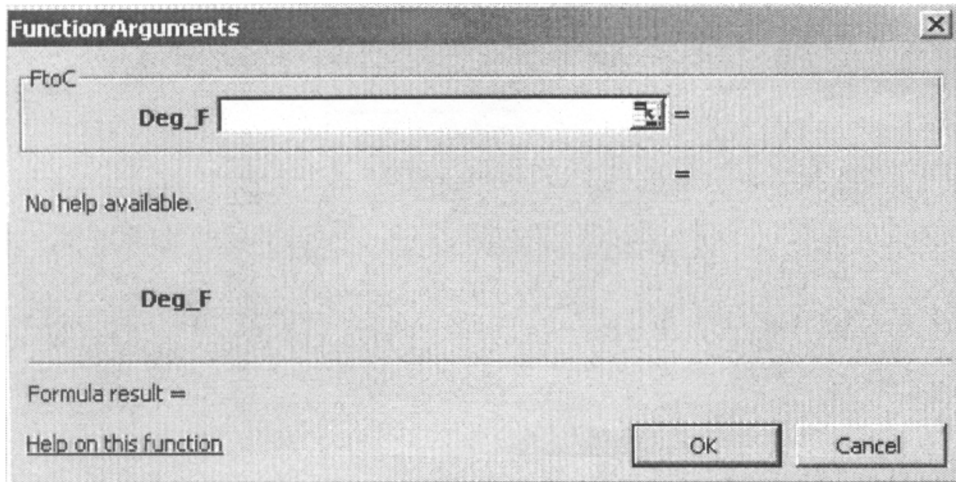


Figure 1-17. The Function Arguments dialog box.

A Shortcut to Enter a Function

You can enter a function without using Insert Function, but still receive the benefit provided by the Function Arguments screen. This is useful if the function takes several (perhaps unfamiliar) arguments. Simply type "=" followed by the function name, with or without the opening parenthesis, and then press CONTROL+A to bypass the Insert Function dialog box and go directly to the Function Arguments dialog box.

If you press CONTROL+SHIFT+A, you bypass both the Insert Function dialog box and the Function Arguments. The function will be displayed with its placeholder argument(s). The first argument is highlighted so that you can enter a value or reference (Figure 1-18).

	A	B	C
1	T, °F	T, °C	
2	212	=FtoC(deg_F)	

Figure 1-18. Entering a custom function by using CONTROL+SHIFT+A.

Unfortunately, if you're entering the custom function in a different workbook than the one that contains the custom function, the function name must be entered as an external reference (e.g., Book1.XLS!FtoC). This can make typing the function rather cumbersome, and it means that you'll probably enter the function by using Excel's Insert Function. But, see "Creating Add-In Function Macros" in Chapter 2.

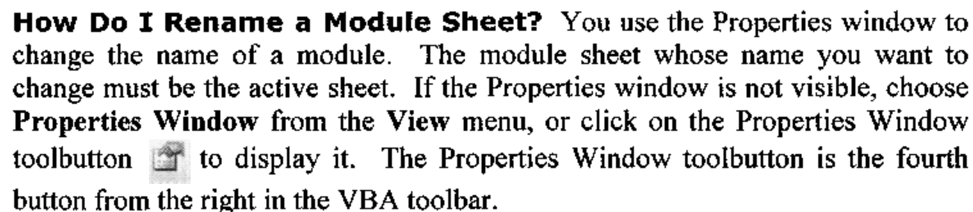
Some FAQs

Here are answers to some Frequently Asked Questions about macros.

I Recorded a Command Macro. Where Did It Go? If you have trouble locating the code module containing your macro, here's what to do "when all else fails": choose **Macro** from the **Tools** menu and **Macros...** from the submenu. Highlight the name of the macro in the Macro Name list box, and press the Edit button. This will display the code module sheet containing the Visual Basic code.

I Can't Find My Function Macro. Where Did It Go? If you're looking in the list of macros in the Macro Name list box, you won't find it there. Only command macros (macros that can be **Run**) are listed. Function macros are found in a different place: in the list of user-defined functions in the Insert Function dialog box. (Choose **Function...** from the **Insert** menu and scroll through the Function Category list and select the User Defined category.)

How Do I Rename a Macro? To rename a **Sub** or **Function** procedure, access the Visual Basic Editor and click on the module containing the procedure. The name of the macro is in the first line of code, immediately following the **Sub** or **Function** keyword. Simply edit the name. Again, no spaces are allowed in the name.



How Do I Rename a Module Sheet? You use the Properties window to change the name of a module. The module sheet whose name you want to change must be the active sheet. If the Properties window is not visible, choose **Properties Window** from the **View** menu, or click on the Properties Window toolbutton to display it. The Properties Window toolbutton is the fourth button from the right in the VBA toolbar.

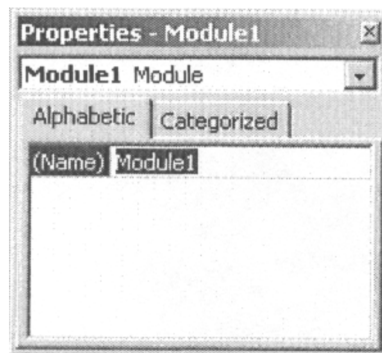


Figure 1-19. Changing the name of a module by using the Properties window.

When you display the Properties window, you will see the single property of a module sheet, namely its name, displayed in the window. Simply double-click on the name (here, Module1), edit the name, and press Enter. No spaces are allowed in the name.

How Do I Add a Shortcut Key? If you decide to add a shortcut key to a command macro "after the fact," choose **Tools→Macro→Macros....** In the Macro Name list box, click on the name of the macro to which you want to add a shortcut key, then press the Options button. In the Shortcut Key box, enter a letter, either lower- or uppercase. To run the macro, use CTRL+<letter> for a lowercase shortcut key, or CTRL+SHIFT+<letter> for uppercase.

Warning: The shortcut key will override a built-in shortcut key that uses the same letter. For example, if you use CTRL+s for the ScientificFormat macro, you won't be able to use CTRL+s for "Save." This will be in effect as long as the workbook that contains the macro is open.

How Do I Save a Macro? A macro is part of a workbook, just like a worksheet or a chart. To save the macro, you simply **Save** the workbook.

Are There Some Shortcut Keys for VBA? Yes, there are several. Here's a useful one: you can toggle between the Excel spreadsheet and the VBA Editor by pressing ALT+F11. A list of shortcut keys for VBA programming is found in Appendix 2.