

EXCEL MACROS

CHAPTER 6



CHAPTER 6 SUMMARY

Overview of Macros

Macro Recorder

Relative vs. Absolute References in Macros

When to Use the Macro Recorder

VBA Programming Basics

Using the Visual Basic Editor

Examples of Programmed Excel Macros

Creating a Macro to Insert the Round Function

Creating User Defined Functions

The Month Book Macro

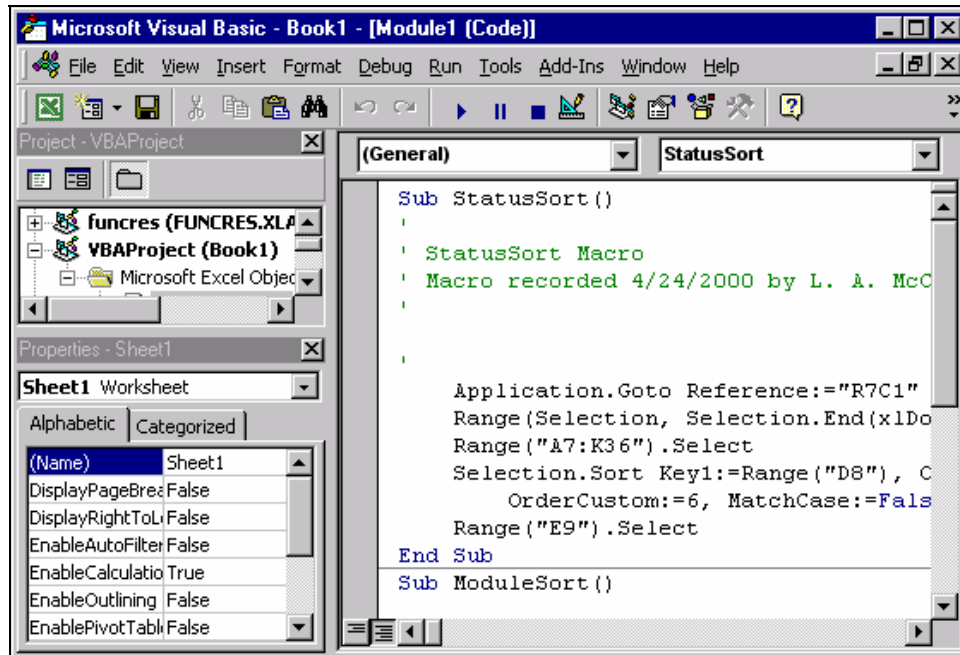
Footer With Path and Filename Macro

OVERVIEW OF MACROS

Starting with Excel version 5 (1993) Microsoft implemented Visual Basic for Applications (VBA) as the Excel macro programming language. While Excel 4 and earlier users were at first distressed that their macros would no longer work (without being converted), this change has proved to be a positive one in the long run.

Visual Basic is a high end programming language with a large following. For Excel users this means that macros can be used to accomplish almost anything. It also means that finding an Excel macro programmer is relatively easy as all you have to do is find someone who can program in Visual Basic.

On the other hand, for non-programmers, macros are generally more confusing than they were in Excel 4 and in Lotus 1-2-3. That is because the keystroke macros (i.e. a series of keystrokes recorded in a cell as a label) no longer exist. All Excel macros have to be Visual Basic programs and the only way to see and/or edit these macros is to use the Visual Basic Editor.



The VBA Editor in Excel

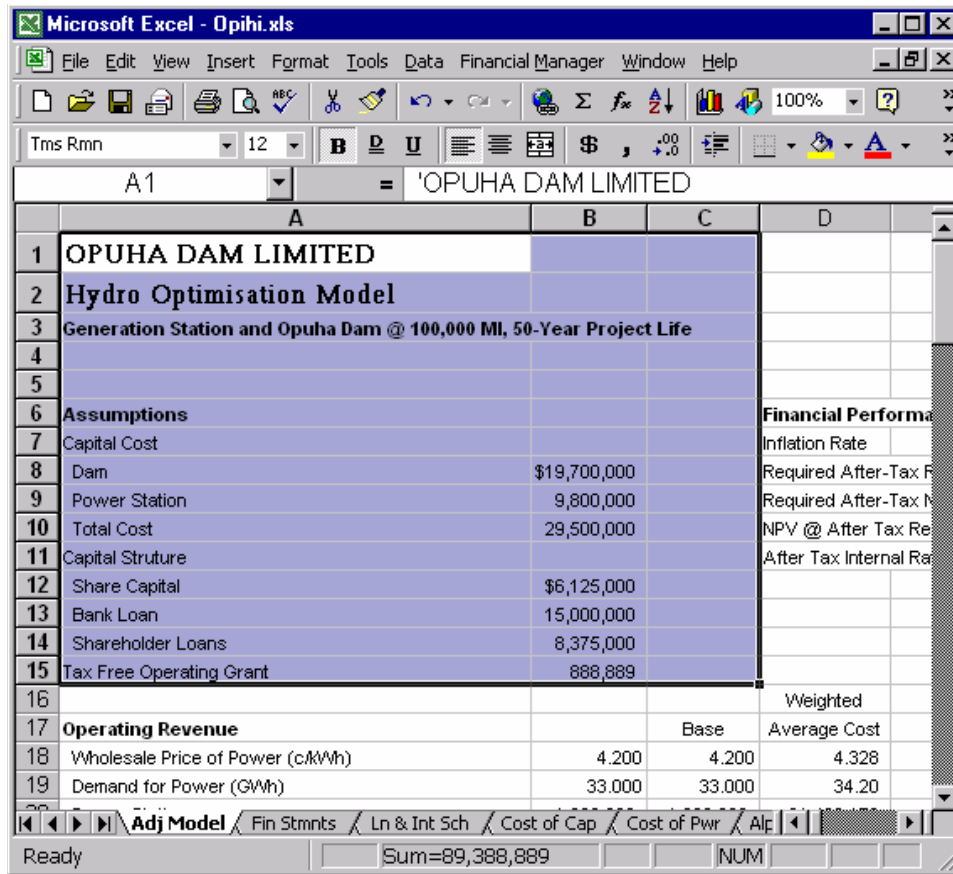
This chapter is designed as a starting point for those who want to create VBA macros in Excel. Because becoming proficient at VBA macros means becoming proficient at VBA programming, this chapter can be no more than a start. However, with the aid of the “Excel Macro Recorder” and a little knowledge of VBA programming basics, you can go a long way.

MACRO RECORDER

The best way to start learning about Excel macros is to use the macro recorder feature to create a simple macro. Actually you don't really create the macro, rather you turn on the macro recorder, execute the steps you want recorded, and Excel generates the VBA code. Then you use the VBA editor to examine the code that has been created and in that way learn about what visual basic programs look like, the syntax of the commands. You will certainly need some type of in-depth reference book to guide you through your journey to becoming a VBA programmer.

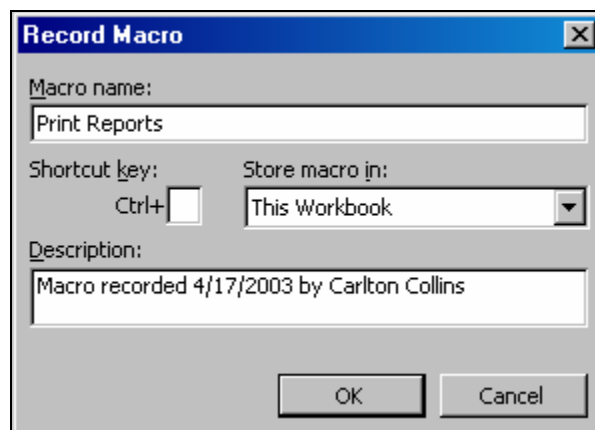
In this first example we will start by recording a simple macro that prints the region of the worksheet that is highlighted when the macro is executed and clears this print range when it is finished. The following are the steps to follow in creating such a macro:

1. Use the mouse to select (i.e. highlight) the range you would like to print.



Using the Mouse to Select the Print Range

2. Turn on the macro recorder (Tools – Macro – Record New Macro...)
3. Assign the following parameters:

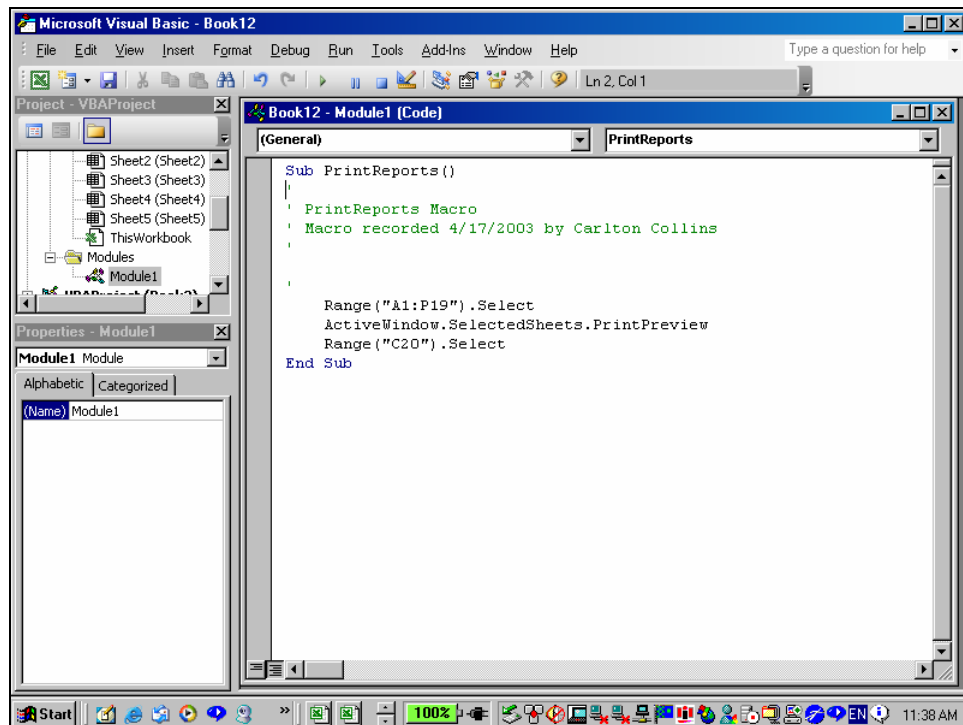


4. Click OK to close the Record Macro dialog box. At this point the following small dialog box will appear on your screen. (The button in the left half of the box is the one you select to stop the macro recorder.)



5. Select print from the File menu, in the Print What area of the print dialog box select Print Selection, and then select OK to execute the print job and close the dialog box.
6. Click on the Stop Recording button to stop the macro recorder.
7. Now take a look at the macro you have recorded by selecting Tools – Macro – Macros. In the dialog box that appears, highlight the Print_Current_Selection macro and select Edit. This will bring up the Visual Basic Editor and the macro you have just created. If you created keystroke macros in the past in Lotus 1-2-3 you will see that this macro does not look anything like those old macros.

Note: Since Excel does not come with a toolbar icon to print the selected range, this simple macro can be very useful if you choose to assign it to a toolbar icon. Although there is a select print area toolbar icon available, using that icon selects a print range which will over ride any previously selected print range. By creating this simple macro and assigning it to a toolbar icon, you can quickly and easily print a selected area with one click of the mouse, without changing any of the previously established print area settings.



A Simple Excel Macro that Prints the Selected Range

RELATIVE VS. ABSOLUTE REFERENCES IN MACROS

When you turn the macro recorder on you will notice that a small button box (like the one pictured below) appears on the screen.



The Macro Recorder Stop/Relative Reference Dialog Box

The button in the left half of the box is used to stop recording the macro. The button in the right half of the box is used to turn on and turn off the relative reference feature. Important! If you turn on the relative reference feature it will stay on and when you go back to record a macro even hours later it will still be on.

The following is an example of a macro created with the relative reference button off:

```
Sub sample1()  
' sample1 Macro  
' Macro recorded 4/19/2003 by Carlton Collins  
  Range("A5").Select  
  ActiveCell.FormulaR1C1 = "Accounting Software Advisor"  
  Range("A6").Select  
  ActiveCell.FormulaR1C1 = "4480 Missendell Lane"  
  Range("A7").Select  
  ActiveCell.FormulaR1C1 = "Norcross, GA 30092"  
  Range("A8").Select  
End Sub
```

This simple macro enters address information into the worksheet. Unfortunately, since I did not press the relative reference button, the macro recorder assumed that I always wanted the address typed in the same cells. Therefore, it included the “Range("A5").Select” command. In other words, you can run this macro just fine, but it will always put the company name and address information in cells A5 through A7.

Alternatively, we could have pressed the relative reference feature and would have gotten the following more useful macro:

```
Sub sample2()  
' sample2 Macro  
' Macro recorded 4/19/2003 by Carlton Collins  
  ActiveCell.Offset(4, 0).Range("A1").Select  
  ActiveCell.FormulaR1C1 = "Accounting Software Advisor"
```

```
ActiveCell.Offset(1, 0).Range("A1").Select
ActiveCell.FormulaR1C1 = "4480 Missendell Lane"
ActiveCell.Offset(1, 0).Range("A1").Select
ActiveCell.FormulaR1C1 = "Norcross, GA 30092"
ActiveCell.Offset(1, 0).Range("A1").Select
End Sub
```

WHEN TO USE THE MACRO RECORDER

The macro recorder works very well when you are creating macros that issue commands. For example, if you have a series of formatting commands (ex. Times New Roman, Bold, Italic, 14 Points, Strikethrough, Underline Style Double) that you frequently issue, you can create a macro that will issue the commands in one action to do all this formatting. And it is not just formatting commands that can be recorded. Most of the menu commands will record well and almost any series of menu commands can be made into macros.

The macro recorder feature allows you to automate a series of tasks you go through frequently. A good example is preparing a series of reports from a single workbook where the different reports may involve not only multiple printouts, but also possibly things like re-sorting, different scenarios, different data filters, etc. Although the Report Manager add-in is useful for this process, it may be simpler to create a Print_Month_End_Reports macro that does it all with one click of the mouse.

The macro recorder is not well suited for creating VBA programs that enter or edit formulas. That is because it does not record keystrokes but rather acts on objects. So, for example, if you wanted a macro that took the contents of a the current cell and enclosed it in the =Round() function, the macro recorder would not create the formula you wanted. Rather, it would create a macro that placed the contents of the current cell (enclosed in the =Round() function) in any cell where you execute the recorded macro.

The following things can be done with VBA macros but can't be recorded:

Interaction - Where you want the user to have input into how the macro will operate. For example, you create a macro to print reports and you want the user to be able to select how many copies of the report are printed.

Custom Functions – If you frequently perform the same complex and lengthy calculations in your worksheets, you can build a macro that will create your own unique Excel function where you only have to provide the inputs.

Decisions – This is where you want the macro to make branching or other decisions based on conditions that may or may not exist in the worksheet data. For example, based on some calculation, you may want some data omitted from the printout the macro is making.

VBA PROGRAMMING BASICS

The VBA programming language provides the tools needed to create solutions when the macro recorder can't get the job done. VBA is such a powerful programming language that you can (if you have the programming ability) create a solution to virtually any problem you encounter in Excel. In fact you can completely customize Excel through VBA and you can even change the entire Excel interface.

OBJECT ORIENTED PROGRAMMING

VBA is a structured programming language where sentences (called statements) are constructed of building blocks such as objects, methods, and properties. These VBA statements are grouped in larger blocks called procedures. A procedure is a set of VBA statements that performs a specific task or calculates a specific result. The first step to learning how to create procedures is to learn about the building blocks.

OBJECTS

VBA is an object-oriented programming language, which means the statements you create in VBA act on specific objects rather than begin general commands. Excel is made of objects that you can manipulate through the VBA statements. The entire workbook file is an object, an individual sheet is an object, a range within a sheet can be an object, and an individual cell is an object. There are many more types of objects, and as you see objects can be containers for (called collections) other objects.

COLLECTIONS

Some objects are collections of other objects. For example, a workbook is a collection of all the objects it contains (ex. sheets, cells, ranges, charts, VBA modules, etc.). A VBA statement that makes reference to the workbook names "Loan Calculator.xls" would appear like this:

Workbooks("Loan Calculator.xls")

METHODS

A method is an action that can be performed on an object. Excel VBA objects are separated from their methods by a period. For example, if you wanted to save a particular file as part of a VBA program you could include the following sentence in the code:

Workbooks("Loan Calculator.xls").Save

PROPERTIES

Properties are used to describe an object. Some properties are read-only, while others are read/write. These properties describe Excel objects. For example, an Excel workbook has a particular path on your hard drive or network where it is saved. That path is a property of the workbook. That path is read-only as it cannot be changed without saving the file to a different location. Properties are separated from objects by periods just as methods are. The following sentence will display the current path of the Loan Calculator.xls file in an onscreen message box:

```
Msgbox Workbooks("Loan Calculator.xls").Path
```

Note: MsgBox is a function. Functions will be discussed later in this section.

If you want to set a read/write property equal to something, it changes the current value of that particular object. If you don't set a read/write property equal to something, Excel will tell you the object's current value. For example, the following sentence will set the sheet name of the first sheet in the workbook to "Cover"

```
Sheets("Sheet1").Name = "Cover"
```

FUNCTIONS

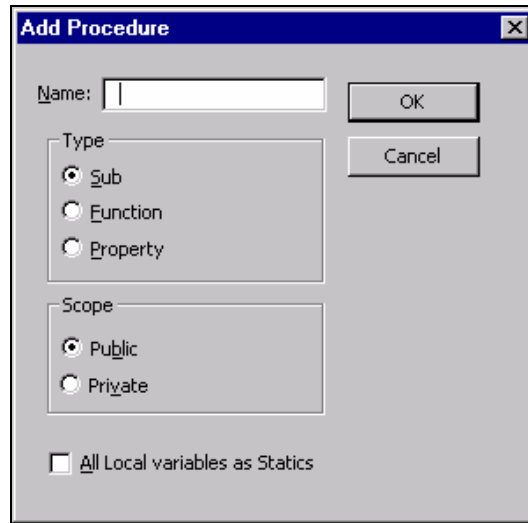
A VBA function is a lot like a workbook function in an Excel spreadsheet. It performs a calculation and then returns the appropriate result. Functions provide information that is useful in building VBA procedures. In the previous example, the MsgBox function was used to display the path information on the screen.

VBA Functions should not be confused with Function Procedures. A Function Procedure is a series of Visual Basic statements enclosed by the Function and End Function statements. A Function procedure is similar to a Sub procedure, but a function can also return a value. A Function procedure can take arguments, such as constants, variables, or expressions that are passed to it by a calling procedure. If a Function procedure has no arguments, its Function statement must include an empty set of parentheses. A function returns a value by assigning a value to its name in one or more statements of the procedure.

PROCEDURES

When you create VBA code inside an Excel workbook it will be stored as a "Module" within the workbook. Unlike, Lotus 1-2-3 macros, these modules are not entered in cells but are still a part of the workbook files. They can only be viewed by using the VBA Editor.

Code within a module is organized into procedures. A procedure tells the application how to perform a specific task. Use procedures to divide complex code tasks into more manageable units. There are three types of VBA procedures: Sub, Function, and Property.



The Insert Procedure Dialog Box

The most common type of procedure is the Sub. A Sub procedure is a series of Visual Basic statements enclosed by the Sub and End Sub statements that performs actions but doesn't return a value. A Sub procedure can take arguments, such as constants, variables, or expressions that are passed by a calling procedure. If a Sub procedure has no arguments, the Sub statement must include an empty set of parentheses. The following is an example of a simple Sub:

```

                Sub Center_Across_Selection()
' Center_Across_Selection Macro
    ' Macro recorded 10/15/2004 by Carlton Collins
    With Selection
        .HorizontalAlignment = xlCenterAcrossSelection
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .ShrinkToFit = False
        .MergeCells = False
    End With
    Selection.Style = "Comma"
    Selection.Font.Underline = xlUnderlineStyleSingleAccounting
End Sub

```

VARIABLES AND CONSTANTS

During the execution of some VBA macros there will be times when information that is either gathered from the user, returned by a function, or defined by the programmer that must be stored temporarily for use later on in the macro. Sometimes this information will change during the execution of the code and sometimes it will be static. Variables and constants are used to store this type of information. In the following macro CoName is a variable:

```

Sub Add_Footer()
    CoName = InputBox("Name your Company?", "Add Company Name to Footer")
    With ActiveSheet.PageSetup
        .LeftHeader = ""
        .CenterHeader = ""
        .RightHeader = ""
        .LeftFooter = CoName
        .CenterFooter = ""
        .RightFooter = "&N"
    End With
End Sub

```

Like a variable, a constant is a temporary holding place for some information that is used in a procedure. However, as the name implies a constant never changes. Constants must be declared. A declaration statement in a VBA macro is used to define the value of a constant. In the following macro the Company Name is declared in the first statement of the VBA code:

```

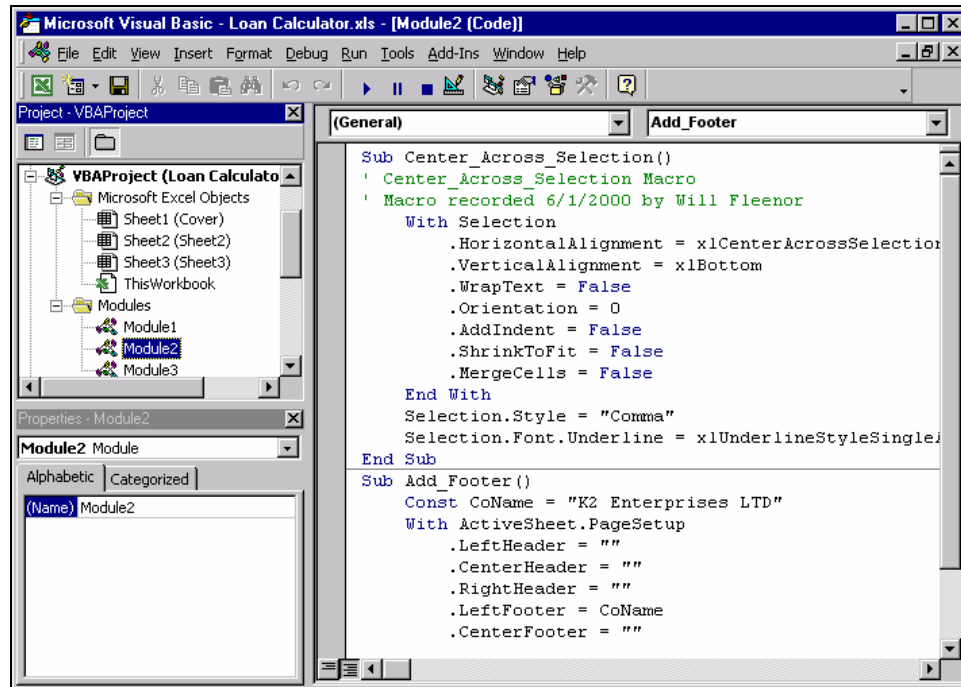
Sub Add_Footer()
    Const CoName = "Accounting Software Advisor"
    With ActiveSheet.PageSetup
        .LeftHeader = ""
        .CenterHeader = ""
        .RightHeader = ""
        .LeftFooter = CoName
        .CenterFooter = ""
        .RightFooter = "&N"
    End With
End Sub

```

The company name is enclosed in quotation marks. All literal text (this is what strings are referred to in VBA programming), which is placed in a procedure, must be surrounded by quotation marks.

USING THE VISUAL BASIC EDITOR

The Visual Basic Editor is the tool you use to display VBA code. It is also the place you go to modify code you have recorded or to create VBA code from scratch.



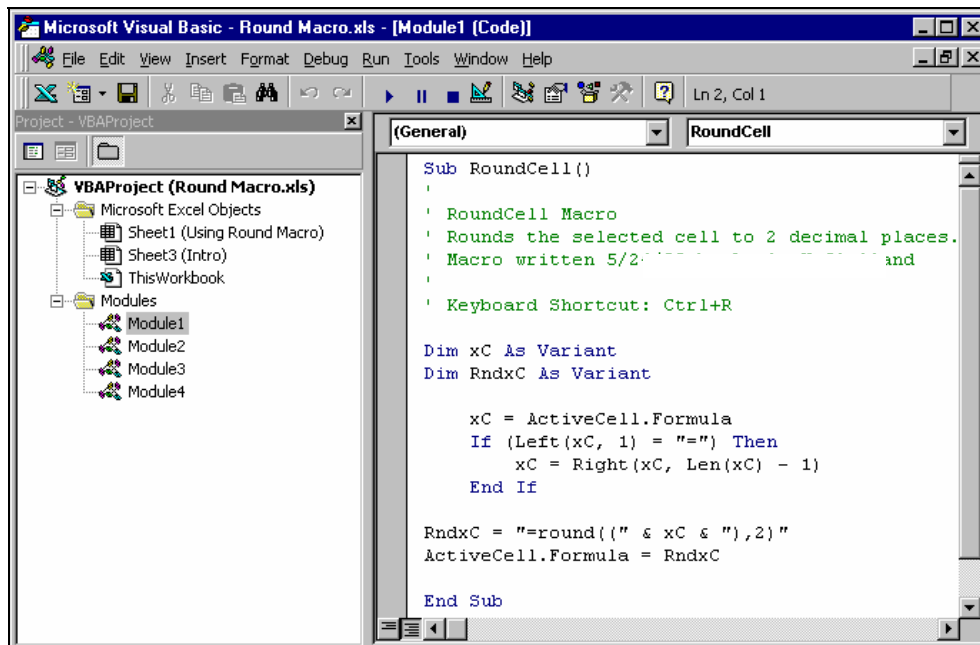
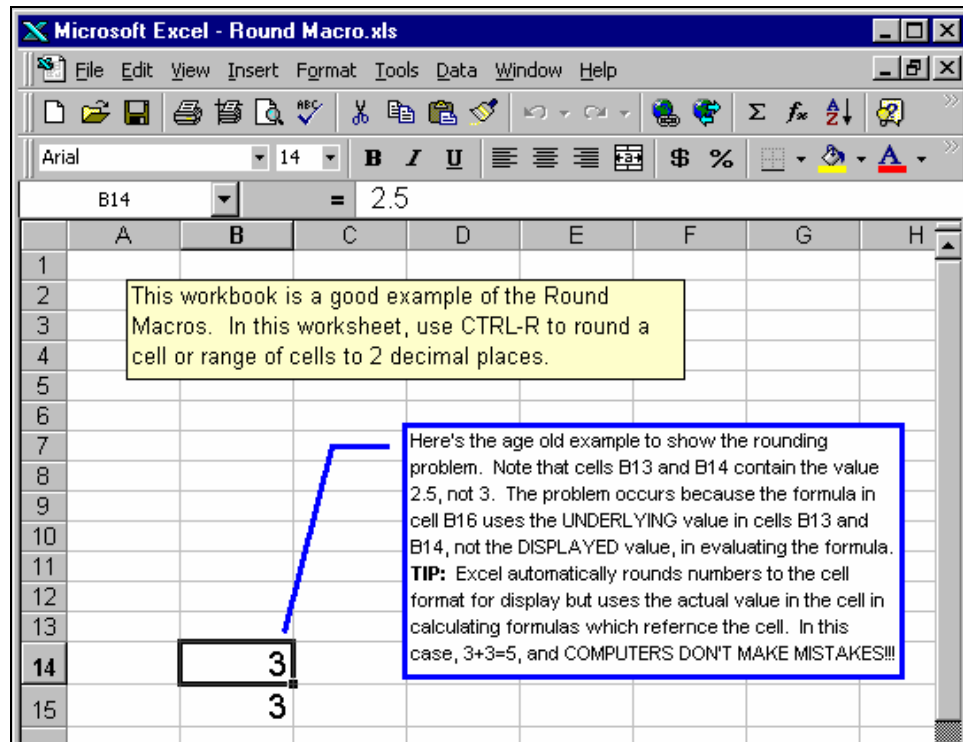
The Visual Basic Editor

The Editor is divided into three windows. The **Project Explorer window** (upper left corner of the screen) displays all open projects and their associated objects. You use this window to browse modules when attempting to locate a particular procedure.

The **Properties window** (lower left corner of the screen) displays the properties of the selected objects in the Project window. If a module is selected in the project window, the Properties window will display the only property of that module, i.e. its Name property.

The **Code window** displays the code that is associated with the object selected in the Project Explorer. Because all procedures appear in the Code window, this is the window that you will use most often.

EXAMPLE MACRO TO INSERT THE ROUND FUNCTION



CREATING USER DEFINED FUNCTIONS WITH A MACRO

Here's the familiar problem. Our model contains floating point calculations that result in values that contain more than 2 decimal places, but the results are displayed with 2 decimal places. The formula seemingly evaluates incorrectly, but this is caused by Excel's use of the actual values in the precedent cells to make the calculation.

	Original Price	100% - Markdown	Sale Price	Units On Hand	Total Inventory at Retail	Actual Cell Values
11	18.00	66.67%	12.00	9	108.01	108.00540
12	21.35	66.67%	14.23	4	56.94	56.93618
13	16.75	33.33%	5.58	6	33.50	33.49665
15					198.44	

Highlight the range F11 through F13 and Press CTRL-R to round the cells to 2 decimal points. Now the formula evaluates correctly.

Note that the formula does not evaluate correctly.

Compare the actual cell values to the values displayed in the Total Inventory at Retail column.

```

Public Sub RoundRange()
' RoundRange Macro
' Rounds the selected range to 2 decimal places.
' Macro written 5/24/98
'
' Keyboard Shortcut: Ctrl+E
Dim nCol As Integer
Dim nRow As Integer
Dim xC As Variant
Dim RndxC As Variant
Dim C As Integer
Dim R As Integer
Dim rng As Range
nCol = Selection.Columns.Count
nRow = Selection.Rows.Count
Set rng = Selection
For C = 1 To nCol
    For R = 1 To nRow
        xC = rng.Cells(R, C).Formula
        If (xC <> "") Then
            If (Left(xC, 1) = "=") Then
                xC = Right(xC, Len(xC) - 1)
            End If
            RndxC = "=round((" & xC & "),2)"
            rng.Cells(R, C).Formula = RndxC
        End If
    Next R
Next C
End Sub
    
```

THE MONTH BOOK MACRO

The Month Book macro adjusts the workbook so that it has 13 sheets and names the sheets, Summary, Jan, Feb, ... , Dec.

```
Sub MonthBook()
```

```
,
```

```
' MonthBook Macro
```

```
' Macro written 10/15/04 by J Carlton Collins
```

```
' Keyboard Shortcut: Ctrl+Shift+M
```

```
,
```

```
Dim A As Variant
```

```
Dim E As Integer
```

```
Dim N As Integer
```

```
Dim shCount As Integer
```

```
Dim shToDel As Integer
```

```
shCount = Sheets.Count
```

```
E = 0
```

```
N = 0
```

```
A = Array("Summary", "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct",  
"Nov", "Dec")
```

```
If (shCount > 13) Then
```

```
    shToDel = shCount - 13
```

```
    Application.DisplayAlerts = False
```

```
    For N = 1 To shToDel
```

```
        Sheets(Sheets.Count).Delete
```

```
    Next N
```

```
    Application.DisplayAlerts = True
```

```
Else
```

```
    If (shCount < 13) Then
```

```
        For N = shCount To 12
```

```
            Sheets.Add After:=Sheets(N)
```

```
        Next N
```

```
    End If
```

```
End If
```

```
For N = 1 To 13
```

```
    Sheets(N).Name = A(E)
```

```
    E = E + 1
```

```
Next N
```

```
Sheets(1).Activate
```

```
End Sub
```

FOOTER WITH PATH AND FILENAME MACRO

The macro, AS WRITTEN, sets a footer with the left-hand footer section containing the information you wanted. If you would rather the file and pathname be in the center or right-hand section, change the object to ".CenterFooter" or ".RightFooter". Set the other sections of your CUSTOM footer in the PageSetup dialog like you normally would. The macro will NOT overwrite your settings in the footer sections, other than the one specified in the macro.

You MUST run the macro each time you CHANGE the filename or change the location of the file. The macro simply automates the insertion of the file and pathname into a custom footer. Here's the macro code. Simple isn't it! If you would rather place the filename and pathname in a cell in your workbook, use the function "=cell("filename")".

```
Sub FooterName()
```

```
    ActiveSheet.PageSetup.LeftFooter = ActiveSheet.Parent.FullName
```

```
End Sub
```

1. If you have never saved a macro to your Personal Macro Workbook, use the Macro recorder to save ANY keystrokes to a dummy macro. You can delete it later.
 2. Choose Tools, Macro, Visual Basic Editor.
 3. In the PROJECT Window (upper left-hand window), double-click on "VBAProject (PERSONAL.XLS)."
 4. Right-click on "Modules." Choose Insert, Module.
 5. A code window will open on the right. Paste the code from this message or type it in.
 6. Click on the SAVE button, and then close the VB Editor.
- You can now choose, Tools, Macro, Macros to Run the macro. Assign a shortcut key or attach it to a button as we did in class. Since you saved it in your Personal Macro workbook, the macro will be accessible from any workbook.

For more examples, visit this Microsoft web site which features Macro examples:

<http://office.microsoft.com/assistance/2002/articles/pwRecordingMacros.aspx>