# Introduction to Macros and Visual Basic in Excel

S. Malpezzi, Real Estate 631
Draft, November 8, 2007. Subject to Revision.


Recording box is checked. Also notice that you can't click on the Stop button while you are editing a cell. As in 2007, once the macro is recorded, you

A macro is a way of automating a procedure in a spreadsheet. You may have used macros before, they also exist in other spreadsheet such as Lotus-123 or Quattro Pro. Generally, macros vary substantially from spreadsheet language to spreadsheet language, and often do not translate automatically across types of spreadsheets. Macros can be extremely simple or extremely complex; that depends upon the task at hand. The purpose of this handout is to give an introduction to macros in the Excel spreadsheet environment. This handout only touches on the basics. For more details, see your Excel manual, or specialty manuals.

Computer manuals have a short "half life" because every few years there's a new version of Excel or other software. In creating the early drafts of this handout I've used two books (now a bit out of date):

> *Microsoft Excel/Visual Basic Programmers Guide* (Redmond, Washington: Microsoft Press, 1995)
>
> *Microsoft Excel/Visual Basic Reference* (Redmond, Washington: Microsoft Press, Second Edition, 1995)

Together these two books give a very detailed look at macros in Excel, but are not particularly good as introductions. I usually avoid the "Dummies" or Intro books, but for Excel macros I can recommend:

> Elisabeth Boonin, *"User Friendly" Using Excel Visual Basic for Applications* (Indianapolis, IN: Que, 1996).

I have drawn on all these references in constructing this handout, but I have drawn especially heavily from Boonin's book. Bookstores like Borders and Barnes and Noble carry lots of these manuals, from introductory to advanced levels. I recommend browsing extensively before you buy, to see which manuals seem to address your own level of expertise.

There have been some changes in VBA as implemented in Excel 2000 (since these books were written) but, other than the look and feel of the newer VBA editor (i.e. the environment in which we write the actual VBA code), so far I haven't found them to be too dramatic. I'll try to indicate some of these changes where relevant.

**Visual Basic**

Microsoft Excel uses a computer language called Visual Basic as its language for writing macros. If you have written macros before in other spreadsheets, you may have used, for example, the technique of writing out a series of letters which represent spreadsheet key strokes in an obscure corner of your spreadsheet; naming the range containing these instructions with a name like \x; then invoking the macro by hitting the Ctrl X keys.

Excel eschews these kinds of crude macro commands for a full-blown programming language called Visual Basic. Those of you with some experience in computer programming know that Basic is itself a full-fledged computer language, and is the computer language most associated with PCs. (Examples of other similar high-level languages include Fortran, Pascal, or PL/1). Visual Basic is a so-called “object oriented” programming language. What this means need not concern us here, except that we will see later how various *objects* defined in our

language have *properties* which can be manipulated in ways which are simultaneously powerful and straightforward.

Visual Basic is often referred to by the acronym VBA. One of the powerful things about the implementation of VBA in Excel is that you can use it to write macros, but you can *also* use it to write your own functions, as well as other kinds of subroutines (to be discussed below). You can also integrate your macros with your worksheets by adding what you create to menus and tool bars. You can also create dialog boxes which interact with users and run the subroutines.

An important externality: Visual Basic (versions not exactly identical in all respects to the MS Office VBA, but similar) is also a widely used standalone programming language. Also, Microsoft uses VBA as the macro environment for all its products (Word, Excel, Access, etc.). A number of other software outfits are following suit. So some of the skills we learn in VBA in the spreadsheet environment can be transferred to other environments.

**The Macro Recorder**

There are two ways to create a macro. If the macro you wish to create is simply a shortcut way of collecting routine tasks in a macro, the easiest way to create it is to use the macro recorder. You start the macro recorder just as you would a tape recorder, perform the task you wish to automate, and Excel records your actions for later playback.

You can also write a macro using the VBA language as you would write any computer program. You can also combine the two methods; for example you can create a draft macro by using the macro recorder and then edit the resulting code which Excel creates for you.

We used the macro recorder in class to create a “toy” macro. The steps involved were:

- Select the Developer tab in the Ribbon, then focus on the Code section of the tab, on the left. Click on **Record Macro**. The **Record Macro** dialogue box will appear.

- Enter the name of the macro, typed as one word, and the description of the macro in the dialogue box. Pay attention to the option "Store macros in:" window of the dialogue box. Make sure that for this class you use the option "This Workbook". This will ensure that your macro is recorded into the workbook you are working in. There are some other options which we will be discussing later in the course.

- Click "OK". The **Record Macro** dialogue box will close. At the very bottom left of the Excel display you’ll see a little rectangular **Stop** button. The recorder is now recording every action that you take with the computer.

- Undertake whichever actions you wish to have recorded. Remember that the recorder will quite literally copy every action you take until you shut the recorder off. Thus, if you make a mistake and correct it, every time you LATER run the macro Excel will make the same mistake and then correct it (*if possible*). Thus I recommend that before using the macro recorder you practice your action or actions several times to make sure you are doing them correctly so that your recorded macro is “clean.” If the macro to be recorded is long and complex, make some shorthand notes on a sheet of paper to remind you of the steps to be taken and the order in which to take them.

- Finally, when you are finished click on the **Stop** button to stop recording the macro.

Once the macro is recorded, you run or play the macro by using the **Macros** button in the **Code** section of the **Developer** tab. A dialogue box will pop up which lists any macros which you have available to you. To run a macro you have created, click on the macro in question and then click the **Run** button.

You can also assign a macro a shortcut key, which is a combination of keys that run the macro and work faster than the dialogue box. I do not recommend using shortcut keys in the classroom and lab environment, since we are often not sure if some particular key combination has already been set to perform some other action. You can also make a frequently used macro as a menu item. Again I do not recommend this as long as you are working in a lab environment.

When you have recorded a macro, where it is stored depends on the version of Excel you're using. In Excel 97 and 2000, macros are stored "out of sight" even though they are (usually, at least in our class) saved as part of the spreadsheet. To view a macro you recorded, click on the **Visual Basic** icon in the **Code** section of the **Developer** tab (in Excel 2003: **Tools, Macro, Visual Basic Editor)**. The VBA editor (a separate program) starts, and three windows pop up. Go to the "Project" window, and click on the "Modules" folder. If you have recorded or written any macros, one or more modules appear, of the form Module1, Module2, etc. Double click on a Module and it opens in the larger window to the right of the Project window. The default is that Excel will create a new module every time you record a new macro; but it is possible to place multiple macros in a single module. Notice that the module sheet does not have rows or columns. It works more or less like a crude word processing program.

If there are no module sheets, you can easily add one. Go to the **Project Explorer** window of the VBA Editor, select your VBA Project (the name of the Excel workbook you are using), and then from top menu, **Insert** a **Module**. One will appear, like a blank sheet of white paper, in the large window to the right.

The next page shows a few annotated screen shots of the VBA explorer, to help you learn how to navigate.

**VBA Programming**

Since recording is so easy, why use the other method of creating macros? Sometimes you may want to write or copy someone else's program. Another reason is that there are many things that can be done in the Visual Basic language that cannot be done using Excel commands *per se*, so they can't be recorded.

The VBA language is not easy to truly master in a short period of time. The best approach is probably not to try to memorize all the elements of the language, but rather to start off by recording some macros that undertake familiar tasks, then look to see how the macro recorder has translated your actions into Visual Basic.

Let's write an extremely simple macro as a little practice. First we need a module sheet to work with. In the VBA editor, go to the top menu bar and Insert a Module. This will create a new macro sheet. You should have a blinking cursor at the top left corner of the module sheet. If not, use the mouse to put the cursor there. Now you can type just as in a simple word processor. Type the following:

```
Sub Sillymacro ()
        Beep
End Sub
```

That's all there is to it. The first line of the macro -- Sub Sillymacro() -- is the title of the macro. The word Sub tells VBA that this is a subroutine. There are several kinds of subroutines in VBA, including macros. Sillymacro is the name and the way we would refer to this macro later. The parentheses are not utilized in macros, but are utilized in other types of subroutines and in functions. (To jump ahead, if we write a non-macro subroutine, or a function, we use the

parentheses to enclose arguments which are past back and forth between Excel and the subroutine or function that we have created.)

The second line contains the "meat" of the subroutine (such as it is). The word “Beep” is the action that the action undertakes; that is to say, the computer beeps. We can place many lines representing many actions, some of them actually useful (believe it or not, there are occasions where "Beep" is quite useful!)

The final line, End Sub, tells VBA that the subroutine has ended.

You run this macro in the same way in which you run a macro which you have recorded. Typing in module sheets is similar to typing in a simple word processor. You can also **Copy** and **Paste** from elsewhere in the spreadsheet, or even other programs. The VBA editor does make some changes to your text if your text has a special meaning to VBA. Words which are functions or have other special meanings are automatically capitalized even if you type them in lower case letters, for example. Some key words are assigned colors, so you will notice some words turn blue or red as you type them. Blue means that what you have typed is a restricted key word. This is a word which has to be used for a special purpose and cannot be used, for example, as the name of a macro.

Green text is for comments. Comments are lines which start with an apostrophe. It’s a good idea to get in the habit of writing lots of comments about your macros within your macros. Comments can also be handy if you think you may want to delete a line of code but aren’t sure. (This happens a lot when the macro recorder writes possibly extraneous lines). Just put comments (apostrophes) in front of the lines you want to consider deleting, and then run the macro. If it works, then delete the lines. If it doesn’t work, delete the apostrophes and the lines

are back as code. You can put a comment on the same line as programming code if you put the code first.

Red text indicates some sort of error. Remember, if you're not making errors, you're not learning anything new.

The VBA text editor doesn't wrap lines. This can be annoying, but it's more or less necessary since a line of code needs to be kept as a separate line for good programming practice. You can write one line of code across two physical lines by using the underscore character at the end of one physical line to indicate continuation of a programming line on the next physical line.

One thing you may often want to do if you use the macro recorder is to take out some of the commands that get recorded which may be extraneous.

Let's look at a simple example. Select a cell in a spreadsheet with some text in it and start the macro recorder. Select **Format, Cells**. Click on the **Font Tab** to bring up the page of the dialogue box that describes the font. Select **Bold** from the list of styles, then click "OK".[1] Then hit the **Stop** button to stop the macro recorder.

All you wanted to do with this macro was to change the font style to Bold. ( I know you don't need a macro to do this but this is just to illustrate a point.) Now look at the module sheet and see what was recorded. It probably looks something like this:

---

1 Don't use the bold button on your menu bar; it will work, i.e. bold the text, but it won't illustrate the principle we want to look at here.

```
Sub Boldness ()
    With Selection.Font
        .Name = “Arial”
        .Fontstyle = “Bold”
        .Size = 10
        .Strikethrough = False
        .Superscript = False
        .Subscript = False
        .OutlineFont = False
        .Shadow = False
        .Underline = x1None
        .ColorIndex = x1Automatic
    End With
End Sub
```

Now, you have probably already figured out what has happened. All you really cared about was changing the font to Bold. But what Excel actually did was in effect take a snap shot of all the current properties of your font. Thus, if later you had a cell where you had typed something in, say, Times Roman, and used this macro, it would bold the text but also change it to Arial! So what you want to do is delete all those lines that describe properties of the font that you do not wish the macro to affect. Exactly which these are depends of course on what you want the macro to do; for example if you want the only thing the macro does is to bold then delete all the lines from .Name = "Arial" through .ColorIndex = x1Automatic, with the exception of .Fontstyle = "Bold".[2]

---

2 If you used the bold button, despite my entreaties, the macro will contain the following:

```
Selection.Font.Bold=True
```

which in fact works better than our example, but doesn't illustrate the principle we're trying to get across.

This also gives us a clue to something we'll see pop up again and again; the "With" code, which specifies the **object** we are going to work with.

**Control Structure**

A macro or subroutine without control structure starts at the beginning and goes to the end without stopping. Using control structure means you add **If** statements, or other statements. These control the order in which statements are run, how many times they run, or whether they run at all. Probably the simplest control structure is to use a simple if statement. For example:

```
If AGI>100000 Then PLL = 25000 - .5*(AGI - 100000)
```

where AGI has been previously defined. Notice this If statement within VBA is different than the =IF function in the worksheet. First of all, there is no equals sign in front of the word "If." Secondly, if the condition is true then the action to the right of the Then statement is carried out. There is no requirement to specify another statement which is carried out if the statement is not true. However, this can be done with a construction called if\then\else:

```
If AGI>100000 Then PLL = 25000 - .5*(AGI - 100000)
Else PLL = 25000
```

If you have multiple lines which need to be carried out if a statement is true, you can use a more general version:

```
If (condition) Then
        (statements)
        (statements)
        (statements)
End if
```

where "condition" is a logical condition to be evaluated, and "statements" are things which run if the condition is true.

To undertake such comparisons you can use the following comparison operators: =, >, <, >=, <=, or <>. These are all familiar with the possible exception of <> which means "not equal to." You can compare text as well as numbers (although we won't be using these comparisons much for a while). You can also test multiple conditions together using "And" and "Or" statements. You can also cause the program to repeat steps. This uses a construction called For\Next. The best way to see this is to look at an example:

```
Sub Testloop()
        Zzz = 5
        For n = 1 to Zzz
                MsgBox "the current number is " & n
        Next n
        MsgBox "now we're out of the loop"
End Sub
```

The variable zzz is given the value 5. The For statement sets up a loop that will repeat until n = the value of zzz. By default the initial value of n is set to 1. The line beginning with next is the signal that it is the end of the loop.

A word of caution: if you use the wrong logic in a loop you can end up in an infinite loop; that is a loop which you never get out of. This use to concern programmers a lot because you might burn up thousands of dollars worth of computer of time before any one realized what was going on.

Of course with PCs all you lose is a little bit of your time. If you find yourself in an infinite loop in Excel, hit the ESC key to jump out. Then take a careful look at your logic and fix the loop.

There are some other constructs which can be used to loop. Another useful one is the do while:

```
Do While (condition)
        (statements)
Loop
```

The first line starts with Do While and is followed by a condition that Visual Basic can evaluate as true or false. Whenever it is true, the statements which follow (up to Loop) will be executed. If it's false it will be skipped. After the statements have been run the Loop line sends the program back to do while to see if the condition is still true.

**Expressions**

Now that we know how to control the program, what do we control? What kinds of statements will we execute? One element we will need to understand in some detail is expressions. An expression is something that returns a value in Visual Basic. Common expressions include arithmetic expressions such as 3 + 2. Another kind of expression is referring to a variable, the expression will be evaluated as whatever value is currently assigned to that variable. A single number is an expression.

Some expressions test for truth or falseness, such as x < 7. A function can also be used to create an expression. Something that is confusing in VBA is that VBA functions are often slightly different than equivalent functions in the worksheet. For example, to take the square root of a number in an Excel worksheet cell, we write:

```
=SQRT(4)
```

which will return the value 2. However within VBA we write:

```
x=sqr(4)
```

which assigns the value 2 to x. Notice you can place variables on both sides of the equal sign, for example:

$$X = X+3$$

In an assignment statement VBA evaluates the right hand side of the equal sign and then assigns it to the variable on the left hand side.

**Types of VBA Procedures**

There are three main types of VBA procedures. The first two are subroutines. A macro is a subroutine which has an empty argument list. That is, after the word Sub and the name of the subroutine we find (). Since there is nothing in the parentheses following the name of the procedure it doesn't look for any values to be passed to it. In fact it's the lack of arguments that *defines* a macro subroutine.

The second type of subroutine is where one or more values are passed to the subroutine when it is invoked. Notice that you can write a macro (a subroutine with no arguments) in which you supply arguments interactively during the running of the subroutine. Generally the type of subroutine that requires arguements be passed to it can only be called by other lines of VBA code; that is you can't run it by using the macro dialogue box or using the play button on the Visual Basic tool bar.

The third type of procedure is a function. A function comes up with a single value when called, known as the return value. This is what distinguishes functions from subroutines. Functions require arguments be passed to them. Here is an example of a function:

```
Function Zap(a,b,c)
Zap = a+b+c
End Function
```

Within the spreadsheet you would invoke this function by typing =ZAP(1,2,3). Of course within the parentheses you could put any three numbers or cells containing three numbers. Zap will return their sum.

**A Comment on Style**

Use lots of comments, and indent your code appropriately. VBA does not *require* that you indent code. However I strongly urge you to indent codes so that it is easy to follow program control. For example indent everything between the beginning and ending of a subroutine; indent twice everything between If statements and End If statements, for example.