MANAGEMENT OF PRODUCTION AND INVENTORY SYSTEMS
MION01

Tutorial and introductory exercise to Visual Basic (VBA) programming in Excel

Introduction

Programming macro's in Excel is a convenient way to build your own customized decision support tools. A macro is essentially a computer program that can be controlled via Excel. It can be used to perform more advanced calculations that are difficult or even impossible to do in the actual spreadsheet. The programming language used by Excel is called Visual Basic. An advantage with using VBA programming in Excel is that it comes with Excel, and no additional compiler or software program is needed.

In this introduction, we focus on how VBA/Macro programming in Excel can be used to implement more advanced inventory control models, and to create useful decision support tools. You are more than welcome to apply this in order to solve the assignments in the course, but it is not required! VBA Programming in general offers almost endless possibilities to build your own cool software applications, but we will here limit ourselves to very basic functionality.

This tutorial consists of 4 parts:

- A general introduction to VBA programming in Excel in terms of excerpts from the book Wølk (2010) available in Appendix A. This provides a general introduction to VBA programming and how to get started.

- An illustrative inventory control problem to be implemented as an Excel/VBA program. A solution to this problem is provided in the Excel file titled "*VBA program template - (R,Q) fillrate evaluation and optimization*" available on the course website. This program is designed as a template that can be used for programming other inventory control models. The solution illustrates most of the basic VBA syntax that is needed to build programs of inventory models. By going through how the given program solves the problem at hand you will get a tutorial of applying VBA programming to solve inventory control problems.

- A small exercise which is based on extending the analysis of the illustrative control problem above.

- A shortlist of useful VBA syntax and commands. This can be seen as a complement to the Help function in the software, and the provided program template discussed above.

To study this material on your own I recommend that you follow the order above. To get the most out of studying the provided program you need to first recap the theoretical solution to the problem which is available, for example, in Axsäter (2006).

The instructor led tutorial will focus on explaining how the provided program is constructed, and step by step how the given problem is solved. To benefit from that ***you need to prepare by reading up on the analytical solution to the problem so that you can recognize how this is implemented in the program.***

An illustrative inventory control problem

We consider a basic single-echelon inventory system with constant replenishment leadtime, complete backordering, normally distributed customer demand, controlled by a continuous review (R,Q)-policy. The problem is to construct a VBA program that will: (i) compute the fillrate for a given policy (R,Q) and (ii) optimize R to minimize the inventory while meeting a specified target fillrate (i.e., a service constraint model). The program should be designed so that an arbitrary number of single-echelon systems can be evaluated in a single program run.

The input data required for the analysis encompass:

- The mean, μ, and standard deviation, σ, of the demand per time unit
- The order quantity, Q (and for (i) above the reorder point, R)
- The replenishment leadtime, L.

This inventory problem is thoroughly analyzed in Axsäter (2006) pp. 91-94, and pp. 98-99. ***You should go through this analysis before continuing***. When you have familiarized yourself with the theoretical solution of the problem you can continue to study the Excel template program "*VBA program template - (R,Q) fillrate evaluation and optimization*" available on the course website. This program contains an implementation of the theoretical solution.

The program consists of an Excel worksheet named "Interface" and an underlying VBA program. The interface sheet is where indata is entered and where the results are displayed. You run the existing program by clicking on the button "Run". The choice between evaluating the expected fillrate of an existing policy, and minimizing the reorder point given a specified target fillrate is done by setting the variable **Choice** to 0 or 1 respectively. The number of inventory systems (or items in the same system) you want to evaluate during one program run is chosen by setting the value of **N** appropriately. Currently **N**=1 and only a single system is considered. Test and see what happens in the sheet when you change the number, say to **N**=5.

When studying the program, you should start by first running it once and see what happens. You run the program by clicking on the "Run" button in the worksheet named Interface. The next step is to open the Visual Basic Editor as explained in Appendix A. In the upper left side of the VBA editor you can see that the program consists of 4 modules:

- **Public_variables:** Contains declarations of all public variables. The values of public variables can be changed anywhere in the program and they do not need to be declared as input and output parameters in subroutines and functions. It is convenient to use public variables but very dangerous for larger programs. Locally declared variables are generally preferred

- **Main_program:** Contains the code for the main program that is executed when the “Run” button in the worksheet “Interface” is clicked on.
- **Input_data_module:** Contains the code for the subroutine Input_data. This subroutine reads the input data specified in the Excel sheet “Interface” and assign it to variables defined in the program.
- **Normal_demand_single_stage_R_Q:** Contains the code for the functions and subroutines called by the main program to solve the problem.

When analyzing the program a recommendation is to do so by looking at the modules in the order specified above. The main logic is specified in the main program and the detailed calculations are performed in the functions and subroutines called by the main program in order to complete the task. Simple ways of stepping through the code line by line are available under the “Debug” tab found on the top of the VBA editor window. You can experiment with different ways to step through the code, but one simple way is as follows:

1) Double click on the module Main_program
2) Set a breakpoint by clicking on the gray border to the left of the line
   ```
   Set interface = Worksheets("interface")
   ```
   The entire line should then be highlighted in dark red.
3) Run the program by clicking on the “Run” button in the Interface worksheet or by clicking on the green “play” icon directly below the “Debug” tab in the VBA Editor.
4) The program will execute and stop at the breakpoint which will be highlighted in yellow. To step line by line through the program you just press the key F8.
5) You can set as many break points you want, and you can run the program stepwise from one breakpoint to the next (instead of stepwise line by line) by pressing the key F5.

The code contains a lot of comments (text in green which starts with ') the first comment in the main program is:

```
'********Different types of variable declarations*************************
```

The comments are not executable code but simply explanations of what the code is currently doing. To document your code by writing explanatory comments about what the code is doing is a very good strategy. It makes it much easier for others (and yourself) to read and understand the code. Hopefully the comments in the current program are sufficient for you to read the code and understand what it does without too much effort.

A small exercise – extending the analysis of the considered inventory problem

Assume that the holding cost per unit and time unit, h, and the shortage cost per unit and time unit, p, are available. Modify the provided VBA program so that it can be used for evaluating the expected costs of the considered inventory system. The theoretical analysis of this problem is available in Axsäter (2006) pp. 103-105. In the resulting program you should be able to choose the option of evaluating the expected costs in the Excel sheet “Interface”.

Shortlist of useful VBA syntax and commands

This incomplete list includes some basic VBA syntax and commands. Good book references for someone who wants to learn more include Wølk (2010), Walkenbach (2004), and more recent editions of the same text. The list below should be seen as a complement to the template program which illustrates how most of the syntax works. To get help and information about how different VBA commands work the "Help" tab at the top of the VBA Editor (to the right) is very useful. It is recommended that it is used to get more thorough explanations of the items in the list below. If you have a command that you want to know more details about, a tip is to mark it in the code and then hit F1. This is a shortcut to the help function.

- **Option Explicit**: If entered at the very top of a module it prevents the code in that module to run if it contains undeclared variables
- Different types of loop statements: For-To-Next and Do-While-Loop

  For *integer_var = lower_limit* **To** *upper limit* *'Unconditional loop*
  :
  Next

  Do While *statement=True* *'Conditional loop*
  :
  Loop

- If-Then-Else-End If statements

  If *statement1=true* **Then**
  :
  ElseIf *statement2=true* **Then** ***'****Optional condition level*
  :
  Else ***'****Optional condition level*
  :
  Endif

- **WorksheetFunction.*object*** : VBA contains a large number of WorksheetFunction objects, which can be thought of as small programs and specialized functions that can be very helpful in building the code. For example,

  y= WorksheetFunction.NormDist(x, 0, 1, True)

  assigns the value of the cumulative distribution function for a normal distribution with mean=0 and standard deviation=1 and the argument x, to the variable y. ($y=P(X\leq x)$ when $X\sim N(0,1)$).

  A list of the available WorksheetFunction objects can be obtained by pressing the key F2 when the marker is in the VBA Editor window.

- **Int(*number*)** and **Fix(*number*)** : Both **Int** and **Fix** remove the fractional part of *number* and return the resulting integer value. The difference between **Int** and **Fix** is that if *number* is negative, **Int** returns the first negative integer less than or equal to *number*, whereas **Fix** returns the first negative integer greater than or equal to *number*. For example, **Int** converts -8.4 to -9, and **Fix** converts -8.4 to -8.
- **Abs(x):** Returns the absolute value of the number x

- **Exp(x):** Returns the value e^x
- **Log(x):** Returns the natural logarithm of x
- **MsgBox:** Displays a message box and (optionally) returns a value (see Help in VBA Editor for more details).
- **Now:** Returns the current system date and time
- **Rnd:** Returns a uniformly distributed random number between 0 and 1
- **Sqr(x):** Returns the square root of the number x
- **Time:** Returns the current system time
- **And:** Logical and that can be used for combining conditions, i.e., **If** (x>5) **And** (x<10) **Then**
- **Or:** Logical either or that can be used for combining conditions
- **Stop:** Stops the program at the line where this command is found and opens the VBA Editor for debugging.

References

Axsäter S. *Inventory Control*, *Second edition*. Springer, New York, 2006

Walkenbach, J., *Excel VBA programming for Dummies*, Wiley Publishing, 2004.

Wølk, S., *VBA Programming in Business Economics*, DJØF Publishing, Copenhagen, 2010

Appendix A
Introduction to VBA programming in Excel

Excerpts from

Wølk, S., *VBA Programming in Business Economics*, DJØF Publishing, Copenhagen, 2010

Chapter 1
VBA and Your First Program

In order to get started there are two things you need to become familiar with: The *Developer tab* in Excel, which is explained in Section 1.2 and the *VBA Editor* which is explained in Section 1.3.

The remainder of this chapter is devoted to creating and executing the first macro. In Section 1.4, we explain how to insert a module that can contain your code and in Section 1.5, the macro is created. In Section 1.6, you learn different methods for executing the macro and finally in Section 1.7, we show how to save a workbook that contains code.

1.1 What is VBA

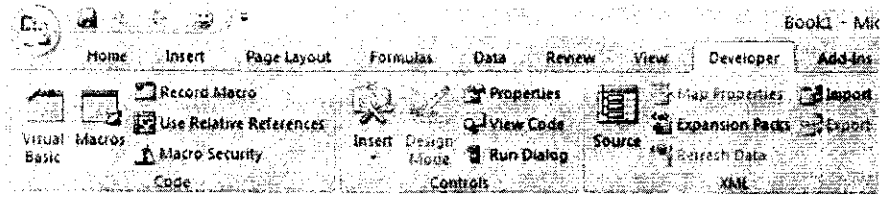
VBA is short for *Visual Basic for Applications* and is a programming language built into the programs in Microsoft Office. In this book, we focus on VBA for the Excel application, but VBA can just as easily be used with Word as the application, for instance.

VBA can be used for manipulating the objects in the application. In the Excel application, the objects are workbooks, worksheets, ranges, charts, etc. The objects of the other applications are different, but the VBA language itself is the same across applications. Like any other programming language, the core of a program created in VBA is variables, loops and decision constructions.

Learning to program is all about learning to write in a language that the computer can understand: If you want to speak Chinese, you must use the Chinese words and phrases, the Chinese grammar, and the Chinese pronunciation. Otherwise, you cannot expect to be understood. Similarly for VBA: In order to create a VBA program that can be executed or to create VBA functions that can be used within the worksheet, it is important to speak VBA to the computer. So far, your computer does not understand plain English. Therefore, you must obey the VBA grammatical rules, which mainly consist of keywords and syntax.

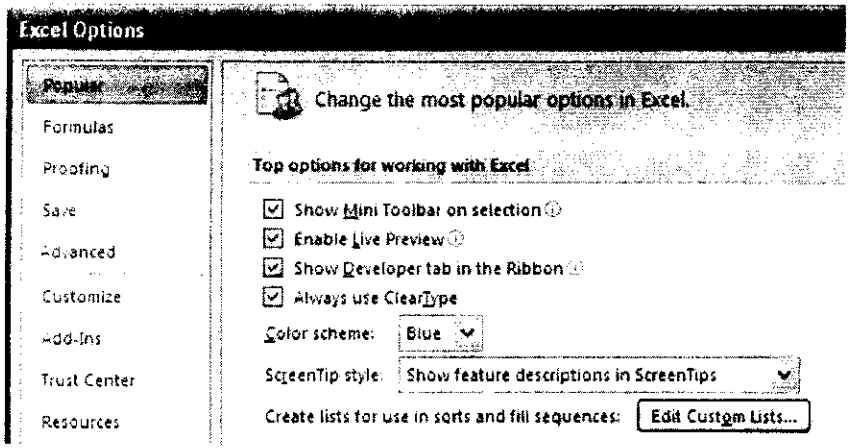
1.2 The Developer Tab

When working with VBA programming, you need the *Developer* tab in Excel, which is shown in Figure 1.1. This is where you find the controls to create and execute macros.



1.1 THE DEVELOPER TAB.

If the Developer tab is not visible in your Excel, locate the *Office* button in the upper left corner of Excel. The Office button is the round one with four colored squares and it is one of the new features in Office 2007. From the Office button, choose *Excel Options* to get the dialog box displayed in Figure 1.2. Select *Popular* in the left-hand side menu and select *Show Developer tab in the Ribbon*. Clicking *Ok* will display the tab as illustrated in Figure 1.1.

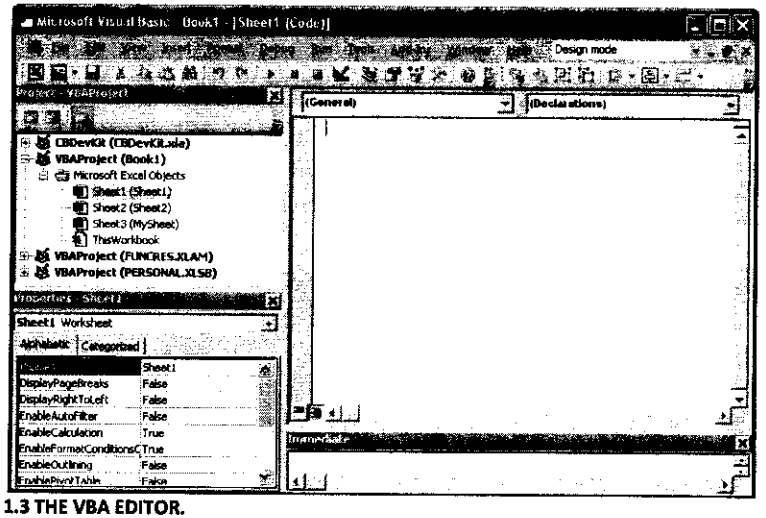


1.2 ACTIVATE THE DEVELOPER TAB FROM THE EXCEL OPTIONS.

The Developer tab consists of three parts. The *Code* part is used to create macros and to execute them. You will learn more about this part below and in Chapter 2. The *Controls* part is used for integrating controls into the worksheets. You will be introduced to this in Section 1.6. In Chapter 16, you will learn a lot about controls on user forms, and you will also learn how they can be integrated into a worksheet. Finally, the *XML* part deals with xml-code and is beyond the scope of this book.

1.3 The VBA Editor

When you develop programs in VBA, you use the *VBA Editor* to do this. There are basically two ways to open the VBA Editor from Excel: Either press the *Visual Basic* control, which is located to the very left on the Developer tab, or press *Alt+F11* from anywhere in Excel. The VBA Editor that appears is shown in Figure 1.3.



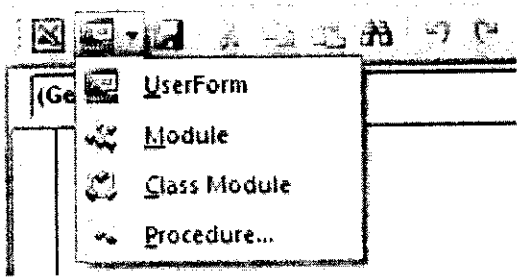
1.3 THE VBA EDITOR.

The VBA Editor has a layout similar to most other Windows programs: At the top you find a *menu bar*, where at least some of the menus are well known: File, Edit, and View, and others that have unfamiliar names such as Debug and Run. After reading through this book, you will be familiar with most of these menus.

Below the menu bar you find a *toolbar*. Figures 1.4 and 1.5 zoom in on the toolbar which we will consider a bit further in the following, so that you know what you are dealing with. From the left-hand side in Figure 1.4, you find the following. The *Excel* control: use this to swap to Excel. *Insert*: use this to insert a module, a user form, or a procedure. *Save*, followed by the traditional *cut*, *copy*, and *paste*. *Find* which is used to search through the code. Finally you have the well-known *undo* and *redo*.

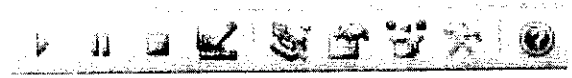
Continuing towards the right, in Figure 1.5 you find the following. Firstly the part that looks like the control panel of a CD player: The triangle, which is used to execute your code, the double vertical bar symbol is used to stop the code from running before it is done, and the square: *reset*, which is mainly used when you try to execute code that contains a bug. In this case, the bug part of your code will be highlighted.

After you have corrected the code by removing the bug, press the square to get out of debugging mode. You will learn a lot more about debugging in Chapter 9. The next control, which looks like tools from a math class, lets you enter and leave design mode. My guess is that you will never want to use this control.



1.4 FIRST HALF OF THE TOOL BAR.

The final five controls are extremely useful. The first two display the *VBA Project Explorer* and the *Properties Window* in case you – more or less by accident – have closed them. These two windows will be explained further below. The control that looks like a yellow box with toys above it opens the *Object Browser*, which will be explained in detail in Chapter 7. The control with the hammer and wrench opens the tool box that is used when designing user forms. You will learn about this tool box in Chapter 16. Finally, the blue question mark control will lead you to the *VBA help system*.



1.5 SECOND HALF OF THE TOOL BAR.

The main part of the VBA Editor consists of three, maybe four, windows. To the right you find a large area that looks much like a note pad. This is the *Code Window*, which is the place where you type in your programming code.

You might, or might not, see an *Immediate Window* below the Code Window. This is used for debugging which is the topic of Chapter 9. We suggest that you close the Immediate Window until you reach that chapter.

At the top to the left, you see the *Project Explorer*, which displays your open Excel files in the same way as the Windows Explorer shows your file system. Let us take a closer look. At this point, if you do not have any Excel file open, please go to Excel and open a new file before returning to the VBA Editor. You now see your Excel file in the Project Explorer.

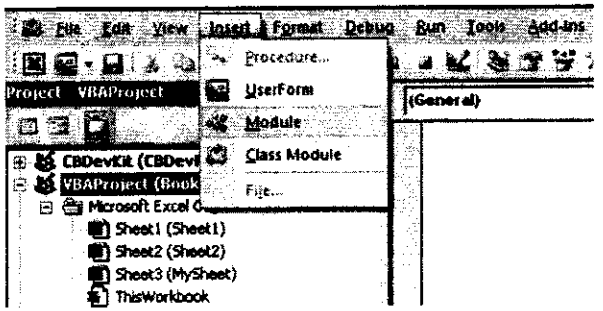
Now a bit of terminology that will ease your understanding of what you see: When an Excel file is open in Excel, it is referred to as a Workbook. The one you just opened is probably called Book1. Each workbook contains a collection of Worksheets. These are probably called Sheet1, Sheet2, and Sheet3 in your workbook. The Project Explorer lists the sheets of your workbook by stating the sheet number followed by the name of the sheet. In Excel I changed the name of sheet 3 to MySheet. Note how that looks in Figure 1.3.

Below the Project Explorer you find the *Properties Window*. Here you can see and change various properties of your workbooks, worksheets, modules, and user forms. You will not need this window for quite a while, more specifically before Chapter 16, so I suggest that you close it until further notice.

1.4 Insert a Module

You have now become somewhat familiar with the VBA Editor, and you know what the various parts of it can be used for. It is now time to create your very first program.

Code is located in something called a *Module*. You can think of a module as a container that contains one or several programs. A module is contained in a workbook just like worksheets, but they are only used for writing code and you cannot see them in Excel, only in the VBA Editor and the Project Explorer. You can have as much code as you wish in one module, and as many modules in a workbook as you wish.



1.6 INSERT A MODULE.

It is possible to have code elsewhere, for example behind a user form as we shall see in Chapter 16. But for now, modules are the only place where you should write code.

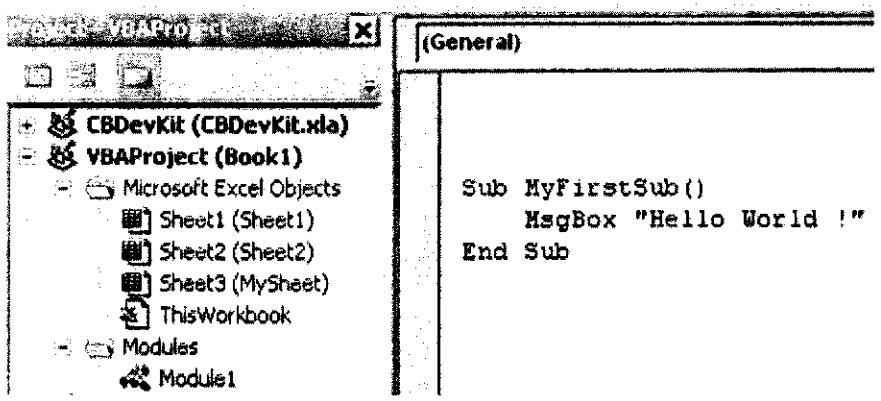
Therefore, the first thing you need to do when you want to write a program is to insert the module. Make sure that the workbook where you

want to insert your module is highlighted in the Project Explorer. There are several ways to insert a module in your project. The easiest way is to open the VBA Editor, choose the menu *Insert*, and select *Module*, as illustrated in Figure 1.6. Alternatively, select the *insert* control on the tool bar and choose *Module*. Notice that it now appears in the Project Explorer as you can see in Figure 1.7.

1.5 Write Your First Macro

Almost every book about programming starts with a small program that displays the text *Hello World!* on the monitor. This book is no exception.

To open a module for writing, double-click on it in the Project Explorer. Now type in the three lines shown in Figure 1.7 in the Code Window. At this point you are not to worry about understanding what you write. After working your way through Chapter 3 you will have the full understanding about those strange looking lines of text.



1.7 THE FIRST MACRO.

1.6 Execute your First Macro

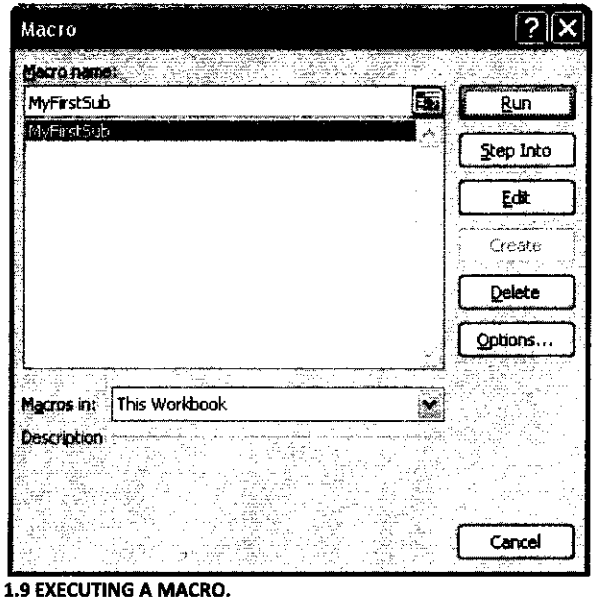
In the previous section you wrote a small program. But writing programs is neither fun nor useful if you cannot execute them. There are many ways to execute a macro. Here we explain some of them. Several more advanced ways to execute a macro are explained in Chapter 11.

In the VBA Editor, place the cursor on one of the three lines of code you just wrote. Press the green triangle control on the tool bar. Alternatively, place the cursor on the code and press F5. In both cases the message box shown in Figure 1.8 will be displayed.

1.8 OUTPUT FROM THE FIRST MACRO

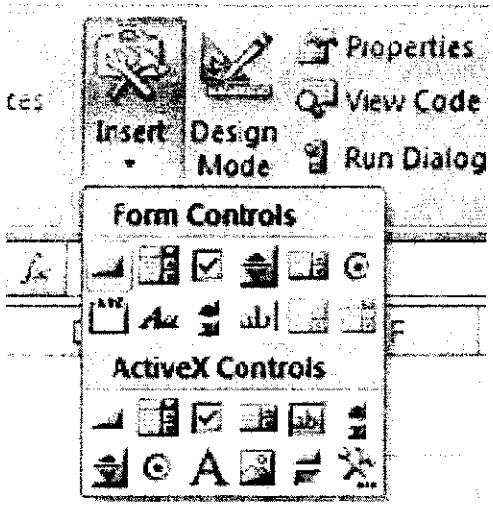
While you are programming, the methods above are good ways to execute your code to see that everything is working correctly. But after creating the program it is very inconvenient to have to open the VBA Editor in order to execute a macro. Therefore, macros can also be executed directly from Excel.

Return to Excel by using the Excel control on the toolbar or by using *Alt+Tab* to swap. On the Developer tab, the second control is called *Macros*. Click it to obtain the dialog box displayed in Figure 1.9. Select the macro you want to execute, in this case the one called MyFirstSub, and press *Run*. This will also execute your macro.



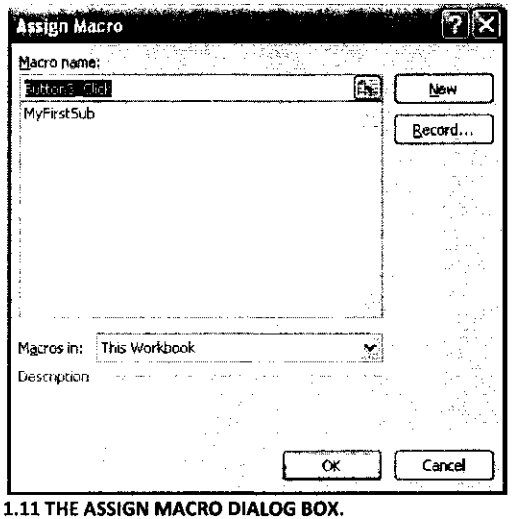
1.9 EXECUTING A MACRO.

The methods above are okay if you only need to execute your macro a couple of times. But if you are going to use it many times, or if someone else is going to use it, you probably would like to be able to execute your macro by pressing a button on the worksheet. Here is how you insert a button: From the Developer tab, select *Insert* to obtain the options displayed in Figure 1.10.



1.10 INSERT A BUTTON.

Choosing the upper left *Form Control*, and clicking somewhere on the worksheet will insert the button. It can be resized and relocated subsequently. The *Assign Macro* dialog box shown in Figure 1.11 is automatically displayed. Select the macro you want to execute when the button is pressed and press *OK*. Now, pressing the button will execute your macro.

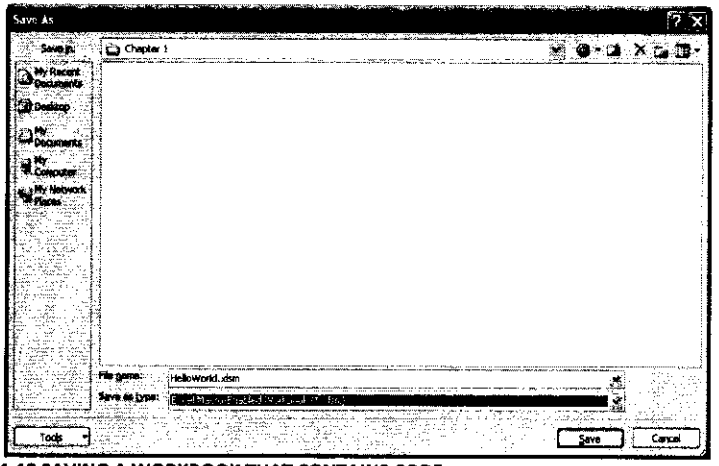


1.11 THE ASSIGN MACRO DIALOG BOX.

You are now familiar with the VBA Editor, you have created your first macro and know how to execute it. In the next chapter you will learn how to use the *macro recorder*, and in Chapter 3 we will write more code and you will start to understand the VBA language.

1.7 Save a Workbook that Contains Macros

When a workbook contains VBA code, it cannot be saved in the standard .xlsx or .xls formats. It must be saved as a macro enabled workbook .xlsm (or xlm if you use a pre-2007 version of Excel) as shown in Figure 1.12.



1.12 SAVING A WORKBOOK THAT CONTAINS CODE.

1.8 Exercises

1.1. Follow the steps described in this chapter and end up with the creation of your own *Hello world!* macro.

1.2. Now, let us see what you learned so far.

1. Insert a new procedure – a sub – and call it *HelloX*.
2. Type the body of the procedure to make it look like this:

```
Public Sub HelloX()
    Dim Name As String
    Name = InputBox("Type your name:")
    MsgBox "Hello " & Name
End Sub
```

3. Make a button and assign the *HelloX* procedure to it.
4. Run the procedure by pressing the button.

Chapter 2
The Macro Recorder

The easiest way to create a macro is to record it. With the *macro recorder* you can record your actions and execute them at a later point. In that way your routine tasks can be performed by a macro, and you can spend your time on other tasks. In Section 2.1, we explained how to use the macro recorder.

The macro recorder generates VBA code. Often you will need to slightly modify the recorded code. In Section 2.2, we will take a look at the generated code and see how it can be modified.

Just like your worksheet functions can use either absolute or relative references, so can the macro recorder. In Section 2.3, we consider the differences between the two.

Even though the macro recorder is simple to use and can generate VBA code, it has many shortcomings. We consider some of these in Section 2.4.

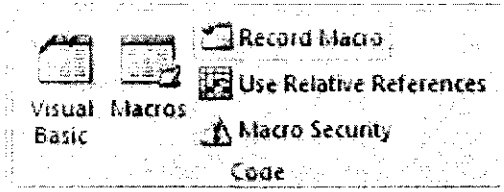
2.1 How to Record a Macro

In this section you will learn how to use the macro recorder to record a macro. The use of the macro recorder will be illustrated through two examples. For both examples, we use the worksheet illustrated in Figure 2.1. The worksheet states the amount of sales per year for each of four areas of a company over a five year period.

	A	B	C	D	E	F
1				Sales		
2						
3		Year 1	Year 2	Year 3	Year 4	Year 5
4	Area 1	230000	245000	256000	156300	174300
5	Area 2	327500	339000	371000	290000	310500
6	Area 3	432000	447900	468000	350500	367800
7	Area 4	210000	225000	234800	224800	221000
8						

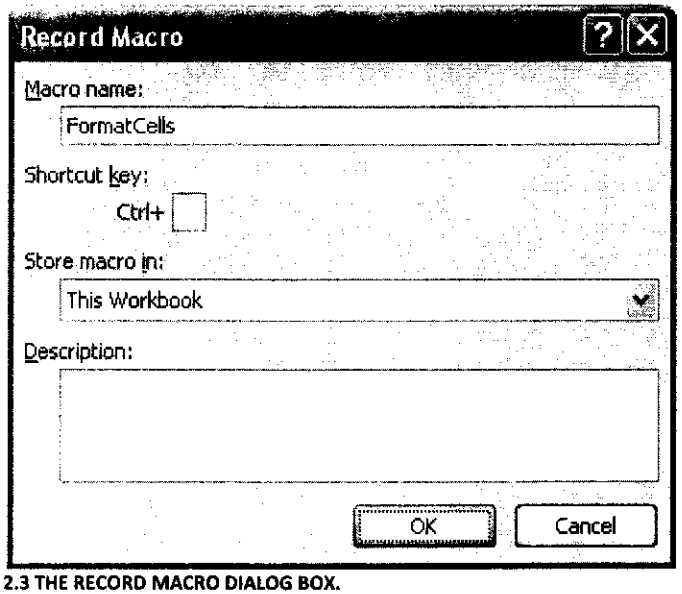
2.1 DATA FOR ILLUSTRATION OF THE MACRO RECORDER.

In the first example we make a macro to format one or several cells. Start by selecting cell B3. Do not change this selection once the recording is started. In order to activate the macro recorder, press *Record Macro* on the Developer tab as illustrated in Figure 2.2.



2.2 PRESS RECORD MACRO TO START RECORDING.

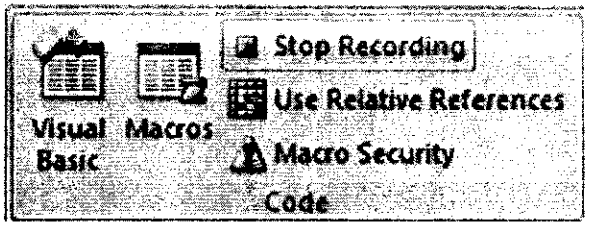
The dialog box shown in Figure 2.3 will be displayed. Here you enter some information about the macro you are about to create: The *name* should be something that relates to the purpose of the macro. We call this macro *FormatCells*. Make sure that the macro is stored in *This Workbook*, which is the default setting. In the *description* field you can write a description of your macro for future use. Click *OK* to start recording.



2.3 THE RECORD MACRO DIALOG BOX.

Note how the Developer tab changes by replacing *Record Macro* by *Stop Recording* as shown in Figure 2.4. Everything you do until you press *Stop Recording* will be recorded and turned into a macro.

We change the format of the selected cell by choosing the *Home* tab and selecting *Bold font, font size 16* and change the *font color* to blue. Click *Stop Recording* to stop the recording of your actions.



2.4 PRESS STOP RECORDING TO STOP THE RECORDER.

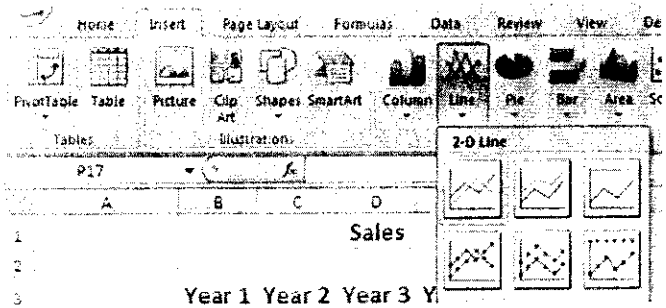
Now it is time to test that the macro is working correctly. Do this by selecting cell C3 and execute the macro as you learned in Chapter 1: On the *Developer* tab, choose *Macros*, select the macro you just recorded and press *Run*. If the macro has been correctly recorded, the formatting of cells B3 and C3 will now be similar. If this is not the case, do try to follow the steps above once again. Note that if you select several cells, the new formatting will apply to all of them when you execute the macro. Executing the macro for every cell with a heading results in the layout shown in Figure 2.5.

	A	B	C	D	E	F
1				Sales		
2						
3		Year 1	Year 2	Year 3	Year 4	Year 5
4	Area 1	230000	245000	256000	156300	174300
5	Area 2	327500	339000	371000	290000	310500
6	Area 3	432000	447900	468000	350500	367800
7	Area 4	210000	225000	234800	224800	221000

2.5 HEADINGS ARE FORMATTED BY THE RECORDED MACRO.

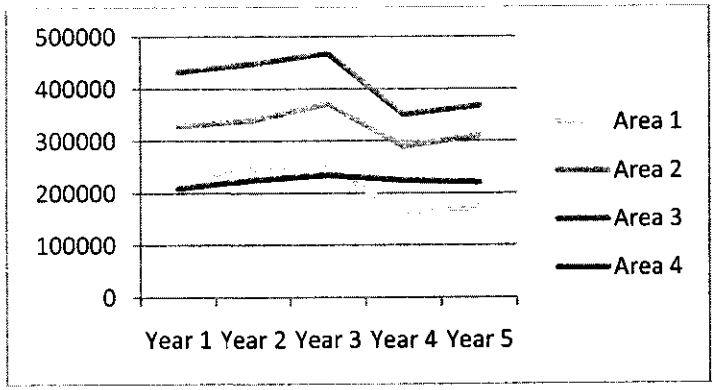
In the second example, we illustrate how a chart can be created by use of the macro recorder. Creating charts is one of the more tricky tasks in VBA programming, mainly because there are so many possibilities. We go into detail with the creation of charts in Chapter 14. Until you reach that chapter, we recommend that you use the macro recorder to create charts. In the next section, we will shortly point out where you need to change the code in order to make it more flexible.

Start by selecting cell A1 and press the *Record Macro* control on the *Developer* tab. Give your macro a name in the Dialog box that appears, for example ShowChart. Select the area A3:F7, which is the area to be displayed. Select the *Insert* tab and choose the first *2-D Line Chart* as illustrated in Figure 2.6. Then stop the macro recorder.



2.6 CREATING A CHART.

When recording this macro, the chart shown in Figure 2.7 will be displayed. Delete the chart and execute the macro to convince yourself that everything is working correctly.



2.7 CHART CREATED BY THE RECORDED MACRO.

2.2 Viewing the Recorded Code

In this section, we will take a look at the code that was recorded in the two examples in the above section. At this point you should not worry about not understanding the details of the code. The purpose is merely to illustrate that even without great programming skills, it is still possible to change the macros slightly.

When you record a macro with the macro recorder, you record your actions, but actually you do a lot more than that: You both record what you do and what you do not do. Here is the explanation. Swap to the *VBA Editor* and take a look at the code you recorded in the first example. It will look like the code in Figure 2.8.

In the code, *Selection* refers to the selected cell or cells. The first real line of code sets Bold to *True*, i.e. states that bold font should be used. The next part of the code changes the Size to 16, but it also states the status of many other font related things which have not been changed, for example that it should not be subscript. This is done even though it is unchanged.

```
Sub FormatCells()
'
' FormatCells Macro
'

'
    Selection.Font.Bold = True
    With Selection.Font
        .Name = "Calibri"
        .Size = 16
        .Strikethrough = False
        .Superscript = False
        .Subscript = False
        .OutlineFont = False
        .Shadow = False
        .Underline = xlUnderlineStyleNone
        .ThemeColor = xlThemeColorLight1
        .TintAndShade = 0
        .ThemeFont = xlThemeFontMinor
    End With
    With Selection.Font
        .Color = -4165632
        .TintAndShade = 0
    End With
End Sub
```

2.8 THE RECORDED CODE.

Every line that states something that is unchanged can be removed without influencing the result. Removing unnecessary lines of code from Figure 2.8 results in the code shown in Figure 2.9.

```
Sub FormatCells()
    Selection.Font.Bold = True
    With Selection.Font
        .Size = 16
    End With
    With Selection.Font
        .Color = -4165632
    End With
End Sub
```

2.9 THE RECORDED CODE AFTER UNNECESSARY CODE HAS BEEN REMOVED.

The code in Figure 2.9 can be cleaned up even further as illustrated in Figure 2.10.

```
Sub FormatCells()
    With Selection.Font
        .Bold = True
        .Size = 16
        .Color = -4165632
    End With
End Sub
```

2.10 FURTHER ADJUSTMENT OF THE CODE IN FIGURE 2.8.

Now consider the *ShowChart* macro, which was recorded in the second example and is shown in Figure 2.11.

```
Sub ShowChart()
'
' ShowChart Macro
'

'
    Range("A3:F7").Select
    ActiveSheet.Shapes.AddChart.Select
    ActiveChart.SetSourceData Source:=Range("'Sheet1'!$A$3:$F$7")
    ActiveChart.ChartType = xlLine
    Range("H3").Select
End Sub
```

2.11 THE RECORDED CODE FOR MAKING CHARTS.

The most interesting line of code for editing purposes is the long one that specifies the *Source*. This line states that the data for the chart should be found on sheet 1 in the range A3:F7. It does not require a lot of imagination to imagine that changing the $F$7 to $F$6 will result in only the first three areas being shown. If you need a chart to display the information that you have stored in the range L9:Q17, enter $L$9:$Q$17 instead.

2.3 Using Relative References

Someone who is used to working with Excel knows the difference between absolute and relative references, but let us shortly refresh this before turning to the use of this in the macro recorder. Consider the small worksheet in Figure 2.12 where the formulas are shown.

	A	B	C
1	1	=A1	=$A$1
2	2		
3			

2.12 ABSOLUTE VERSUS RELATIVE REFERENCES.

Now copying B1:C1 to B2:C2 results in the worksheet shown in Figure 2.13. Why? Well the reference in B1 is relative and points to the cell one step to the left of itself, which when copied to B2 is A2, but the reference in C1 is absolute and points to the specific cell A1, also when copied. The $-sign is the trick.

	A	B	C
1	1	1	1
2	2	2	1

2.13 RESULT OF USING ABSOLUTE VERSUS RELATIVE REFERENCES.

To illustrate the use of *Relative References* which is found right below the *record macro* control on the *Developer* tab, we use the workbook shown in Figure 2.14.

	A	B	C	D	E
1	1			1	
2	2			2	

2.14 DATA FOR USING RELATIVE VERSUS ABSOLUTE REFERENCES.

We record two macros. Firstly select cell A1 and start the macro recorder. Select cell A1 again and press *Ctrl+c* (copy). Then select cell B1 and press *Ctrl+v* (paste). Then stop the recorder. You will see a 1 in cell B1.

Clear cell B1 to be able to test how the macro you just recorded works. Firstly, select cell A1 and execute the macro, then select cell A2 and execute the macro. The result is shown in column B of Figure 2.15.

	A	B	C	D	E
1	1	1		1	1
2	2			2	2

2.15 ILLUSTRATION OF ABSOLUTE AND RELATIVE REFERENCES.

Now activate *Use Relative References* as shown in Figure 2.16. To see the difference we will perform exactly the same steps in columns D and E as follows: Select cell D1 and start the macro recorder. Select cell D1 again and copy. Then select cell E1 and paste. Then stop the recorder. Clear cell