# Introducing VBA Message Boxes

## It's All About Communication

When you build a tool for someone else to use it is important that that person, "the user", knows what's going on and feels confident using it. For this reason it is important that you communicate with the user. The VBA Message Box is an ideal tool for doing this and requires only basic programming skills to create.

Computer users are used to seeing message boxes during the course of their everyday work. If I try to copy a multiple selection in Excel the program displays a message to tell me I can't do that (*Fig. 1*).

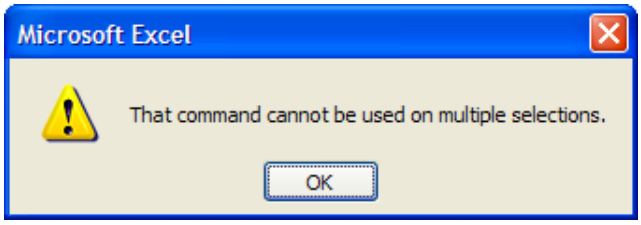


*Fig. 1 Microsoft Excel displays a simple message.*

If I were to try to close the document I'm currently working on Microsoft Word would show me a message box (*Fig. 2*) reminding me that I have not saved the document and asking me if I want to do so.

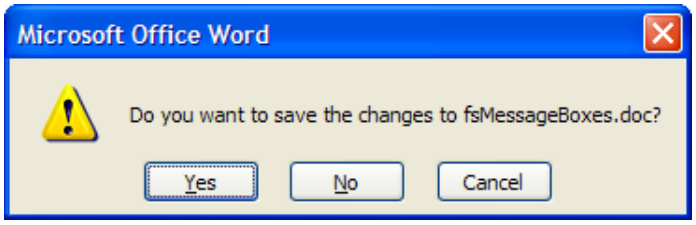


*Fig. 2 Microsoft Word displays a standard Yes/No/Cancel message box.*

Whether you record your own macros or write procedures from scratch with VBA, you can create message boxes like these to enhance your macros with just a few lines of VBA code.

## The Two Basic Types of Message Box

In VBA Message Boxes fall into two basic categories, the *MsgBox method* and the *MsgBox function.*

### *The MsgBox Method*

The *method* is the "verb" of the VBA language and as such carries out some sort of action, in this case displaying a message to the user. It has a button marked **OK** to allow the user to dismiss the message and they must do so before they can continue working in the program.

### *The MsgBox Function*

Like any other function, this one returns a value. Use the MsgBox function to ask the user a question. The question must be one that can be answered with **Yes**, **No**, **OK**, **Cancel**, **Abort**, **Retry** or **Ignore** and the MsgBox function displays a message accompanied by two or more buttons (in certain pre-defined combinations) for the user to make their response. The value returned by the *MsgBox* function identifies which button the user clicked.

Both types of Message Box can also display a button marked **Help** allowing you to direct the user to the appropriate part of a Help file.

## Writing the Code for a Message Box

When you type the keyword *MsgBox* followed a space the Visual Basic Editor displays a panel listing the various parameters appropriate to a Message Box (*Fig. 3*). The parameters are separated by commas. Those enclosed in square brackets are optional.

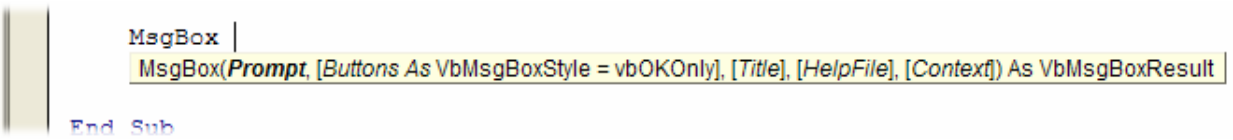


*Fig. 3 The Visual Basic Editor's* Auto Quick Info *feature helps you provide the required parameters.*

The parameter currently highlighted in ***bold*** is the one that the Visual Basic Editor is expecting you to supply now. After entering a parameter you must type a comma to move to the next one. Since some parameters are optional you might want to skip one and move to the next in the list. Type another comma to do this. Do not end the code statement with a comma. It is not necessary to type a comma if you are not going to supply any more parameters. The following table describes the Message Box parameters:

| Parameter | Description |
|---|---|
| **Prompt** | The *Prompt* is the only parameter you must supply. It is the message itself and must be supplied as a text string enclosed in quotes, for example:<br>`"The macro has finished."`<br>If your prompt is longer than the maximum width of a Message Box it will automatically wrap to form as many lines as necessary. To enter a multi-line message use the *vbCrLf* constant to force a line break, for example:<br>`"The macro has finished." & vbCrLf & "A new sheet was added."` |
| **Buttons** | The *Buttons* parameter is optional. If no button is specified a single **OK** button (`vbOKOnly`) is shown. This parameter can be used to specify a number of features: the required combination of buttons, which icon to display (*Fig. 4*), and which button is the default one.<br>  <br>*Fig. 4 The Buttons parameter offers a choice of icons: vbInformation, vbQuestion, vbExclamation and vbCritical.*<br>The options are specified using named constants. When specifying multiple constants enter a plus sign (+) between them, for example:<br>`vbYesNo + vbQuestion`<br>The Visual Basic Editor displays a list of options when you enter the *Buttons* parameter (*Fig. 5*). Click on the one you want to use. To add another type a plus sign to display the list again. |
| **Title** | With this parameter you can optionally specify some text to appear in the *Title Bar* at the top of the Message Box. Supply the *Title* as a text string enclosed in quotes. If no *Title* is specified the name of the host application is displayed. |
| **HelpFile** | If you have created a dedicated Help file to accompany your application you can use the *Help* parameter to specify its location. This automatically causes a **Help** button to be displayed on the Message Box. |
| **Context** | Used in conjunction with the *Help* parameter the *Context* is a number referring to the location within the Help file of the relevant section of help to which the user will be taken when they either click the **Help** button on the Message Box or press the **[F1]** key on their keyboard when the Message Box is open. |

NOTE: If the Visual Basic Editor does not provide the context-sensitive help (*Auto Quick Info* as in *Fig. 3* or *Auto List Members* as in *Fig. 5*) first check your typing. If you made a typing error the Visual Basic Editor will not be prompted to display its help. If help still fails to appear check that it is enabled by going to **Tools > Options > Editor** in the Visual Basic Editor. If the help disappears because, perhaps, you switched to another window or clicked somewhere else, you can force it to return by typing a **[Backspace]** then retyping the comma or space that prompted the help originally.

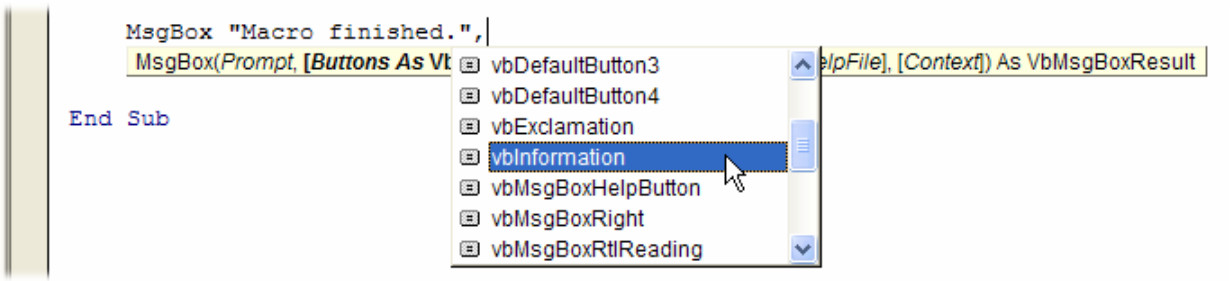


*Fig. 5 The Visual Basic Editor's* Auto List Members *feature offers a list of choices.*

## Displaying a Message Using the MsgBox Method

Use the *MsgBox* method when you want to display a message providing information or maybe a warning that requires no other response from the user than a simple acknowledgement. A very useful application of this tool is to let the user know when a macro has finished its work. When you run a macro it is not always apparent when the macro has completed or, indeed if anything at all has happened! A simple message displayed by the macro on completion will confirm that it has run its course.

In this simple example, the macro adds a new worksheet to the active workbook in Excel. If you want to make it really simple you only have to provide the prompt (*Listing 1*) and your program will display a simple Message Box (*Fig. 6*).

*Listing 1:*

```
Sub AddWorksheet()
    Worksheets.Add
    MsgBox "Macro finished."
End Sub
```

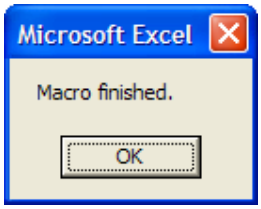


*Fig. 6 A simple Message Box with only the Prompt supplied.*

You can customise the Message Box further by adding an icon and a title, and maybe a little more information in the message (*Listing 2, Fig. 7*).

*Listing 2:*

```
Sub AddWorksheet()
    Worksheets.Add
    MsgBox "A new worksheet has been added." _
        , vbInformation + vbOKOnly, "Macro finished"
End Sub
```

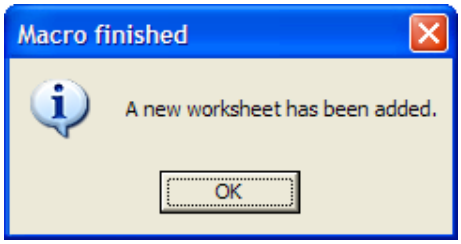


*Fig. 7 This Message Box displays an icon and a custom Title*

NOTE: In Listing 2 I have specified *vbOKOnly* in the *Buttons* parameter but since this is the default I could have omitted it and achieved the same result. Also, I have written the MsgBox statement on two lines, breaking the statement with a **[Space]** and **[Underscore]**. This was done only to fit the code on this written page. The statement could equally have been written as a single line.

With some modification to the macro additional information can be automatically displayed in the Message Box (*Listing 3*, *Fig. 8*).

*Listing 3:*

```
Sub AddWorksheet()
    Dim sht As Worksheet
    Set sht = Worksheets.Add
    MsgBox "A new worksheet has been added." & vbCrLf & _
        "The new sheet is: " & sht.Name _
        , vbInformation + vbOKOnly, "Macro finished"
End Sub
```

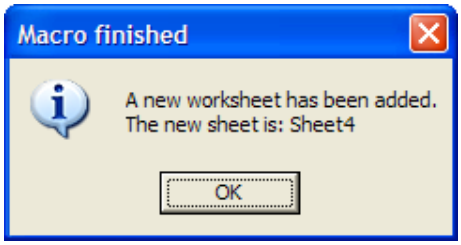


*Fig. 8 This message contains a forced line break and some variable data.*

In this example, the new worksheet is declared as an object variable so that its name can be retrieved and included in the message. The *MsgBox* statement also includes a forced line break using the *vbCrLf* constant to display the message on two lines.

## Offering a Choice Using the MsgBox Function

A multi-button Message Box allows your macro to interact with the user by offering them a number of choices. When the user clicks one of the buttons it returns the value of the constant represented by that button. Your code can then make use of this value to take an appropriate course of action. This is achieved by reading the value returned by the MsgBox function directly into a variable then using an *If Statement* or *Case Statement* to execute the relevant code.

Note that, when *MsgBox* is used as a function its parameters must be enclosed in parentheses (round brackets).

### *Checking the User's Response with an If Statement*

In this example the user is asked to confirm their request to add a new worksheet (*Fig. 9*).

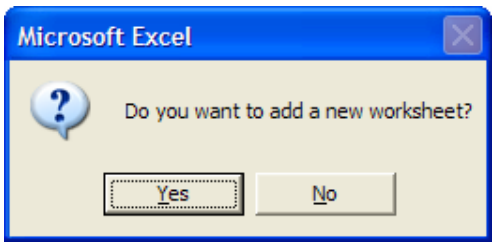


*Fig. 9 A Yes/No Message Box asks the user for confirmation.*

The code (*Listing 4*) shows that the user's choice is recorded in the variable named "Response" (I usually use this name but any relevant and VBA legal name, such as "Answer" or "Choice" will do) which is then examined by the *If Statement*. If the user clicked the **No** button all that is necessary is to cancel the macro. The expression *Exit Sub* is used for this. Since there can only be one other response (**Yes**) any code following the *If Statement* will be executed if that choice is made.

*Listing 4:*

```
Sub AddWorksheet()
    Dim Response As VbMsgBoxResult
    Response = MsgBox("Do you want to add a new worksheet?", vbQuestion + vbYesNo)
    If Response = vbNo Then Exit Sub
    Worksheets.Add
    MsgBox "A new worksheet has been added.", vbInformation
End Sub
```

The *If Statement* can be elaborated if, for example, you want to confirm the cancellation of the macro. In this example (*Listing 5*) the *Exit Sub* expression has been omitted since the macro terminates immediately after the *If Statement* anyway.

*Listing 5:*

```
Sub AddWorksheet()
    Dim Response As VbMsgBoxResult
    Response = MsgBox("Do you want to add a new worksheet?", vbQuestion + vbYesNo)
    If Response = vbNo Then
        MsgBox "No worksheet was added", vbInformation
    Else
        Worksheets.Add
        MsgBox "A new worksheet has been added.", vbInformation
    End If
End Sub
```

If more than two choices are offered (e.g. **Yes**, **No**, **Cancel**) the *If Statement* must have an additional clause. The next example (*Fig. 10*) offers to give a particular name to the new worksheet.

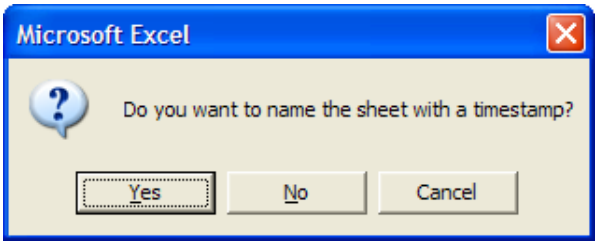


*Fig. 10 This Message Box offers three choices.*

There are three possible courses of action so the *If Statement* is a little more complex (*Listing 6*).

*Listing 6:*

```
Sub AddWorksheet()
    Dim Response As VbMsgBoxResult
    Dim sht As Worksheet
    Response = MsgBox("Do you want to name the sheet with a timestamp?" _
        , vbQuestion + vbYesNoCancel)
    If Response = vbYes Then
        Set sht = Worksheets.Add
        sht.Name = Format(Now, "yyyy-mm-dd_hhnnss")
    ElseIf Response = vbNo Then
        Worksheets.Add
    Else
        Exit Sub
    End If
    MsgBox "Macro finished.", vbInformation
End Sub
```

Whether the user chooses **Yes** or **No** a sheet is created and both choices continue after the *If Statement* to show a final Message Box confirming completion of the macro. If the user did not click either **Yes** or **No** then they must have clicked **Cancel** so this response can be dealt with in the *Else* part of the *If Statement*.

NOTE: I chose to name the worksheet using a timestamp since this is one way to ensure that each is given a unique name. Attempting to give a worksheet a name that already exists would cause an error. This isn't a problem since I could write code to deal with this eventuality but for these examples I wanted to keep the code simple.

### *Checking the User's Response with a Case Statement*

The decision to use an *If Statement* or a *Case Statement* is often a matter of personal choice. I usually prefer to use a *Case Statement* when there is a larger number of choices because the syntax is simpler and easier to read and understand.

In this example (*Listing 7*) a clause, or *Case*, has been included for each possible button choice, although I could have used the expression *Case Else* as a "catch all" for the third choice instead of *Case vbCancel* since if the user had not chosen **Yes** or **No** the only remaining option is **Cancel**.

*Listing 7:*

```
Sub AddWorksheet()
    Dim Response As VbMsgBoxResult
    Dim sht As Worksheet
    Response = MsgBox("Do you want to name the sheet with a timestamp?" _
        , vbQuestion + vbYesNoCancel)
    Select Case Response
        Case vbYes
            Set sht = Worksheets.Add
            sht.Name = Format(Now, "yyyy-mm-dd_hhnnss")
        Case vbNo
            Worksheets.Add
        Case vbCancel
            Exit Sub
    End Select
    MsgBox "Macro finished.", vbInformation
End Sub
```

## Defining the Default Button

When a Message Box has more than one button, one of them is the default. It appears highlighted when the Message Box is displayed and is automatically "clicked" if the user presses the **[Enter]** key on their keyboard. Unfortunately, people are often eager to proceed and press **[Enter]** without properly reading the message, sometimes with disastrous results!

Unless you decide otherwise the first (leftmost) button is the default which is usually **Yes** or **OK** but you can help the user inadvertently making the wrong choice by defining which button is the default, and making this the safest option. Do this by adding an additional constant to the *Buttons* parameter.

Compare the two message boxes illustrated here (*Fig. 11*). For the Message Box on the left no default button was specified so the first button (**Yes**) is highlighted. The *MsgBox* statement for the Message Box on the right uses the *vbDefaultButton2* constant to make the **No** button the default. You can choose *vbDefaultButton1* to *vbDefaultButton4* (counting from left to right).

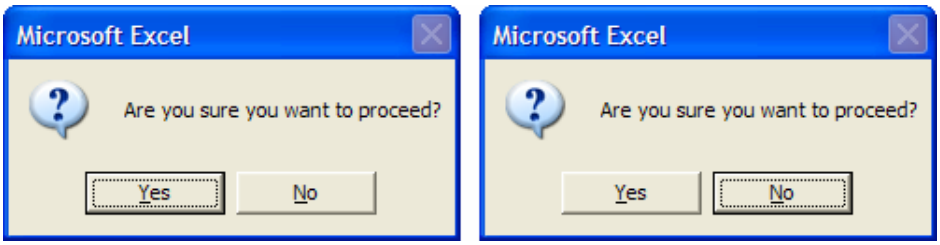


*Fig. 11 You can specify which button is the default.*

NOTE: For single-button Message Boxes and multi-button Message Boxes having a **Cancel** button the user can dismiss the message by clicking its close button or by pressing the **[Esc]** key on their keyboard. If a multi-button Message Box does not include a **Cancel** button its close button is automatically disabled to force the user to make a choice from the available buttons.

## Programming Note

In these examples the *Response* variable has been declared as the VbMsgBoxResult data type (indicating that it refers to one of the collection of VBA Message Box constants). This data type was introduced with Office 2000 so, if you are programming for Office 97 (or might require your macros to be compatible with this version) you should use the *Integer* data type instead, for example:

```
Dim Response As Integer
```

Since each constant also has a numerical value the Integer data type will suffice in this context.