



**Publication**

**How to integrate VBA into a spreadsheet model**

## Introduction

The purpose of this article is to share some experience of integrating VBA functionality into complex spreadsheet models. A typical model is a financial model, which has many input assumptions, performs calculations and produces outputs, for example, in the form of financial projections. Most calculations will be performed through formulae in Excel, but sometimes VBA will be needed to automate certain calculations or to perform some complex calculations.

As soon as you decide to enhance your model with VBA, you would need to find answers to the following questions:

- 1) How do I read data from the spreadsheet into VBA?
- 2) How do I trigger execution of the VBA code?
- 3) How do I output data from VBA back to the spreadsheet?

Obviously, there are many possible solutions to the above problems. But not all of them would be equally robust and reliable. This article aims at clarifying what, in our view, are good practices of integration of VBA code with spreadsheets.

### 1. How to read data from the spreadsheet

A spreadsheet is organised in rows and columns. In fact the only way to locate data in it is somehow to specify the row and column numbers. The simplest way of doing that is simply to type the required references in VBA, e.g.: `Cells(2,6)`, or `Range("F2")`, or `Range("$A$2:$F$4")`, etc. The problem here is that such references will break as soon as the user inserts or deletes columns or rows in the spreadsheet before the referenced range. This is because Excel does not update any VBA code when such changes occur, in contrast to updating all affected spreadsheet formulae automatically.

For this reason, the only robust way of accessing spreadsheet cells from VBA is through defined names. They are treated by Excel in the same way as formulae, i.e. if a name contains a reference to a range, the reference is automatically updated for any changes in the row and/or column numbers.

So, you would define a name, e.g. "RangeName1", make it refer to some range, e.g. "`=Sheet1!$A$2:$F$4`", and simply refer to the name in VBA instead of the actual reference, i.e. `Range("RangeName1")` instead of `Range("Sheet1!$A$2:$F$4")`.

#### ***Two types of defined names***

When using defined names in VBA code it is important to remember that there are two types of them:

- a. *Workbook level names*, i.e. which belong to the whole spreadsheet (Excel creates such names by default), and
- b. *Worksheet level names*, i.e. which belong to a particular sheet in the spreadsheet (Excel creates them when you indicate the sheet name with an exclamation mark before the name when defining it, e.g. `Sheet1!RangeName1`).

To ensure VBA finds the right defined name, you should specify the right parent object before the Range object that you are trying to access. For example, if you are using a defined name that belongs to a worksheet with the code name "Sheet1", you should type `Sheet1.Range("Name1")`. If

you are using a workbook level name, you should type `Application.Range("Name1")` or just `Range("Name1")`. But remember that the second option will not work if some other workbook is active during the macro execution. So you may want to put the `ThisWorkbook.Activate` command at the start of all your macros.

### ***Calculating column numbers***

So now we know how to access a particular cell range from VBA. But what if that range contains a whole table of data with many rows and several columns? You would probably want to read the whole table into a variant array and then work with that array, rather than access the cells individually. This is much more efficient, usually saving a lot of execution time. So we would use something like this:

```
Dim Table as Variant
Table = Sheet1.Range("Table1").Value
```

But how would you access particular columns in this array, for example the third one called "Customer Name"? You should not use the number 3 as a plain constant in your code, because there is a risk that the user may insert another column in the table and make this number incorrect. So what do you do?

The answer is again to use defined names. You would need to name every column in the table which you will be accessing. You would then calculate the right column number as follows:

```
MyColumnNumber = Sheet1.Range("CustomerNameColumn").Column - _
    Sheet1.Range("Table1").Column + 1
```

You can then access this particular column in the table using `Table(SomeRowNumber, MyColumnNumber)`.

### ***Accessing a large number of ranges***

The above methods are good if you have relatively few ranges to access from VBA. But what if your VBA code required data from dozens of different ranges located in different parts of the spreadsheet? Of course, you could just use many defined names. But it would probably be time consuming to set up and may be inconvenient to maintain in the long run.

A good alternative would be to create a special table somewhere on the spreadsheet which would contain links to all the areas required by the VBA code. Since Excel automatically updates cell links for any column or row changes, the table will always point to the right ranges.

Such table would have two columns: a unique name for the range (we will call it a key, e.g. "Name1") and a link to the range itself (e.g. "Sheet1!\$A\$2:\$F\$6"). The links may point to many cells at once showing #VALUE as a result. Let us assume that the table has a defined name "Map". We could read all the references from the table into a collection and then use it as a middle interface to access the ranges. This is achieved by the following VBA code:

```
Dim colMap As Collection, vMap As Variant
Dim sKey As String, sLink As String
Dim i As Long

Set colMap = New Collection
vMap = Application.Range("Map").Formula
For i = LBound(vMap, 1) To UBound(vMap, 1)
    sKey = vMap(i, 1)
    sLink = vMap(i, 2)
    colMap.Add sLink, sKey
Next i
```

You can then access a particular range via its key in the collection as follows:

```
Range(colMap("Name1"))
```

The suggested approach provides the same result as if all the ranges were named with their "keys", but it is usually easier to set up and maintain.

### ***Ring-fencing the data access code***

It has become a standard practice in software development to separate code in any project into three logical tiers: data access, business logic and user interface. This allows to extend or upgrade any of the tiers independently in the future. It also improves the robustness of the software by reducing the number of hidden dependencies between different parts of the code. It is usually worth following the same principles in Excel development.

So we should separate the code that reads data from the spreadsheet from the rest of the code. It is probably best achieved by placing it in a separate *class module*. You can have module level variables in it, such as the colMap collection described above, or some data storage arrays that you would use instead of spreadsheet cells to speed up access. A class module is more suitable than a regular module for two reasons. First, by explicitly creating and destroying the instance of the class, you can control when the module level variables are dimensioned and populated and also make sure they are disposed of properly. Second, a class module will provide you with good means of exposing the data to the rest of the code – through custom properties and functions of the class.

## **2. How to trigger execution of the VBA code**

There are four ways how a macro execution can be started in Excel:

- 1) Through controls placed on worksheets, such as buttons
- 2) Through keyboard shortcuts
- 3) Through controls added to the Excel menu and toolbars
- 4) Through events

Only some of the above methods will be needed or appropriate for a particular VBA project. In order to choose the optimal solution, it is important to understand their key characteristics.

### ***Controls on worksheets***

This is probably the easiest and most evident way of providing access for the user to VBA macros – by putting buttons and other controls (e.g. list boxes, option selectors) directly on a worksheet. However, it may not be always optimal for the following reasons:

- The user would always need to select the right worksheet before they can execute the macro
- It is not always easy to organise controls on a worksheet in a clear way, especially if it also contains data.
- It ties the VBA code to a particular spreadsheet, which may not always be desirable.

Overall, placing controls on worksheets is usually appropriate for simple VBA projects and spreadsheet models. But it may not be the right choice for projects that are designed to work with many spreadsheets, such as Excel add-ins.

### ***Keyboard shortcuts***

Assigning keyboard keys to macros allows the user to execute them quickly and without the need to locate the right controls on a worksheet. The disadvantage is that the user must first be aware of the shortcuts. So it is a good method to use in combination with the controls, but probably not as the only one. It is also a good way of addressing the first issue with worksheet controls noted above, i.e. the need for the user to activate the right worksheet before executing the macro.

You would typically place the assignment of keyboard shortcuts in the auto-start routine of your project. You would use the Application.OnKey method for that. It is important to remove the shortcuts when your spreadsheet is closed. You can use the same Application.OnKey method (but with the second parameter empty) placed in the Workbook\_BeforeClose event handler for the workbook.

### ***Controls on Excel menus and toolbars***

This is arguably the most professional way of implementing user controls in Excel. And it is usually the only appropriate method for Excel add-ins. However, it requires more effort to implement and may not be worth it for other types of VBA projects, such as financial models. In particular, there are the following difficulties with this method:

- The application must ensure that the custom menus and toolbars are appropriately created when the program loads and removed when it is closed.
- Creating custom controls often requires programming relatively lengthy code which may not be easy to maintain. There are techniques to address this issue (e.g. a table driven approach), but they are quite complex.
- Professionally looking menus would also require icons and they are not easy to obtain. You should avoid using the built-in Excel icons for custom controls as it may lead to confusion.

Overall, the costs and benefits should be weighed carefully before deciding to go down this route for a particular project.

### ***Events***

Another way of starting a macro in Excel is by "trapping" a workbook or worksheet event. You can do so by placing code in a special routine, called an event handler, which is created in the code section of either a worksheet or a workbook in Project Explorer. Examples of event handlers are the routines Worksheet\_SelectionChange, Worksheet\_Activate, and Workbook\_Open.

This approach can be very helpful if you want the macro to run when a certain action is performed either by the user (e.g. when a particular cell is selected) or by Excel itself (e.g. when recalculation occurs). For example, by trapping the Worksheet\_Change event and checking whether a particular

cell with a drop-down validation on it has changed, you can effectively transform this cell into a macro selector. However, there are a couple things to bear in mind about events:

- Users usually do not expect any VBA code to be triggered when they just perform an action on a spreadsheet. So bear in mind unexpected error messages or slow execution of macros.
- Make sure you do not place any slow code in events that are triggered many times in Excel, such as `Worksheet_Calculate`.

### 3. How to output data back to the spreadsheet

If you are developing a spreadsheet model where some calculations are performed in VBA, you will need to somehow output the results of those calculations back to the spreadsheet. Most often these outputs will be used in further calculations in the spreadsheet. For reasons described in the first part of this article, you should not simply use constant references to spreadsheet ranges in VBA. So you would use defined names in a way similar to:

```
Range("Name2").Value = OutputTable
```

Here the `OutputTable` variable would be a two-dimensional variant array, populated with the right calculated data. You could also use the "map" approach described above for locating output ranges, similarly to input ranges.

#### *Using custom formulae*

An alternative approach of transferring data back to the spreadsheet is to use custom worksheet formulae. It is arguably better than simply writing values to cell ranges, as it fits well with the structure of the model where most of the calculations are performed via formulae. You also won't need to keep any placeholders for the output ranges, ensuring that their dimensions are not changed during the model development (especially if you use array formulae as described below).

However, outputting data through custom formulae is not very straight forward to implement. There are several things to be aware of.

First, the routines behind the custom formulae must have access to the necessary data to be able to return it. Assuming that your main calculations are performed when some events are triggered (e.g. the user clicks on a "Calculate" button or changes some inputs), the results of those calculations should be available to the custom formulae via a global variable. The formulae routines could then simply access that variable and return the right data from it. It is probably best to make the global variable an instance of a special class module, devoted to storing the calculation results. This approach would be in line with the principle of separating the code for data input and output.

Second, you should remember that Excel recalculates a custom formula only when any of its input parameters change. So you need to provide at least one input parameter to all the output formulae and ensure that it changes every time your VBA calculations take place. This can be relatively easily implemented by making the formulae refer to a single cell, where your VBA calculation routine would write a large random number every time it is executed.

Third, whenever possible you should use formulae returning arrays of data rather than values for individual cells. This is because custom formulae are inevitably slow and returning data for numerous cells individually through them can dramatically slow down calculation in Excel. For example, if you need to output values for a series of time periods, say from 1 to 60, you should return the whole series of values by one array formula covering the 60 cells at once, rather than having 60 separate formulae, each with a period number as an input.

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher. Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks. While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.