PhUSE EU Connect 2018

Paper SM07

How to write SAS code using Microsoft Visual Basic for Applications

Hansjörg Frenzel, PRA Health Sciences, Mannheim, Germany

ABSTRACT

Complex tasks like creating tables, figures, and listings, are usually solved in Clinical Programming departments by writing bespoke SAS® programs. Even when using validated macros, bespoke SAS programs require special considerations and attention during quality control. Alternatively, Microsoft Excel® may be chosen to organize metadata that controls program execution. Often SAS macros are used to read this data and write SAS programs automatically. With Excel this task can be rather cumbersome.

But there is not only SAS. The paper demonstrates at a beginner level how to use Visual Basic for Applications™ (VBA) to write SAS programs: how to read metadata from an Excel workbook and how to use them to write a text file. Special emphasis is put on the advantages VBA has compared to SAS with regards to reading cells of different data type and dealing with characters that require macro quoting in SAS.

INTRODUCTION

Clinical programmers are used to working with the SAS programming language. SAS is quite powerful and provides tools to access the file system, access a multitude of files created by other applications (like Microsoft Office® applications), write SAS code, and of course create graphics and calculate various statistics. But just because SAS can do a lot, does not automatically mean that it is the best tool for a job.

When building an application using Microsoft Office, however, SAS might not be the first choice. Not only is it cumbersome to integrate SAS processes into an interactive application, it also has nothing to offer to add interactive elements to the application. Often, Excel is solely used for data storage, and SAS processes are executed from outside the application to extract data from it and finish a task. In such cases, SAS struggles coping with Excel's ability of changing data types between individual cells. Trying this without appropriate precautions can lead to data loss[1]. Finally, it can be break down when the workbook contains metadata that demand macro quoting.

But luckily, Excel has its own programming language, Visual Basic for Application (VBA). VBA facilitates the automation of processes when using Microsoft Office Applications, and is easily accessible for most SAS Programmers. In most of its modules SAS is a procedural programming language, while VBA is regarded as an object-oriented programming language. In many respects it is more like other programming languages, e.g. C®, C++®, Java®, Python®, or Perl®. Using VBA, SAS programmers can hone some programming techniques seldomly used with SAS, and thus enhance their toolbox of options and programming solutions.

This paper introduces a simple interactive application that stores data of different data types and uses them to write a simple SAS "Hello World" program. While the application does not seem very sophisticated, it possesses the minimum functionality needed to build a system that e.g. stores tables, listings and figures (TFL) metadata and uses it to write complex SAS programs. It is the first step of building a metadatadriven application to automate aspects of SAS programming while making SAS programs part of the deliverable

INTRODUCTION TO VBA

As already mentioned, VBA differs from SAS in some important aspects:

1. Rather than providing functionality through a finite list of procedures, VBA is extensible by using libraries of class definitions to expose elements of the Office applications as objects.
2. While SAS code is often written linearly, with one data step or procedure call following the next, VBA is sub-routine based, and requires capsuling almost all code within sub-routines. Consequently, VBA programs start within a sub-routine and often immediately jump out to the next sub-routine.
3. SAS stores data in data sets, which exist outside program execution, and only rarely uses them to control the execution of code. For this purpose, it uses the macro language. In VBA, data are stored in variables that almost always influence code execution. Variables only live within their sub-routines, unless they are declared as `Global`. They are only accessible from within their sub-routine and they cease to exist once the sub-routine finished execution. In addition, data can be stored in the respective application that stores or executes the VBA code.
4. While SAS code is in most cases submitted from the SAS Editor, or executed from the command line, Excel VBA is event driven, i.e. the execution of code is always triggered by an event (e.g. open the workbook, close the workbook, move to another cells, push a button). This gives a programmer many options to let users trigger the execution of code from the graphical user interface.

Its flexibility and these features lead to VBA being the superior application programming language when automating processes based on Microsoft Office applications.

OBJECT-ORIENTED VBA

For a programmer with limited exposure to object-oriented programming (OOP) languages, its terminology can be quite confusing. *Classes*, *Objects*, *Properties* and *Methods* are terms used within the realm of those languages and sound alien to most SAS users. What makes VBA especially confusing for beginners is that objects, properties, and methods often have similar names. In addition, VBA allows specific variables of data type *Object*, which are generally used to access object properties and methods during programming. This leads to beginners often being confused with how to handle a certain element.

VBA ELEMENTS

Before starting to describe the application, a short introduction to some object related terminology:

Classes: A class is a container for VBA code to define objects. Therefore, classes are often referred to as being *blueprints* for objects. Within classes *properties* of objects are defined as well as the *methods* to access, manipulate, or return these properties.

Objects: Objects are instances of a class[2]. Within a project, multiple objects can be created from the same class definition and used in parallel in the same session. VBA is set up so that objects defined in the Excel Object library represent elements of the Excel application, e.g. workbooks, worksheets, Excel tables, ranges, etc. In addition, programmers can define their own classes, which do not need to coincide with the Excel application, but can e.g. be complex data types used to store data efficiently in memory.

Properties: Properties are variables stored within an object. The Excel object library defines properties of objects in such a way, that changing their value leads to changes in the (appearance of) application elements. Properties can represent simple variables, like the *Range.Value* property, which stores the value entered in cells. But a property can also be *object variables*, like *Worksheet.Range,* which itself stores a *Range* object, which in turn has specific properties[3]. Although being very powerful, this feature makes VBA to a certain degree confusing for the OOP novice.

Methods: Methods are sub-routines included in the class definition, used to access/alter properties (e.g. Worksheet.Delete, Workbook.Open, Range.ClearFormat). In the Excel object library, their names often resemble the functionality available to the user via ribbon elements or the context menu.

The VBA documentation is a little inconsistent in its terminology. In some of the documentation[4], "Property" refers to a specific set of sub-routines, declared by using the keyword "Property", which are used to access the "Member Variables" defined within the class definition. In this paper, we will follow the commonly used terminology.

VARIABLES AND PROGRAMMING STATEMENTS:

Beside using objects, VBA provides a wide range of elements that can also be found in other programming languages, like primitive data types, arrays, lists, looping, branching, and sub-routines. Below only those VBA language elements are described that are used within the applications described in this paper. For a more exhaustive list, please refer to the suggestions in the recommended reading section.

Variables (primitive data types):

Variables are containers for values that need to be stored and processed during program execution. Variables must be declared to assign them a specific data type. This is done using the `Dim` statement[5]:

```
Dim <variable name> As <data type>
```

To avoid mistyping a variable name, in Excel the compiler option `Explicit` should always be used, which forces the programmer to declare variables, and subsequently checks whether there are undeclared variables (typing errors) and issues a compilation error[6].

Data Type	Comment
Integer	Integer ranging from –32,768 to 32,767
String	0 to 2 billion characters (Fixed String: 1 to 65,400 characters)
Object	Any Access object, ActiveX component or Class object
Variant	This is a universal data type, and the default data type in case no data type is specified during initialization of a variable. It adapts to the contents of the variable, thus can be integers, numbers in floating point representation, strings, or objects.

Please note that this list of VBA data types is incomplete. Other data types are described in the VBA literature.

A good variable naming convention conveys the type, scope and purpose of the variable with a simple visual inspection of its name[7]. A good starting point is the Reddick VBA Naming Conventions[8].

Arrays:

Arrays are a special type of variable, as they can hold multiple values of the same data type. Individual values are accessed via an index rather than an individual name. Arrays can have up to 32 dimensions. The nearest element in SAS resembling a VBA Array is a SAS temporary array. Arrays are declared similarly to variables using a `Dim` statement but require their dimensions to be defined within the statement. Below an example for a two-dimensional array:

```
Dim intPosition(0 to 20, 0 to 50) As Integer
```

Collections:

A Collection Object is an ordered set of items that can be referred to as units[9]. It is simplest to view collections as one-dimensional lists. Different from Arrays, these members of a collection do not need to be of the same data type. Collections can contain complex data types as well as other objects. Collections are used in VBA to access parts of the application, such as workbooks, worksheets, tables (ListObjects), and many others. A collection must be declared as a collection object, and assigned using the `Set` statement:

```
Dim myCollection As New Collection
Set myCollection = someOtherCollection
```

Members can be added to a collection using the `.Add` method, removed using the `.Remove` method, or accessed using the `.Item` method. The `.Count` method returns the number of members in a collection.

As with every user defined object, it is good practice to destroy it and free up memory space once a collection fulfilled its purpose:

```
Set myCollection = Nothing
```

Branching:

Branching is used to execute code conditionally.

Statement	Description	Example

If – then - else	Statements are wrapped conditionally in IF (starts the branching), ELSEIF (alternative branch), ELSE (remaining cases), END IF (end Statement)	`Dim i As Integer`
		
		`If (I < 5) Then`
		`  Debug.Print "Value I is small"`
		`Elseif (i < 10) Then`
		`  Debug.Print "Value I is bigger"`
		`Else`
		`  Debug.Print "Value I is large"`
		`End If`

Looping:

Looping allows to execute the same code multiple times.

Statement	Description	Example

For – loop	The for loop cycles through a series of values within a given name. Step-size can be controlled by the Step instruction.	`Dim intSum As integer, i As Integer`
		`intSum = 0`
		`For i = 1 To 10`
		`    intSum = intSum + 1`
		`Next I`
		`Debug.Print intSum`

Sub-Routines

Sub-routines are named code blocks, that are executed by calling their names, often with additional parameters. Functions are a subset of sub-routines which return a value. Keywords for defining a sub-routine are:

```
<Private|Public> Sub <Subroutine Name>(<optional parameter declarations>)
    ...
End Sub
```

Similarly, functions are defined as:

```
<Private|Public> Function <Function Name>(<optional parameter declarations>) As
<Return data type>
    ...
    ' Return the result of the calculation
    <Function Name> = resultOfTheCalculation
End Sub
```

In contrast to sub-routines, function definitions require the declaration of a data type for its return value. If not declared, the default return type is Variant. To be able to return a value, somewhere in the body of the function it has to be assigned using the function name.

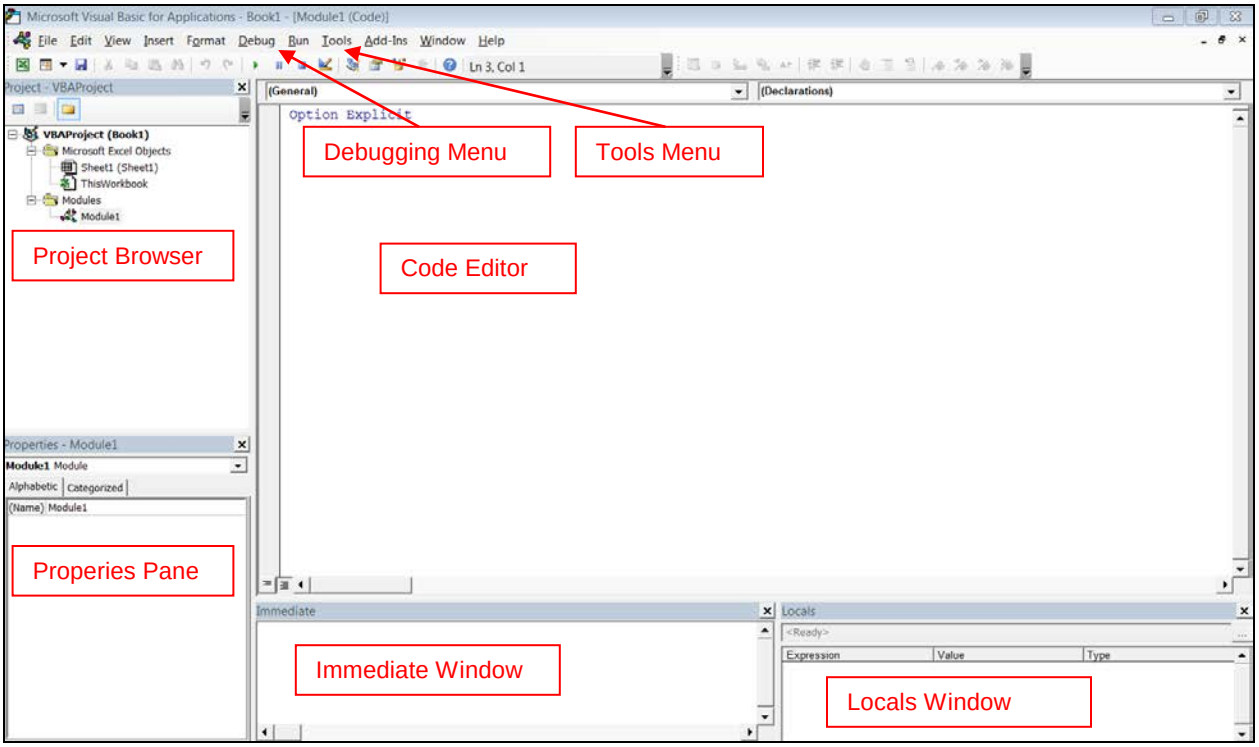
Comments

Comments are text within a VBA program without immediate functionality, often used to further explain a step or a section of a program. Comments are a vital part of good programming practices and help programmers who are not the author to understand the purpose of a specific section of code. This text must be preceded using an apostrophe.

THE VBA EDITOR

Each Microsoft Office application comes with its own VBA editor built into the application. To be able to write VBA programs, a developer simply needs to open the editor (using the **ALT + F11** key combinations, choosing **Macros**

from the **View** ribbon, or choosing **Visual Basic** from the **Developers** ribbon). In case the written program needs to be stored with the Office file, it has to be a macro enabled file. For Excel workbooks this would be an Excel Macro-enabled Workbook (*.xlsm).
After opening the editor, this is how the editor window appears:



Project Browser: Lists all elements of the workbook, modules, forms, and class modules that can contain VBA code.
Properties Pane: Lists all properties that are related to the element selected in the project browser. It contents can change depending on the selected element. The Name property allows changing the name of a VBA module.
Code Editor: This is where VBA code is entered. The editor auto-completes variable names and statements or provides drop down menus to select appropriate entries.
Immediate Window: When printing messages using `Debug.Print`, this window contains the results (the strings). It can also be utilized when entering e.g. a function call directly in the window, preceded by a "?" character.
Locals Window: Lists all variables defined in a sub-routine or outside of a sub-routine, and their current value. Very helpful when debugging a program.
Debugging Menu: Contains menu entries to step-wise execute a program or stop the execution, to set and delete break-points, or run a program to a pre-defined break-point.
Tools Menu: The entry "References" allows loading libraries (e.g. for other Office applications). Entry "Options" allows setting useful options, like "Require Variable Declaration" in the "Editor" tab, which results in an `Option Explicit` entry in the declaration area of each newly added code module. This results in the debugger stopping the compilation of a code module when there are variables used in the code that were not previously declared (e.g. using a `Dim` statement), helping programmers to detect typing errors in variable names.

BUILDING THE APPLICATION

In this section the VBA elements needed to achieve the goal of this paper are described, and all the VBA sub-routines and Excel forms needed for the application are described and developed.

VBA CODE ELEMENTS

The objects that are needed for the application stem from the following libraries :

- From the Microsoft Excel Object Library: Application, Workbook, Worksheet, Range
- From the Microsoft Scripting Runtime: FileSystemObject and TextStream
- From the Microsoft Forms Object Library: CommandButton

Application Object: The Application object contains application wide settings and options, and methods that return top level objects, like ActiveWorkbook, ActiveSheet, or ActiveRange, and the property ThisWorkbook[10]. If not coded explicitly, `Application.` is added implicitly to statements beginning with `ActiveWorkbook.`, `Workbooks.` and similar.

Workbook Object: The Workbook object is a member of the Workbooks collection. The Workbooks collection contains all the Workbook objects currently open in an instance of Microsoft Excel[11]. An open workbook can either be accessed using the workbooks collection in code:

```
Application.Workbooks("Cogs.xlsm").Worksheets("Sheet1").Activate
```

or by use of an object variable:

```
Dim objWB as Excel.Workbook
Set objWB = Application.Workbooks("Cogs.xlsm")
objWB.Worksheets("Sheet1").Activate
```

In case the workbook containing the VBA code needs to be referenced, the `ThisWorkbook` property of the `Application` object can be used:

```
Application.ThisWorkbook.Worksheet("Sheet1").Activate
ThisWorkbook.Worksheet("Sheet1").Activate
```

Worksheet Object: The Worksheet object is a member of the Worksheets collection. The Worksheets collection contains all the Worksheet objects in a workbook[12]. A specific worksheet can be accessed using the following ways:

```
Application.ThisWorkbook.Worksheets("Sheet1").Activate
```

or by assigning it to an object variable:

```
Dim objWS as Excel.Worksheet
Set objWS = ThisWorkbook.Worksheets("Sheet1")
```

Range Object: The Range Object represents a cell, a row, a column, or a selection of multiple cells[13]. A range can be defined by use of the `Range` property of the `Worksheet` object:

```
Dim rngFirst as New Excel.Range
Set rngFirst = ThisWorkbook.Worksheets("Sheet1").Range("A1:B3")
```

There are three ways to specify a range:

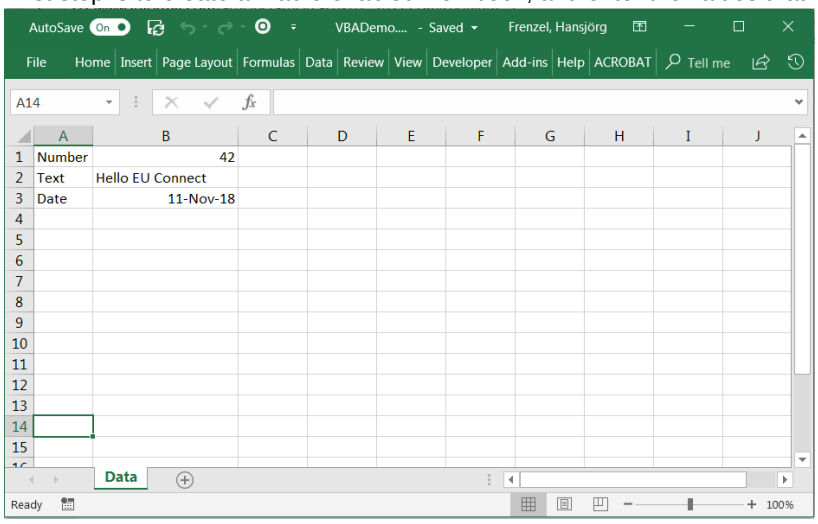
```
ThisWorkbook.Worksheets("Sheet1").Range("A1:B3")
ThisWorkbook.Worksheets("Sheet1").Range(Cells(1,1),Cells(2,3))
ThisWorkbook.Worksheets("Sheet1").Range("namedRange")
```

SCOPE OF VARIABLES

When `Option Explicit` is declared within a VBA code module[6] variables must be declared before they can be used. It is advisable to add this option, as it prevents programmers from accidently creating new variables due to typing errors. Variables can be declared within the declaration area of a module, or within sub-routines. When using the `Dim` statement at the module level, this has the same effect as using the `Private` statement: The variable will be accessible from all procedures within the module, but not from another module. When using the `Global` statement at this point, the variable is accessible from within the whole application. When declared within a sub-routine, a variable is only accessible from within the sub-routine, and only exists as long as the sub-routine is executed. For this application, without exception variables will be declared locally in sub-routines.

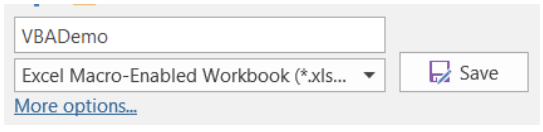
CREATING THE WORKBOOK

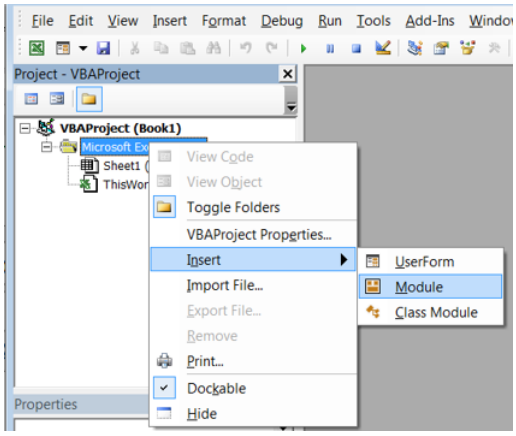
First step is to create a macro enabled workbook, and enter the values that will be used in a SAS program:



The workbook "VBADemo.xlsm" contains a worksheet "Data" with two columns, with three rows each. Column A contains three parameter names ("Number", "Text", "Date"), column B the corresponding values (42, "Hello EU Connect", 11-Nov-18), which are of different data types (Number, String, Date).

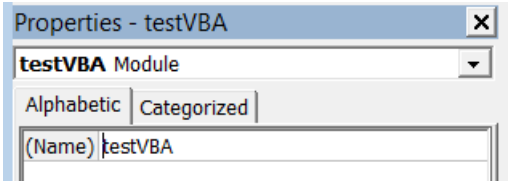
The workbook must be saved as macro enabled workbook:





After opening the VBA Editor, a VBA Module must be added. This is done by right clicking on the Microsoft Excel Objects folder symbol in the Project Browser and selecting "Insert" and "Module".

After inserting the Module, its name can be updated in the properties pane to "testVBA":



FUNCTION LOADDATA()

The code for function loadData() can be added to module testVBA in the VBA Code Editor:

```
Function loadData(strWSName As String) As Variant()
    ' Select the current region of cell A1 and load its
    ' contents into an array of data type variant

    Dim objWS As Worksheet
    Dim rngData As Range

    Set objWS = Application.ThisWorkbook.Worksheets(strWSName)

    Set rngData = objWS.Range("A1").CurrentRegion
    loadData = rngData

    Set rngData = Nothing
    Set objWS = Nothing
End Function
```

Function `loadData` finds all non-emtpy cells around cell A1 and returns their values within an array of data type Variant.

The array that is returned by the function can be visualized like this:

Number	42
Text	Hello EU Connect
Date	11-Nov-18

The function is defined between the `Function` and the `End Function` statements. `Function` allows the programmer to assign the function a name, define parameters, and the data type:

```
Function <function name>(<ByRef|ByVal><parameter declarations>) As <data type>
```

If no parameters are required, the brackets after the function name can remain empty; if multiple parameters are needed, their declaration must be separated by comma. By default, VBA passes a reference to a variable rather than its value. This leads to changes made to the parameter in the called sub-routine being passed through to the variable in the calling sub-routine. Including keyword `ByVal` before the parameter name results in passing the value instead. In the latter case, the value of the variable is not modified by changes in the calling sub-routine[14].

If no data type is declared, i.e. the `As <data type>` is omitted, the function will return a single value of type Variant.

Function loadData() will not return a single value, but an array of values of data type variant. This is why the array notation was chosen as data type: `As Variant()`. No specification of dimensions is required. The function has one parameter to allow for specifying which worksheet contains the required information. The parameter is called `strWSName`, which is a variable of type String which stores text.

Within the function, the worksheet that contains the data must be accessed. This is done by defining a Worksheet object and a range object:

```
Dim objWS as Excel.Worksheet
Dim rngData as Excel.Range
```

Because the Excel object library is the only Microsoft office library used in this application, the `Excel.` part of the data type can be omitted[15].

To access the only worksheet currently available in the workbook, which is named "Data", its name must be assigned as a parameter to function `loadData` (`= loadData("Data")`).

Via the `strWSName` variable the "Data" worksheet is assigned to an object variable `objWS`. This is done by using the `ThisWorkbook` property of the `Application` object, which returns a Worksheet object from the workbook containing the VBA module with the VBA Code:

```
Set objWS = Application.ThisWorkbook.Worksheets(strWSName)
```

Once worksheet "Data" can be accessed using variable objWS, it can be used it to find all cells which form a continuous area around cell "A1" and can assign the range of cells to variable `rngData.` This can be accomplished by using the `.currentRegion` method.

By assigning the range to an array of type variant (`loadData = rngData`), the contents of the range are

automatically copied to the array and returned by the function as an array of data type variant.
As a last step before the function finishes execution, the objects assigned variables `rngData` and `objWS` are deleted by setting their values to `Nothing`, to destroy the objects and free up memory.

SUB-ROUTINE WRITESASCODE

The second element of the application takes the array delivered by `loadData("Data")`, parses the result array for specific values, and uses them to write the following SAS code:

```
/*****************************************************
Program: mySASFile.sas
*****************************************************/

data _null_ ;
    do i=1 to <Number>;
           put '<Text>' ;
           put 'on: <Date>';
    end ;
run ;
```

The values for <Number>, <Text>, and <Date> stem from worksheet "Data". The complete code can be found in the appendix.

Two new objects from the Microsoft Scripting Runtime library are required for this sub-routine:
FileSystemObject: This object provides access to a computer's file system, to read, write, or modify files[16].
TextStream: This object facilitates sequential access to files[17].

Again, the functionality must be encapsulated into a sub-routine. A sub-routine executes code, when called, but does not return a value. A `sub` statement is needed to define a sub-routine:

```
sub writeSASCode()
```

The statement allows for defining parameters, as the function statement used earlier, but in contrast to it, it doesn't require a return data type, as it does not return a value. The sub-routine definition ends with the following statement:

```
end sub
```

For sub-routine writeSASCode() no parameters are required to control the execution of its code.

Next step is the declaration of variables needed within the sub-routine:

```
Dim varArray() as Variant, intRow as Integer

Dim fso as FileSystemObject
Dim tsSASProgram as TextStreat
```

Array `varArray()`stores the values returned by function `loadData("Data")`. There is no need to define its dimensions, as it copies the dimension from `loadData("Data")`.
`intRow` is declared to store integer values.

Object `fso` is needed to open a file on the file system of the computer that hosts the application:

```
Set fso = New FileSystemObject
```

The `New` statement creates a new instance of the file system object[18].

The application writes the SAS program in the same folder in which the Excel Workbook is stored. It retrieves the name of the folder by using the `Path` property of `ThisWorkbook`:

```
ThisWorkbook.Path
```

A backslash character is concatenated to the end of the path name, which is available via the PathSeparator property of the Application object.

Next, a new text file is opened using the CreateTextFile method of the FileSystemObject, by creating a textStream object:

```
Set tsSASProg = fso.CreateTextFile(Filename:=strPathName & "mySASFile.sas",
Overwrite:=True)
```

Parameter `Filename` requires a fully qualified file name (including the file location). Parameter `Overwrite` has two possible values: `True` (allows for overwriting) and `False`.

The textStream object has two methods that can be utilized to write text to the SAS program: `.writeLine` and `.writeBlankLines`.

- `.writeLine` writes a single line and ends it with a new line character
- `.writeBlankLines` has one parameter to assign the number of blank lines to be created.

The sub-routine starts writing the SAS program by adding a program header and writing a `data _null_` statement:

```
tsSASProg.WriteLine "/*****************************************************"
tsSASProg.WriteLine " Program: mySASFile.sas"
tsSASProg.WriteLine "*****************************************************/"
tsSASProg.WriteBlankLines 1
tsSASProg.WriteLine "data _null_; "
```

To write the code for the data step, the contents of worksheet "Data" is loaded into array varArray():

```
varArray = loadData("Data")
```

The array is parsed in a loop to write the do statement, followed by a loop to write two put statements. The sequence of SAS statements in the data step is controlled by explicitly coding a for-loop for the do statement. Regardless of where the parameter "Number" appears in worksheet "Data", the do statement shall always be printed before the put statements.

```
For intRow = 1 To UBound(varArray, 1)
    If varArray(intRow, 1) = "Number" Then
        tsSASProg.WriteLine "  do i = 1 to " & varArray(intRow, 2) & ";"
    End If
Next intRow
```

The for-loop parses the first column of the array, until it finds an entry named "Number". Once it is found, the loop retrieves the value of the second column to place it within the SAS program. To accomplish this, the.WriteLine method is used together with the string concatenation operator "&" to write our SAS code line.

The second for-loop functions similarly. However, here the order of the parameters in the excel workbook decides which statement is printed first:

```
For intRow = 1 To UBound(varArray, 1)
    If varArray(intRow, 1) = "Text" Then
        tsSASProg.WriteLine "      put '" & Trim(varArray(intRow, 2)) & "';"
    ElseIf varArray(intRow, 1) = "Date" Then
        tsSASProg.WriteLine "      put 'on: " & Format(varArray(intRow, 2),
"YYYY-MM-DD") & "';"
    End If
```

The date value is stored as a number in the worksheet and is displayed in its cell using a date format. For output into the SAS program a format similar to the ISO8601 format is defined: `"YYYY-MM-DD"`. Please note that in contrast to using SAS no date transformation is needed.

Last steps are closing the do loop, adding a run statement, and closing the file:

```
tsSASProg.WriteLine "  end;"
tsSASProg.WriteLine "run; "

tsSASProg.Close
```

Before the sub-routine finishes execution, the object variables and the array must be cleaned up:

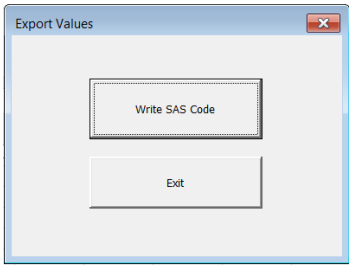
```
Set tsSASProg = Nothing
Set fso = Nothing

Erase varArray
```

The complete code for sub-routine `writeSASCode` can be found in the appendix.

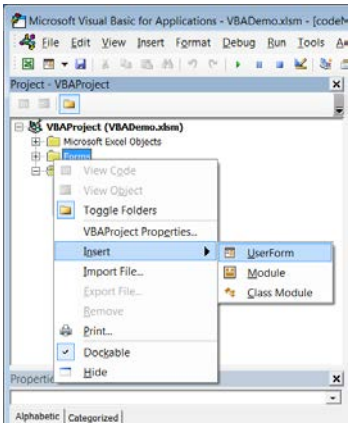
THE USER INTERFACE

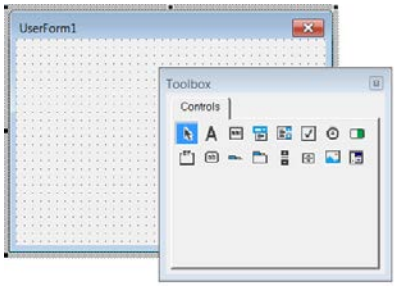
As mentioned in the introduction to VBA, Excel VBA is event driven. This renders it unnecessary for the users to open the VBA Editor to execute sub-routine `writeSASCode()` by hitting the "F5" key every time they want to create new SAS code. Instead, other means can be utilized to write the SAS program.



Excel has a long list of events associated to specific elements of the user interface. The application uses the click event of the command button control, activated by a click on button "Write SAS Code". A second button allows for exiting the application.

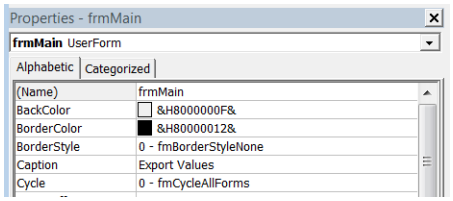
Forms can be created from the VBA Editor in the same way as VBA Modules can be created: Within the Project Browser, by right clicking on the Forms folder, and selecting **Insert** >> **Userform**.





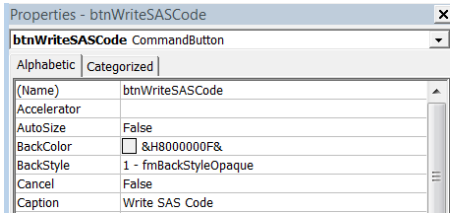
A new User Form is added, which can be resized to fit its purpose and all the controls that should be added. An additional Menu, the "Toolbox" provides a list of different controls that can be added to the forms. Two controls with the name "CommandButton" are required.

Before this is done, though, the name of our user form must be updated: By changing the value of the "Caption" property in the Properties Pane to "Export Values".



In addition, the value of the property "(Name)" is updated to "frmMain". This changes the name appearing in the Project Browser and the name under which it can be controlled from within VBA.

Next, two command buttons are created using the toolbox, by selecting the CommandButton control and drawing two buttons. Once added, "(Name)" and "Caption" properties are updated:



Top Button	(Name)	btnWriteSASCode
	Caption	Write SAS Code
Bottom Button	(Name)	btnExit
	Caption	Exit

Now that a form and two buttons are created, they must be programmed. This is done by writing two small sub-routines within the VBA Container of `frmMain`.

To make the VBA Code visible in the Code Editor window, `frmMain` must be selected in the Project Explorer and the "View Code" button must be clicked in the bar above the window.

```
Option Explicit

Private Sub btnWriteSASCode_Click()
    writeSASCode
End Sub

Private Sub btnExit_Click()
   ThisWorkbook.Save
    Application.Quit
End Sub
```

There are two sub-routines defined within the code module, both are private to the VBA module embedded in the form. This means they can only be accessed from within the embedded module, but not from any other module.

Each of these sub-routines is tied to the event "Click", which means its code is executed as soon as the button is clicked.

The functionality of `btnWriteSASCode` contains a call to sub-routine `writeSASCode`. Please note that the sub-routine name is not followed by a pair of parentheses. In contrast to a function call, parameters for a sub-routine call are just appended to its name following a blank. Multiple parameters are separated by commas. Now this sub-routine can be executed by the

event "Push of a Button" and does not require being executed from the VBA editor anymore.

Sub-routine `btnExit` contains two statements, which are self-explanatory: `ThisWorkbook.Save` saves the workbook, while `Application.Quit` closes the application.

ACTIVATION OF THE USER INTERFACE

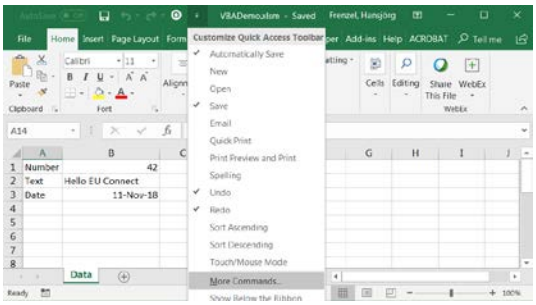
One small step remains to call the application finished: users need a means to open the user form when they are working in worksheet "Data".

First step is to define a new sub-routine, whose only purpose is to make the form `frmMain` visible to users.

```
Sub callUserForm()
    frmMain.Show
End Sub
```

To make the form accessible, a button is added to the "Quick Access Toolbar" at the top of the application window. This can be accomplished by clicking on the little triangle left of the center of the top row of our application.

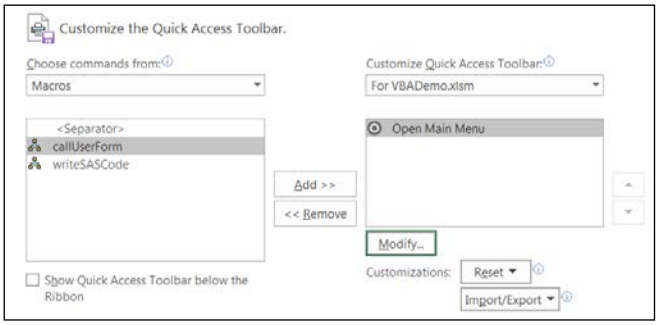
In the drop-down list that opens, we select "More Commands…"



This opens the Quick Access Toolbar options within Excel Options dialog.

Entry "Macros" must be selected in the selection box at the left side, and "For VBADemo.xlsm" on the right side. Then the "Add" button must be clicked on to copy the entry "callUserForm" to the right pane.

The "Modify…" button allows us to change the symbol that will appear in the Quick Access Toolbar, and the text displayed when the cursor hovers over the symbol. We changed it to "Open Main Menu".



CONCLUSION

Automating the creation of SAS programs by using metadata stored in Excel is not restricted to using SAS. SAS programs are text files, and while the SAS macro language is superb in writing SAS code while utilizing data stored in macro variables or SAS datasets, it has some shortcomings when reading such data from Excel. In particular, the necessity to quote strings, when they contain characters that confuse the SAS macro compiler, can make writing SAS Code from metadata a daunting task. But also the characteristic of Excel of varying the data type by cell rather than by column makes it difficult for SAS to read Excel Workbooks.

Visual Basic for Applications has some advantages here: first, it can write text files, and SAS programs are text files. But more importantly, it does not use the SAS Macro compiler, and thus need not be concerned with the ambiguity the SAS Macro compiler faces with some special characters. Finally, it provides a huge set of options to control all aspects of the application and automate tasks, that are simply not available when relying on SAS.

Starting to use Visual Basic for Applications enriches the SAS programmers tool box by introducing them at a minimum to sub-routine programming and an alternative way of using (temporary) arrays in SAS.

REFERENCES

[1] Zhang lists potential stumbling blocks and which precautions to take when using the Excel libname engine in SAS 9.3 and newer for reading data from Excel: Lianbo Zhang, "A simple way to access the data in Excel through SAS v9/ Access libname and Excel engine", PharmaSUG China 2014, PT09. In addition, when developing an application that writes SAS code by using SAS macros, one needs to be careful to handle characters that the macro compiler can confuse with macro instructions. Obviously, there is no problem when using VBA.
[2] Object (computer science). From: https://en.wikipedia.org/wiki/Object_(computer_science)
[3] Range.Value Property (Excel): https://msdn.microsoft.com/en-us/VBA/Excel-VBA/articles/range-value-property-excel
Worksheet.Range Property (Excel): https://docs.microsoft.com/en-us/office/vba/api/Excel.Worksheet.Range
Range Object (Excel): https://docs.microsoft.com/en-us/office/vba/api/Excel.Range(object)
[4] An example for the alternative terminology can be found here: https://excelmacromastery.com/vba-class-modules/. In their developer documentation Microsoft mostly complies to the more common terminology used in this paper.
[5] In addition, `Gobal`, `Private`, and `Const` can be used for declaring variables.
[6] In the editor settings enabling option *Require Variable Declaration* has the consequence that `Option Explicit` is automatically added to the declaration section of every new module. Editor Settings can be accessed by selecting *Options* from the *Tools* menu of the VBA editor.
[7] VBA Development Best Practices: https://www.spreadsheet1.com/vba-development-best-practices.html
[8] Reddick VBA Naming Conventions: http://www.xoc.net/downloads/rvbanc.pdf
[9] VBA Collections: https://msdn.microsoft.com/en-us/vba/language-reference-vba/articles/collection-object
[10] Application Object: https://msdn.microsoft.com/en-us/VBA/Excel-VBA/articles/application-object-excel
[11] Workbook Object: https://msdn.microsoft.com/en-us/vba/excel-vba/articles/workbook-object-excel
[12] Worksheet Object: https://msdn.microsoft.com/en-us/vba/excel-vba/articles/worksheet-object-excel
[13] Range Object: https://msdn.microsoft.com/en-us/vba/excel-vba/articles/range-object-excel
[14] A very concise introduction to this topic can be found here: http://www.cpearson.com/excel/byrefbyval.aspx
[15] This would be different if we wanted to control Word from our Excel Application, as the Word object model also knows a Range object.
[16] FileSystemObject: https://msdn.microsoft.com/en-us/vba/language-reference-vba/articles/filesystemobject-object
[17] TextStream Object: https://msdn.microsoft.com/en-us/vba/language-reference-vba/articles/textstream-object
[18] The `New` statement enables implicit creation of a new object. The New statement can be omitted, if an existing object (e.g. the workbook, or an existing worksheet) is assigned to an object variable. (https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/dim-statement)

RECOMMENDED READING

Pearson Software Consulting, LLC, Comprehensive Excel Information:
http://www.cpearson.com/Excel/MainPage.aspx
Microsoft's own documentation:
https://docs.microsoft.com/en-us/office/vba/api/overview/
Good source for specific VBA questions:
https://www.stackoverflow.com
About error handling in VBA:
https://docs.microsoft.com/en-us/office/vba/access/concepts/maintenance/handle-run-time-errors-in-vba
http://www.cpearson.com/excel/errorhandling.htm
https://msdn.microsoft.com/en-us/library/ms234761(v=vs.90).aspx

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Hansjörg Frenzel
PRA Health Sciences
Gottlieb Daimler Str. 10
68165 Mannheim
Email: frenzelhansjoerg@prahs.com

APPENDIX

SUB-ROUTINE WRITESASCODE

```
Sub writeSASCode()
    Dim varArray() As Variant, intRow As Integer

    Dim fso As FileSystemObject
    Dim tsSASProg As TextStream

    ' find the current path and start writing to the .sas program
    Set fso = New FileSystemObject
    Dim strPathName As String
    strPathName = ThisWorkbook.Path & Application.PathSeparator
    Set tsSASProg = fso.CreateTextFile(Filename:=strPathName & "mySASFile.sas", Overwrite:=True)

    ' load data from worksheet "Data"
    varArray = loadData("Data")

    ' write the .sas program header
    tsSASProg.WriteLine "/*******************************************************"
    tsSASProg.WriteLine " Program: mySASFile.sas"
    tsSASProg.WriteLine "*******************************************************/"
    tsSASProg.WriteBlankLines 1
    tsSASProg.WriteLine "data _null_; "

    ' write the do-loop, by using the value associated to entry "Number"
    For intRow = 1 To UBound(varArray, 1)
        If varArray(intRow, 1) = "Number" Then
            tsSASProg.WriteLine "  do i = 1 to " & varArray(intRow, 2) & ";"
        End If
    Next intRow

    ' write the put - statements using values associated to entries "Text" and "Date"
    For intRow = 1 To UBound(varArray, 1)
        If varArray(intRow, 1) = "Text" Then
            tsSASProg.WriteLine "      put '" & Trim(varArray(intRow, 2)) & "';"
        ElseIf varArray(intRow, 1) = "Date" Then
            tsSASProg.WriteLine "      put 'on: " & Format(varArray(intRow, 2), "YYYY-MM-DD") & "';"
        End If
    Next intRow

    ' closing the do-loop, ending the data-step, and closing the .sas program
    tsSASProg.WriteLine "  end;"
    tsSASProg.WriteLine "run; "

    tsSASProg.Close

    ' unload the objects and delete the array
    Set tsSASProg = Nothing
    Set fso = Nothing

    Erase varArray
End Sub
```