Prioritise your tasks

Improve your way of working and become more efficient

Communicator

The Institute of Scientific and Technical Communicators
Autumn 2016

Create compliant manuals for the US

Implement Swagger with API docs

Write a winning proposal

Misunderstandings around women in techcomm

Procedures and functions

Saving the repetition of VBA code.

By Mike Mee

Introduction

As your VBA library grows with each new chunk of code that you add, you may find that when reading back through the code, some of it repeats itself. This could be because you copied and pasted a working piece of code from one area into another.

This article will give you some ideas on how to break down routines, or create stand-alone routines that you can use multiple times in your own code. The end result is that your code becomes easier to follow.

This article refers to the use of global variables, via the `Public` command, which were covered in the previous article (*Communicator*, Summer 2016).

Procedures

Procedures are what is known within VBA (and general BASIC coding standards) as a subroutine. VBA calls them SUBs to indicate their heritage.

Procedures tend *not* to return a value, unless you are using globally defined variables. Generally, a procedure will do *something* and then come back to the line in your code after the point from where you originally called it.

Functions

Functions, on the other hand, are enhanced versions of procedures which can return a value (or values) based on some calculations that you have asked them to perform.

They are probably used more in Excel-based macros than Word-based ones, such as simple 'How much VAT is embedded in that price?' or 'How much fuel did I use on that journey?'.

Word can still use functions and I have included some examples that are fairly simple, and you do not need a degree in maths (or Excel) to understand them.

Why should you use them?

Here are some reasons why you would use procedures and/or functions in your project:

1. Avoiding the repetition of code. You could have the same code in multiple areas, so moving it into a single procedure saves on memory.
2. Reusing procedures in your other programs. Creating a library of code that you use in all of your VBA macros makes for speedier development.
3. Helping you out when you need to debug a piece of code.

Example procedures

I will cover procedures first as they are a little bit easier than functions.

I have included some examples of fairly simple procedures; the type that just *do something* and then return to your main code.

Track Changes

As mentioned in the second part of my VBA Loops article, I have a simple function that I use to: store the current state of Track Changes, switch it off and then restore it after a piece of code has finished. This is set up in the global area of your code like this:

```
Public blnTrackChanges As Boolean
```

You can now store the current setting for Track Changes, by querying the `ActiveDocument.TrackRevisions` command. If the result is `True`, Track Changes is switched on; if the result is `False`, Track Changes is switched off.

The following code example is taken from my Word Toolbox. It contains both the store and restore functions.

These functions are stored in a separate module, along with all my other procedures. This is how you can create a single module that contains all of your 'standard' procedures and functions, enabling you to reuse the same code in your current and any future macros.

Either procedure can be called whenever it is needed, but I suggest you store the setting at the start of your code, and then restore it afterwards.

```
' Store current Track Changes setting then
ensure it is switched off
Sub Track_Off ()
With ActiveDocument
        blnTrackChanges = .TrackRevisions
        .TrackRevisions = False
End With
End Sub

' Restore Track Changes to its previous
state
Sub Track_On()
With ActiveDocument
        .TrackRevisions = blnTrackChanges
End With
End Sub
```

Calling `Track_Off` before making bulk changes to your document will ensure that none of the alterations is recorded. Once the code is finished, `Track_On` is called and the previous state of Word's Track Changes is restored.

Clearing Find and Replace

If you are going to use macros to replace or enhance the standard Find and Replace function within Word, when your macro has finished, the previous settings you used will still be there.

The following procedure could be called after doing any Find and Replace operations in VBA.

```
Sub ClearFindAndReplaceParameters()

With Selection.Find
  .ClearFormatting
  .Replacement.ClearFormatting
  .Text = ""
  .Replacement.Text = ""
  .Forward = True
  .Wrap = wdFindContinue
  .Format = False
  .MatchCase = False
  .MatchWholeWord = False
  .MatchWildcards = False
  .MatchSoundsLike = False
  .MatchAllWordForms = False
End With

End Sub
```

The code above ensures that every field and/or check box within the Find and Replace dialog is reset back to the defaults.

Checking the protection

This section describes the code I use to check the protection status of the current document.

I have defined `strProtectionType` and `blnProtected` as global variables at the top of my module.

```
Public strProtectionType As String
Public blnProtected As Boolean
```

This allows any piece of code within the Toolbox to ensure it is not going to try and apply changes to a protected document and display an error message with the current protection level, if there is one.

```
Public Sub CheckProtection()
Dim lngProtType As Long

lngProtType = ActiveDocument.ProtectionType

' Check if the current status is not equal
to -1 (which means that the document is
currently protected in one way or another).

If lngProtType <> -1 Then
  Select Case lngProtType

    Case wdAllowOnlyComments
      strProtectionType = "Comments Only
Allowed"

    Case wdAllowOnlyFormFields
      strProtectionType = "Form Fields Only"

    Case wdAllowOnlyReading
      strProtectionType = "Reading Only
Allowed"

    Case wdAllowOnlyRevisions
      strProtectionType = "Revisions
Allowed Only"

End Select
    ' Yes, it's protected
    blnProtected = True
Else
    ' No protection was found
    strProtectionType = "Unprotected"
    blnProtected = False
End If

End Sub
```

All of my routines (which amend a document's content) call the `CheckProtection` routine first. This will result in the type of protection used, or not as the case may be, being stored in `strProtectionType` and the `True/False` status of the protection being stored in `blnProtected`. See Figure 1 for an example.

The various routines in my add-in then check the value of `blnProtected` using code similar to this at the start of each routine:

```
' Examine document's protection status
CheckProtection
' Display appropriate message if it is
If blnProtected = True Then
  MsgBox "This document is protected with:
" & strProtectionType
  Exit Sub
End If
```

If it is `False`, the rest of the routine continues as normal, but if `blnProtected` is `True`, a message box appears telling the end user that the current document is protected and what the level of protection is via the contents of `strProtectionType`. After the user clicks the message box's OK button, the routine is aborted, via the `Exit Sub` command.

Passing a parameter

You can pass through any type of value when calling your procedure. This could be a number that you need to do something with, or, because

Table 1. Types of variables with Leszynski naming conventions

Variable Type	What can be stored in it	Naming convention
boolean	True or False values.	bln
Byte	0 to 255.	byt
Integer	-32,768 to +32,768.	Int
Long	Values can be from approximately -2 billion to +2 billion.	lng
Currency	Decimal numbers with as many as 19 digits.	cur
Single	Single-precision, floating-point numbers.	sng
Double	Double-precision, floating-point numbers.	dbl
Date	A date or time value.	dtm
String	Text	str
Object	Any object, such as a Word document or a window.	obj
Variant	A generic number or string.	var

***Note**: Single or Double variables are more useful in Excel than Word.*

```
Sub ReadyForReview()

' Examine document's protection status
CheckProtection
' Display appropriate message if it is
If blnProtected = True Then
  MsgBox "This document is protected with: " & strProtectionType
  Exit Sub
End If

'Store current Track Changes setting and switch off
Track_Off

' Clear the document's find and replace settings
ClearFindAndReplaceParameters
' Find "techcomm" and replace with "technical communication"
    With Selection.Find
        .text = "techcomm"
        .Replacement.text = "technical communication"
    End With
    Selection.Find.Execute Replace:=wdReplaceAll

'Select the whole document and apply UK English
Selection.WholeStory
Selection.LanguageID = wdEnglishUK
Selection.NoProofing = False
Application.CheckLanguage = True

' Restore Track Changes to its previous state
Track_On

End Sub
```

Figure 1. A macro which gets a document ready for review by calling the `CheckProtection`, `TrackOff`, `Track_On` and `ClearFindAndReplaceParameters` macros that have already been defined.

we are covering Word VBA macros, you are more likely to send a string. This could be the name of a font, the text from a particular header, or whatever you need to work with.

Example procedures with parameter(s)

You can enhance a procedure by adding the ability to send a value, or values, into it. The procedure will then work with the values submitted.

Passing parameter(s) will be covered in a future article.

Example functions

Now that we have covered subroutines, we will move onto functions. I have tried to find some Word-related functions, but the majority of the examples out there assume you will be using them in your Excel-based VBA projects.

Get the square of an integer

This is a very simple function and it will calculate the square of an integer that you supply.

```
Function GetSquare(intNumber As Integer)
As Long
GetSquare = intNumber * intNumber
End Function
```

You would call the above procedure from within your code. For example, calling `GetSquare(9)`, would return the value of `81` inside the variable `GetSquare`. You can then use it as part of a calculation elsewhere in your code.

The following code will expand on the use of the above function; it asks the user to enter a number (via an `InputBox`) and then displays the square of the number entered via a `MessageBox` command.

```
Sub ShowSquare()
Dim intSource As Integer
intSource = CInt(InputBox("Enter a number to
get the square of"))
MsgBox "The square of " & intSource & " is
" & GetSquare(intSource)
End Sub
```

The `CInt` command ensures that if the user enters a non-integer value, such as `4.2`, it will be converted to the integer value of `4`.

Calculating fuel usage (in gallons or litres)

This final example of a function that will calculate the fuel consumption (as miles per gallon) of a road trip.

This could be used on a Word form via a content control calculation – both of which I know I haven't covered yet.

The function is called by sending three parameters: the starting odometer reading, the finishing odometer reading and the amount of fuel used in litres:

```
' Get Miles Per Gallon
Function GetMPG(intStartMiles As Integer,
intFinishMiles As Integer, sngLitres As Single)

GetMPG = (intFinishMiles - intStartMiles) /
sngLitres * 4.546

End Function
```

The result is displayed by putting a reference to `GetMPG (intStart, intFinish, sngLitres)` in a message box or inserted into a cell on a table within a form. Within the function, the amount of litres supplied are multiplied by `4.546` to generate the total miles per gallon result.

Alternatively, you can launch it from the Immediate Window by entering the following command:

```
Msgbox GetMPG(100, 200, 10)
```

If you prefer to stick with litres, you can remove the `* 4.546` part of the calculation. Below is a copy of the function which I have renamed accordingly as `GetMPL`:

```
' Get Miles Per Litre
Function GetMPL(intStartMiles As Integer,
intFinishMiles As Integer, sngLitres As Single)

GetMPL = (intFinishMiles - intStartMiles) /
sngLitres

End Function
```

You call the function, `GetMPL`, using the same parameters as you would for `GetMPG`, except you will get the result back in litres.

```
Msgbox GetMPL(100, 200, 10)
```

That is the basics of procedures and functions covered, but they can be far more powerful than the examples I have given.

The end

Despite the sub heading, this is just an introduction and I hope to be able to return to this subject matter in a follow-up article at some point in the future.

You can see that there are two examples of code at the end that expand upon the areas covered in this article.

1. Figure 1 is a procedure called `ReadyForReview` which shows you how to: use the protection check, examine the status of track changes, perform a find and replace, change the language of the document to UK English and then switch Track Changes back to its previous state.
2. Figure 2 is a procedure called `ShowMPGandMPL`. This expands on the Miles per Gallon/Miles per Litre functions and includes the code to get the required input from the user, before displaying the two calculations in a single message box.

Next article

As it has been a while since my articles covered anything actually happening with the contents of your document, the next article will take you into the realm of sending data from your macro into your document(s). This is where the fun begins!

Further reading

Mee M (2015) 'Variables and screen output/input' *Communicator*, Autumn 2015: 44-47

Mee M (2016) 'Creating VBA loops: part 1' *Communicator*, Spring 2016: 34-36

Mee M (2016) 'Creating VBA loops: part 2' *Communicator*, Summer 2016: 52-55

Microsoft 'Declaring Variables' https://msdn.microsoft.com/en-us/library/office/gg264241.aspx?f=255&MSPPError=-2147217396 (accessed April 2016)

Mike Mee MISTC is a technical author working at CDL in Stockport.
E: mike.mee@cdl.co.uk
T: @Mug_UK
W: http://mikestoolbox.weebly.com – my toolkit for Word 2007-2016 (the full source code is also available on the website).

```
Sub ShowMPGandMPL()

Dim intStart As Integer
Dim intFinish As Integer
Dim sngLitresUsed As Single

intStart = CInt(InputBox("Enter the starting odometer reading"))
intFinish = CInt(InputBox("Enter the finishing odometer reading"))
sngLitresUsed = InputBox("Enter the number of litres used")

MsgBox "Starting odometer reading: " & intStart & vbCr & _
"Finishing odometter reading: " & intFinish & vbCr & _
"Total miles driven: " & intFinish - intStart & vbCr & _
GetMPG(intStart, intFinish, sngLitresUsed) & " miles per gallon" & vbCr & _
GetMPL(intStart, intFinish, sngLitresUsed) & " miles per litre", vbOKOnly, "Fuel usage"

End Sub

' Get Miles Per Gallon
Function GetMPG(intStartMiles As Integer, intFinishMiles As Integer, sngLitres As
Single)
        GetMPG = Format((intFinishMiles - intStartMiles) / sngLitres * 4.546, "0.00")
End Function

' Get Miles Per Litre
Function GetMPL(intStartMiles As Integer, intFinishMiles As Integer, sngLitres As
Single)
        GetMPL = Format((intFinishMiles - intStartMiles) / sngLitres, "0.00")
End Function
```

Figure 2. An example showing a procedure macro which passes parameters to functions.