8. Programming in Excel with VBA.

Introduction

Hiding “behind the scenes” in Microsoft Office products is a powerful programming environment that most users are unaware of. This programming language is Visual Basic for Applications (VBA). Thus far in the course we have made use of two VBA capabilities: (1) user defined functions and (2) user defined macros (subprograms). VBA is closely related to, but not identical to, Visual Basic. Persons who are proficient with VBA will have little difficulty becoming proficient with Visual Basic.

Visual Basic for Applications (VBA) Overview

Although VBA is a complete programming environment there is one significant limitation and several operational issues.

The limitation is that one cannot develop standalone modules (executable programs) that can be distributed. Code developed using VBA in Excel (for example) is a part of the Excel worksheet and must be run from Excel. This can cause problems if one distributes a “program” developed under one version of Excel and one attempts to run it under a different version. Normally the differences are small between VBA versions but even a small difference can cause major problems. This also means that “programs” developed with “today’s Excel” may not run properly when one upgrades one’s software to “tomorrow’s Excel”. Generally code which uses straightforward features (like using the “SIN” function or the “TODAY” function) will cause little problems. Most problems arise from use of functions and features much closer to the “operating system”.

Since we have already used the VBA Editor, we will not detail how to get into it. Once in the editor one sees three or more “panels” where different activities take place:

- The most obvious panel is the Development Panel (or development area) where code is written and forms are developed. Forms are similar to “menus” and similar “popups” where data is entered or results are displayed.
- The upper panel on the left is called the Project Panel. In the area one selects what will be displayed in the Development Panel. By default there are four items in a new worksheet.
Material developed for the “sheet1” development area are available ONLY to sheet1. Items to be common (shared) between ALL worksheets should be associated with the “ThisWorkbook” development area.

- The bottom left panel is called the “Properties Panel” which provides information about the properties of the currently selected item (more about this later).
- Another frequently used panel is the “Immediate Window (Panel)” which is not displayed by default. Use the “View” menu to show/hide this window or press Cntl-G. One can immediately evaluate or cause events to happen by typing in this window.

![Immediate Window Example]

**Projects, Forms and Modules**

In VBA, a **project** is a collection of components that make up a complete program. Projects typically contain:

- one or more **forms** to collect (input) and present (output) information.
• one or more **modules** that contain variable definitions (pi=3.141592) and program code (functions, subprograms, etc).
• a single **workbook** shown as “This Workbook” in the project panel.
• one of more individual **sheets** (again used to collect and present information).

**Visibility Issues**

Proper use of these functional components is important in that the location where data and code imposes limits (controls) over where the data and code can be used.

Program elements stored in a **module** are accessible from any project source. As we have seen, having several Excel xls files open at the same time makes all the functions written in any **module** available to any of the **workbooks** or **sheets**. Generally program **functions** and **subprograms** are written in **modules** rather than in specific sheets.
Program elements stored in the **workbook** are available to all sheets but not to other elements of the **project**.

Program elements stored in a **sheet** are only available within that **sheet**. This means that one can have different functions with the same name used in different sheets. Sheet1!Fun1, Sheet2!Fun1, etc.

**Example Subprogram**

Let’s begin by writing a subprogram procedure called “prog1” which demonstrates a few structural elements of the VBA language:

Starting with a blank worksheet, Insert a new module (Module1) and add the following subprogram to the module.

```vba
Public Sub prog1()
     firstletter = "A"
     For i = 1 To 10
         For j = 1 To 10
             red = Int(256 * Rnd())
             blue = Int(256 * Rnd())
             green = Int(256 * Rnd())
             Cells(i, j).Value = Chr(i + Asc(firstletter) - 1)
             Cells(i, j).Interior.Color = RGB(red, green, blue)
         Next j
     Next i
End Sub
```

Press F5 or use the Run / Run Sub/UserForm to execute the subprogram prog1.

Click on the “Xls” icon to see the results. Note that your code will be executed in Sheet1 since it is the default sheet.
Now go to Sheet2 and execute the subprogram by selecting Tools / Macro / Macros / Run or simply press Alt F8. Do this repeatedly to see the random nature of this subprogram.

To simplify the process of executing our prog1() subprogram we will add a button and “attach it” to the prog1 code.

Begin by returning to Sheet1 (this is where we will put our Button).

Click the “Add Button” button from the Forms toolbar. (If this menu is not available, you can add it from View Toolbars.)

Drag over the region where you want the button to appear or just click to get the “default size”.
Associate clicking the button with our prog1 subprogram. (Select and click OK).

Resize the button if you desire.
Right-Click and “Edit Text” to change the caption for the button.
You can also change other “appearance issues” for the button by using the Format Control menu.

**Example Function**

The author presents (on pp. 303-306) an example function. As we have already created many functions in class we will skip this material.

**Flowcharts**

The author discusses “flowcharts” on pages 306-309. Although flowcharts are widely accepted, they are not a very good “development tool”. Many other diagramming techniques are available to assist with writing structured programs including the Nassi-Shneiderman (NS) method that is discussed next. We will exclusively employ NS diagrams in this course.

**Several references will be consulted for NS diagrams:**

http://www.smartdraw.com/resources/centers/software/nassi.htm
http://www.cbu.edu/~lschmitt/I351/Nassi%20Schneiderman.htm
One problem with flowcharts is that they do not force a person to do well structured programming. Another problem with flowcharts is the lack of space in a block for writing information about its function. Nassi-Schneiderman charts overcome these problems. The following figure shows the three basic constructs using Nassi-Schneiderman Charts.

![Flowchart Diagram]

<table>
<thead>
<tr>
<th>SEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SELECTION or IF / THEN / ELSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOOPS such as WHILE and REPEAT-UNTIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

The following figure shows a Nassi-Schneiderman Chart For Taking a Test

<table>
<thead>
<tr>
<th>Clear off desk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive test questions from instructor</td>
</tr>
<tr>
<td>Get your name from memory</td>
</tr>
<tr>
<td>Print name on test</td>
</tr>
<tr>
<td>Do while you haven't finished test</td>
</tr>
<tr>
<td>Read next test question</td>
</tr>
<tr>
<td>Repeat until you understand question</td>
</tr>
<tr>
<td>Convert input into thoughts</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Do you understand question?</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>Reread question</td>
</tr>
<tr>
<td>Search memory for answer</td>
</tr>
<tr>
<td>Mark answer on test paper</td>
</tr>
<tr>
<td>Turn in test paper</td>
</tr>
<tr>
<td>Leave classroom</td>
</tr>
</tbody>
</table>
NS diagrams are independent of computer language. In this respect, they are called pseudo-programming languages (or psuedocode).

In order to see the value of NS diagrams, let’s take a relative simply problem and solve it using NS diagrams.

**Program: Generating Unique Random Numbers**

Program Statement: Write a procedure that develops a set of “m” integer numbers each of which is in the range n1...n2 with no duplicates. Include logic to detect invalid situations.

For example:  m=5  n1=10  n2=20
Solution:  13, 11, 19, 20, 18

Another:  m=10  n1=5  n2=10
Solution:  NONE (invalid)

Another:  m=55  n1=0  n2=100
Solution:  TOO BIG FOR DIMENSIONING (invalid)

(Program follows on next page)

*Note: A button was added to call the subprogram uniqueRandom().*
Public Sub uniqueRandom()
Dim m As Integer
Dim n1 As Integer
Dim n2 As Integer
Dim maxN As Integer
Dim nRand As Integer
Dim found As Integer
Dim i As Integer
Dim IsUnique As Boolean
Dim IsInvalid As Boolean
Dim soln(50) As Integer

' Acquire Problem Data
m = 55
n1 = 10
n2 = 20

' Determine If Valid Case
maxN = n2 - n1 + 1
IsValid = (m <= maxN) And (m <= 50)

' Process Valid Case or Report Error
If IsValid Then

' Determine Solution
found = 1
soln(1) = GenerateRandomNumber(n1, n2)
Do While found < m
nRand = GenerateRandomNumber(n1, n2)
IsUnique = True
For i = 1 To found
If soln(i) = nRand Then IsUnique = False
Next i
If IsUnique Then
found = found + 1
soln(found) = nRand
End If
Loop

' Report Solution
For i = 1 To m
Cells(2 + i, 2).Value = soln(i)
Next i
Else
' Report As Invalid
Cells(3, 2).Value = "No solution... input data in error!"
End If

End Sub
Public Function GenerateRandomNumber(a As Integer, b As Integer) As Integer

Dim range As Integer
Dim floatran As Single

range = b - a
floatran = range * Rnd()
GenerateRandomNumber = Int(a + floatran)

End Function
Here is the output from a valid case \((m=5, n1=10, n2=20)\):

![Microsoft Excel - Book8b.xls](image1)

Here is the output from an invalid case \((m=55, n1=10, n2=20)\):

![Microsoft Excel - Book8b.xls](image2)

**Getting Back The Power of Excel**

One of the initial frustrations of using VBA is the “lack” of corresponding functions and procedures that one may be used to from using Excel itself.

Remembering that each product (Excel and VBA) are separate independent products one should not be too surprised that each has its own unique capabilities. For example, Excel is a “number cruncher” and has hundreds of mathematical and statistical functions. VBA, on the other hand, is a general purpose computing language whose audiences is much broader than “number crunching engineers”. Therefore, in order to access the Excel functions we have gotten used to, we need to appropriately “reference them”.

Simple Excel functions are fairly easy to “get to”. For example, consider the “Sum” function that might be used to add the values in the range A1:C5.

You can call most standard Excel worksheet functions in VBA procedures using the syntax:

\[ \text{Result} = \text{Application.WorksheetFunction.Sum(Range("A1:C5"))} \]

where Application.WorksheetFunction allows VBA to locate the function “Sum”. We also need to “adjust” the argument information.

On the other hand, other functions are “less standard” and may belong to various installed programs (add-ins).

For example, suppose we wish to round a number in VBA to the closest “2” using the MROUND function. Since this function comes from the Analysis Toolpak we need to take two steps to be able to use that function.

1. From the Excel window, activate the Analysis ToolPak (VBA) from the Add-Ins menu (note this is a separate add-in from the ATP for Excel).
2. From the VBA window, activate the Analysis ToolPak VBA (ATPVBA) in the Tools/References menu. That item is atpvbaen.xls (EN=English)

![Image of References - VBAProject dialog box]

You will now be able to access the `mround` and other math functions in the following fashion:

```vba
Sub DemoRound
    Dim result As Single
    Dim x as Single
    x = 3.2
    result = mround(x)
    Cells(2,2) = result
End Sub
```

**Viewing the Members of the ATPVBA (or other Libraries)**

In order to view the procedures available one can view the members (Objects) that are currently loaded in VBA. Simply press F2 (or use the View/Object Browser). By default, one sees “<All Libraries>” but we can select “atpvbaen.xls” to see what we are interested in:
We can see our old favorites such as BesselJ as well as MRound and RandBetween.

Another example, suppose we wish to employ the following “Excel” formula when we are using VBA:

\[ = \text{norminv}(0.3, 70, 10) \]

Suppose we have a normally distributed variable (Gaussian distribution) where the average exam grade is 70 and the standard deviation of grades is 10 (that is, 95% of students have a grade between 50 and 90). 30% of students have “what grade or less”? The answer is: 64.756

```vba
Sub test()
    Dim result As Variant
    result = Application.WorksheetFunction.NormInv(0.3, 70, 10)
    Cells(3, 1) = "Answer:"
    Cells(3, 2) = result
End Sub
```
Note that we can also use the Object Browser to search for items we may not be sure of the syntax or location of.

For example, we will search in “All Libraries” for the string “norm”.

Use the “binoculars” to initiate the search.

Notice that “norm” matches a lot of different “members” (LogNormDist, NormDist, NormInv, etc). These familiar (sort of) functions are members of the “WorksheetFunction” group. Clicking on the “Class” WorksheetFunctions will show ALL members in the “bottom” window as shown below:
Here are the majority of the “expected” functions we apply that are not in the Analysis ToolPak VBA.

(continued in Notes 8b)