PhUSE 2011

Paper TS05

A Comparison of SAS versus Microsoft Excel and Access's Inbuilt VBA Functionality

Jozef Tarrant, Amadeus Software Ltd., Oxford, UK

ABSTRACT

There are a great variety of business situations where it is necessary to automatically export data from a large number of similar Microsoft® Excel spreadsheets (perhaps reports, forms etc.) into a central database, without access to a SAS® installation.

This paper will show how to use Microsoft® Excel and Microsoft® Access's inbuilt VBA functionality to program this type of export. The export process will run with user-friendly dialogue boxes that gather parameters from the user such as where the source files are located, and the destination database.

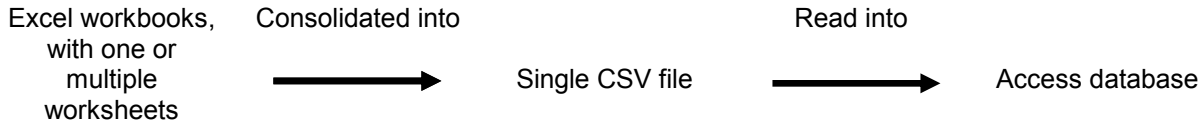
Finally, a brief comparison of a SAS approach to consolidating data will be presented, allowing the strengths of the alternative approaches to be evaluated.

No prior knowledge of VBA is necessary for the programmer or end user. All the code required is supplied within this paper.

INTRODUCTION

Microsoft Excel and Access contain a lot of functionality that few users know how to implement, including the ability to program user-friendly dialogue boxes and prompts to perform consolidation of data, including customisable error handling. This paper will detail the steps needed to use VBA to program a two-part export of data from a large number of Excel spreadsheets to an Access database. The first part will use VBA modules programmed in Excel to sequentially read an unlimited number of external spreadsheets and consolidate the results into a single CSV file. The second part will use VBA programmed in Access to read the CSV file and write to an Access database. This will be done through user-friendly dialogue boxes, which are fully customizable.

Finally, the Analyze Table function will be demonstrated as a way of creating a true relational database from the data.



AN INTRODUCTION TO CODING IN VBA

VBA (Visual Basic for Applications) is an event-driven programming language built into most Microsoft Office applications, often informally referred to as 'Macro'. It allows us to script and automate almost any procedure that can be done manually. It lends itself to any procedure that the user finds themselves having to do repetitively, or a task that an administrator would prefer to be automatic and not subject to user input and associated error.

VBA code is entered inside code modules, which are in turn embedded into the Excel / Access file in question. VBA can be hand-entered and edited through Microsoft Visual Basic IDE, which is a VBA development and editing studio built into Microsoft Office. It also contains some useful debugging tools.

STRUCTURE OF A CODE MODULE AND SUBROUTINES

It is usual practice to break down a program into a sensible number of procedures, then put each procedure into its own subroutine. It is also good practice to put each subroutine into its own coding module.

Once entered, a subroutine compiles automatically and can be instantly used. It can be executed (or 'called') by the user, or it can be called by another VBA program.

A subroutine begins with *sub subname(),* where subname is any name. It ends with *end sub*.

```
sub ExampleSubroutine()
( ... Further Programming Statements ... )
end sub
```

DIM STATEMENTS AND VARIABLE TYPES

Many programming languages, such as SAS, allow the user to introduce variables as and when the program needs them. VBA is different in that it requires all variables to be defined, in name and type, at the beginning of the subroutine. Below are the types that we will be needing in this paper, although more types do exist in the wider world of VBA.

```
Dim anyname1 as Integer
Dim anyname2 as String
Dim anyname3 as Worksheet
Dim anyname4 as Database
Dim anyname5 as TableDef
Dim anyname6 as Boolean
Dim anyname7 as Object
Dim anyname8 as DAO.Database
Dim anyname9 as DAO.Recordset
Dim anyname10 as DAO.Field
Dim anyname11
```

This will create the following variables, or in some cases objects, as the following types:

Name	***Type***
anyname1	Numeric variable
anyname2	Character variable
anyname3	Excel worksheet
anyname4	ODBC database
anyname5	Table view or editing interface
anyname6	Boolean variable
anyname7	Any other object
anyname8	DAO (Data Access Object) enabled database
anyname9	DAO (Data Access Object) enabled recordset
anyname10	DAO (Data Access Object) enabled field
anyname11	Call to application, no type needs to be specified

PART 1: CONSOLIDATING MULTIPLE EXCEL WORKBOOKS

In part 1, we will be consolidating data from multiple Excel workbooks into a single CSV file. We are using the fictional example of a warehouse survey project.

A company that stocks car accessories has recently acquired several warehouses, together with their existing stock, in two new countries. A survey must be performed of the stock in each warehouse.

Each country will be entered into a separate workbook. Each workbook will have one worksheet per warehouse in that country. Each item of stock will be entered as one row, with information on the product name, supplier, and condition. We need to consolidate these data to an Access database.

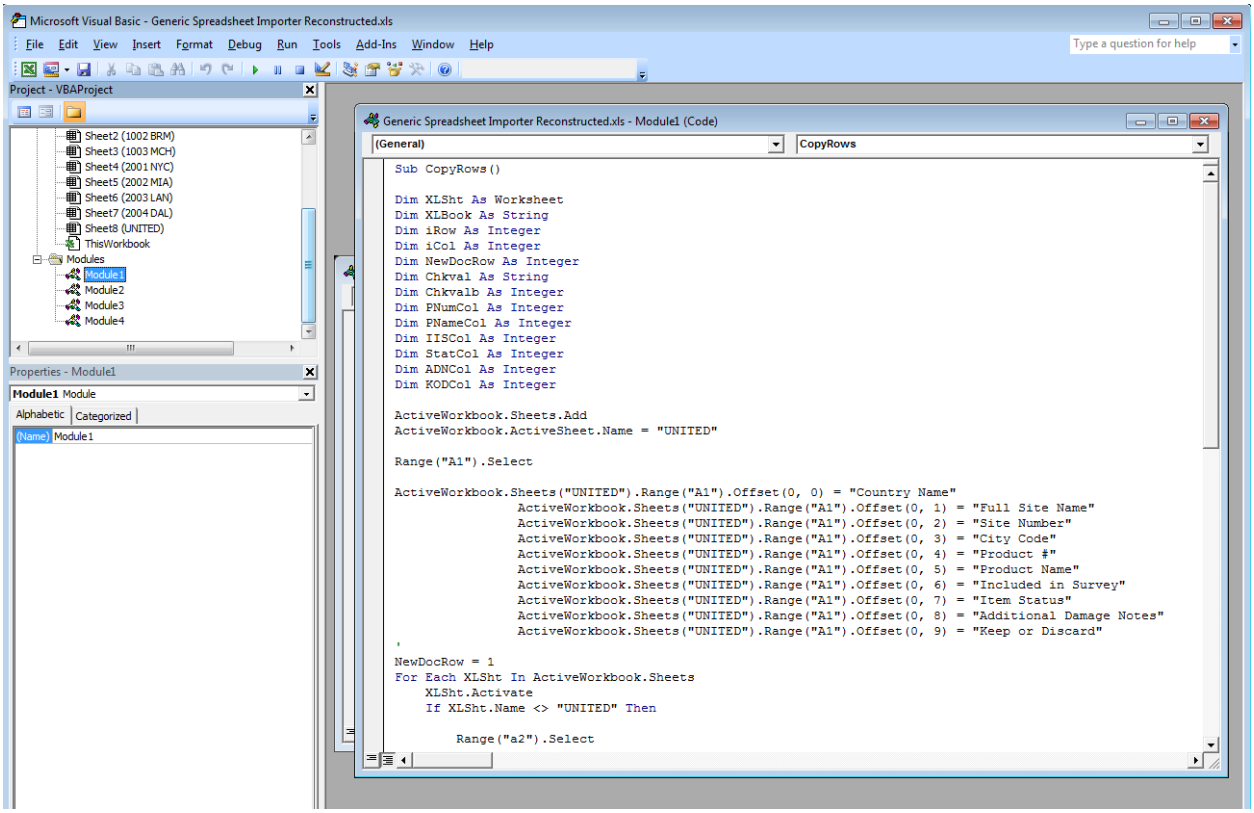
To start the process, open a new Excel workbook. This will be the workbook that all our data will be imported into, so we will give it an appropriate name such as All Data.

SECURITY SETTINGS

By default, Excel and Access are highly restrictive of custom VBA code, as it could accidentally (or deliberately) be used to modify or delete many files on the user's PC. We need to add an exception for each trusted source (i.e. per workbook or per database.)
Clicking on the Excel Button → Excel Options → Popular → Show Developer Bar In The Ribbon will bring up a new tab in the ribbon named Developer. From here, click on Macro Security. Enable the Macro and ActiveX controls to their most open settings.

Return to the spreadsheet and click the Visual Basic button in the Developer tab. This will open up Microsoft Visual Basic IDE, our editing and development studio.

To manage the coding code modules, we use the VBA Project tree towards the top left. Under the spreadsheet name, and within the Modules folder, there should be one empty coding module already present, named Module1. Right clicking this module and selecting Insert → New Module will insert a new coding module. We will need a total of six. To open a module for editing, double click it.



CODE MODULE 1 – "COPYROWS"

The first coding module needs to run through each worksheet in the workbook and copy all rows we are interested in to one new worksheet, which we will call "UNITED".

The code needed to do this is copied below. It has been interspersed with notes (starting "Note:") explaining and expanding on how to further customize its functionality; these notes should be removed if the user wishes to copy and paste this code directly into a module.

```
Sub CopyRows()
```

Note: The variables we need are defined in name and type at the beginning of the code module.

```
Dim XLSht As Worksheet
Dim XLBook As String
Dim iRow As Integer
Dim iCol As Integer
Dim NewDocRow As Integer
Dim Chkval As String
Dim Chkvalb As Integer
Dim PNumCol As Integer
Dim PNameCol As Integer
Dim IISCol As Integer
Dim StatCol As Integer
Dim ADNCol As Integer
Dim KODCol As Integer
```

Note: The name of the new consolidated worksheet, “UNITED”, is specified here.

```
ActiveWorkbook.Sheets.Add
ActiveWorkbook.ActiveSheet.Name = "UNITED"
```

Note: Column titles of the “UNITED” worksheet are added next. The basic logic is to select cell A1 and then use Offset to move X rows and Y columns away before adding the text. X and Y are iterated at various points throughout the loop. Note that VBA moves the offset (1, 3) by 1 row and 3 columns, not 3 columns and 1 row as we might expect a Cartesian coordinate to be written.

```
Range("A1").Select

ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(0, 0) = "Country Name"
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(0, 1) = "Full
Site Name"
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(0, 2) = "Site
Number"
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(0, 3) = "City
Code"
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(0, 4) = "Product
#"
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(0, 5) = "Product
Name"
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(0, 6) = "Included
in Survey"
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(0, 7) = "Item
Status"
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(0, 8) =
"Additional Damage Notes"
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(0, 9) = "Keep or
Discard"
'
```

Note: A loop begins here and is executed for every worksheet in the workbook, with the exception of the newly created “UNITED” worksheet.

```
NewDocRow = 1
For Each XLSht In ActiveWorkbook.Sheets
    XLSht.Activate
    If XLSht.Name <> "UNITED" Then
```

Note: Column positions are dynamically calculated here, based on a name match. It is a defensive programming technique that ensures the program functions correctly even when users enter unexpected extra columns in the source worksheets. If we are absolutely certain this has not happened, then the names could be hardcoded.

```
        Range("a2").Select
```

```
        iCol = 1
        Do
            Chkval = ActiveCell.Offset(0, iCol).Text
             If Chkval = ("Product #") Then PNumCol = iCol
             If Chkval = ("Product Name") Then PNameCol = iCol
             If Chkval = ("Included In Survey") Then IISCol = iCol
             If Chkval = ("Item Status") Then StatCol = iCol
             If Chkval = ("Additional Damage Notes") Then ADNCol = iCol
             If Chkval = ("Keep or Discard") Then KODCol = iCol
             If iCol > 40 Then Exit Do
         iCol = iCol + 1
        Loop

        Range("a3").Select
        iRow = 0
        Do
```

Note: A loop within the first loop begins here and copies data into the UNITED worksheet. We can use conditions here to specify which rows to include and exclude. The variables Chkval and Chkvalb allow us to select only the rows that have Included In Survey = Yes and a Product Number between 101-106 or 201-220.

```
            Chkval = ActiveCell.Offset(iRow, 4).Text
            Chkvalb = ActiveCell.Offset(iRow, 0)
            If (Chkval = "Yes") And ((Chkvalb >= 101 And Chkvalb <= 106) Or (Chkvalb
>= 201 And Chkvalb <= 220)) Then
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(NewDocRow, 0) =
Range("B1")
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(NewDocRow, 1) =
XLSht.Name
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(NewDocRow, 2) =
Left(XLSht.Name, 4)
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(NewDocRow, 3) =
Right(XLSht.Name, 3)
```

Note: As we can see above, the value for Country Name is taken from the value of cell B1. The value for Full Site Name is taken from whatever the name of the worksheet is. The value for Site Number uses the ‘Left’ function to extract the first four characters of the worksheet name. Similarly, the value for City Code uses the ‘Right’ function to extract the last three characters of the worksheet name.

```
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(NewDocRow, 4) =
ActiveCell.Offset(iRow, PNumCol)
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(NewDocRow, 5) =
ActiveCell.Offset(iRow, PNameCol)
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(NewDocRow, 6) =
ActiveCell.Offset(iRow, IISCol)
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(NewDocRow, 7) =
ActiveCell.Offset(iRow, StatCol)
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(NewDocRow, 8) =
ActiveCell.Offset(iRow, ADNCol)
                ActiveWorkbook.Sheets("UNITED").Range("A1").Offset(NewDocRow, 9) =
ActiveCell.Offset(iRow, KODCol)
                NewDocRow = NewDocRow + 1
            End If
            iRow = iRow + 1
            If ActiveCell.Offset(iRow, 0).Text = "" Then Exit Do
        Loop
    End If
Next XLSht

End Sub
```

CODE MODULE 2 – "COPYROWS"

The first code module will append together the data from all worksheets within the current workbook. However, we still have to get the worksheets, which could reside in multiple exterior files, into the current workbook. Code Module 2 will copy all worksheets from any number of exterior workbooks, which the user selects through a dialogue box featuring a file browser, to the current workbook. We need to enter the following into a new code module:

```
Sub CombineWorkbooks()
    Dim FilesToOpen
    Dim x As Integer
    Dim GrabFileName As String
    Dim XLSht As Worksheet

    On Error GoTo ErrHandler
    Application.ScreenUpdating = False
```

Note: This code module uses a call to a built-in VBA function called GetOpenFilename, which passes the file names as text strings for subsequent use in the program The "FileFilter" option allows us to screen what type of files the user can add, eliminating potential error. "Multiselect" is enabled to allow multiple selections. "Title" names the dialogue box with a useful prompt name.

```
    FilesToOpen = Application.GetOpenFilename _
      (FileFilter:="Microsoft Excel Files (*.xls), *.xls", _
      MultiSelect:=True, Title:="Protocols to Combine")

    If TypeName(FilesToOpen) = "Boolean" Then
        MsgBox "No Files were selected"
        GoTo ExitHandler
    End If
```

Note: A loop starts at the 'While' statement for every workbook. 'While' loops begin with a While statement. If the condition being evaluated is true, the code executes up to the Wend statement, before re-evaluating the condition and re-entering the loop if it is still true. A loop within the first loop starts at the 'For Each XLSht' statement and copies all worksheets in the selected external file to our current workbook.

```
    x = 1
    While x <= UBound(FilesToOpen)
        Workbooks.Open Filename:=FilesToOpen(x)

        GrabFileName = ThisWorkbook.Name

        For Each XLSht In ActiveWorkbook.Sheets
            XLSht.Activate
        Next XLSht

        Sheets().Move After:=ThisWorkbook.Sheets _
          (ThisWorkbook.Sheets.Count)
        x = x + 1
    Wend

ExitHandler:
    Application.ScreenUpdating = True
    Exit Sub

ErrHandler:
    MsgBox Err.Description
    Resume ExitHandler
End Sub
```

Another useful procedure is to replace all blank values in our composite spreadsheet with the string "Nothing Selected", in order to make it clear to those performing an analysis that the data is not missing. Create a new module and insert the following code:

```
Sub ReplaceBlanks()

Sheets("UNITED").Select
Cells.Replace What:="", Replacement:="Nothing Selected", LookAt:=xlPart, _
SearchOrder:=xlByRows, MatchCase:=False

End Sub
```

Finally, we will need to execute (or 'call') the macros we have just created in the correct order. The best way of doing this is to create another macro, named RunEverything, in a new code module and have it call the others. The code for doing this can be found below:

```
Sub RunEverything()

Call CombineWorkbooks
Call CopyRows
Call ReplaceBlanks

End Sub
```

To run the finished program, we can go to the Excel workbook where the code modules reside and select Developer –-> Macros. Select the macro RunEverything.

The program should run correctly, and a worksheet of consolidated data named "UNITED" should be produced. We will save this manually as a .CSV file, for the next stage of export to a Microsoft Access database. Access can read both XLS and CSV files, but we will use CSV as many other programs, including SAS, can easily read this file format as well.

PART 2: CREATING AND UPDATING AN ACCESS DATABASE

CREATING A NEW DATABASE

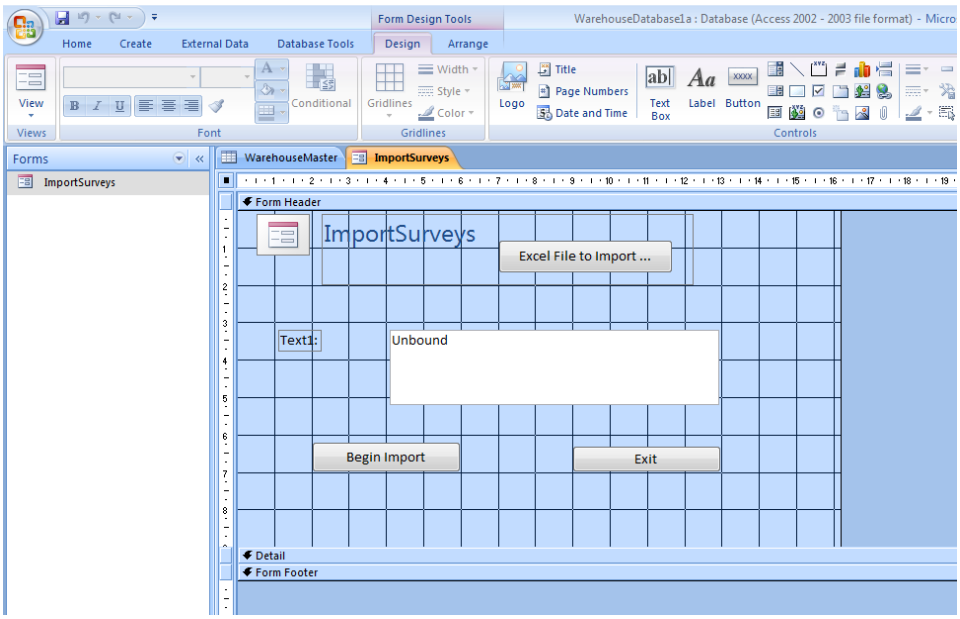
To create a new database, we start Microsoft Access and select New → Blank Database. To ensure maximum compatibility, we create it as a 2002-2003 compatible format (.mdb) rather than the default .accmdb format. We will name our new database WarehouseDatabase1.

CREATING TABLES

We will need to create a new table (Create → Table) to import the data from the CSV file into. Switch to Design View (Home → View → Design View) and create one new field for each of our columns in the CSV file, with the type 'Text'. Note that the ID field is created automatically; this uniquely identifies each record with an automatically generated sequential number. We will call this table WarehouseMaster. Its name is important as we will need to refer to it programmatically later.

CREATING FORMS

Access allows records to be updated in two basic ways: directly adding rows to tables, or through a form. Creating a form allows greater ease of use for any user wishing to upload additional data to the database, and also allows an administrator to control how and where users can enter data. To start the process, we select Create → Form Design. Our form will need four buttons and one text field, as shown below. The Design View of form creation is reasonably intuitive, and we can create text fields and buttons simply by clicking and dragging the appropriate tools onto the design area. We create the buttons, forms etc. first and we will bind them to functions later.



When closing the form, we will be prompted to give it a name. In this example, we have chosen the name ImportSurveys.

CODE MODULE 1 – THE LAUNCHCD CUSTOM FUNCTION

The text field in the form, labeled Text1 by default, supplies the filename and filepath of the CSV file to import. The user can type this in manually, but to avoid error and give greater usability, we can let users browse to a file. This functionality is supplied by the code module below, which needs to be copied into any free code module, e.g. Code Module 1. It creates a custom function called LaunchCD. Custom functions are very much like parameterized macros in SAS. It is, essentially, a program that can be executed multiple times from within another program.

```
Option Compare Database

Private Declare Function GetOpenFileName Lib "comdlg32.dll" Alias _
```

```
"GetOpenFileNameA" (pOpenfilename As OPENFILENAME) As Long

Private Type OPENFILENAME
    lStructSize As Long
    hwndOwner As Long
    hInstance As Long
    lpstrFilter As String
    lpstrCustomFilter As String
    nMaxCustFilter As Long
    nFilterIndex As Long
    lpstrFile As String
    nMaxFile As Long
    lpstrFileTitle As String
    nMaxFileTitle As Long
    lpstrInitialDir As String
    lpstrTitle As String
    flags As Long
    nFileOffset As Integer
    nFileExtension As Integer
    lpstrDefExt As String
    lCustData As Long
    lpfnHook As Long
    lpTemplateName As String
End Type


Function LaunchCD(strform As Form) As String
    Dim OpenFile As OPENFILENAME
    Dim lReturn As Long
    Dim sFilter As String
    OpenFile.lStructSize = Len(OpenFile)
    OpenFile.hwndOwner = strform.Hwnd
```

Note: Here, we can restrict the file types that a user can select. In this example, it has been left as All Files, including image files. For a real application, it is recommended that the search be restricted to only the expected file types.

```
 sFilter = "All Files (*.*)" & Chr(0) & "*.*" & Chr(0) & _
   "JPEG Files (*.JPG)" & Chr(0) & "*.JPG" & Chr(0)
 OpenFile.lpstrFilter = sFilter
 OpenFile.nFilterIndex = 1
 OpenFile.lpstrFile = String(257, 0)
 OpenFile.nMaxFile = Len(OpenFile.lpstrFile) - 1
 OpenFile.lpstrFileTitle = OpenFile.lpstrFile
 OpenFile.nMaxFileTitle = OpenFile.nMaxFile
```

Note: The default directory and user prompt message can be specified below, along with the message to be displayed if the user does not select anything.

```
    OpenFile.lpstrInitialDir = "C:\"
    OpenFile.lpstrTitle = "Select an Excel file to import"
    OpenFile.flags = 0
    lReturn = GetOpenFileName(OpenFile)
        If lReturn = 0 Then
            MsgBox "A file was not selected!", vbInformation, _
              "Select an Excel file to import"
         Else
            LaunchCD = Trim(Left(OpenFile.lpstrFile, InStr(1, OpenFile.lpstrFile,
vbNullChar) - 1))
         End If
End Function
```

ATTACHING PROGRAMS TO BUTTONS

Next, we need to attach the program to one of the buttons that we created in our form, so that it executes when the user clicks it. Switch on Design View, right click the button, go to Properties, Event Builder, Code Builder, and insert the following code:

```
Private Sub BrowseButton_Click()
Me!Text1 = LaunchCD(Me)
End Sub
```

We then need to import the CSV file as populated in the text box. The code to do so also needs to be attached to click of a button. In this case, the button is called Command3. We can re-caption this button to display "Begin Import".

```
Private Sub Command3_Click()

Dim ImportYesNo As String
Dim NewFileName As String
Dim NewFileNameBrowsed As String
```

Note: A loop is created here that will keep prompting the user to enter more protocols to import until they enter "No" in the accompanying dialogue box.

```
ImportYesNo = "Y"
Do Until ImportYesNo <> "Y"
    NewFileNameBrowsed = Me!Text1
    Dim dbsTemp As Database
    Dim tdfLinked As TableDef
    Dim Cnct As String
        Cnct = "Excel 5.0;HDR=YES;IMEX=2;DATABASE=" & NewFileNameBrowsed
    Set dbsTemp = CurrentDb
```

Note: A temporary table called tblSheet1 is created, populated and then deleted for every iteration through the loop.

```
    Set tdfLinked = dbsTemp.CreateTableDef("tblSheet1")
    tdfLinked.Connect = _
    Cnct
```

Note: The worksheet within the workbook we want to import needs to be specified here, followed by a $ sign.

```
  tdfLinked.SourceTableName = "UNITED$"
  dbsTemp.TableDefs.Append tdfLinked
```

Note: VBA, like SAS, supports SQL as a programming "language within a language." SQL is used here to append the data we have just imported to the Access table we created earlier, named WarehouseMaster.

```
  DoCmd.RunSQL "Insert Into WarehouseMaster Select * From tblSheet1"
  DoCmd.DeleteObject acTable, "tblSheet1"

  MsgBox ("The following Excel file has been added to the database: ") &
NewFileNameBrowsed

    ImportYesNo = InputBox("Do you want to import another protocol? Enter Y for Yes
or N for No", "Title", "Y")
    If ImportYesNo = "Y" Then
        Me!Text1 = LaunchCD(Me)
    End If

    Loop

End Sub
```

We will attach a command to the final unused button that will close the form. Re-caption the button to display the text "Exit." Programmatically, the button is still called Command4.

```
Private Sub Command4_Click()
DoCmd.Close acForm, "ImportSurveys"
End Sub
```

LAUNCHING THE PROCESS

All required code modules have now been entered. To start the import process, view the form we have created in Form View and click on the Begin Import button. After completing the process, Access will confirm the number of rows we have added to the database. Rows are added to the table WarehouseMaster. We can browse to this table to check that it has been updated as we are expecting.

A single table can only ever be a 'flat' listing as opposed to a true relational database. Access contains a very useful tool named Table Analyzer, which splits a table into multiple relational tables without amending the original table. To start the process, simply click Table Analyzer from within the Database Tools tab. This will result in a much more manageable database. However, it is recommended that all data be imported first, as it is not straightforward to import additional listings once the relational structure has been created and populated.

CONCLUSION

We can see that there are several key differences between VBA and SAS data step approaches to data consolidation.

- In brief, a SAS-based solution would probably begin with a PROC IMPORT step, LIBNAME statement or FILENAME statement to read the external files. Various WHERE and IF clauses in the import step, or accompanying data steps, could be used to control which records are read. Finally, we would probably write out directly to an external database using one of SAS's many engines designed to interface with external databases. PROC SQL and an associated CONNECTION TO ODBC would be a good method in this case.
- SAS programs tend to be controlled from a data step. SAS data steps contain an implied loop which allows us to perform the same actions on each observation in a table by default. In VBA, a loop has to be explicitly specified.
- In VBA, variables have to be defined in type and number before a program can commence. SAS allows the user to create variables as and when needed. Only on occasion do length or character attributes need to be defined in advance.
- SAS uses built-in, or sometimes additionally licensed, engines to interface with external objects, with the user typically only needing to specify any options which are not defaults. VBA requires interfaces with external objects to be much more explicitly programmed.
- VBA allows us to create custom functions which can have several key parameters passed to them by the user. In SAS, parameterized macros are used to accomplish similar results.
- VBA is slightly better when it comes to conditionally executing code depending on the value of an observation in an input dataset. SAS requires an understanding of the SAS macro facility to do this.

RECOMMENDED READING

This site gives a good grounding in VBA and explains much of the logic behind the code:

http://www.excel-vba.com/excel-vba-solutions-beginners.htm

CONTACT INFORMATION

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

Jozef Tarrant
Amadeus Software Ltd.
Oxford Science Park
Cowley
+44 (0) 1993 848010
jozef.tarrant@amadeus.co.uk
www.amadeus.co.uk