The Reddick VBA (RVBA) Coding Conventions (version 0.90)

What follows are the Reddick VBA (RVBA) Coding Conventions. The objectives of the conventions are to make code:

- More readable: Conventions allow a reader to understand the meaning of the code with less effort.
- More maintainable: The code can be more reliably changed to fix bugs and enhance functionality.
- More reliable: The code is more likely to perform as expected.
- More efficient: The code performs faster or consumes fewer resources.

These conventions are separate from the RVBA Naming Conventions and may be adopted without adopting the naming conventions. The current version of these conventions can always be found on the Xoc Software web site: http://www.xoc.net.

More rational is provided for these recommendations than is given in the RVBA Naming Conventions. In most cases, there are good rationales for the given conventions. However, in some cases an arbitrary decision was made to select one convention from a set of reasonable alternatives. The other reasonable alternatives to the conventions placed in {braces} at the end of a section. In some cases, a topic only relates to the Visual Basic 6.0 development environment, as opposed to VBA in general. In those cases, the topic is marked with [VB6] after the topic heading.

No set of conventions can cover every case or every consideration. The general rule is that exceptions to the conventions can be made with the approval of the programming team after careful consideration.

The sections are listed in alphabetical order to facilitate their use as a reference work. However, this makes the flow of the document unusual for casual reading as some topics are much more technical than others.

Arrays

Always specify the both the lower and upper bound of an array. This makes explicit whether element zero of the array is a valid element or not. For example:

```
Dim astrValue(1 To 10) As String
```

By convention the index variable used to walk an array should always be a Long data type. This assures that if the array size grows past 32767 elements when maintaining the program that the index variable can still address all elements in the array

When walking an array, always use the VBA LBound and UBound functions to visit each item. This makes sure that every item in the array is visited. For example:

```
Dim iastrValue As Long
For iastrValue = LBound(astrValue) To UBound(astrValue)
    MsgBox astrValue(iastrValue)
Next iastrValue
```

Assertions

VBA provides a built-in assertion mechanism through Debug.Assert. If the expression following the Debug.Assert evaluates to True, the code continues. If the expression evaluates to False, VBA enters Break mode as if a breakpoint had been set on that line. The line shown here acts as a hard coded breakpoint:

```
Debug.Assert False
```

Assertions that do not include a function call in the expression are removed by the compiler when an executable is made, so they only apply to debugging inside the VBA environment. Assertions with a function call in the expression will remain in the executable, but the resulting value of the expression is discarded. VBA doesn't remove function calls because they may have side effects, but discards the return value from the function.

Any time that there is an assumption in the code about the state of the program, there should be an assertion that states the assumption. For example, suppose that a procedure includes this code:

```
Select Case intValue
Case 1
    MsgBox "Aircraft"
Case 2
    MsgBox "AutoMobile"
Case 3
    MsgBox "SnowMobile"
End Select
```

This code assumes that the value of intValue is between one and three. However, if through some bug, intValue had the value of zero or four, this code doesn't work right. The result is that no MsgBox appears at all. Tracking down why the MsgBox doesn't appear is time consuming. Instead, the code could be written one of two other ways. Either:

```
Debug.Assert intValue >= 1 And intValue <= 3
Select Case intValue
Case 1
    MsgBox "Aircraft"
Case 2
    MsgBox "AutoMobile"
Case 3
    MsgBox "SnowMobile"
End Select
```

Or

```
Select Case intValue
Case 1
    MsgBox "Aircraft"
Case 2
    MsgBox "AutoMobile"
Case 3
    MsgBox "SnowMobile"
Case Else
    Debug.Assert False
End Select
```

In general, every Select/End Select block should have a Case Else to trap unexpected values. If the Case Else should never occur, then a Debug.Assert False should be inserted into the block. If the code is correctly written to handle 1 To 3, but zero and four are allowed values, the code should be written with a comment in the Case Else block to indicate that this is expected, like this:

```
Select Case intValue
Case 1
    MsgBox "Aircraft"
Case 2
    MsgBox "AutoMobile"
Case 3
```

```
        MsgBox "SnowMobile"
    Case Else
        'Do nothing
    End Select
```

Assertions trap logic errors early. Rather than waiting to see the results of a bug in the use interface, there is immediate feedback that the bug has occurred. Assertions are only effective if they are present, which means that they have to be added when writing the code. Any logic error that is fixed in the code is a good indication that some additional assertions need to be added.

Comments

A comment in VBA starts with an apostrophe and ends at the end of the line. Comments may be placed on a line by themselves or at the end of a line. A comment starts with the apostrophe followed immediately by the text with no space between the two.

The comment at the end of a line should be used in only a few places:

- At the end of a declaration line
- On a Case line
- On the line that ends a block to indicate what block is being ended. For example on a set of nested If/End If blocks, a comment on the End If line may say what If block is completed. This is especially useful if the block spans several screens.

Examples:

```
Dim dateUTC As String 'time in Univeral Coordinated Time
Case 11 'Division by Zero
```

If the end of a line comment line exceeds the 80 characters line limit, continue the comment on the next line indented by one tab stop. For example:

```
Case 35602 'This key is already associated with an element of this
    'collection
    Set nodChild = tvw.Nodes.Item(cci.Guid)
```

All other comments should be placed on a separate line above the line they are documenting and indented to the same level. A comment of this sort is generally preceded by a blank line unless it is the first line of an indented block. For example:

```
vt = vti.VarType

'Special hack for analyzing my code
If LCase$(Left$(strParamName, Len(strcDecPrefix))) = strcDecPrefix Then
    strDataType = strDecimal
End If
```

If it is the first line of an indented block, it is not preceded by a blank line. For example:

```
If mboolShowProperties Then
    'Show properties for each member
    For Each mi In ci.Members
```

Comments should state the intention of the code not how it performs the task. This is an example of a worthless comment:

```
'Place the VarType into the vt variable
vt = vti.VarType
```

It is worse than no comment at all. The comment is wrong if the code changes to use the variable name vtCur instead of vt without changing the comment. When reading a comment that doesn't match the code, the question becomes whether the comment is correct or the code is correct. Usually it is the comment that

is wrong, but it may take some time to prove that. A wrong comment can be worse than no comment at all. A comment that says the same thing as the following line of code is worthless. In general, don't write comments that have to be maintained, because in the real-world comments frequently aren't maintained.

A comment that states the intention of the code, though, may be useful. For example:

```
'Store VarType for recovery in error condition.
vt = vti.VarType
```

However, use these comments only when the intention is not immediately clear when reading the code. Instead strive to make the code self explanatory, through good naming and coding conventions.

Constants

Always give constants an explicit data type. For example:

```
Private Const dblcPi As Double = 3.14159265358979
```

If a literal value other than zero or one appears in the code, consideration should be given to whether it makes things more readable and maintainable to replace it with a constant. Replace a magic number used more than once in the code with a constant.

Global constants are allowed and encouraged. Replace sets of constants of the data type long with enumerated types using Enum.

Date Functions and Date Variables

Be careful about using the VBA date functions: Date, DateAdd, DateSerial, DateValue, and Now. These functions return a variant containing a date. If implicit type conversion to turns the return value into a string, the string representation of the date displays a two-digit year number. That year number is, of course, not Y2K compliant. This also applies to allowing a variable of type Date to be converted into a string. Instead, use the Format$ function to convert the date into a string. For example:

```
strValue = Format$(Date, "mm/dd/yyyy")
```

Default Properties

Using default properties makes code difficult to read. VBA allows you to just use the name of a textbox and looks up the default property, Text. For example:

```
MsgBox txtValue
```

This prints the value of the txtValue textbox. On the other hand, it is much clearer to say:

```
MsgBox txtValue.Text
```

To even be more explicit, it could even be expressed as:

```
MsgBox Me.txtValue.Text
```

This, however, does not add any additional worth because all references to a control in a module from a form are implicitly on Me.

The reason to be explicit about default properties is to keep the programmer from having to figure out what property is being referenced. This is especially true when referencing ActiveX controls and ActiveX DLLs where the default properties are obscure. For example, when an ADO field is referenced, you are allowed to say:

```
varValue = rst!strFirstName
```

This references the Value property of the strFirstName field. However, it is much clearer to say:

```
varValue = rst.Fields.Item("strFirstName").Value
```

This code doesn't use any default properties and retrieves the same value.

Deprecated Features

Avoid using features Visual Basic supports only for backwards compatibility. Avoid using undocumented features. Also, avoid using functionality that VBA has replaced with functionality that is more modern. Some examples of these kinds of features:

- %, &, $, Etc. in declaration of variables
- Rem statements
- Line numbers (except in conjunction with the Erl function in special error handling situations)
- Single line If statements (use If/End If blocks instead). For example, don't use:

```
If boolValue Then MsgBox "Hi There"
```

- While/Wend loops (replace with Do While/Loop)
- Variables declared with Global (use Public instead). Using Dim in the General Declarations section (use Private instead)
- Using user defined types except in the case of Windows API calls or reading fixed width record files (use Class modules instead)
- Gosub
- The End statement in most cases (simply unload the last form in a standard EXE instead)

Disambiguation

When referencing classes from an ActiveX library, always use the library name to explicitly tell VBA from what library to get the class. If you don't, then VBA will use the order of the libraries in the References dialog to determine from which library it gets the class. The library name always appears in the upper left-hand listbox of the VBA object browser. For example, if there are references to both the Access and Excel object libraries, then this is ambiguous:

```
Dim appObj As Application
```

Because both the Access and Excel libraries include an Application class, which Application class is referenced depends on which one appears first in the References dialog. Instead, it should be declared like this:

```
Dim appObj As Excel.Application
```

Microsoft refers to this as "disambiguation". With this declaration, it does not matter what the order of the libraries is inside the References dialog, as appObj will always refer to the Excel Application object. All references to class names in libraries should include the disambiguating library name.

DLL Base Address [VB6]

The base address is the location that the DLL is loaded into memory. If two DLLs are loaded into the same base address, then VBA moves the second DLL to a new address. VBA then has to modify the binary code within the DLL's address space to reflect the new address. This slows down loading the second DLL.

Libraries used together should start at different base addresses. In Visual Basic, enter the Base Address for a library in the Project Properties dialog Compile tab. Enter a random number base address different than any other used at the same time.

Dollar Sign ($) Functions

If the result of a function is used as a string or assigned it to a string variable, use the $ form of the function. This results in faster executing code, because a conversion from a variant to a string is unnecessary. For example, this is proper usage of dollar sign functions:

```
    If LCase$(Left$(strParamName, Len(strcDecPrefix))) = strcDecPrefix Then
```

This example calls the LCase$ and Left$ functions instead of the LCase and Left functions because the result is used as a String. If the result is used as a Variant, then call the LCase and Left functions instead.

The $ version of the function returns the same value as the Variant version. The one except to the rule is the VBA Date function. The Date function should always be used because the Date$ function doesn't behave correctly. The Date$ always returns information in mm-dd-yyyy format regardless of the Windows localization settings, whereas the Date function uses the localization settings.

Error Handling

A procedure should always include runtime error handling. In general, Error handling should be blocked out the same way in every procedure, as shown in this example:

```
Private Sub Test()
    On Error GoTo ErrorHandler

    'Code for the procedure goes here

ExitProcedure:
    On Error Resume Next
    'Cleanup code for the procedure goes here
Exit Sub
ErrorHandler:
    Select Case Err.Number
    'Case statements for expected errors goes here
    Case Else
        Call UnexpectedError(Err.Number, Err.Description, Err.Source, _
            Err.HelpFile, Err.HelpContext)
    End Select
    Resume ExitProcedure
End Sub
```

Use the label names shown in the example, although the label names have been arbitrarily chosen. Notice that the Exit Sub and ErrorHandler label are left justified making them easily findable. Case statements for expected errors should be given with the error number and a comment with the error message. For example:

```
    Select Case Err.Number
    'Case statements for expected errors go here
    Case 11 'Division by zero
        MsgBox "Zero isn't a valid divisor", vbExclamation, Me.Caption
    Case Else
```

The UnexpectedError routine is a global routine that is only called in a condition where a runtime error that isn’t expected is received, so that there is a bug in the problem. This procedure should log the error message. At the absolute minimum it should just look like this, but ideally it should do a lot more to log the error:

```
Public Sub UnexpectedError(ByVal lngNumber As Long, _
```

```
    ByVal strDescription As String, ByVal strSource As String, _
    ByVal strHelpfile As String, ByVal lngHelpContext As Long)
    On Error Resume Next
    MsgBox "[" & strSource & "]" & vbCrLf & "Run-time error '" _
        & CStr(lngNumber) & "':" _
        & vbCrLf & vbCrLf & strDescription, vbExclamation, App.Title, _
        strHelpfile, lngHelpContext
    Debug.Print "Case " & CStr(lngNumber) & " '" & strDescription
    Debug.Assert False
End Sub
```

The first executable line of every procedure should be the On Error GoTo ErrorHandler line. The only exception to the rule is when a procedure checks the values of its arguments and generates a runtime error when they are invalid. In this case, the checking code comes before the On Error GoTo line. For example

```
Public Sub Test(ByVal intValue As Integer)
    If intValue < 1 Or intValue > 10 Then
        Call Err.Raise(Number:=lngcInvalidValue, _
            Description:=strcInvalidValue)
    End If
    On Error Goto ErrorHandler
```

Exiting a Procedure

In general, a procedure should only have one exit point. Having one exit point makes it easier to read the code and understand when and where it exits. If you use the code mentioned in the Error Handling code section, that exit point is the Exit Sub, Exit Function, or Exit Property line at the top of the error handling. The only other way to exit the procedure should be through using Err.Raise. These Err.Raise lines should occur either before the On Error GoTo line when validating the parameters (see Error Handling) or inside the error handler.

In a few cases, there may be a need to raise an error inside the body of the procedure. In such cases, you should explicitly set any object variables to Nothing (see Nothing), and then exit the procedure. In such cases, the exiting the procedure should be explicitly detailed by a comment that shows the exit, consisting of an arrow stretching to 80 character right margin. The On Error GoTo 0 statement has to be used to turn off error handling for this procedure before executing the Err.Raise. For example, if this code appears somewhere after the On Error GoTo line, it should be written like this to make it explicit that there is an exit point in the middle of the procedure:

```
    If intValue > 1000 Then
        'Raise an Error------------------------------------------------------>
        Set rst = Nothing
        On Error Goto 0
        Call Err.Raise(Number:=lngcInvalidValue, _
            Description:=strcInvalidValue)
    End if
```

For/Next and For Each/Next Loops

The index variable used in the For/Next loop should be specified on the Next line. This makes it explicit which For loop is being completed. For example:

```
    Dim iastrValue As Long
    For iastrValue = LBound(astrValue) To UBound(astrValue)
        MsgBox astrValue(iastrValue)
    Next iastrValue
```

The object variable used to walk the collection should be placed on the Next line in a For Each/Next loop. For example:

```
    Dim frm As Form
    For Each frm in Forms
```

```
        If Not (frm Is Me) Then
            Unload Me
        End If
    Next frm
```

GoTo Statements

You can usually avoid using GoTo statements in VBA code. Use GoTo statements only when the alternative code is not as clear as the GoTo statement. A common reason to use a GoTo statement is to jump out of nested loops. For Example:

```
    For iastrOuterLoop = 1 To 10
        For iastrInnerLoop = 1 To 100
            'some other code
            If astr(iastrOuterLoop, iastrInnerLoop) = "Done" Then
                GoTo ExitNestedLoops
            End If
            'some other code
        Next iastrInnerLoop
    Next iastrOuterLoop
ExitNestedLoops:
    'More code here
```

Headers

Each module should start with a header code that looks something like this:

```
'$Header: $
'******************************************************************************
Option Explicit
'This module includes definitions of Windows API calls
```

The line of asterisks is an apostrophe followed by 79 asterisks. See the section on Long Lines.

Each Public procedure should begin with a header block that looks something like this:

```
Public Sub Almanac(ByVal lngTrecena As Long, ByVal vein As veinc, _
    ByVal lngRows As Long, ByRef alngBlack() As Long, _
    ByRef alngRed() As Long, ByRef aveinRowStart() As veinc, _
    ByRef aveinAlmanac() As veinc, ByRef lngComplete As Long)
    'Generates a Maya almanac
    'lngTrecena [in]        Upper left corner trecena
    'vein [in]              Upper left corner veintena
    'lngRows [in]           Number of rows in the almanac
    'alngBlack() [in]       Black distance numbers across almanac
    'alngRed() [out]        Calculated Red trecena numbers across almanac
    'aveinRowStart() [out]  Calculated Leftmost shown veintenas in almanac
    'aveinAlmanac() [out]   Actual veintenas implied by almanac
    'lngComplete [out]      Number almanac misses completing by.
    'Return value:          None

    'If lngComplete returns zero then it is an almanac, if it is non-zero,
    'then it misses completing and you'll need to report that. You will still
    'need to handle the error encNotAnAlmanac because the black numbers
    'in alngBlack must wrap back to the starting lngTrecena.
    On Error GoTo ErrorHandler
```

Read/write values allowed are [in], [out], and [inout].

Event procedures do not need a header unless the scope is changed to Public. Private procedures may need the header depending on the context. Note that the name of the routine is not referenced in the comments, making it possible to change the name of the procedure without changing the comments. No change history

or coding history is included. Histories should be maintained by source code control systems, not by programmers since they are rarely properly kept up to date.

The comments are addressed to the person calling the procedure, and should include just enough information to tell the person how to call the procedure and use the returned values. After the On Error GoTo, other comments can be placed describing algorithms and other implementation details, if needed (although see the section on Comments).

Indenting

Tab stops should be set at four spaces. No member of a programming team should vary this number, as it makes editing other members of the team's code difficult.

All code inside a block should be indented one tab stop from the surrounding code, with exceptions noted elsewhere in this document. Indenting blocks makes finding the start and end of the block easy. A block is defined as the code that falls between the following keywords:

- Do/Loop
- Enum/End Enum
- For/Next
- For Each/Next
- Function/Exit Function/End Function
- If/Else/ElseIf/End If
- #If/#Else/#ElseIf/#End If
- Property/Exit Property/End Property
- Sub/Exit Sub/End Sub
- Type/End Type
- With/End With

For example:

```
    For Each ci In tlio.Constants
        Set nodChild = tvw.Nodes.Add(Relative:=nod.Key, _
            Relationship:=tvwChild, _
            Key:=ci.Guid & ci.Name, Text:=ci.Name, Image:=strcEnum)
        nodChild.EnsureVisible
        DoEvents
        If mboolShowProperties Then
            For Each mi In ci.Members
                Set nodEnumChild = tvw.Nodes.Add(Relative:=nodChild.Key, _
                    Relationship:=tvwChild, Text:=mi.Name & strcEquals & _
                    mi.Value, Image:=strcConstant)
                nodEnumChild.EnsureVisible
                DoEvents
            Next mi
        End If
    Next ci
```

See also the section on Select/End Select Blocks.

{Alternative: The entire programming team may standardize on another number of spaces.}

{Alternative: Exit Function, Exit Property, and Exit Sub statements may be indented to the level of the surrounding code.}

Instantiation

An object variable should not be declared with New on the line it is declared on, unless there is a good reason to do so. The declaration should instead be broken into two lines. For example:

```
    Dim rst As ADODB.Recordset
    Set rst = New ADODB.Recordset
```

Not this:

```
    Dim rst As New ADODB.Recordset
```

Breaking it into two lines causes each reference to the rst variable to execute slightly faster. In addition, the object variable can be tested to see if it contains the value Nothing. For example:

```
    If rst Is Nothing Then
        MsgBox "rst not initialized"
    End If
```

If a one-line declaration is used, the above code would never execute the MsgBox because the reference to the rst variable in the If statement causes the object to be instantiated before the Is operator is evaluated. For Private and Public object variables, occasionally the convenience of using the New keyword outweighs the performance benefit, so the one-line declaration may still be used.

Labels

Labels in the code should be left justified, regardless of the indenting level of the surrounding code. They should appear on a line by themselves. For example:

```
ExitProcedure:
    On Error Resume Next
```

Long Lines of Code

VBA code editors will scroll a line of code to make the end visible. However, this makes it difficult to read the code quickly. It also means the code is not understandable if placed into a media that doesn't scroll, such as a paper print out or a book. For these reasons, the length of lines should be restricted.

A physical line of code should not exceed 80 characters. If a logical line of code exceeds 80 characters, then the line should be broken into two or more physical lines using the underscore line continuation character. All physical lines in the logical line following the first physical line should be indented one tab stop (four spaces) from the first physical line.

It may help to place a line at the top of the module with an apostrophe followed by 79 asterisks. Then the code window of the VBA editor can be sized to barely make the last asterisk visible. A fixed width font, such as Courier New, should be used to display the code in the VBA code window.

You should choose an appropriate place to break the line to enhance the maximum readability of the remaining code. When breaking lines that have a list separated by commas, you should break the line after a comma and before the next non-space character. For example:

```
Private Sub GetFiles(ByRef fso As Scripting.FileSystemObject, _
    ByRef fld As Scripting.Folder)
```

When breaking a line that is an expression built by operators, break the line before an operator of the expression. For example when the expression is built of string concatenation operators, break it like this:

```
    strParameters = strParameters & strAdd _
        & strPassingConvention & pmi.Name & strArray _
        & strcAs & strDataType & strDefault
```

The next line becomes more readable this way.

If you have a long literal string, you may have to break the line like this:

```
    strValue = "This is a very, very long string that will cause the code " _
        & "to wrap. Because of this, you will need to break it."
```

In such cases, break it before the start of a word. Note that VBA performs the string concatenation at runtime, so this has performance considerations. In many cases, the string should be placed into a constant, an entry in resource file, or a database field and retrieved from there.

Comments should never be continued. When a comment exceeds 80 characters, continue the comment on the next line preceded by another apostrophe. See the section on comments.

Don't overly indent lines. Move overly indented code to a new procedure and call it from the original. In general, code should not need to be indented more than eight tab stops.

{Alternative: Place operators at the end of the line before the line continuation character instead of on the next line.}

Nothing

Explicitly set Object variables to Nothing before allowing the variable to be destroyed. This is especially true of object variables declared with the Dim keyword. For Example:

```
Public Sub Test(ByVal intValue As Integer)
   'error handling omitted for clarity
    Dim rst As ADODB.RecordSet
    Set rst = New ADODB.RecordSet
    'More code here
    Set rst = Nothing
End Sub
```

Setting the object variable to Nothing is not just good programming practice. If the rst object has code in its Class_Terminate event handler, that code can mess with global variables and objects.

In addition, set the object variable to Nothing is before exiting the procedure with Err.Raise. For example:

```
Private Sub Test(ByVal intValue As Integer)
   'error handling omitted for clarity
    Dim rst As ADODB.RecordSet
    Set rst = New ADODB.RecordSet
    'More code here
    If intValue > 1000 Then
        'Raise an Error------------------------------------------------------>
        Set rst = Nothing
        On Error Goto 0
        Call Err.Raise(Number:=lngcInvalidValue, _
            Description:=strcInvalidValue)
    End if
    Set rst = Nothing
End Sub
```

In the example just shown, if the rst object variable is not set to Nothing before performing the Err.Raise, the Class_Terminate of the rst object likely will change the properties of the Err object so that it no longer reflects the number given in lngcInvalidValue. This Class_Terminate code executes before the calling routine's error handler is invoked. This weird flow of execution has caused a number of very difficult to track down bugs.

Parameters to a Procedure

Every parameter to a procedure should be given an explicit data type, including variants. Every parameter should be passed by value using the ByVal keyword, with a few exceptions. These are:

- VBA doesn't allow certain data types to be passed by value, such as arrays, user-defined types, and objects.
- You specifically want to allow the changed value of the parameter to be passed back to the calling routine.
- The parameter to event procedure is specified as being by reference when VBA creates it.
- The arguments to a Declare statement must match the definition in the DLL.

Even in the cases where the argument should be passed by reference, you should explicitly prefix the parameter with ByRef, even though this is the default in VBA. This makes it explicit that you meant to pass that parameter by reference.

After VBA inserts an event procedure, the parameters to the event procedure should be changed to include ByVal and ByRef keywords, and change the parameter names to use the appropriate naming conventions. For example, VBA inserts the event procedure like this (with the line wrapped in this document):

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y
As Single)

End Sub
```

This should be changed to read like this:

```
Private Sub Form_MouseMove(ByRef intButton As Integer, _
    Byref intShift As Integer, ByRef sngX As Single, ByRef sngY As Single)

End Sub
```

By changing it to read like this, the naming conventions indicate the data type and the ByRef keywords indicate that VBA may see the changes to the parameters.

Parentheses

Always use parentheses where the reading of the line may be unclear. For example, suppose that a line is written:

```
    If Not frmTest Is Nothing Then
```

It may not be clear that the Is operator has higher precedence than the Not operator in this line. Recode it to read:

```
    If Not (frmTest Is Nothing) Then
```

This makes it clear what order the operators are evaluated. The general rule is that if there is any question what the operator precedence is, use parentheses to make it clear.

Procedure Scope

Always use the Private scope on a procedure unless you need to expose the procedure outside the current module. In a library, use the Friend scope when you need a larger scope. Use Public only when access to the procedure is required outside the library. For Example:

```
Private Sub Test()
```

Project Properties [VB6]

In Visual Basic, the Project Properties dialog should always be filled in. These values may not apply to VBA hosts other than Visual Basic. Most of these fields can be retrieved from the EXE, DLL, or OCX file by right clicking on it in the Windows Explorer, then selecting Properties, then clicking on the Version tab in the dialog that appears. The values can be retrieved from within the program by getting properties of the App global system object. The following fields should be always be filled in:

- Project Name: The name of the library or standard EXE name. The library name should always start with a short word or abbreviation indicating the company or organization that is developing the library. For example, the Maya Calendar engine library from Xoc Software, might be named XocEngine or XocMayaEngine. This term is used for disambiguation of libraries and shows in the Object Browser. See the Disambiguation section. This is the internal name of the library. This may have abbreviations in it.

- Project Description: This should be the same name as the Project Name, except with spaces between the words. Abbreviations and the company or organization name should be spelled out. For example, use Xoc Maya Engine. These words show up in the VBA References dialog.

- Major/Minor/Revision number: These should be filled in with appropriate values. The version number should never be set to a smaller value as installation programs depend on it to determine if they should overwrite an older version with a newer one.

- Auto Increment: In most cases this should be checked. This automatically increments the revision number by one every time the project is compiled to a file.

- Application Title: The application title should be the name of the product that you expect to show externally, on the Windows Start menu, the Windows task list, the Windows Task Bar, and should be copied to the Caption of the main form in the application when the program starts. For example: Xoc Maya Engine.

- Comments: If Visual SourceSafe is used to maintain the project, this should be filled in with $Header: $. If keyword expansion in files is used, then Visual SourceSafe will place the expansion into the comments section of the executable. This gives the source name of the project is, the SourceSafe version number of the VBP file, the date and time the project file was changed, and by whom. This helps roll back the project to a given release to test for bugs. See the section on Source Code Control to configure SourceSafe.

- Company Name: Should be filled with your company or organization name. For example: Xoc Software. This is used on splash screens and about dialogs. Therefore, if your company is XYZ Software, Inc., you probably want to use XYZ Software.

- File Description: This is the description of how this file fits into the entire package. For example: Xoc Maya Calendar calculation engine or Xoc Maya Calendar UI.

- LegalCopyright: Enter the copyright notice for the program. For example: Copyright © 1999 by Xoc Software. You may find it useful to type Alt+0169 on the keypad (not the main keyboard) to get the © symbol in the dialog.

- LegalTrademarks: Enter any trademarks or registered trademarks for the company or product. For example: Xoc™ is a trademark of Xoc Software. You may find it useful to type Alt+0153 on the keypad (not the main keyboard) to get the ™ symbol and Alt+0174 to get the ® symbol. Note that these symbols may or may not show correctly in the application depending on the font you choose to display them.

- Product Name: This is the name of the product, without the company name. Therefore, if the name of the product elsewhere is Xoc Maya Calendar, the name here should be just Maya Calendar. This value may be used in splash screens and about dialogs.

Raising Errors

When you raise a runtime error from a component, to be trapped in the calling code, the error number that you raise should have a unique error number. For this purpose, VBA defines a constant vbObjectError that guarantees that errors that you generate will not conflict with ones that VBA defines. However, all libraries use errors in the range larger than vbObjectError, so you should strive to be different from the other libraries with your numbers. There is no way to guarantee this; the chances can be reduced by starting your errors at a random number in the range 512 to 32767 larger than vbObjectError. No library that an organization produces should ever have conflicting error numbers with another library from the same organization. For example: XYZ Software might start numbering its errors at vbObjectError + 4096. The first library produced from XYZ software might generate errors in the range from vbObjectError + 4096 to vbObjectError + 4146, the second library from vbObjectError + 4147 to vbObjectError + 4196, etc.

Select/End Select Blocks

The Select/End Select block is indented differently from other blocks (see Indenting). The Case blocks within the Select/End Select are lined up with the Select/End Select keywords. Code within a Case block is indented one tab stop from the Case statement. For Example:

```
Select Case Err.Number
Case tliErrCantLoadLibrary
    Err.Raise Number:=Err.Number, Description:=Err.Description, _
        Source:=Err.Source
Case 35602 'This key is already associated with an element of this
    'collection
    Set nodChild = tvw.Nodes.Item(cci.Guid)
    nodChild.Image = "InstClass"
    Resume NextItem
Case Else
    Call UnexpectedError(Err.Number, Err.Description, Err.Source, _
        Err.HelpFile, Err.HelpContext)
End Select
```

In non-RVBA coding standards, it is more common to indent Case blocks one tab stop from the surrounding Select/End Select. However, this causes the actual executing code to be indented two tab stops from the surrounding Select/End Select. The readability of the code is just as good, if not better with this scheme, although it takes some getting use to the first Case block being indented to the same level as the Select line.

See also the note about Case Else blocks in the section on Assertions.

{Alternative: Indent the Case blocks one tab stop from the surrounding Select/End Select. Then indent the code in the case blocks one more tab stop.}

Source Code Control [VB6]

Code should be maintained using some sort of Source Code Control. Microsoft Visual SourceSafe is the most common product used for this. When using Visual SourceSafe, the Administrator should configure it to expand keywords in files in the SourceSafe Administrator Options dialog. The following files should be expanded: *.bas,*.cls,*.ctl,*.frm,*.pag,*.vbp. Entries such as $Header: $ can then be placed into the code and are expanded automatically. See the SourceSafe documentation on keyword expansion. Also, see the use of the Comments entry in the section on Project Properties in this document.

Type Conversion

VBA is considered a weakly typed language. You can construct expressions such as this one:

```
strValue = "Your order came to " & intQuantity * curPrice
```

VBA will automatically convert the result of the expression into a string to make the expression work. However, it is better programming practice to make explicit what VBA is doing using the type conversion functions: CBool, CInt, CLng, CStr, etc., plus the Format$ function. For example:

```
strValue = "Your order came to " _
    & Format$(CCur(intQuantity) * curPrice, "$#,###.00")
```

The RVBA naming conventions will help point out possible bugs. If you see a line that looks like this, you may have a potential bug:

```
intValue = lngInput
```

If the value in the variable lngInput is 90,000, this line will cause an Overflow runtime error. The fact that the types of the variables are different is a clear warning sign. If, however, you knew the value in lngInput could only be in the range 1 to 1000, it might be acceptable to do the assignment like this:

```
Debug.Assert lngValue >= 1 And Debug.Assert lngValue <= 1000
intValue = CLng(lngInput)
```

See also the section on Assertions.

Variable Declaration

Every variable should be explicitly declared. Using the Option Explicit keyword at the top of the module will have VBA enforce that. The VBA editor's Tools Options dialog has a setting that will make this be automatically inserted in all new modules.

Every variable should be given an explicit data type. This includes variants, which are the default. For example, a variant should be declared as:

```
Dim varValue As Variant
```

Rather than letting it be implicitly defined or declaring it as:

```
Dim varValue
```

Every variable should be declared on a line by itself. This precludes running into this bug:

```
Dim intValue, intTest As Integer
```

That declaration makes it obvious that the two variables were meant to be declared as integers, but the first variable is defined as a variant. If instead the declarations were made on a line by themselves, the problem goes away. For example:

```
Dim intValue As Integer
Dim intTest As Integer
```

In addition, by declaring each variable on a line by itself, you can use the Ctrl+Y keyboard shortcut to cut the declaration to the clipboard regardless of where the caret is on the line, then paste it somewhere else. If there are multiple declarations on a line then editing is not as easy.

Variable Initialization

Some languages allow declaring a variable and giving it an initial value at the same time. Visual Basic doesn't allow that. A variable has a default value at the time that it is declared based on its data type. However, a syntax variation can be used inside of a procedure to give the feel of initializing it and assigning the default value.

Visual Basic allows you to place multiple logical lines of code on the same physical line if you separate them with colons. It also allows you to mix the declaration of variables with executable lines of code. Therefore, inside a procedure you can initialize a variable and give it a default value in one physical line like this:

```
Public Sub Test()
    'Error handling omitted for clarity
    Dim intValue As Integer: intValue = 7
    Dim strTest As String: strTest = "Default Value"
    'other code
End Sub
```

Variables Scope and Lifetime

Variables should always be declared with the smallest level of scope and the shortest lifetime possible. Thus, you should declare variables with Dim inside of procedures by preference. If you need a longer lifetime, then use Static. If you need a wider scope, use Private. Only use Public as a last resort. Public variables declared in standard modules are global and can be changed by any piece of code throughout the entire project. This makes debugging changes to their value very difficult. Global variables of this sort should only be used in the context where they are set during initialization of the program, then remain static for the rest of the time the program executing.

Global variables should never be changed in one part of the program to be retrieved in another part of the program. In such cases, you should use parameters of procedures or properties of forms or objects to pass the information. If there are more than 20 global variables in the program, it is a warning sign that the program design is wrong.

Having Public constants are allowed and encouraged. See the section on constants.

Version Compatibility [VB6]

After building an ActiveX control or ActiveX DLL for the first time in Visual Basic, the project compatibility should be set to Binary Compatibility in the Project Properties dialog. Each time the component is "released," a copy of the component should be made to the same directory, but with the filename extension set to CMP. The binary compatible file entry should point to this file. That means that you can modify the interface to the file within a release as long as you are still backwards compatible with the last release. The CMP file should be checked into the Source Code Control project, whereas the current copy of the component itself probably should not be checked in (see the section on Source Code Control).

Greg Reddick is the President of Xoc Software, a software development company developing programs in Visual Basic, Microsoft Access, C/C++, and for the web. He leads training seminars in Visual Basic for Application Developers Training Company (AppDev). In a previous life, he worked for four years on the Access development team at Microsoft. Greg can be reached at mailto:grr@xoc.net or from the Xoc Software web site, http://www.xoc.net.