See also:

Access Techniques for VBA code techniques, some is specific to Access, some are general.

This document presumes you know something about VBA. It is not a primer. It is my personal resource.

CONTENTS

About Macros 3
Organizing Macros 3
Export and Import Code 3
Get to Know the Object Model 3

Miscellaneous Subjects 4
Anatomy of a Word Document 4
Moving Around in a Word Document 4
Working with Documents 6
Working with Templates 7
Working With Files 7
Working With Styles 7
Working With Path 8
Working With Text 9
Portrait and Landscape 13
Document Properties 14
Page Numbering 15
Lists 16
Working with Tables 16
Addressing Table Cells 19

Dialog Boxes 20
Message Box Object 20
Present Information 21
Prompt User for Choices 21
Dialog Object 21
Prompt User for File(s) or Directory with FileFind Dialog – OBSOLETE 22
FileDialog Object 23
Prompt User to Select Folder with FileDialog FolderPicker 24

Documenting Your Shortcut Keys 24
About Keyboard Shortcut Keys 25
Run This Code 25
DOM Background 26

Field Codes in VBA 27
Index Object 28
Table of Contents 29
RD Field Code 29
Field Codes and Page Numbers 30

Page Headers and Footers 30

Assembling Multi-file Documents 32
Approach 32
Code Samples 33
Prompt User for Directory of New File 33
Insert Chapter Files 33
Insert Section Break Odd Page 34
Update Main Table of Contents 34
Update Main Table of Figures 34
Save New Document 34
Update Chapter Tables of Contents 34
Update Indices 34
Delete Macros in New Document 35
Subroutines 35
Variations 36

Shapes 36
About Shapes 36
Anchoring a Shape 36
Positioning a Shape 37
Formatting a Shape 37
Other Important Shape Properties 37
Remarks 37
Converting Visio Picture into Inline Shape 38
Key Properties 38
RelativeVerticalPosition Property 38
RelativeHorizontalPosition Property 38
Top Property 38
Left Property 39
Shrink Inline Shapes 40

Watermarks 42
Background Printed Watermark 42
Watermark as Text Box 43
Code Created by Insert Print Watermark Background Wizard 44

Iterative Document Editing 45
Reformat Text in Square Brackets 45
Insert RD Field Codes 46
Change Styles in all Word Files in a Given Directory 49

Passwords and Protection 50

Interacting with an Access Database.....51

Automation 52

Inventorying Macros 52
Relevant Object Models.....53
Iterating Procedures in Modules 55
Basic Logic to Inventory Macros 56

Inventorying Files 56
Relevant Objects.....57
Code.....57

ABOUT MACROS

Recording keyboard actions is a good way to get the needed code, then you can edit it into an efficient macro.

Organizing Macros

Macros (in newer versions of Word) are stored in modules. They may reside in any kind of Word file, document or template. All the macros in a file comprise a Project.

By default new recorded macros are created in the NewMacros project of normal.dot.

It can be helpful to organize macros into modules. Perhaps you put all macros used together in the same module. Perhaps you put utility macros, like those that transpose two adjacent characters, in the same module. But if you have 100 macros, it is better if they are distributed across 5 or so modules.

You can create a module in the VBA editor with menu Insert, Module.

You can rename a module: First select the module, then use menu View, Properties Window. to open the same-named window. Properties are listed in tabular format with the names in the left column and the values in the right column. Select the value for the Name property and retype it. Click elsewhere for it to take effect.

You can move a macro from one module to another: In the Code window select the text of the macro, cut it, open the Code window of the new module, position the cursor, and paste the text.

You can delete a module: select it then use menu File, Remove.

Export and Import Code

A module's code can be exported as a .bas text file. Such a file can be imported into a different project. This can be a convenient method for copying code from one Word file to another.

Get to Know the Object Model

Because VBA is an object-oriented language, you will be most effective if you understand the Word object model and how to manipulate it. Basically the model has objects. Objects are grouped into collections which are an object unto themselves. Objects may have child objects and/or parent objects. Objects also have methods and properties. When you are trying to figure out how to do something, you must identify the relevant object and property or method. Sometimes you start with the property or method and work backward to the object. Look in the help file for a diagram.

MISCELLANEOUS SUBJECTS

Anatomy of a Word Document

When you are editing a Word document with VBA, it can be important to understand the different parts—objects—and how to deal with them. You could study the Word Object Model, which you can find in the VBA Help file. Key objects that I have needed to handle include:

Object	*Parent Object*	*Contents*
Window	Application	all the (open) windows
Pane	Window	the window panes for a given window
Document	Application	an open document
Section	Document	sections are used to hold text formatted differently than the base document, including multiple columns and different page orientation and/or margins
Footnotes	Document	

There are special properties that return a particular object:
ActiveDocument. Returns a Document object for the document that is active, i.e., the document with the focus.
ActiveWindow. Returns a Window object for the active window, i.e., the window with the focus.
ActivePane. Returns a Pane object for the active pane in specified window.
SeekView. Returns or sets a View object with the document element displayed in print layout view. It uses the WdSeekView constant to specify which view: main document, header, footer, endnotes, footnotes. There are several variations of header and footer: current, first, even, primary. This property can incur a run time error of 5894 if the view is not Print Layout.

Examples:
```
ActiveDocument.ActiveWindow.ActivePane
ActiveDocument.ActiveWindow.View.SeekView = wdSeekMainDocument
If ActiveDocument.ActiveWindow.View.SeekView <> wdSeekMainDocument Then . . .
```

The following code will change the view/pane to Normal view, even when the document is in print layout view and the header is open. Hence, it is very useful for resetting the state of the document prior to editing.
```
ActiveDocument.ActiveWindow.View.Type = wdNormalView
```

Moving Around in a Word Document

By "moving" I mean moving the cursor. This is accomplished with methods of the **Selection** object. The selection can be extended or collapsed to an insertion point. The principal methods are those that reflect the keyboard direction keys: Home, End, Up, Down, Left, Right.

Use the **Select** property to return the **Selection** object. If the Selection property is used without an object qualifier, the object is assumed to be the active pane of the active document window. It may be advisable to not assume what the active pane is.

Which, unfortunately, gets us into panes. A window has more than one pane if the window is split or the view is not print layout view and information such as footnotes or comments are displayed. From this it seems safe to assume that the first pane is the one that opens when the document is first opened.

To close all but the first pane:

```
MsgBox ActiveDocument.ActiveWindow.Panes.Count
If ActiveDocument.ActiveWindow.Panes.Count > 1 Then
    For i = 2 To ActiveDocument.ActiveWindow.Panes.Count
         ActiveDocument.ActiveWindow.Panes(i).Close
    Next
End If
```

Similarly don't assume the cursor is in the main body of the document, it might be in a header/footer or footnote. The following code will incur a run time error if the document is not in print layout view.

```
If ActiveDocument.ActiveWindow.View.SeekView <> wdSeekMainDocument Then
    ActiveDocument.ActiveWindow.View.SeekView = wdSeekMainDocument
```

Better:

```
If ActiveDocument.ActiveWindow.View.Type = wdPrintView Then
If ActiveDocument.ActiveWindow.View.SeekView <> wdSeekMainDocument Then
    ActiveDocument.ActiveWindow.View.SeekView = wdSeekMainDocument
End If
End If
```

To go to the first character:

```
Selection.HomeKey Unit:=wdStory
```

In the previous example, wdStory is one value of the WdUnits constants. Other values: wdCharacter, wdWord, wdSentence, wdParagraph, wdSection, wdCell, wdColumn, wdRow, wdTable.

To go to the first character and select the first paragraph:

```
Selection.HomeKey Unit:=wdStory
Selection.MoveDown Unit:=wdParagraph, Count:=1, Extend:=wdExtend
```

To release selection by moving cursor to the right:

```
Selection.MoveRight Unit:=wdCharacter, Count:=1
```

To move down 8 paragraphs:

```
Selection.MoveDown Unit:=wdParagraph, Count:=8
```

To move to the start of the current line:

```
Selection.HomeKey Unit:=wdLine
```

To move to the end of the current line:

```
Selection.EndKey Unit:=wdLine
```

To move to the end of the document:

```
Selection.EndKey Unit:=wdStory
```

Move cursor:

```
Selection.MoveRight Unit:=wdCharacter, Count:=1, Extend:=wdExtend
```

Collapse a range or selection to the starting or ending position. After a range or selection is collapsed, the starting and ending points are equal. You can optionally specify the direction in which to collapse the range, as start or end.

- If you use wdCollapseEnd to collapse a range that refers to an entire paragraph, the range is located after the ending paragraph mark (the beginning of the next paragraph).
- Using wdCollapseEnd to collapse a selection that refers to the last table row causes the cursor to be located after the end of the table.
- Using wdCollapseStart to collapse a selection that refers to a table row causes the cursor to be located in the first cell of that row. If you collapsed a group of rows, say after inserting them, the cursor is located in the first cell of the first row.

```
Selection.Collapse
```

Collapse the selection to an insertion point at the beginning of the previous selection.

```
Selection.Collapse Direction:=wdCollapseStart
```

Working with Documents

Documents collection consists of all open documents.

Open a named document:

Documents.Open *FileName, ConfirmConversions, ReadOnly, AddToRecentFiles, PasswordDocument, PasswordTemplate, Revert, WritePasswordDocument, WritePasswordTemplate, Format, Encoding, Visible, OpenConflictDocument, OpenAndRepair, DocumentDirection, NoEncodingDialog*

Example:

```
Documents.Open FileName:="C:\MyFiles\MyDoc.doc", ReadOnly:=True
Documents.Open "c:\data\this.doc"
```

Close the active document:

ActiveDocument.Close *SaveChanges, OriginalFormat, RouteDocument*

Example: `ActiveDocument.Close  wdSaveChanges`

Save the active document:

expression.Save(*NoPrompt, OriginalFormat*)

Example: `ActiveDocument.Save True`

Create new empty document:

expression.Add(*Template, NewTemplate, DocumentType, Visible*)

Example: `Documents.Add`

Make a document the active one:

```
Documents(1).Activate
Documents("Report.doc").Activate
```

Run-time errors for files and directories:

53 - file not found
55 - file already open
57 - device I/O error
75 - path/file access error
76 - path not found

Sometimes you need to know the actual filename. The following properties apply to a Document object and to a Template object.

Property	*Example*
FullName	c:\Data\Templates\normal.dot
Name	normal.dot
Path	c:\Data\Templates
PathSeparator	\

Working with Templates

Generally there are two kinds of templates:
1. global template
2. document template

Both templates are identified in the "Templates and Add-Ins" dialog box. They are opened automatically when you start Word. The universal document template is commonly called **normal.dot** and may be located anywhere on a local drive. When a particular document template is attached to a particular document, the template is opened when the document is opened.

The Template object applies to the Application and Global objects. It is described somewhat in the section Inventorying Macros on page 52.

Templates have the same filename properties as Documents.

Working With Files

ActiveDocument is a property that returns a Document object. **ActiveDocument.Name** returns the filename of the active document.

You can perform some task for all the files in a given directory. See section Change Styles in all Word Files in a Given Directory on page 49 for a sample.

You can perform some task for all the files in a given directory structure. This is described in *Microsoft Access Techniques* in the section titled "Walking A Directory Structure" (page 186 the last time I looked).

Working With Styles

Copy a style to a document:

```
Dim strFilename As String
Dim strTemplate As String
strFilename = ActiveDocument.FullName
strTemplate = "C:\Data\CMS\Template for CMS Operations Guide.dot"
Application.OrganizerCopy Source:=strTemplate, Destination:=strFilename,
Name:="Default Paragraph Font", Object:=wdOrganizerObjectStyles
```

```
Application.OrganizerCopy Source:=strTemplate, Destination:=strFilename,
Name:="Bold", Object:= wdOrganizerObjectStyles
```

Built-in styles cannot be deleted from a document.

Delete a custom style from the document:

```
Dim strFilename As String
strFilename = ActiveDocument.FullName
Application.OrganizerDelete Source:=strFilename, Name:=”unwantedStyleName”,
   Object:=wdOrganizerObjectStyles
```

List the styles in use in a given document:

```
Sub ListStylesInUse()
Dim docTem As Document
Set docTem = Documents.Open("c:\Data\From Others\Stacey
Duke\NGI_Detailed_Design_Template_v01.doc")
Dim docNew As Document
Set docNew = Documents.Add
Dim s As Style
For Each s In docTem.Styles
    If s.InUse Then
        docNew.Activate
        Selection.InsertAfter s.NameLocal & vbCrLf
        Selection.Collapse Direction:=wdCollapseEnd
    End If
Next
End Sub
```

Does a given style exist in the document? If you try to access a named style that does not exist, you will get a 5122 run time error, “This style name does not exist.”

```
Msgbox StyleExists("Heading 1")

Function StyleExists(strStyle as string) as Boolean
Dim t
On Error Resume Next
StyleExists=True
Set t=ActiveDocument.Styles(strStyle)
If Err.Number<>0 then StyleExists=false
Err.Clear
End Function
```

Working With Path

There are several functions which can be used to work with a path:

- CurDir returns or sets the current path-directory. NOT ALWAYS RELIABLE
- ActiveDocument.Path is reliable. Use it to get the path of a document that is open.
- ChDrive changes the current default drive.
- ChDir changes the current default directory.
- MkDir creates a directory.
- RmDir deletes an empty directory.
- Kill deletes one or more files in a drive or directory. If argument is a directory, all the files in that directory are deleted. Wildcards may be used in filename argument.
- ChangeFileOpenDirectory sets the directory in which Word searches for files. The specified directory’s contents are listed the next time the File Open dialog box is opened.

Function CurDir returns or sets the current path-directory. If there is no argument, it returns the current directory. If there is an argument, it sets the current directory to the value of the argument.
Syntax:
CurDir[(drive)]

Example:

```
' Assume current path on C drive is "C:\WINDOWS\SYSTEM" (on Microsoft Windows).
' Assume current path on D drive is "D:\EXCEL".
' Assume C is the current drive.
Dim MyPath
MyPath = CurDir         ' Returns "C:\WINDOWS\SYSTEM".
MyPath = CurDir("C")    ' Returns "C:\WINDOWS\SYSTEM".
MyPath = CurDir("D")    ' Returns "D:\EXCEL".

ChDrive "D"             ' Make "D" the current drive.
ChDir "D:\TMP"          ' Make "TMP" the current directory on D drive.
ChDir ".."              ' Moves up one directory in Microsoft Windows.
ChDir "MYDIR"           ' Make "MYDIR" the current directory on current drive.
MkDir "MYDIR"           ' Create directory "MYDIR" on current drive.
MkDir "D:\MYDIR"        ' Create directory "MYDIR" on D drive.
RmDir "MYDIR"           ' Delete directory "MYDIR" on current drive.
Kill  "*.TXT"           ' Delete all files ending in .TXT in current directory.
Kill  "c:\DATA\THIS.DOC"  ' Delete named file in named path.
ChangeFileOpenDirectory "c:\Data\this.doc"
```

Working With Text

Refer to all text in the current document:

```
ActiveDocument.Content
```

Select the whole word in which the cursor is located; cursor may be at the beginning, end, or in the middle:

```
Selection.Expand wdWord
```

InsertAfter is a method of Selection or Range object.

The essential tasks are
(1) position the cursor
(2) enter the text

Interspersing fields in text characters can make this more complex.

You can insert special characters such as quotation marks, tab characters, and nonbreaking hyphens by using the Visual Basic Chr function. You can also use the following Visual Basic constants: vbCr, vbLf, vbCrLf, and vbTab.

Entering text is done with a range or selection object. The relevant methods and properties are:

- Method `Selection.TypeText`. Optionally the text can replace the original contents of the selection.
- Method `Selection.TypeParagraph` inserts a new, blank paragraph. If the selection isn't collapsed to an insertion point, it is replaced by the new paragraph.

- Property `Selection/range.Text` returns or sets the text.
- Method `Selection/range.InsertBefore` and `Selection/range.InsertAfter` which inserts the specified text at the beginning/end of a selection/range. After this method is applied, the selection/range expands to include the new text. For the InsertAfter method, if the selection/range ends with a paragraph mark that also happens to be the end of the document (and perhaps a footer), the text is inserted before the final paragraph mark.
- Method `Selection/range.InsertParagraph` replaces the specified range or selection with a new paragraph.
- Method `Selection/range.InsertParagraphAfter` inserts a paragraph mark after a selection/range.
- Method `Selection/range.InsertParagraphBefore` inserts a paragraph before after a selection/range.
- Method `Selection/range.InsertSymbol` inserts a named symbol in place of the specified selection/range.
- Method `Selection/range.Collapse` collapses a selection/range to the starting or ending position. After a selection/range is collapsed, the starting and ending points are equal. If you use wdCollapseEnd to collapse a range that refers to an entire paragraph, the range is located after the ending paragraph mark (the beginning of the next paragraph).
- Method `Selection/range.Move` collapses the specified selection/range (must be object variable) to its start or end position and then moves the collapsed object by the specified number of units. If Count is a positive number, the object is collapsed to its end position and moved forward in the document by the specified number of units. If Count is a negative number, the object is collapsed to its start position and moved backward by the specified number of units. The default value is 1. You can also control the collapse direction by using the Collapse method before using the Move method. If the range or selection is in the middle of a unit or isn't collapsed, moving it to the beginning or end of the unit counts as moving it one full unit.

This method seems to be a generic version of methods MoveLeft and MoveRight. It seems to be a way of setting the cursor for actions that necessitate a range, such as inserting a field.

Entering a field is done:

- Method `Selection.Fields.Add` inserts a field in a named range. If the range isn't collapsed, the field replaces the range.

Entering a field after text and following it with text requires:

- before adding a field: collapse the range
- after adding a field: set the range end to the field end

Examples of entering text:

a. With method Selection.TypeText. In the example below a field is entered in between text characters:

```
Selection.TypeText Text:="Revision:  "
Selection.Fields.Add Range:=Selection.Range, Type:=wdFieldEmpty, Text:= _
    "SAVEDATE  \@ ""M/d/yyyy"""
Selection.TypeText Text:=vbTab & "Page "
```

b. With property range.Text This technique requires additional actions to move the cursor before and/or after entering text characters and fields.

```
rf.Text = "My entered text"
```

For a footer, the text is usually either new or replaces existing text. Thus the text is written at the beginning of the area in a sequential manner.

1.1 set a header-footer object = footer

```
Dim ftr As HeaderFooter
Set ftr = ActiveDocument.Sections(1).Footers(wdHeaderFooterPrimary)
```

1.2 set a range object = the range of the header-footer object

```
Dim rf As Range
Set rf = ftr.Range
```

1.3 is there text present? If so prompt user to keep or replace.

```
Dim appTitle As String
Dim msgText As String
appTitle = "Initialize Footer"
msgText = "Footer is not empty. Do you want to replace it?"
If rf.Text <> vbCr Then    ' not empty
    If MsgBox(msgText, vbYesNo, appTitle) = vbYes Then rf.Text = "" Else Exit
Sub
End If
```

1.4 enter text characters

```
rf.InsertBefore Text:="Revision:  "
rf.InsertAfter Text:=”Concatenated text.”
```

The problem is inserting fields in the desired location. My preferred footer is:
Revision: <savedate> , <tab> Page <pagenum> of <numpages><paragraph break>
<filename>
It seems as though the sequence is:

1. insert text “Revision: “ into range (assuming the range is empty, InsertBefore is fine)
2. collapse range to end point
3. insert field savedate
4. collapse range to end point
5. insert tab as vbTab and “page “ with InsertAfter
6. collapse range to end point
7. insert field pagenum
8. collapse range to end point
9. insert text “ of “ with InsertAfter
10. collapse range to end point
11. insert field numpages
12. collapse range to end point
13. insert paragraph break as vbCrLf with InsertAfter
14. collapse range to end point
15. insert field filename

The code that works:

```
Sub InitFooterNew()
'
' Initialize page footer with save date, page number, and file name.
'
Dim appTitle As String
Dim msgText As String
Dim ftr As HeaderFooter
```

```
Dim rf As Range
Dim fld As Field
appTitle = "Initialize Footer"
msgText = "Footer is not empty. Do you want to replace it?"
Set ftr = ActiveDocument.Sections(1).Footers(wdHeaderFooterPrimary)
Set rf = ftr.Range
If rf.Text <> vbCr Then    ' not empty
    If MsgBox(msgText, vbYesNo, appTitle) = vbYes Then rf.Text = "" Else Exit
Sub
End If
rf.InsertBefore Text:="Revision:  "
rf.Collapse Direction:=wdCollapseEnd

Set fld = ActiveDocument.Fields.Add(Range:=rf, Type:=wdFieldSaveDate,
Text:="\@ ""M/d/yyyy""", _
PreserveFormatting:=False)
rf.End = fld.Result.End
rf.InsertAfter Text:=vbTab & "Page "
rf.Collapse Direction:=wdCollapseEnd
ActiveDocument.Fields.Add Range:=rf, Type:=wdFieldPage,
PreserveFormatting:=False
rf.End = fld.Result.End
rf.InsertAfter Text:=" of "
rf.Collapse Direction:=wdCollapseEnd
ActiveDocument.Fields.Add Range:=rf, Type:=wdFieldNumPages,
PreserveFormatting:=False
rf.End = fld.Result.End
rf.InsertParagraphAfter
rf.Collapse Direction:=wdCollapseEnd
ActiveDocument.Fields.Add Range:=rf, Type:=wdFieldFileName, Text:="\p",
PreserveFormatting:=False
End Sub
```

I found this code to add text to every footer. The StoryType property has a wd constant, the values in the macro are

8 = wdEvenPagesFooterStory
9 = wdPrimaryFooterStor
11 = wdFirstPageFooterStory

```
Sub InsertTextAfterCurrentTextInFooters()
Dim myRng As Range
For Each myRng In ActiveDocument.StoryRanges
 Select Case myRng.StoryType
   Case Is = 8, 9, 11
     Do
       myRng.Collapse Direction:=wdCollapseEnd
       myRng.InsertAfter vbCr & "Your text here"
       Set myRng = myRng.NextStoryRange
     Loop Until myRng Is Nothing
   Case Else
     'Do nothing
 End Select
Next
End Sub
```

Portrait and Landscape

Macros are useful for changing the orientation of document text. In addition the margins and styles of the page headers and footers must be changed accordingly.

```
Sub MakeSectionLandscape()
' First, set orientation and margins
' Second, correct headers and footers: link to previous, styles, page numbering
Dim cntSections, thisSection, nextSection
cntSections = ActiveDocument.Sections.Count
thisSection = Selection.Information(wdActiveEndSectionNumber)
Landscape
'x = MsgBox("Count of sections = " + CStr(cntSections), vbOKOnly)
'y = MsgBox("Number of this section = " + CStr(thisSection), vbOKOnly)
If thisSection < cntSections Then
 nextSection = thisSection + 1
ActiveDocument.Sections(nextSection).Headers(wdHeaderFooterPrimary).LinkToPrevi
ous = False
ActiveDocument.Sections(nextSection).Footers(wdHeaderFooterPrimary).LinkToPrevi
ous = False
End If
With ActiveDocument.Sections(thisSection).Headers(wdHeaderFooterPrimary)  '
this section
 .LinkToPrevious = False
 .Range.Style = "HeaderLandscape"
End With
With ActiveDocument.Sections(thisSection).Footers(wdHeaderFooterPrimary)
 .LinkToPrevious = False
 .PageNumbers.RestartNumberingAtSection = False
 .Range.Style = "FooterLandscape"
End With
End Sub


Sub MakeSectionPortrait()
' First, set orientation and margins
' Second, correct headers and footers: link to previous, styles, page numbering
Dim cntSections, thisSection, nextSection
cntSections = ActiveDocument.Sections.Count
thisSection = Selection.Information(wdActiveEndSectionNumber)
Portrait
'x = MsgBox("Count of sections = " + CStr(cntSections), vbOKOnly)
'y = MsgBox("Number of this section = " + CStr(thisSection), vbOKOnly)
If thisSection < cntSections Then
 nextSection = thisSection + 1
ActiveDocument.Sections(nextSection).Headers(wdHeaderFooterPrimary).LinkToPrevi
ous = False
ActiveDocument.Sections(nextSection).Footers(wdHeaderFooterPrimary).LinkToPrevi
ous = False
End If
With ActiveDocument.Sections(thisSection).Headers(wdHeaderFooterPrimary)  '
this section
 .LinkToPrevious = False
 .Range.Style = "Header"
End With
With ActiveDocument.Sections(thisSection).Footers(wdHeaderFooterPrimary)
 .LinkToPrevious = False
```

```
 .PageNumbers.RestartNumberingAtSection = False
 .Range.Style = "Footer"
End With
End Sub


Sub Portrait()
' sets page layout to Portrait
'
 With Selection.PageSetup
     .Orientation = wdOrientPortrait
     .TopMargin = InchesToPoints(1.1)
     .BottomMargin = InchesToPoints(1.1)
     .LeftMargin = InchesToPoints(1.25)
     .RightMargin = InchesToPoints(1.25)
     .HeaderDistance = InchesToPoints(0.4)
     .FooterDistance = InchesToPoints(0.3)
 End With
End Sub


Sub Landscape()
With Selection.PageSetup
     .Orientation = wdOrientLandscape
     .TopMargin = InchesToPoints(1.1)
     .BottomMargin = InchesToPoints(1.1)
     .LeftMargin = InchesToPoints(0.5)
     .RightMargin = InchesToPoints(0.5)
     .HeaderDistance = InchesToPoints(0.4)
     .FooterDistance = InchesToPoints(0.3)
 End With
End Sub


Sub MakeHeaderLandscape()
 If ActiveWindow.View.SplitSpecial <> wdPaneNone Then
     ActiveWindow.Panes(2).Close
 End If
 If ActiveWindow.ActivePane.View.Type = wdNormalView Or ActiveWindow. _
     ActivePane.View.Type = wdOutlineView Then
     ActiveWindow.ActivePane.View.Type = wdPrintView
 End If
 ActiveWindow.ActivePane.View.SeekView = wdSeekCurrentPageHeader
 Selection.WholeStory
 Selection.Style = ActiveDocument.Styles("HeaderLandscape")
 Selection.EscapeKey
 Selection.MoveDown Unit:=wdLine, Count:=1
 Selection.WholeStory
 Selection.Style = ActiveDocument.Styles("FooterLandscape")
 Selection.EscapeKey
 ActiveWindow.ActivePane.View.SeekView = wdSeekMainDocument
End Sub
```

Document Properties

A Document object can have one or more DocumentProperty objects. There are two kinds of document properties: built-in and custom. Each kind has its own collection of DocumentProperty objects. The

basic elements of a document property are its name and value; the DocumentProperty object has corresponding properties of the same name.

You can return a collection of document properties. You can return a specific document property. And you can add, change, and delete members of the collection.

- Use the BuiltInDocumentProperties property to return the collection of built-in document properties.

- Use the CustomDocumentProperties property to return the collection of custom document properties.

- Use a constant to refer to a specific document property:

```
Dim intWords As Integer
intWords = ActiveDocument.BuiltInDocumentProperties(wdPropertyWords)
```

- Set the value of a specific document property:

```
ActiveDocument.BuiltInDocumentProperties("Title").Value = "CMS Operations
Guide"
```

- Use a literal to refer to a specific document property. But be careful because if the document property does not exist, an error will occur.

```
x = ActiveDocument.CustomDocumentProperties("VersionNum").Value
x = Documents("Sales.doc").CustomDocumentProperties("VersionNum").Value
```

- Use the Add method to add a property to a collection.

```
Documents("Sales.doc").CustomDocumentProperties.Add _
    Name:="YourName", LinkToContent:=False, Value:=thename, _
    Type:=msoPropertyTypeString
```

Page Numbering

Page numbering is usually done with field codes: Page, SectionPages, and NumPages. The field codes have optional switches for the style of the number (e.g., roman or arabic).

PageNumbers collection. Use PageNumbers(index), where index is the index number, to return a single PageNumber object. In most cases, a header or footer contains only one page number, which is index number 1.

```
PageNumbers(1).Alignment = wdAlignPageNumberCenter
```

PageNumbers properties: | Application Property | ChapterPageSeparator Property | Count Property | Creator Property | DoubleQuote Property | HeadingLevelForChapter Property | IncludeChapterNumber Property | NumberStyle Property | Parent Property | RestartNumberingAtSection Property | ShowFirstPageNumber Property | StartingNumber Property. Note that formatting properties apply to all objects in the collection.

PageNumbers methods: | Add Method | Item Method

PageNumbers parent Objects: | HeaderFooter

Determine the page number at the current cursor position

```
Selection.Information (wdActiveEndPageNumber)
```

Determine the number of pages in a document

```
Selection.Information(NumberOfPagesInDocument)
ActiveDocument.BuiltInDocumentProperties("Number of Pages")
ActiveDocument.Content.ComputeStatistics(wdStatisticPages)
```

The third method is the most reliable, but the slowest.

Lists

`ActiveDocument.Lists`	returns the Lists collection
`For Each li In ActiveDocument.Lists`	returns an individual list; lists are in reverse order, from the end of the document forward
`List.ListParagraph`	returns all the paragraphs in the list
`List.SingleListTemplate`	returns True/False if the entire list uses the same list template
`List.StyleName`	returns the name of the style
`List.CanContinuePreviousList`	returns a WdContinue constant that indicates whether formatting from previous list can be continued: wdContinueDisabled wdContinueList wdResetList

Working with Tables

The hierarchy of relevant objects. Collection names are followed by en empty pair of parentheses:

Tables()
Table
Rows()
Row
Cells()
Cell
Columns()
Column
Cells()
Cell
Borders()
Border
Shading
Range
Cells()
Cell

How many tables are there in the document:

```
ActiveDocument.Tables.Count
```

How to tell if the cursor is in a table:

```
If Selection.Information(wdWithInTable) = True.  . .
```

Select the table in which the cursor is located:

```
Selection.Tables(1)
```

Select the first table.

```
ActiveDocument.Tables(1).Range.Select
```

or

```
ActiveDocument.Tables(1).Select
```

Deselect the table while leaving cursor at the beginning of it:

```
Selection.MoveLeft Unit:=wdCharacter, Count:=1
Selection.MoveRight Unit:=wdCharacter, Count:=1
```

Create a table at the current cursor position. The Range object is required, it is the range where you want the table to appear. The table replaces the range, if the range isn't collapsed.

```
ActiveDocument.Tables.Add  Range:=Selection.Range, NumRows:=1, NumColumns:=4,
DefaultTableBehavior:=wdWord9TableBehavior
```

Create a table at the beginning of the document:

```
Set myRange = ActiveDocument.Range(Start:=0, End:=0)
ActiveDocument.Tables.Add Range:=myRange, NumRows:=3, NumColumns:=4
```

Move to the next cell on the right:

```
Selection.MoveRight Unit:=wdCell
```

Select the last row in a table:

```
ActiveDocument.Tables(1).Rows.Last.Range.Select
```

Insert one row when the cursor is in the table:

```
Selection.InsertRows 1
```

Insert one row regardless of where cursor is, add a row after the first row:

```
ActiveDocument.Tables(1).Rows.Add  BeforeRow:=.Rows(1)
```

Insert a row at the bottom of a table:

```
ActiveDocument.Tables(1).Rows.Add
```

Make the first row in the selected table a repeating heading row:

```
Selection.Tables(1).Rows(1).HeadingFormat = True
```

Is the second row in the selected table a repeating heading row:

```
If Selection.Tables(1).Rows(2).HeadingFormat = True
Then . . .
```

Change the font formats within the selected table:

```
Selection.Tables(1).Range.Font.Size = 9
```

Change the font formats within a row:

```
Selection.Tables(1).Rows(1).Range.Font.Italic = True
```

Set a paragraph format within a row:

```
Selection.Tables(1).Rows(1).Range.ParagraphFormat.KeepWithNext = True
```

Generally formatting all table rows is done like:

```
With Selection.Tables(1)
    .AllowAutoFit = False
    .TopPadding = InchesToPoints(0.03)
    .BottomPadding = InchesToPoints(0.03)
    .LeftPadding = InchesToPoints(0.08)
    .RightPadding = InchesToPoints(0.08)
    With .Borders(wdBorderLeft)
        .LineStyle = wdLineStyleDouble
        .LineWidth = wdLineWidth050pt
        .Color = wdColorGray25
    End With
End With
```

Select the table that is the subject of a variable (set via the object model):

```
Dim i As Long
Dim docTables As Tables
Set docTables = ActiveDocument.Tables
With docTables
    For i = 1 To .Count
    With docTables(i)
        . . .
        docTables(i).Select
        . . .
    End With
    Next
End With
```

Iterate through all tables in the active document:

```
Dim i as Long
With ActiveDocument.Tables
    For i = 1 To .Count
    . . .
    Next
End With
```

Delete a single row:

```
ActiveDocument.Tables(1).Rows(1).Delete
```

Caution when deleting a group of rows: if you use a For loop to delete rows 3 through 7 of a 7-row table, when you delete row 3, the next row becomes 3; your code will fail trying to access a row that is no longer in the table (4). In this case use a reverse loop, starting with the last row and deleting the previous ones.

Delete the contents of a row, leaving the row empty:

```
ActiveDocument.Tables(1).Rows(2).Select
Selection.Delete
```

Delete all but the first row (this example is a backwards loop):

```
For i = ActiveDocument.Tables(1).Rows.Count To 2 Step -1
    ActiveDocument.Tables(1).Rows(i).Delete
Next
```

Sort a table:

```
ActiveDocument.Tables(1).Sort ExcludeHeader:=True, FieldNuumber:=1,
SortFieldType:=wdSortFieldAlphanumeric, SortOrder:=wdSortOrderAscending
```

Up to two more columns may be specified for the sort as FieldNumber2 and FieldNumber3. ExcludeHeader is by default false. SortFieldType is by default alphanumeric. SortOrder is by default ascending. The default "field" is the first column. Hence this next example is the same as the first example:

```
ActiveDocument.Tables(1).Sort ExcludeHeader:=True
```

Count rows and columns., This makes a handy tool to add to a toolbar or menu.

```
Dim rCnt1, rCnt2, x
rCnt1 = Selection.Tables(1).Rows.Count
rCnt2 = Selection.Tables(1).Columns.Count
x = MsgBox("Row count = " + CStr(rCnt1) + ", Column count = " + CStr(rCnt2),
vbOKOnly)
```

Addressing Table Cells

Use **Cell(row, column)**, where row is the row number and column is the column number, or **Cells(index)**, where index is the index number, to return a Cell object. The following example applies shading to the second cell in the first row.

```
Set myCell = ActiveDocument.Tables(1).Cell(Row:=1, Column:=2)
myCell.Shading.Texture = wdTexture20Percent
```

Word adds a end-of-cell mark, Chr(07), at the end of each cell.

The contents of a cell can be accessed with the Range object: Note, this also returns the end-of-cell marker.

```
Set myRange = ActiveDocument.Tables(1).Cell(1,2).Range
```

this may not work, you may have to do more:

```
Set myTable = ActiveDocument.Tables(1)
Set myRange = ActiveDocument.Range(myTable.Cell(1, 1).Range.Start,
myTable.Cell(1, 2).Range.End)
```

Or:

```
ActiveDocument.Tables(1).Cell(1,2).Range.Text = "wow"
```

Return cell contents without end-of-cell marker (not sure this is necessary):

```
Set myRange = ActiveDocument.Tables(1).Cell(1, 2).Range
myRange.MoveEnd Unit:=wdCharacter, Count:=-1
cellText = myRange.Text
```

Set value of a cell:

```
ActiveDocument.Tables(1).Cell(1,1).Range.Text = "this text"
```

Delete the contents of a cell:

```
ActiveDocument.Tables(1).Cell(1,1).Select
Selection.Delete
```

Delete a cell (why would you ever do this?):

```
ActiveDocument.Tables(1).Cell(1,1).Delete
```

Return a numbered cell in the last row of a table:

```
ActiveDocument.Tables(1).Rows.Last.Cells(1)
ActiveDocument.Tables(1).Rows.Last.Cells(2)    )
```

DIALOG BOXES

Message Box Object

The basic syntax is

```
MsgBox(prompt[, buttons] [, title] [, helpfile, context])
```

The button constants are:
The first group of values describes the number and type of buttons displayed in the dialog box; the second group describes the icon style; the third group determines which button is the default; and the fourth group determines the modality of the message box. Buttons can be combined—no more than one button from each group—by adding their numbers You can add numbers:

vbYesNo + vbCritical

vbOKOnly	Display OK button only.
vbOKCancel	Display OK and Cancel buttons.
vbAbortRetryIgnore	Display Abort, Retry, and Ignore buttons.
vbYesNoCancel	Display Yes, No, and Cancel buttons.
vbYesNo	Display Yes and No buttons.
vbRetryCancel	Display Retry and Cancel buttons.

vbCritical	Display Critical Message icon.
vbQuestion	Display Warning Query icon.
vbExclamation	Display Warning Message icon.
vbInformation	Display Information Message icon.

vbDefaultButton1	First button is default.
vbDefaultButton2	Second button is default.
vbDefaultButton3	Third button is default.
vbDefaultButton4	Fourth button is default.

vbApplicationModal	Application modal; the user must respond to the message box before continuing work in the current application.
vbSystemModal	System modal; all applications are suspended until the user responds to the message box.
vbMsgBoxHelpButton	Adds Help button to the message box
VbMsgBoxSetForeground	Specifies the message box window as the foreground window
vbMsgBoxRight	Text is right aligned

MsgBox can return an integer corresponding to the button selected:

vbOK	OK
vbCancel	Cancel
vbAbort	Abort
vbRetry	Retry
vbIgnore	Ignore
vbYes	Yes

vbNo No

Present Information

```
Dim msgTitle as String, msgText as String
msgTitle = "the process title"
msgText = "Count of sections = " + CStr(cntSections)
MsgBox msgText, vbOKOnly, msgTitle
```

Prompt User for Choices

You can use MsgBox to prompt the user for simple choices by directing the user to press a button that corresponds with a certain condition.

```
msgTitle = "Aggregate Document Files for NPI Business Requirements"
msgText = "Where are the the document files located? " + vbCrLf
msgText = msgText + "Select [Yes] for " & HomeDirectory + vbCrLf
msgText = msgText + "Select [No] for " & WorkDirectory
c = MsgBox(msgText, vbYesNoCancel, msgTitle)
On Error Resume Next
Select Case c
Case vbYes
    ChangeFileOpenDirectory (HomeDirectory)
Case vbNo
    ChangeFileOpenDirectory (WorkDirectory)
Case Else
    Exit Sub
End Select
If Err.Number = 4172 Then
    msgText = "That directory does not exist. Stopping."
    MsgBox msgText, vbOKOnly, msgTitle
    Exit Sub
End If

On Error GoTo ErrorHandler
```

Dialog Object

The object model has a Dialogs collection that represents all the built-in dialog boxes in Word. The collection has two parents: Application and Global.

Use Dialogs(index), where index is a wdWordDialog constant that identifies the dialog box, to return a single Dialog object. Dialog boxes you are likely to use include:

wdDialogEditFind
wdDialogFileFind
wdDialogFileOpen
wdDialogFilePrint
wdDialogFileSaveAs

The Dialog object has methods:

- Display: Displays the specified built-in Word dialog box until either the user closes it or the specified amount of time has passed.
- Execute: Applies the current settings of a Microsoft Word dialog box.
- Show: Displays and carries out actions initiated in the specified built-in Word dialog box.

- Update: Updates the values shown in a built-in Microsoft Word dialog box.

The Show method of the Dialog object displays and executes any action taken in a built-in Word dialog box. The return value (Long) indicates which button was clicked to close the dialog box:

Return value	*Description*
-2	The Close button.
-1	The OK button.
0 (zero)	The Cancel button.
> 0 (zero)	A command button: 1 is the first button, 2 is the second button, and so on.

Open a particular dialog box:

```
Dialogs(wdDialogFileOpen).Show
```

This example displays the built-in Find dialog box, with "Hello" in the Find What box.

```
Dim dlgFind As Dialog
Set dlgFind = Dialogs(wdDialogEditFind)
With dlgFind
    .Find = "Hello"
    .Show
End With
```

This example displays the built-in Open dialog box showing all file types.

```
With Dialogs(wdDialogFileOpen)
    .Name = "*.*"
    .Show
End With
```

This example prints the active document, using the settings from the Print dialog box.

```
Dialogs(wdDialogFilePrint).Execute
```

Prompt User for File(s) or Directory with FileFind Dialog – OBSOLETE

This used to work, but the better approach with Word 2003 is to use the FileDialog object.

```
Dim pn, r, t, m
r = MsgBox("Select directory from the following window, then [Open].",
vbOKCancel, t)
If r = vbCancel Then End
With Dialogs(wdDialogFileFind)
    .Display
    .Update
    pn = .SearchPath
End With
m = "Confirm directory:  " + pn
r = MsgBox(m, vbOKCancel, t)
If r = vbCancel Then End
```

FileDialog Object

New with Office 2003 is the FileDialog object. It provides file dialog box functionality similar to the functionality of the standard Open and Save dialog boxes found in Microsoft Office applications.

The object can to used in four ways, determined by a single parameter, DialogType:

Constant	*Action*
msoFileDialogOpen	lets users select one or more files that you can then open in the host application using the Execute method
msoFileDialogSaveAs	lets users select a single file that you can then save the current file as using the Execute method
msoFileDialogFilePicker	lets users select one or more files; the file paths that the user selects are captured in the FileDialogSelectedItems collection
msoFileDialogFolderPicker	lets users select a path; the path that the user selects is captured in the FileDialogSelectedItems collection

It has two methods:

- Execute: carries out a user's action right after the Show method is invoked.
- Show: Displays a file dialog box and returns a Long indicating whether the user pressed the action button (-1) or the cancel button (0). When you call the Show method, no more code will execute until the user dismisses the file dialog box. In the case of Open and SaveAs dialog boxes, use the Execute method right after the Show method to carry out the user's action.

Has properties:

- AllowMultiSelect: if the user is allowed to select multiple files from a file dialog box. Has no effect on Folder Picker dialog boxes or SaveAs dialog boxes because users should never be able to select multiple files in these types of file dialog boxes.
- InitialFileName: Set or returns a String representing the path and/or file name that is initially displayed in a file dialog box.
- Title: Sets or returns the title of a file dialog box displayed using the FileDialog object.

Display the Open dialog box and limit the user to select only one item:

```
Dim dlgOpen As FileDialog
Set dlgOpen = Application.FileDialog( FileDialogType:=msoFileDialogOpen)
With dlgOpen
    .AllowMultiSelect = False
    .Show
End With
```

The FileDialog object has two sub objects.

- FileDialogSelectedItems
- FileDialogFilter

FileDialogFilter object represents a file filter in a file dialog box displayed through the FileDialog object. Each file filter determines which files are displayed in the file dialog box. It has a collection FileDialogFilters.

Access the collection:

```
Application.FileDialog(msoFileDialogOpen).Filters
```

Clear the default filters:

```
Application.FileDialog(msoFileDialogFilePicker).Filters.Clear
```

Add a filter that includes all files:

```
Application.FileDialog(msoFileDialogFilePicker).Filters.Add "All files", "*.*"
```

Add a filter that includes GIF and JPEG images and make it the first item in the list:

```
Application.FileDialog(msoFileDialogFilePicker).Filters.Add "Images", "*.gif;
*.jpg; *.jpeg", 1
```

Prompt User to Select Folder with FileDialog FolderPicker

This code can be used in a number of places.

```
dim strDir As String
strDir = PromptForDirectory
If strDir = "" Then Exit Sub
...

Function PromptForDirectory()
‘ returns empty string if no directory selected
Dim strDir As String
Dim r As Long
Dim dlgOpen As FileDialog
Set dlgOpen = Application.FileDialog(FileDialogType:=msoFileDialogFolderPicker)
repeat:
With dlgOpen
    .Title = "Select Directory"
    .InitialFileName = "C:\"
    .AllowMultiSelect = False
    If .Show = 0 Then
        r = MsgBox("Do you want to cancel this macro?", vbYesNo, "Select
Directory")
        If r = vbYes Then
            Exit Function
        Else
            GoTo repeat
        End If
    End If
    strDir = .SelectedItems(1)
End With
r = MsgBox("You selected " & strDir, vbYesNo, "Select Directory")
If r = vbNo Then GoTo repeat
PromptForDirectory = strDir
End Function
```

DOCUMENTING YOUR SHORTCUT KEYS

This can be done with VBA.

About Keyboard Shortcut Keys

Shortcut keys—combinations of keys—can be assigned to menu items, toolbar items, macros, Word commands, styles, fonts, and common symbols. Menu and toolbar items have Alt shortcut keys which are identified by underlining. For example, the File menu is presented in the menu bar as File where the underlining indicates that [Alt+F] is the shortcut key.

There are two ways of creating custom shortcut keys:

- For menu and toolbar items, open the Customize dialog box, select the desired menu or toolbar item, then click [Modify Selection] to open a context menu. In the Name text box, type the ampersand character (&) to the left of the letter you want to use for an Alt shortcut key.
- For all objects, open the Customize Keyboard dialog box, select the desired object as an item in a category, enter the desired shortcut key combination, and save it. If you assign an Alt shortcut key that is already in use (as visible on the menu or a toolbar), your new assignment has precedence (until you delete it). So use caution.

Run This Code

1. Open an empty Word document.
2. Run the code:

```
CustomizationContext = NormalTemplate
For Each aKey In KeyBindings
    Selection.InsertAfter aKey.Command & vbTab _
        & aKey.KeyString & vbCr
    Selection.Collapse Direction:=wdCollapseEnd
Next aKey
```

3. Select all text in the Word document and convert text to a table separating text into columns with tabs.
4. Reformat the table to suit yourself.

Or run:

```
Sub DocumentKeys()
Documents.Add DocumentType:=wdNewBlankDocument
CustomizationContext = NormalTemplate
For Each aKey In KeyBindings
    Selection.InsertAfter aKey.Command & vbTab _
        & aKey.KeyString & vbCr
    Selection.Collapse Direction:=wdCollapseEnd
Next aKey
MsgBox "Count of custom shortcut keys = " & KeyBindings.Count
Selection.WholeStory
Selection.ConvertToTable Separator:=wdSeparateByTabs, NumColumns:=2, _
    AutoFitBehavior:=wdAutoFitContent
With Selection.Tables(1)
    .AllowPageBreaks = False
    .AllowAutoFit = True
    .Style = "Table Grid"
    .ApplyStyleHeadingRows = True
    .ApplyStyleLastRow = True
    .ApplyStyleFirstColumn = True
    .ApplyStyleLastColumn = True
```

```
End With
Selection.MoveRight Unit:=wdCharacter, Count:=1
End Sub
```

You can differentiate between types of commands easily. They are typically either style names, Word command names, or macro names. The latter have names like Normal.General.Keep; Normal is the name of the template file, General is the name of the module, and Keep is the name of the macro.

DOM Background

The relevant DOM collection is **KeyBindings**. This is a collection of KeyBinding objects that represent the custom key assignments in the current context. (Custom key assignments are made in the Customize Keyboard dialog box.) The collection excludes Alt keys assigned to menu and toolbar items. The collection is returned by property KeyBindings, a property of the Application and Global objects.

Property Context: Returns an object that represents the storage location of the specified key binding. Note that built-in key assignments (for example, CTRL+I for Italic) return the Application object as the context. Any key bindings you add will return a Document or Template object, depending on the customization context in effect when the KeyBinding object was added.

Methods:
Item: returns a single item; expression.Item(Index)
Key: Returns a KeyBinding object that represents the specified custom key combination. If the key combination doesn't exist, this method returns Nothing.

```
MsgBox KeyBindings(1).Command
```

Property CustomizationContext: Returns or sets a Template or Document object that represents the template or document in which changes to menu bars, toolbars, and key bindings are stored. Corresponds to the value of the Save in box on the Commands tab in the Customize dialog box. Applies to the Application object and to the Global object.

Examples:

```
CustomizationContext = NormalTemplate
CustomizationContext = ActiveDocument.AttachedTemplate
```

Object KeyBinding: Represents a custom key assignment in the current context.
Application Property
Command Property: the command assigned to the key combination
CommandParameter Property: the command parameter
Context Property: the storage location (example: normal.dot)
Creator Property
KeyCategory Property: WdKeyCategory constants:
- wdKeyCategoryAutoText
- wdKeyCategoryCommand
- wdKeyCategoryDisable
- wdKeyCategoryFont
- wdKeyCategoryMacro
- wdKeyCategoryNil
- wdKeyCategoryPrefix

wdKeyCategoryStyle
wdKeyCategorySymbol
KeyCode Property: unique number for first key in the combination
KeyCode2 Property: unique number for second key in the combination
KeyString Property: the key combination string for the specified keys (for example, CTRL+SHIFT+A)
Parent Property
Protected Property

FIELD CODES IN VBA

The Fields collection belongs to three objects: Document, Range, and Selection.

There is a Fields collection and a Field object. The Fields collection is a child of Range and Selection. The Field object has properties:

- Code property returns a Range object that contains all the text enclosed by the { } including leading and trailing spaces.
- Result property returns a Range object that represents the field's result.
- Type property returns an expression that is a value of the WdfieldType constant. Sample values: wdFieldNumPages, wdFieldPage, wdFieldSaveDate, wdFieldSectionPages, wdFieldSequence, wdFieldIndex, wdFieldIndexEntry, wdFieldTOC.
- Kind property identifies the field as a constant value:

wdFieldKindCold	has no result
wdFieldKindHot	result is automatically updated
wdFieldKindNone	invalid
wdFieldKindWarm	result auto updated when source changes or can be manually updated

Field methods:
- Select
- Copy
- Cut: removes the field and puts it on the Clipboard.
- Delete
- Add
- Update: updates the field's result.

A Range object's Text property returns the text in the range. Can be useful for extracting the text in a field's result.

The Add method: Adds a Field object to the Fields collection. Returns the Field object at the specified range.

expression.Add(*Range*, *Type*, *Text*, *PreserveFormatting*)
expression Required. An expression that returns a Fields object.
Range Required Range object. The range where you want to add the field. If the range isn't collapsed, the field replaces the range.
Type Optional Variant. Can be any WdFieldType constant. The default value is wdFieldEmpty.
Text Optional Variant. Additional text needed for the field. For example, if you want to specify a switch for the field, you would add it here.

PreserveFormatting Optional Variant. True to have the formatting that's applied to the field preserved during updates.

```
Selection.Collapse Direction:=wdCollapseEnd
ActiveDocument.Fields.Add Range:=Selection.Range, _
    Type:=wdFieldListNum, Text:="\s 3"
```

To remove all XE field codes:

```
Dim fld As Field
For each fld in ActiveDocument.Fields
    If fld.Type = wdFieldIndexEntry Then fld.Delete
Next
```

If you iterate through field codes at the Document level, you only access those in the Main Text story. To iterate through all field codes in the document, you have to iterate through the StoryRanges at the top level, then for each StoryRange you can iterate through its Fields collection.

You can access fields in page headers/footers with the HeadersFooters collection.

How to access a particular field code? How to learn if a document has a particular field code?

- Access the nth field code: ActiveDocument.Fields(n)
- x = ActiveDocument.StoryRanges(wdPrimaryFooterStory).Fields.Count

Field methods:

- Unlink: replaces the field with its most recent result (converts it to plain text)

When a document has a table of contents created by the TOC field code, an iteration of the Fields collection will return one field for the TOC itself (13) and two fields for each TOC entry (37 FieldPageRef and 88 Hyperlink).

```
Dim sty as Range
Dim fc As Field
For Each sty in MyDoc.StoryRanges
If sty.Fields.Count <> 0  Then
For Each fc In sty.Fields
    MsgBox "
    Select Case fc.Type
    Case wdFieldSaveDate
        fc.Unlink
    Case wdFieldDate
        fc.Delete
    Case wdFieldPrintDate
        fc.Delete
    End Select
Next fc
End If
Next sty
```

Index Object

The Index object is a member of the collection Indexes. It has properties:

- Type: constant wdIndexType has values wdIndexRunin, wdIndexIndent.

Methods:
- Update: updates the values.

Table of Contents

The object is TableOfContents. It belongs to the collection TablesOfContents, which belongs to the object Document.

Methods:
- UpdatePageNumbers: updates only the page numbers.
- Update: updates the entries ("update entire table").

```
ActiveDocument.TablesOfContents(1).Update
```

The following code updates the page numbers in TOCs other than the first one. This is useful in a multi-chapter document with a master TOC and individual chapter TOCs.

```
Dim  t as TableOfContents
For Each t in ActiveDocument.TablesOfContents
    If not t is ActiveDocument.TablesOfContents(1) Then
        t.UpdatePageNumbers
    End If
Next
```

or

```
Dim  t as TableOfContents
For Each t in ActiveDocument.TablesOfContents
    If t is ActiveDocument.TablesOfContents(1) Then
        t.Update
    Else
        t.UpdatePageNumbers
    End If
Next
```

RD Field Code

URL strings always encode space characters to prevent the possibility of being misunderstood. URL encoding of a character consists of a "%" symbol, followed by the two-digit hexadecimal representation (case-insensitive) of the ISO-Latin code point for the character. As hex 20 is the representation of a space, "%20" is used in lieu of the space character in URL-encoded strings.

When you copy a filename from a website into an RD field, it will be encoded. And while you can create an RD field with spaces in the filename, the filename becomes encoded after the document with the RD field is saved. For instance,
"RD "Using%20SharePoint.doc" \f"

The encoded value may be useful when the file in question resides on a web server, but not on a file server.

Use a simple function like the following example to replace "%20" with a space.

```
Private Function URLDecode(URLtoDecode As String) As String
    URLDecode = Replace(URLtoDecode, "%20", " ")
End Function
```

Field Codes and Page Numbers

If you needed to change a switch in a field code, you would have to edit the Result property. If you needed to change the numbering style, you might do some string manipulation

NUMPAGES * roman
NUMPAGES * arabic

```
Replace(Field.Code, “arabic”, “roman”)
```

Or you could replace one field with another. In the following example, the cursor is immediately to the left of a Page field:

```
Selection.MoveRight Unit:=wdCharacter, Count:=1, Extend:=wdExtend
Selection.Fields.Add Range:=Selection.Range, Type:=wdFieldEmpty, Text:= _
    "PAGE  \* roman ", PreserveFormatting:=True
```

Why would you need to do this when the HeaderFooter.PageNumbers property is supposed to return a PageNumbers collection that represents all the page number fields included in the specified header or footer? Because when I changed the PageNumbers.NumberStyle property, the NumPages field did not change, only the Page field.

How would you find the page number field codes?

PAGE HEADERS AND FOOTERS

HeadersFooters is the main collection, has HeaderFooter objects. A collection belongs to each section, e.g., ActiveDocument.Sections(n).

Each section can have several HeaderFooter objects represented by the following WdHeaderFooterIndex constants: wdHeaderFooterEvenPages, wdHeaderFooterFirstPage, and wdHeaderFooterPrimary (returns an odd-numbered header/footer when there are different odd and even ones).

Headers property returns a HeadersFooters collection that represents the headers for the specified section

```
With ActiveDocument.Sections(1).Headers(wdHeaderFooterFirstPage)
    .Range.InsertAfter("First Page Text")
    .Range.Paragraphs.Alignment = wdAlignParagraphRight
End With
```

Footers property returns a HeadersFooters collection that represents the footers for the specified section

```
With ActiveDocument.Sections(1)
    .Headers(wdHeaderFooterPrimary).Range.Text = "Header text"
    .Footers(wdHeaderFooterPrimary).Range.Text = "Footer text"
End With
```

The Exist property indicates if the specified type of header/footer exists.

```
If secTemp.Headers(wdHeaderFooterFirstPage).Exists = True Then . . .
```

The InsertAfter method inserts the specified text at the end of a Range object or Selection object. After this method is applied, the range or selection expands to include the new text. If you use this method with a range or selection that refers to an entire paragraph, the text is inserted after the ending paragraph mark (the text will appear at the beginning of the next paragraph). To insert text at the end of a paragraph, determine the ending point and subtract 1 from this location (the paragraph mark is one character), as shown in the following example.

```
Set doc = ActiveDocument
Set rngRange = _
    doc.Range(doc.Paragraphs(1).Start, _
    doc.Paragraphs(1).End - 1)
rngRange.InsertAfter _
    " This is now the last sentence in paragraph one."
```

HeaderFooter objects can have a Range sub-object but not a Selection sub-object. The Range object is established with the Range property:

```
ActiveDocument.Section(1).Headers(wdHeaderFooterPrimary).Range
```

Text can be inserted at the end of a header with the InsertAfter property on a Range object.

```
ActiveDocument.Section(1).Headers(wdHeaderFooterPrimary).Range.InsertAfter
“Draft”
ActiveDocument.Section(1).Headers(wdHeaderFooterPrimary).Range.InsertAfter
vbCrLf + “Draft”
```

To select the inserted paragraph, use the Paragraphs collection:

```
ActiveDocument.Section(1).Headers(wdHeaderFooterPrimary).Range.Paragraphs(Last)
```

To apply a style to a Range object:

```
Selection.Range.Style = "Bolded"
ActiveDocument.Section(1).Headers(wdHeaderFooterPrimary).Range.Paragraphs.Last.
Style = "Watermark"
Selection.Style = ActiveDocument.Styles("Watermark")
```

To do something to every header, do it for each header in each section (but perhaps not to headers linked to previous ones, see below):

```
Sub PutWatermarkTextInAllHeaders()
' inserts text at end of each header
WorkDirectory = "C:\Data\NPI Requirements\"
ChangeFileOpenDirectory (WorkDirectory)
Documents.Open FileName:="All Business Requirements.doc"
Dim sec As Section
Dim cnt As Integer
Dim s As Integer
cnt = ActiveDocument.Sections.Count
'For Each sec In ActiveDocument.Sections    1 skip first page
For s = 2 To cnt
    Set sec = ActiveDocument.Sections(s)
    sec.Headers(wdHeaderFooterPrimary).Range.InsertAfter vbCrLf + "Draft"
    sec.Headers(wdHeaderFooterPrimary).Range.Paragraphs.Last.Style =
ActiveDocument.Styles("Watermark")
Next
End Sub
```

Property LinkToPrevious returns True/False reflecting if current HeaderFooter object is linked to the previous one. Can be used to change the setting. If you put a two-column table inside a continuous

section break, you do not want to change that section's header. Continuous sections are linked to the previous heading. The following code works when only continuous sections are linked to the previous heading. (On the other hand there's no point in changing a header that is linked to a previous one.)

```
If sec.Headers(wdHeaderFooterPrimary).LinkToPrevious = False Then   ' ignore
section break continuous
    sec.Headers(wdHeaderFooterPrimary).Range.InsertAfter vbCrLf + "Draft"
    sec.Headers(wdHeaderFooterPrimary).Range.Paragraphs.Last.Style =
ActiveDocument.Styles("Watermark")
    End If
```

ASSEMBLING MULTI-FILE DOCUMENTS

Large and/or complex documents can be written such that each chapter is its own file and the entire document is assembled by concatenating the chapter files with VBA code and saved as a single file. The key issues for this assembly are:

- maintaining page headers and footers
- maintaining page margins
- maintaining page numbers and styles
- handling a mix of page orientations (landscape and portrait)

Approach

The approach I use these days is as follows:

a. Use a master document that contains the front matter (title page, revision history, TOC, list of figures) and the macro that concatenates the chapter files. The user opens this file and runs the macro. The macro does the rest.
b. Prompt user for directory in which to save the finished document.
c. Move cursor to the end of the document.
d. Insert each file. Before all but the first file insert a section break next page.
e. When an inserted file has a landscape orientation, before inserting it change the orientation of the open document and change the styles of the page header and footer text. After inserting the file and the section break next page, reset the orientation to portrait and change the styles of the page header and footer.
f. After the last file:
g. Update the table of contents.
h. For any chapter table of contents, update only its page numbers.
i. Save the file with a new name in the location from step b.
j. Remove all macros from the new file.
k. Leave document open on first page in print layout view.

Now you should check the new document for correctness. Check the table of contents for correct page numbering. Check the page headers and footers to be sure that the title page has none, that they match each page's orientation, and that they reflect the correct chapter name. If you find problems, you must make a note of them, close this file, correct the chapter files, and start over. When all is well, change the version number and revision date on the title page and **save the document with a different name.**

Some formatting details:

- The master document has portrait orientation.

- Each page is in its own section, i.e., there is a section break next page between each page, including after the last page.
- The title page has no header and footer.
- The other front matter pages share the same header and footer. The last page—where the first chapter file will be inserted—is not linked to the previous header and footer.
- Page numbering starts on the second page with "ii". Page numbering on the last page is reset to start with "1".
- The chapter files have the same size of header and footer so they will look seamless when concatenated.
- The chapter files have one header and footer. They do not have different header/footer for the first page, nor do they have different ones for odd and even pages. If your chapters have several headers and/or footers, you may have to tinker with the code here.

Code Samples

Prompt User for Directory of New File

```
Sub GetDir()
Dim strDir As String
With Application.FileDialog(msoFileDialogFolderPicker)
    .InitialFileName = "C:\"
    .Show
    strDir = .SelectedItems(1)
End With
MsgBox "You selected " & strDir
End Sub
```

Insert Chapter Files

The key method is InsertFile. Each file is inserted in order, preceded by a page break next section. Care must be taken to know where the cursor is. After InsertFile the cursor is at the end of the contents of the inserted file. Because this may change with different versions of Word, be sure to confirm this with a test. If the cursor does not move you will need code to move the cursor: Selection.EndKey Unit:=wdStory.

```
‘ use strChapDir only if documents are in a different directory
Dim strChapDir As String
strChapDir = "\\www.your-url.com\whee\more\"

Selection.EndKey Unit:=wdStory          ' go to end of doc
ActiveWindow.ActivePane.View.Type = wdPrintView

Selection.InsertFile FileName:=strChapDir & "Component Design.doc"
LinkFooterToPrevious
Selection.InsertBreak Type:=wdSectionBreakNextPage
```

When a chapter begins in landscape orientation, change the orientation in the new document to match.

```
Landscape                         ‘ a subroutine
Selection.InsertFile . . .         ‘ landscape file
```

When the previous chapter ended in landscape and the new one is portrait, change the orientation in the new document to match.

```
Selection.InsertBreak Type:=wdSectionBreakNextPage
Portait
Selection.InsertFile . . .        ‘ portrait file
```

Insert Section Break Odd Page

When you are compiling a document to be printed on both sides (duplex), you may want it to adhere to the book publishing verso-recto convention: the first page of the "book," of each section, and the first chapter of a section, is a recto page (right-handed). All recto pages will have odd numbers and all verso pages (left-handed) will have even numbers.

The way to make this happen is by setting Mirror Margins on and using section break next page (yes, there is a wdSectionBreakOddPage, but it does not work the way you want or expect.)

```
Dim r As Integer
r = MsgBox("Will document be printed duplex?", vbYesNo, "Build Operations
Guide")
If r = vbYes Then ActiveDocument.PageSetup.mirrormargins = True
```

Then when you

```
Selection.InsertBreak Type:=wdSectionBreakNextPage
```

it will insert a section break odd page at the correct point.

Update Main Table of Contents

```
ActiveDocument.TablesOfContents(1).Update
```

Update Main Table of Figures

```
ActiveDocument.TablesOfFigures(1).Update
```

Save New Document

```
ActiveDocument.SaveAs FileName:=strDir & "All One File.doc", FileFormat:=_
wdFormatDocument, AddToRecentFiles:=True
```

Update Chapter Tables of Contents

```
Sub UpdateChapterTOCs()
Dim t As TableOfContents
For Each t In ActiveDocument.TablesOfContents
    If Not t Is ActiveDocument.TablesOfContents(1) Then
        t.UpdatePageNumbers
    End If
Next
End Sub
```

Update Indices

```
Sub UpdateIndexes()
Dim it As Index
For Each i In ActiveDocument.Indexes
    i.Update
Next
End Sub
```

Delete Macros in New Document

In this code, the macros reside in a module named BuildDocuments.

```
Sub DeleteModuleInAll()
Application.OrganizerDelete Source:=ActiveDocument.Name, _
     Name:="BuildDocuments", Object:=wdOrganizerObjectProjectItems
End Sub
```

Subroutines

```
Sub LinkFooterToPrevious()
With Selection.Sections(1).Footers(wdHeaderFooterPrimary).PageNumbers
    .NumberStyle = wdPageNumberStylearabic
    .RestartNumberingAtSection = False
End With
With Selection.Sections(1)
    .Footers(wdHeaderFooterPrimary).LinkToPrevious = True
End With
End Sub
```

```
Sub Portrait()
' First, set orientation and margins
' Second, change styles

With Selection.PageSetup
    .Orientation = wdOrientPortrait
    .TopMargin = InchesToPoints(1.1)
    .BottomMargin = InchesToPoints(0.9)
    .LeftMargin = InchesToPoints(1.25)
    .RightMargin = InchesToPoints(1.25)
    .HeaderDistance = InchesToPoints(0.5)
    .FooterDistance = InchesToPoints(0.4)
End With
ActiveWindow.ActivePane.View.SeekView = wdSeekCurrentPageHeader
Selection.WholeStory
Selection.Style = ActiveDocument.Styles("Header")
Selection.EscapeKey
Selection.MoveDown Unit:=wdLine, Count:=1
Selection.WholeStory
Selection.Style = ActiveDocument.Styles("Footer")
Selection.EscapeKey
ActiveWindow.ActivePane.View.SeekView = wdSeekMainDocument
End Sub

Sub Landscape()
With Selection.PageSetup
     .Orientation = wdOrientLandscape
     .TopMargin = InchesToPoints(1.1)
     .BottomMargin = InchesToPoints(0.9)
     .LeftMargin = InchesToPoints(0.5)
     .RightMargin = InchesToPoints(0.5)
     .HeaderDistance = InchesToPoints(0.5)
     .FooterDistance = InchesToPoints(0.4)
End With
ActiveWindow.ActivePane.View.SeekView = wdSeekCurrentPageHeader
Selection.WholeStory
```

```
Selection.Style = ActiveDocument.Styles("HeaderLandscape")
Selection.EscapeKey
Selection.MoveDown Unit:=wdLine, Count:=1
Selection.WholeStory
Selection.Style = ActiveDocument.Styles("FooterLandscape")
Selection.EscapeKey
ActiveWindow.ActivePane.View.SeekView = wdSeekMainDocument
End Sub
```

Variations

```
Sub SetVersionNumber()
msgText = "Which version of the document is this?"
Dim v As String
v = InputBox(msgText, msgTitle, cv)
ActiveDocument.CustomDocumentProperties.Add _
    Name:="VersionNum", LinkToContent:=False, Value:=v, _
    Type:=msoPropertyTypeString
End Sub

Sub DraftWatermark()
' insert draft watermark
msgText = "Do you want the DRAFT watermark to appear in the document?"
c = MsgBox(msgText, vbYesNo, msgTitle)
If c = vbYes Then SetDraftWatermark
End Sub
```

SHAPES

The detail here is meant to support writing macros to manipulate shapes, hence the discussion of the object model.

Represents an object in the drawing layer, such as an AutoShape, freeform, OLE object, ActiveX control, or picture. The Shape object is a member of the Shapes collection, which includes all the shapes in the main story of a document or in all the headers and footers of a document.

A shape is usually attached to an anchoring range. You can position the shape anywhere on the page that contains the anchor.

There are three objects that represent shapes: the Shapes collection, which represents all the shapes on a document; the ShapeRange collection, which represents a specified subset of the shapes on a document (for example, a ShapeRange object could represent shapes one and four on the document, or it could represent all the selected shapes on the document); the Shape object, which represents a single shape on a document. If you want to work with several shapes at the same time or with shapes within the selection, use a ShapeRange collection.

About Shapes

Anchoring a Shape

Every Shape object is anchored to a range of text. A shape is anchored to the beginning of the first paragraph that contains the anchoring range. The shape will always remain on the same page as its anchor.

You can view the anchor itself by setting the ShowObjectAnchors property to True. The shape's Top and Left properties determine its vertical and horizontal positions. The shape's RelativeHorizontalPosition and RelativeVerticalPosition properties determine whether the position is measured from the anchoring paragraph, the column that contains the anchoring paragraph, the margin, or the edge of the page.

If the LockAnchor property for the shape is set to True, you cannot drag the anchor from its position on the page.

Positioning a Shape

The position of the shape can be described in English by its vertical and horizontal position relative to a line, paragraph, margin, or page. Four properties can be used:

- RelativeVerticalPosition identifies what object controls the vertical position
- Top specifies the vertical position
- RelativeHorizontalPosition identifies what object controls the horizontal position
- Left specifies the horizontal position

You use a combination of these properties. For example, to center a shape vertically relative to the page: Top = wdShapeCenter, RelativeVerticalPosition = wdRelativeVerticalPositionPage

Formatting a Shape

Use the Fill property to return the FillFormat object, which contains all the properties and methods for formatting the fill of a closed shape. The Shadow property returns the ShadowFormat object, which you use to format a shadow. Use the Line property to return the LineFormat object, which contains properties and methods for formatting lines and arrows. The TextEffect property returns the TextEffectFormat object, which you use to format WordArt. The Callout property returns the CalloutFormat object, which you use to format line callouts. The WrapFormat property returns the WrapFormat object, which you use to define how text wraps around shapes. The ThreeD property returns the ThreeDFormat object, which you use to create 3-D shapes. You can use the PickUp and Apply methods to transfer formatting from one shape to another.

Use the SetShapesDefaultProperties method for a Shape object to set the formatting for the default shape for the document. New shapes inherit many of their attributes from the default shape.

Other Important Shape Properties

Use the Type property to specify the type of shape: freeform, AutoShape, OLE object, callout, or linked picture, for instance. Use the AutoShapeType property to specify the type of AutoShape: oval, rectangle, or balloon, for instance.

Use the Width and Height properties to specify the size of the shape.

The TextFrame property returns the TextFrame object, which contains all the properties and methods for attaching text to shapes and linking the text between text frames.

Remarks

Shape objects are anchored to a range of text but are free-floating and can be positioned anywhere on the page. InlineShape objects are treated like characters and are positioned as characters within a line of text. You can use the ConvertToInlineShape method and the ConvertToShape method to convert shapes from one type to the other. You can convert only pictures, OLE objects, and ActiveX controls to inline shapes.

WordArt objects may not be anchored.

Converting Visio Picture into Inline Shape

```
Sub ConvertVisioPicture()
' When Visio drawing is pasted as Picture, it becomes a shape with
an anchor and
' is essentially free-floating.
' This macro converts it to an Inline Shape which is treated like
text characters
' and positioned within a line of text.

If Selection.ShapeRange.Type = msoPicture Then
    Selection.ShapeRange.ConvertToInlineShape
End If
End Sub
```

Key Properties

RelativeVerticalPosition Property

Specifies to what the vertical position of a frame, a shape, or a group of rows is relative. Read/write

Can be one of the following WdRelativeVerticalPosition constants.

wdRelativeVerticalPositionLine	Relative to line.
wdRelativeVerticalPositionMargin	Relative to margin.
wdRelativeVerticalPositionPage	Relative to page.
wdRelativeVerticalPositionParagraph	Relative to paragraph.

RelativeHorizontalPosition Property

Specifies to what the horizontal position of a frame, a shape, or a group of rows is relative. Read/write

Can be one of the following WdRelativeHorizontalPosition constants.

wdRelativeHorizontalPositionCharacter	Relative to character.
wdRelativeHorizontalPositionColumn	Relative to column.
wdRelativeHorizontalPositionMargin	Relative to margin.
wdRelativeHorizontalPositionPage	Relative to page.

Top Property

Returns or sets the vertical position of the specified shape or shape range in points. Can also be any valid constant, especially WdShapePosition (see Left Property for values). Read/write Single.

expression.Top
expression Required. An expression that returns one of the above objects.

Remarks

The position of a shape is measured from the upper-left corner of the shape's bounding box to the shape's anchor. The RelativeVerticalPosition property controls whether the shape's anchor is positioned alongside the line, the paragraph, the margin, or the edge of the page.

For a ShapeRange object that contains more than one shape, the Top property sets the vertical position of each shape.

Example

As it applies to Shape object.

This example sets the vertical position of the first shape in the active document to 1 inch from the top of the page.

```
With ActiveDocument.Shapes(1)
    .RelativeVerticalPosition = wdRelativeVerticalPositionPage
    .Top = InchesToPoints(1)
End With
```

This example sets the vertical position of the first and second shapes in the active document to 1 inch from the top of the page.

```
With ActiveDocument.Shapes.Range(Array(1, 2))
    .RelativeVerticalPosition = wdRelativeVerticalPositionPage
    .Top = InchesToPoints(1)
End With
```

This example was created automatically by wizard:

```
Selection.ShapeRange.Top = wdShapeCenter
```

Left Property

Returns or sets a Single that represents the horizontal position, measured in points, of the specified shape or shape range. Can also be any valid constant. Read/write.

WdShapePosition can be one of these WdShapePosition constants.

wdShapeBottom	At the bottom.
wdShapeCenter	In the center.
wdShapeInside	Inside the selected range.
wdShapeLeft	On the left.
wdShapeOutside	Outside the selected range.
wdShapeRight	On the right.
wdShapeTop	At the top.

expression.Left

expression Required. An expression that returns one of the above objects.

Remarks

The position of a shape is measured from the upper-left corner of the shape's bounding box to the shape's anchor. The RelativeHorizontalPosition property controls whether the anchor is positioned alongside a character, column, margin, or the edge of the page.

For a ShapeRange object that contains more than one shape, the Left property sets the horizontal position of each shape.

Example

As it applies to the Shape object.

This example sets the horizontal position of the first shape in the active document to 1 inch from the left edge of the page.

```
With ActiveDocument.Shapes(1)
```

```
    .RelativeHorizontalPosition = _
        wdRelativeHorizontalPositionPage
    .Left = InchesToPoints(1)
End With
```

This example sets the horizontal position of the first and second shapes in the active document to 1 inch from the left edge of the column.

```
With ActiveDocument.Shapes.Range(Array(1, 2))
    .RelativeHorizontalPosition = _
        wdRelativeHorizontalPositionColumn
    .Left = InchesToPoints(1)
End With
```

Shrink Inline Shapes

In my work I frequently insert Visio drawings into Word documents. I insert the drawings as Picture (Enhanced Metafile), because it conserves file size while being able to print on different printers. I style the drawings by inserting a border with an inside margin. When the drawing is wider than the text boundaries, I resize it to fit within the text boundaries (text and diagrams are left justified).

I prefer to use macros to do this. I have a macro which adds the border with inside margin (to put some space between it and the diagram), but have been adjusting the size manually. Now I want to do this automatically. I'll create a new macro for the shrinking and call it from the border macro.

Approach:

1. Make sure the requirements are met. If any are not, display an error message and stop.
    a. Only one pane open. This is so the code can assume the first pane, reasonable for my use.
    b. Only one column of text. For ease of programming, but reasonable for my use.
    c. Cursor is immediately to the right of an inline shape. The program has to know where to find the drawing and be sure it is an inline shape.
    d. Drawing is wider than column width. Otherwise no shrinking would be necessary.
2. Set width of inline shape = text column width.
3. Collapse selection to the insertion point.

Doing this manually uses the following UI features:

- "Paste Special" dialog box to insert the drawing. On the Edit menu.
- "Format Picture" dialog box to change the size settings and insert inside margins. On the Format menu.
- "Borders and Shading" dialog box to insert the border. On the Format menu.

Relevant VBA objects:

- TextColumns collection. A child object of the Page Setup object.
- TextColumns(x).Width property returns/sets the width of the actual line of text in points.
- InlineShapes collection. A child object of the Selection, Document, and Range objects.
- InlineShapes(x).LockAspectRatio method returns/sets value to true/false. True is needed here. While Word may set this by default, it is wise not to count on it. I want the aspect ratio of my drawings to remain unchanged.
- InlineShapes(x).Reset method removes changes made to the inline shape. In the context of an inserted Visio drawing, a drawing wider than the column is automatically resized to fit. When I add

a border with inside margins, it no longer fits. So it is best to reset the shape to 100% of its natural size before making my own changes.

- InlineShapes(x).Width property returns/sets the width of the shape in points.
- InlineShapes(x).ScaleWidth property returns/sets the scale of the width as a percentage of its original size.
- InlineShapes(x).ScaleHeight property returns/sets the scale of the height as a percentage of its original size.
- InchesToPoints() function converts inches to points.
- Selection.Type property returns the type of the selection. Value wdSelectionInlineShape is needed here.
- Panes collection. A child object of the ActiveWindow object.
- Panes.Count property returns the number of panes open.

Surprises:

- I first tried to shrink the shape by decrementing the ScaleWidth, as this is what I do with Word's UI. But the code `.ScaleWidth = .ScaleWidth - 1` actually incremented the value.
- When I ran the macro to set the border and inside margin before resizing, the ScaleWidth changed differently from the ScaleHeight, thus distorting the aspect ratio.
- I found it necessary to run InlineShapes(x).Reset before comparing the shape width with the text column width.
- Before setting the border and inside margins but after doing the reset and setting the shape width, the ScaleHeight was 100% while the ScaleWidth was less. If I did this manually, doing the reset after inserting the drawing changed the ScaleHeight and ScaleWidth to 100%. Then changing the picture width caused the ScaleHeight and ScaleWidth to change equally. The solution in the macro is to set ScaleHeight = ScaleWidth after changing the width.
- I notice that Reset is on both the Size and Picture tabs of the "Format Picture" dialog box. They differ in their effect on the size of the drawing: Reset on the Picture tab does not change the size, only the cropping and image controls.

Design:

- Because this code is run before that for the border and inside margin, the actual width of the shape needs to be the column width less the combined left and right margins.

Code:

```
Sub ShrinkPictureToPageWidth()
Dim title, msg As String
Dim x, y, z As Long
title = "Shrink Picture to Page Width"
z = ActiveDocument.ActiveWindow.Panes.Count
If z > 1 Then
    msg = "Cannot shrink picture while more than 1 pane is open. Pane count = "
& z
    MsgBox msg, vbOKOnly, title
    Exit Sub
End If
Selection.Collapse (wdCollapseEnd)
z = Selection.PageSetup.TextColumns.Count
If z > 1 Then
    msg = "Cannot shrink picture when there is more than one column. Column
count = " & z
    MsgBox msg, vbOKOnly, title
    Exit Sub
End If
```

```
Selection.MoveLeft Unit:=wdCharacter, Count:=1, Extend:=wdExtend
If Selection.Type <> wdSelectionInlineShape Then
    msg = "Object is not an inline shape, cannot continue."
    MsgBox msg, vbOKOnly, title
    Exit Sub
End If

x = Selection.PageSetup.TextColumns(1).Width
With Selection.InlineShapes(1)
    .LockAspectRatio = msoTrue
    .Reset
    y = .Width - InchesToPoints(0.4)
    If y <= x Then
        Selection.Collapse (wdCollapseEnd)
        Exit Sub
    End If
    .Width = x - InchesToPoints(0.4)
    .ScaleHeight = .ScaleWidth
End With
Selection.Collapse (wdCollapseEnd)
End Sub
```

WATERMARKS

Background Printed Watermark

This is inserted manually with menu Format, Background, Printed Watermark. The watermark is a WordArt Shape object. When I recorded a macro for this, it put the shape into the header of the first section; this might be a problem with a multi-section document. Inserting it with code into ActiveDocument.Shapes puts it on the last page only.

Apparently the only way to get something to appear on every page is to put it in a header/footer.

When you add WordArt to a document, the height and width of the WordArt are automatically set based on the size and amount of text you specify.

The key method is AddTextEffect of the Shape object. Adds a WordArt shape to a document. Returns a Shape object that represents the WordArt and adds it to the Shapes collection. Once the shape is added, you must select it in order to set its properties.

expression.AddTextEffect(*PresetTextEffect*, *Text*, *FontName*, *FontSize*, *FontBold*, *FontItalic*, *Left*, *Top*, *Anchor*)

expression Required. An expression that returns a Shapes object.
PresetTextEffect Required MsoPresetTextEffect constant. A preset text effect. The values of the MsoPresetTextEffect constants correspond to the formats listed in the WordArt Gallery dialog box (numbered from left to right and from top to bottom).
Text Required String. The text in the WordArt.
FontName Required String. The name of the font used in the WordArt.
FontSize Required Single. The size, in points, of the font used in the WordArt.
FontBold Required. MsoTrue to bold the WordArt font.

FontItalic Required . MsoTrue to italicize the WordArt font.
Left Required Single. The position, measured in points, of the left edge of the WordArt shape relative to the anchor.
Top Required Single. The position, measured in points, of the top edge of the WordArt shape relative to the anchor.
Anchor Optional Variant. A Range object that represents the text to which the WordArt is bound. If *Anchor* is specified, the anchor is positioned at the beginning of the first paragraph in the anchoring range. If this argument is omitted, the anchoring range is selected automatically and the WordArt is positioned relative to the top and left edges of the page.

Example:

```
Sub NewTextEffect()
    ActiveDocument.Shapes.AddTextEffect _
        PresetTextEffect:=msoTextEffect11, _
        Text:="This is a test", FontName:="Arial Black", _
        FontSize:=36, FontBold:=msoTrue, _
        FontItalic:=msoFalse, Left:=1, Top:=1, _
        Anchor:=ActiveDocument.Paragraphs(1).Range
End Sub


Sub SetDraftWatermark()
' DOES NOT WORK AS DESIRED
' ONLY PUTS WATERMARK ON FIRST PAGE
Dim wm As Shape
Set wm = ActiveDocument.Shapes.AddTextEffect(powerpluswatermarkobject1, _
    "DRAFT", "Arial Black", 40, False, False, 0, 0)
wm.Name = "Watermark"
wm.Rotation = 315
wm.LockAspectRatio = True
wm.Height = InchesToPoints(0.77)
wm.Width = InchesToPoints(2.04)
wm.RelativeHorizontalPosition = wdRelativeVerticalPositionMargin
wm.RelativeVerticalPosition = wdRelativeVerticalPositionMargin
wm.Left = wdShapeCenter
wm.Top = wdShapeCenter
wm.TextEffect.NormalizedHeight = False
wm.Line.Visible = False
wm.Fill.Visible = True
wm.Fill.ForeColor.RGB = RGB(196, 120, 120)
wm.Fill.Transparency = 0.5
wm.WrapFormat.AllowOverlap = True
wm.WrapFormat.Side = wdWrapNone
wm.WrapFormat.Type = 3
End Sub
```

Watermark as Text Box

This approach works. There are two versions, the first is a subroutine, the second is standalone.

```
Sub PutWatermarkTextInAllHeaders()
' inserts text at end of each header
' acts on current document, should be called
Dim sec As Section
Dim cnt As Integer
```

```
Dim s As Integer
cnt = ActiveDocument.Sections.Count
' For Each sec In ActiveDocument.Sections
' skip first section which is title page
For s = 2 To cnt
    Set sec = ActiveDocument.Sections(s)
    If sec.Headers(wdHeaderFooterPrimary).LinkToPrevious = False Then   '
ignore section break continuous
        sec.Headers(wdHeaderFooterPrimary).Range.InsertAfter vbCrLf + "DRAFT"
        sec.Headers(wdHeaderFooterPrimary).Range.Paragraphs.Last.Style =
ActiveDocument.Styles("Watermark")
    End If
Next
End Sub


Sub PutWatermarkTextInNamedDocument()
' inserts text at end of each header
WorkDirectory = "C:\Data\NPI Requirements\"
ChangeFileOpenDirectory (WorkDirectory)
Documents.Open FileName:="All Business Requirements.doc"
Dim sec As Section
Dim cnt As Integer
Dim s As Integer
cnt = ActiveDocument.Sections.Count
'For Each sec In ActiveDocument.Sections
For s = 2 To cnt
    Set sec = ActiveDocument.Sections(s)
    If sec.Headers(wdHeaderFooterPrimary).LinkToPrevious = False Then   '
ignore section break continuous
        sec.Headers(wdHeaderFooterPrimary).Range.InsertAfter vbCrLf + "DRAFT"
        sec.Headers(wdHeaderFooterPrimary).Range.Paragraphs.Last.Style =
ActiveDocument.Styles("Watermark")
    End If
Next
```

Code Created by Insert Print Watermark Background Wizard

This watermark was defined with text in a given font, size, and color and positioned diagonally. Apparently it can be positioned differently.

```
Sub SetDraftWatermark()
'
' SetDraftWatermark Macro
' Macro recorded 3/22/2006 by Susan J. Dorey
'
    ActiveDocument.Sections(1).Range.Select
    ActiveWindow.ActivePane.View.SeekView = wdSeekCurrentPageHeader
    Selection.HeaderFooter.Shapes.AddTextEffect(PowerPlusWaterMarkObject1, _
        "DRAFT", "Arial Black", 40, False, False, 0, 0).Select
    Selection.ShapeRange.Name = "PowerPlusWaterMarkObject1"
    Selection.ShapeRange.TextEffect.NormalizedHeight = False
    Selection.ShapeRange.Line.Visible = False
    Selection.ShapeRange.Fill.Visible = True
    Selection.ShapeRange.Fill.Solid
    Selection.ShapeRange.Fill.ForeColor.RGB = RGB(196, 120, 120)
    Selection.ShapeRange.Fill.Transparency = 0.5
```

```
    Selection.ShapeRange.Rotation = 315
    Selection.ShapeRange.LockAspectRatio = True
    Selection.ShapeRange.Height = InchesToPoints(0.77)
    Selection.ShapeRange.Width = InchesToPoints(2.04)
    Selection.ShapeRange.WrapFormat.AllowOverlap = True
    Selection.ShapeRange.WrapFormat.Side = wdWrapNone
    Selection.ShapeRange.WrapFormat.Type = 3
    Selection.ShapeRange.RelativeHorizontalPosition = _
        wdRelativeVerticalPositionMargin
    Selection.ShapeRange.RelativeVerticalPosition = _
        wdRelativeVerticalPositionMargin
    Selection.ShapeRange.Left = wdShapeCenter
    Selection.ShapeRange.Top = wdShapeCenter
    ActiveWindow.ActivePane.View.SeekView = wdSeekMainDocument
End Sub
```

ITERATIVE DOCUMENT EDITING

Here are some samples of iterative document editing.

Reformat Text in Square Brackets

The following macro illustrates several points:

- how to make the same kind of change throughout a document
- use of FindText

In this example, the text enclosed by square brackets is set to italics. I use square brackets to insert my editor/author comments in a document that is in development. Sometimes I highlight this text.

```
Sub SetNotesToItalics()
'
' Loops through active document, finds instances of paired [], then sets text
inbetween to italics.  This is necessary because Word in all its wisdom removes
the italics in an inconsistent and unpredictable manner; reinstating it each
time I edit a document is a time-consuming nuisance.

Dim x As Integer
x = 0
Do While x = 0
    With Selection.Find
        .ClearFormatting
        .Execute FindText:="["
    End With
    If Selection.Find.Found = False Then
        x = 1
        Exit Do
    End If
    Selection.MoveRight Unit:=wdCharacter, Count:=1
    ' next code extends selection through next occurrence of "]",
    ' then decreases selection by one character to unselect the ] character
    With Selection
        .Extend Character:="]"
        .MoveLeft Unit:=wdCharacter, Count:=1, Extend:=wdExtend
        If .Font.Italic = False Then
```

```
            .Font.Italic = True         ' applies italics to selection
        End If
        Selection.MoveRight Unit:=wdCharacter, Count:=2  'unselect text
    End With
Loop
End Sub
```

Insert RD Field Codes

This code constitutes a single module.

```
Attribute VB_Name = "TOC"
Option Compare Text
Private cntAll As Integer
Private cntMod As Integer
Private title As String
Private thisDoc As Document
Private strDir As String


Sub PutTCInEachRDFile()

' Context: active document contains field codes to build TOC from referenced
documents, that is TOC and RD field codes.
' Referenced documents do not have high-level heading that identifies them,
instead the Title property and header do.
' So, an improved TOC is built based on both the heading styles in the
referenced documents and a TC field code
' at the beginning of each referenced document that uses the text of the Title
property; this is effected with
' the TITLE field code embedded within the TC field code.

' This macro (1) determines the path of the referenced documents and (2) reads
the RD field codes in active document.
' For each, it opens the file and inserts a TC field code if one is not already
present.

Dim oField As Field
Dim strCode As String
Dim strMsg As String
Dim oDoc As Document

cntAll = 0
cntMod = 0
title = "Put TC in Each RD File"
Set thisDoc = ActiveDocument
If DirectoryOK = False Then
    MsgBox "Macro cancelled because no directory selected.", vbOKOnly, title
    Exit Sub
End If

'With ActiveDocument.ActiveWindow.View
'    .ShowAll = False
'    .ShowHiddenText = False
'    .ShowFieldCodes = False
'    .Type = wdPrintView
'End With
```

```
For Each oField In ActiveDocument.Fields
    If oField.Type = wdFieldRefDoc Then
        cntAll = cntAll + 1
'  get filename, strip off RD text
        strCode = Trim$(oField.Code)
        strCode = Trim$(Mid$(strCode, InStr(strCode, " ")))
'  strip off leading \f switch
        If LCase$(Left$(strCode, 2)) = "\f" Then
            strCode = Trim$(Mid$(strCode, 3))
        End If
' strip off trailing \f switch
        If LCase$(Right$(strCode, 2)) = "\f" Then
            strCode = Trim$(Left$(strCode, Len(strCode) - 2))
        End If
' strip off leading double prime
        If Asc(strCode) = 34 Then
            strCode = Trim$(Mid$(strCode, 2, Len(strCode) - 2))
        End If
' open file
        strCode = URLDecode(strCode)
        Set oDoc = Documents.Open(FileName:=strDir & "\" & strCode)
        oDoc.Activate
        'With oDoc.ActiveWindow.View
        '    .ShowAll = False
        '    .ShowHiddenText = False
        '    .ShowFieldCodes = False
        '    .Type = wdPrintView
        'End With
' run macro to insert TC field
        If AlreadyHasTC = False Then
            InsertTCTitle
            oDoc.Close wdSaveChanges
            cntMod = cntMod + 1
        Else
            oDoc.Close wdDoNotSaveChanges
        End If
        thisDoc.Activate
    End If
Next oField
strMsg = "Count of referenced documents:  " & cntAll & vbCrLf & "Count of docs
with new TC field code:  " & cntMod
MsgBox strMsg, vbOKOnly, title
End Sub


Function DirectoryOK()
' the original design had this function performing a ChDir, but its results
proved unreliable
If MsgBox("Are documents in directory: " & ActiveDocument.Path, vbYesNo, title)
= vbYes Then
    strDir = ActiveDocument.Path
    DirectoryOK = True
    Exit Function
End If
DirectoryOK = GetDir()
End Function
```

```
Function GetDir()
With Application.FileDialog(msoFileDialogFolderPicker)
    .InitialFileName = ActiveDocument.Path
    If .Show = -1 Then
        strDir = .SelectedItems(1)
        MsgBox "You selected " & strDir
        GetDir = True
    Else                'The user pressed Cancel.
        GetDir = False
    End If
End With
End Function


Function AlreadyHasTC()
' is the first line a TC field?  If created automatically it will comprise 2
field codes, TC and TITLE, types 9 and 15.
' first select first paragraph
Selection.HomeKey Unit:=wdStory
Selection.MoveDown Unit:=wdParagraph, Count:=1, Extend:=wdExtend
With Selection
    If .Fields.Count = 0 Then
        AlreadyHasTC = False
    ElseIf .Fields.Count > 0 And .Fields(1).Type = wdFieldTOCEntry Then
        AlreadyHasTC = True
    Else
        AlreadyHasTC = False
    End If
End With
Selection.MoveRight Unit:=wdCharacter, Count:=1             ' release selection
End Function


Private Function URLDecode(URLtoDecode As String) As String
    URLDecode = Replace(URLtoDecode, "%20", " ")
End Function


Sub InsertTCTitle()
' Macro inserts TC field code with value = TITLE field code.
' To be used with documents being included with RD field code into consolidated
TOC
' and having no Heading 1 styled paragraph.
' Creates TC formatted like:  { TC "{ TITLE }" \l 1 }, then generates value of
TITLE field code.
'
Selection.HomeKey Unit:=wdStory
Selection.TypeParagraph
Selection.MoveUp Unit:=wdLine, Count:=1
Selection.Range.Style = ActiveDocument.Styles(wdStyleNormal)
Selection.Fields.Add Range:=Selection.Range, Type:=wdFieldEmpty,
PreserveFormatting:=False
Selection.TypeText Text:="TC """
Selection.Fields.Add Range:=Selection.Range, Type:=wdFieldEmpty,
PreserveFormatting:=False
Selection.TypeText Text:="TITLE"
Selection.MoveRight Unit:=wdCharacter, Count:=2
```

```
Selection.TypeText Text:=""" \l 1"
Selection.MoveLeft Unit:=wdCharacter, Count:=7
Selection.Fields.Update
Selection.MoveLeft Unit:=wdCharacter, Count:=1
Selection.Fields.Update
Selection.MoveLeft Unit:=wdCharacter, Count:=8
Selection.Fields.Update
Selection.HomeKey Unit:=wdLine
Selection.MoveDown Unit:=wdLine, Count:=1
End Sub
```

Change Styles in all Word Files in a Given Directory

My need was to change certain styles in all Word files in a given directory. I found, through trial and error, that the best way to change styles is with Organizer.

Note that in this example the name of the directory is hard-coded. You could instead prompt the user for the directory name. The first SUB can be used to apply other changes to all files in a directory structure.

```
Sub ChangeFilesInDirectoryWithFSO()
Dim cntAll As Long
Dim cntChanged As Long
cntAll = 0
cntChanged = 0
Dim strPath As String
Dim fso As Object
Set fso = CreateObject("Scripting.FileSystemObject")
strPath = "C:\Data\CMS\Ops Guide Chapters"
'strPath = "C:\Data\Ops Guide Chapters"
Dim ThisFolder
Set ThisFolder = fso.GetFolder(strPath)
Application.ScreenUpdating = False
Dim fs
Set fs = ThisFolder.Files
For Each f In fs
    cntAll = cntAll + 1
    If Right(f.Name, 3) = "doc" Then
        Documents.Open f.Path, AddToRecentFiles:=False
        cntChanged = cntChanged + 1
        OrganizerCopyStyles
        ActiveDocument.Close SaveChanges:=wdSaveChanges
    End If
Next f
Application.ScreenUpdating = True
MsgBox "Count of files read: " & cntAll & vbCrLf & "Count of files changed: " &
cntChanged, vbOKOnly, "Change Files in Directory"
End Sub
```

The following macro was recorded. I selected all the styles in the dialog box, so they all were named individually in the macro. I then deleted the lines for the styles I did not want to copy. I also added code to use variables. I could not figure out how to do a Find-and-Replace to get rid of the continuation marks.

```
Sub OrganizerCopyStyles()
Dim strFilename As String
```

```
Dim strTemplate As String
strFilename = ActiveDocument.FullName
strTemplate = "C:\Data\CMS\Template for CMS Operations Guide.dot"
Application.OrganizerCopy Source:=strTemplate, Destination:=strFilename,
Name:="Default Paragraph Font", Object:=wdOrganizerObjectStyles
Application.OrganizerCopy Source:=strTemplate, Destination:=strFilename,
Name:="Bold", Object:= _
    wdOrganizerObjectStyles
Application.OrganizerCopy Source:= _
    strTemplate, Destination:= _
    strFilename, Name:="Normal", Object:= _
    wdOrganizerObjectStyles
Application.OrganizerCopy Source:= _
    strTemplate, Destination:= _
    strFilename, Name:="CaptionAutoNum", _
    Object:=wdOrganizerObjectStyles
. . .
End Sub
```

PASSWORDS AND PROTECTION

There can be three passwords in a Word document:
a. password to open the document, set on Tools, Options, Save tab
b. password to modify the document, set on Tools, Options, Save tab
c. document protection password, set on Tools, Protect Document; this password can apply to tracked changes, comments, or forms

VBA modules can have a password. It can lock the project for viewing and/or restrict viewing properties. It is set from menu Tools, Project Properties, Protection tab.

So far I have not learned how to detect these with VBA.

Protection

The Document object has property ProtectionType which returns the protection type for the specified document. Can be one of the following WdProtectionType constants: wdAllowOnlyComments, wdAllowOnlyFormFields, wdAllowOnlyReading, wdAllowOnlyRevisions, or wdNoProtection.

The Document object has two methods related to protection:

- Protect: Applies protection. If the document is already protected, this method generates an error.
- Unprotect: Removes protection from the specified document. If the document isn't protected, this method generates an error.

expression.Protect(Type, NoReset, Password, UseIRM, EnforceStyleLock)
where:
Type: is a WdProtectionType constants
NoReset: applies only to Type = wdAllowOnlyFormFields
Password: the password required to remove protection from the specified document;optional.
UseIRM: Specifies whether to use Information Rights Management (IRM) when protecting the document from changes; optional.
EnforceStyleLock: Specifies whether formatting restrictions are enforced in a protected document; optional.

expression.UnProtect(Password)
where
Password: The password string used to protect the document; optional. Passwords are case-sensitive. If the document is protected with a password and the correct password isn't supplied, a dialog box prompts the user for the password

```
If Doc.ProtectionType <> wdNoProtection Then Doc.Unprotect

If ActiveDocument.ProtectionType := wdNoProtection Then
   ActiveDocument.Unprotect Password:="readonly"
End If
```

Read-Only

There are two document properties that involve read-only:

- ReadOnlyRecommended property
- ReadOnly property

The ReadOnly property only returns the value, it cannot be used to set the value.

The ReadOnlyRecommended property is read/write meaning it returns the current value and can change the current value.

You can turn off Read-Only Recommended when saving a document:

```
ActiveDocument.SaveAs FileName:="C:\Temp\MyFile.doc",
   Password:="",  WritePassword:="",  ReadOnlyRecommended:=False
```

```
ActiveDocument.ReadOnlyRecommended = True
```

INTERACTING WITH AN ACCESS DATABASE

If you are using data in an Access database to control processing of one or more Word documents, you can use DAO and not Access directly. In this case
first set a reference to Microsoft DAO 3.6 Object Library.

Define an object corresponding to an Access database using DAO:

```
Private dbsThis As Database
```

Open database:

```
Set dbsThis = DBEngine.OpenDatabase("\\server\share\Data\Migrate.mdb")
```

Use a recordset to iterate through a table or query:

```
Private rstQ As Recordset
Set rstQ = dbsMigrate.OpenRecordset("WordFilesNotChecked")
With rstQ
Do Until .EOF
    txtFilename = ![FullName]
    Documents.Open FileName:=txtFilename, ReadOnly:=False,
AddToRecentFiles:=False
    . . .
    .Edit
    ![ControlDate] = Now()
    .Update
Loop
```

```
.Close
End With
Set rstQ = Nothing
Set dbsThis = Nothing
```

AUTOMATION

Automation is the running of one application from a second. Examples: (1) editing a Word document with code in an Access database and (2) using data from an Access table to control processing of a Word document.

Per Microsoft: You can use another Office application's objects (or the objects exposed by any other application or component that supports Automation) without setting a reference in the References dialog box by using the CreateObject or GetObject function and declaring object variables as the generic Object type. If you use this technique, the objects in your code will be late-bound, and as a result you will not be able to use design-time tools such as automatic statement completion or the Object Browser, and your code will not run as fast.

Early-Bound Declarations

Early binding allows you to declare an object variable as a programmatic identifier, or class name, rather than as an Object or a Variant data type. The programmatic identifier of an application is stored in the Windows registry as a subkey below the \HKEY_CLASSES_ROOT subtree. For example, the programmatic identifier for Access is "Access.Application"; for Excel it is "Excel.Application."

When you are using early binding, you can initialize the object variable by using the CreateObject or GetObject function or by using the New keyword if the application supports it. All Office 2000 applications can be initialized by using the New keyword. Because the Outlook 2000 programming environment for Outlook items supports only scripting, you can't use early binding declarations of any sort in its VBScript programming environment; however, you can use early binding in VBA code in a local Outlook VBA project or COM add-in, or in Automation code that works with Outlook from another host application.

Use early binding whenever possible. Early binding has the following advantages:

- Syntax checking during compilation rather than at run time.
- Support for statement-building tools in the Visual Basic Editor.
- Support for built-in constants. If you use late binding, you must define these constants in your code by looking up the values in the application's documentation.
- Better performance—significantly faster with early binding than with late binding.

INVENTORYING MACROS

Word provides you with two tools to see which macros you have: the Macros dialog box and the Visual Basic IDE (with the Project Explorer). Both make it difficult to actually find a macro when you have more than one module.

It is possible to write VB code to inventory your macros in a non-hierarchical list.

You need a reference to the VBIDE—Microsoft Visual Basic for Applications Extensibility, **VBE6EXT.OLB**.[1] This library provides objects that you can use to work with the Visual Basic Editor and any VBA projects that it contains. Help for this object model is available in **VBOB6.CHM**.[2] It can be useful to place a shortcut to this help file on the Desktop.

Relevant Object Models

The relevant Word object model objects:

Entity	*Type*	*Description*
Templates	collection	all templates that are currently available including open template files, document template attached to open documents, and loaded global templates
Template	object	a DOT file
VBProject	object	see the VBIDE object model

VBProject	property	returns the VBProject object for the specified template or document; applies to Document and Template objects
		`Set normProj = NormalTemplate.VBProject`
		`Set currProj = ActiveDocument.VBProject`

FullName	property	name of document/template file including the path
NormalTemplate	property	returns a Template object that refers to the Normal template (normal.dot)[3]; applies to the Application and Global objects
AttachedTemplate	property	returns a Template object that refers to the template attached to the specified document, this is the document template and is usually "normal.dot"; applies to the Document object

A problem with AttachedTemplate is that it is often used prefixed with ActiveDocument. If there is no document open, there will be an error. The solution is to test the value of the count of open documents:

```
If Documents.Count >= 1 Then
    MsgBox ActiveDocument.Name
Else
    MsgBox "No documents are open"
End If
```

The relevant VBIDE object model:

VBE
+VBProjects
VBProject
+References
Reference
+VBComponents
VBComponent

[1] The VBIDE file is located in directory **C:\Program Files\Common Files\Microsoft Shared\VBA\VBA6\.**

[2] The help file is located in directory **C:\Program Files\Common Files\Microsoft Shared\VBA\VBA6\1033**.

[3] It is safe to assume there is ALWAYS a normal.dot file. Even if you delete it, Word recreates it.

+Properties
Property
CodeModule

Entity	*Type*	*Description*
VBE	object	root object (in Excel this is VBIDE)
ActiveVBProject	property	returns active project (in the Project window); applies to VBE object
VBProjects	collection	all projects that are open in the VBIDE
Count	property	returns count of items in a collection
VBProject	object	refers to a VBA project that is open in the Visual Basic Editor
VBComponents	collection	refers to all components in the current project
VBComponent	object	component in the project, such as a standard **module**, class module, or form
VBComponent.Type	property	returns type of module
Find	method	searches active module for a specified string
GetSelection	method	returns the selection in a code pane
Item	method	returns the indexed member of a collection
CountOfLines	method	returns count of lines in procedure (macro)
ProcCountLines	property	returns the count of all lines in a named procedure, including blank or comment lines preceding the procedure declaration and, if the procedure is the last procedure in a code module, any blank lines following the procedure; applies to a CodeModule
ProcOfLine	property	returns the name of the procedure that the specified line is in
Lines	property	returns a string containing the specified number of lines; applies to a CodeModule
VBE	property	returns the root of the VBE object; applies to all collections and objects
Name	property	returns/sets name of the object
FileName	property	full path name of the document/template file for the current project; see NOTE below
Type	property	type of object ; "1" for code module

Examples of use:

```
MsgBox VBE.ActiveVBProject.Name
MsgBox ActiveDocument.AttachedTemplate.FullName
MsgBox NormalTemplate.FullName
MsgBox VBE.VBProjects(1).VBComponents(1).Name
Set normproj = NormalTemplate.VBProject
Dim norm As VBProject
Set norm = VBE.VBProjects("Normal")
```

NOTE: After the following code located in normal.dot was run:

```
Set norm = VBE.VBProjects("Normal")
strFileName = norm.FileName
```

variable strFileName = "c:\documents and settings\owner\normal"
This is in error because this normal.dot is located in c"\data\template\".
An accurate approach is to use the FullName property that applies to a document or template file.

Iterating Procedures in Modules

In this context it is important to remember that a module can contain declaratives, subs, and functions. The subs and functions are procedures. How to iterate procedures?

Found basic idea on http://www.cpearson.com/excel/vbe.aspx. It inventories macros in a module by iterating through the lines of code, recognizing a macro, getting its information, and moving on to the next macro. It wrote the information to an Excel spreadsheet. My code, below, writes information to a table in a Word document.

In this example I limit the template file to **normal.dot** wherever it may be.

```
Sub InventoryMacros()
Dim LineNum As Long
Dim NumLines As Long
lngLineCount = 0

NewDocForInventory        ' opens blank doc, inserts table

Dim norm As VBProject
Set norm = NormalTemplate.VBProject
strFileName = NormalTemplate.FileName
Dim modl As VBComponent
For Each modl In norm.VBComponents
    strModuleName = modl.Name
    With modl.CodeModule
    LineNum = .CountOfDeclarationLines + 1
    Do Until LineNum >= .CountOfLines
        strMacroName = .ProcOfLine(LineNum, vbext_pk_Proc)
        LineNum = .ProcStartLine(strMacroName, vbext_pk_Proc) + _
            .ProcCountLines(strMacroName, vbext_pk_Proc) + 1
        lngLineCount = .ProcCountLines(strMacroName, vbext_pk_Proc)
        WriteLine
    Loop
    End With
Next modl
End Sub

Sub WriteLine()
Selection.TypeText Text:=strFileName
Selection.MoveRight Unit:=wdCell
Selection.TypeText Text:=strModuleName
Selection.MoveRight Unit:=wdCell
Selection.TypeText Text:=strMacroName
Selection.MoveRight Unit:=wdCell
Selection.TypeText Text:=lngLineCount
Selection.MoveRight Unit:=wdCell
Selection.MoveLeft Unit:=wdCharacter, Count:=1
```

```
Selection.InsertRows 1
Selection.Collapse Direction:=wdCollapseStart
End Sub
```

This code is fine as long as you are using only the document template. You can expand it to select a different template (which I did).

Basic Logic to Inventory Macros

1. Which document/template file
use the Normal template normal.dot?
if No prompt for file — must be normal.dot
open file
set object for template:
if Documents.Count = 0 Then
    Set tmpl = NormalTemplate.VBProject
Else
    Set tmpl = ActiveDocument.AttachedTemplate.VBProject
End If
create new Word doc, create table with one row of 4 columns (filename, module name, macro name, count of lines)
for each module
    get macro name
    write in

INVENTORYING FILES

Goal: Dynamically create an index of files in a given directory, save it as a Word document. Provide for portability of the program code so that it can be easily shared with others.

Functionality: Index is created as a table populated with file properties. Index can be re-created any time.

Architecture: Program logic is composed of Word VBA macros that reside in a Word document file named Index.doc. The content of the document is title, table (holding index), directions, and macrobutton (used to run the indexing macro). The Word document is thus self-sufficient. A copy can be placed into any directory for which an index is desired, and emailed to others.

Example of index:

Filename	*Date Last Saved*
Last Tango in Paris.doc	6-1-1988 3:48:34 PM

Logic:
Document file is open when macro is run. Table exists and is formatted; the first row is a repeating header row.
BEGIN
1. Get current directory.
2. IF table exists
    delete all but first two rows (row #1 is heading row)

ELSE
create table and format it
3. Position cursor at the end of the table. Insert rows to hold all files in the current directory, except for the Index file itself. (This technique keeps new rows from being flagged as repeating header rows.)
4. Initialize row counter to 1.
5. For each file in the current directory, except the active one and temporary ones (filename begins with "~")
   a. increment row counter
   b. set value of first cell
   c. set value of second cell
6. Sort table on first column.
7. Delete empty rows. These will exist if any files were excluded (the temporary files).
8. Display message box with count of files.

END

Relevant Objects

Document
Table, Row
FileSystemObject

Code

```
Private txtDir As String
Private txtFile As String
Private cntFiles As Integer
Private rowIdx As Integer
Private idxTbl As Table



Sub IndexCurrentDirectory()
' Get current directory and filename
txtDir = ActiveDocument.Path
txtFile = ActiveDocument.Name
cntFiles = 0

If ActiveDocument.ActiveWindow.Panes.Count > 1 Then
    For i = 2 To ActiveDocument.ActiveWindow.Panes.Count
         ActiveDocument.ActiveWindow.Panes(i).Close
    Next
End If

ActiveDocument.ActiveWindow.View.Type = wdNormalView

' if table exists
If ActiveDocument.Tables.Count = 0 Then CreateIndexTable
Set idxTbl = ActiveDocument.Tables(1)

If ActiveDocument.Tables.Count > 1 Then
    MsgBox "Fatal error: There should be only one table!", vbOKOnly, "Index the
Current Directory"
    Exit Sub
```

```
End If

' delete all but first two rows, row 2 will be overwritten by first indexed
file
If idxTbl.Rows.Count > 1 Then
For i = idxTbl.Rows.Count To 3 Step -1
    idxTbl.Rows(i).Delete
Next
End If

' establish connection to the current directory
Dim fso As Object
Set fso = CreateObject("Scripting.FileSystemObject")
strPath = txtDir
Dim ThisFolder
Set ThisFolder = fso.GetFolder(strPath)
Set fs = ThisFolder.Files

' size the table to hold the actual number of files
If fs.Count > 2 Then
    n = fs.Count - 1       ' exclude the Index file
    idxTbl.Rows.Last.Select   ' selects last row
    Selection.Collapse Direction:=wdCollapseEnd
    Selection.MoveLeft Unit:=wdCharacter, Count:=1
    Selection.InsertRows n
    Selection.Collapse Direction:=wdCollapseStart
End If

' position on first row
rowIdx = 1

' For each file in the current directory
For Each f In fs
    IndexThisFile f.Name, f.DateLastModified
Next f

' sort table on first column
idxTbl.Sort ExcludeHeader:=True
' Make first table row a repeating header
idxTbl.Rows(1).HeadingFormat = True

' delete empty rows
' these will correspond to the temporary files that were not indexed
With idxTbl
For j = .Rows.Count To 1 Step -1
    If Len(.Cell(j, 1).Range.Text) = 2 Then .Rows(j).Delete
Next j
End With

' Display message box with count of files
MsgBox "Files indexed = " & cntFiles, vbOKOnly, "Index the Current Directory"
End Sub


Sub IndexThisFile(n, d)
' insert row in table for file in directory
If n = txtFile Then Exit Sub            ' exclude the active Index file
If Left(n, 1) = "~" Then Exit Sub       ' exclude temporary files
```

```
rowIdx = rowIdx + 1
idxTbl.Rows(rowIdx).Cells(1).Range.Text = n
idxTbl.Rows(rowIdx).Cells(2).Range.Text = d
cntFiles = cntFiles + 1
End Sub


Sub CreateIndexTable()
' create table and format it
Selection.EndKey Unit:=wdStory           'move to end of document
ActiveDocument.Tables.Add Range:=Selection.Range, NumRows:=1, NumColumns:=2,
DefaultTableBehavior:=wdWord9TableBehavior
ActiveDocument.Tables(1).Cell(1, 1).Range.Text = "Filename"
ActiveDocument.Tables(1).Cell(1, 2).Range.Text = "Last Save Date"
Set idxTbl = ActiveDocument.Tables(1)
End Sub
```