Office Automation Tips, Tricks and Traps

Tamar E. Granor, Ph.D.
Tomorrow's Solutions, LLC
Voice: 215-635-1958
Email: tamar@tomorrowssolutionsllc.com

Once you get started automating Microsoft Office, you're likely to find more and more uses for Automation. But you also start running into peculiarities of the individual servers, file format issues, and other complications. In this session, we'll look at a number of issues related to automating the Office servers, including exploring why recording a macro can lead you astray. Most of the topics will apply to multiple Office servers, but we'll also look at the so-called Outlook "hell patch" and how you can automate Outlook email with a minimum of fuss.

Automation, the ability to operate the Microsoft Office applications programmatically, is an extremely powerful tool. It lets your applications create, modify and read Word documents,

Excel spreadsheets, PowerPoint presentations, and Outlook calendars, contacts and email messages. (In fact, you can automate other Office applications as well, but they're not covered in this document.)

Getting started with Office Automation is fairly simple. You instantiate the right server and talk to it. To find out what you can do and how to do it, you have a number of options, including the Office Help files, the Object Browser of the Visual Basic Editor, VFP's Object Browser, VFP's IntelliSense and the macro recorders of the various Office applications (though Outlook doesn't offer a macro recorder). Once you have some idea what to do, you may find lots of places that Automation makes sense.

However, when you get past the most basic operations, or try to transfer your knowledge from one Office server to another, you start running into some issues. There are differences among the servers and places where recording a macro doesn't really tell you what you need to know. Thanks to spammers and scammers, automating Outlook has its own peculiarities.

This paper will cover similarities and differences among the Office servers and among different Office versions. We'll look at what a recorded macro doesn't tell you, and tools for making Outlook automation work. We'll also look at how your applications can respond to events in an Office application.

The Same, Only Different

Since Office is a suite, if you've automated one of the servers, you might assume that working with the others is a breeze. In many ways, that's true. Once you understand the nature of Automation, and know how to tease information out of the Office documentation, you're ahead of the game.

But, there are some subtle variations (and some not so subtle) among the different applications that means code that works in one application might not work in another. (This is mostly because the apps have different heritages, and their Automation models have to accommodate pre-existing differences.) To complicate matters further, while Word, Excel and PowerPoint have a lot in common, Outlook is different from the others in many ways.

There are also some differences among versions of the Office apps. The items below highlight a few of them. In addition, each new version brings changes to the object model, with some PEMs being deprecated ("hidden," in Microsoft-ese) or removed and new ones being introduced. Table 1 provides links to "What's New" pages in MSDN for the Office 2010 applications discussed in this white paper. Each of these pages has three links: one for a list of new objects, one for a list of new PEMs in existing objects, and one for a list of changed, hidden, or removed PEMs. (To find corresponding articles for the other Office applications, drill down from http://msdn.microsoft.com/en-us/library/cc313152%28v=office.12%29.aspx; choose the product you're interested in, and then choose the Developer Reference for that product. In most cases, the home page for the developer reference includes a What's New link.)

Table 1. These articles point to the version changes in the object model for the Office apps.

Application	Object Model changes article
Excel	http://msdn.microsoft.com/en-us/library/ff846371.aspx
Outlook	http://msdn.microsoft.com/en-us/library/ff870434.aspx
PowerPoint	http://msdn.microsoft.com/en-us/library/ff746843.aspx
Word	http://msdn.microsoft.com/en-us/library/ff841699.aspx

Starting in Office 2007, the default format for Office documents is XML-based. This can cause problems if your application makes assumptions about the file formats it's working with. In addition, there are particular problems in importing Excel spreadsheets from recent versions into VFP.

Getting started

The first major difference has to do with instances and creation. Excel and Word let you create multiple instances, while PowerPoint and Outlook are restricted to single instances. This has several implications. First, if you call CreateObject() more than once for Excel or Word, you have more than one instance of the server running. However, with PowerPoint and Outlook, all the object references refer to the same instance. More significantly, if PowerPoint or Outlook is running before you call CreateObject(), the function returns a reference to the running instance.

Not surprisingly, if you create an instance of PowerPoint or Outlook via Automation and then start the same application through the Windows interface, there's still only one instance running. Excel, on the other hand, opens a separate instance in this situation.

Word's behavior on this front is quite odd. If you create an instance via Automation and then start Word from the Windows interface, there's still only one instance running. However, if you create two instances via Automation, then shut down the first one you created, and then start Word from Windows, you get two instances. That is, starting Word from the Windows interface will attach to your first Automation instance, but not to a later one. The code in Listing 1 demonstrates the different behaviors. To see how many instances are involved, open the Task Manager and watch for WinWord.EXE.

Listing 1. This code demonstrates how Word behaves when you start it first with Automation and then from the Windows interface.

```
* First, just one Automation instance
oWord = CREATEOBJECT("Word.Application")
? oWord.Documents.Count  && shows 0 because opening via automation doesn't
                         && create a blank document to start with
* Now start Word from the Windows desktop. Then:
? oWord.Documents.Count  && shows 1 because opening from the desktop creates
```

```
                          && a blank document
oWord.Quit()              && closes visible instance, as well

* Now, try using second Automation instance
oWord = CREATEOBJECT("Word.Application")
oWord2 = CREATEOBJECT("Word.Application")
? oWord.Documents.Count   && shows 0 because opening via automation doesn't
                          && create a blank document to start with
? oWord2.Documents.Count  && shows 0 because opening via automation doesn't
                          && create a blank document to start with
* Now close the first instance
oWord.Quit()
* Now start Word from the Windows desktop. Then:
? oWord2.Documents.Count  && still 0 because the desktop instance didn't connect
                          && to this one.
oWord2.Quit()             && visible instance is still open
```

Getting out

The differences continue when it comes to exiting the applications. When you shut down Word or PowerPoint, the executable is unloaded, but when you close Excel or Outlook, the executable stays in memory until you release the variable that held the reference. The object reference is still good, and you can talk to a number of its properties.

This has an impact on the code you need to check for a valid object reference to the server. For Word and PowerPoint, you can use code along the lines of Listing 2.

Listing 2. When you close Word or PowerPoint, the executable is unloaded, so code like this lets you check whether you have a good reference to the server.

```
IF VARTYPE("oWord") = "O" AND TYPE("oWord.Name") = "C"
   * Server is running
ENDIF
```

But analogous code won't work for Excel or Outlook. The best solution I've found is to maintain an extra property or variable to track the server's expected visibility and test whether expected visibility matches actual visibility. (The issue is that, once you make the server visible, a user might shut it down. As long as you keep it invisible, your application controls its existence and you don't need to check whether your reference is valid, though a sophisticated user might use the Task Manager to close the application.) In Listing 3, lShouldBeVisible is a variable; you could make it a property if you're using a wrapper around the server. Any code that affects the server's visibility needs to set lShouldBeVisible appropriately.

Listing 3. Because Excel.EXE stays in memory, you have to work a little harder to find out whether you have a good reference to Excel or not. The same thing applies to Outlook.

```
* FUNCTION IsExcelRunning
* Assumes lShouldBeVisible exists and is in scope.
```

```
LPARAMETERS oXL
LOCAL lReturn

IF IsNull(m.oXL)
  * No instantiated server
  lReturn = .F.
ELSE
  * Compare actual Visible value to tracked visibility
  IF oXL.Visible = m.lShouldBeVisible
    * They match, so the server is open and good
    lReturn = .T.
   ELSE
    * Visibility doesn't match. User must have
    * shut server down.
    lReturn = .F.
  ENDIF
ENDIF

RETURN m.lReturn
```

Show me

If you've done much Automation work, you've undoubtedly learned that Automation code runs faster when the server is invisible. However, keeping PowerPoint invisible can be tricky. First, once you show PowerPoint by setting its Visible property to True, you can't hide it again. The command in Listing 4 generates an error.

Listing 4. This command generates an error. You can't make PowerPoint invisible once it's visible.

```
oPowerPoint.Visible = .F.
```

Second, a number of PowerPoint's properties, including ActivePresentation, aren't accessible unless PowerPoint is visible. To address individual presentations with PowerPoint hidden, you must use the Presentations collection.

The Add and Open methods of the Presentations object have an optional WithWindow parameter that determines whether the newly opened presentation appears in a window. In PowerPoint 2007 and earlier, when PowerPoint is not visible, even if you pass .T. (or -1) for that parameter, PowerPoint stays hidden. In PowerPoint 2010, passing .T. or -1 or omitting the parameter makes PowerPoint visible.

What's in a template?

Another significant difference among the servers is the relationship of templates to documents. In Word and Excel, you can create a new document based on a template, while in PowerPoint, using Automation, you create the presentation first, then apply the template to it using the ApplyTemplate method.

In Excel and PowerPoint, basing a document on a template simply makes a copy of the template as a starting point. In Word, a new document contains copies of the template's

boilerplate text and styles, but retains a connection to the template's macros. This means that changes to the macro definitions in a Word template affect documents based on that template. That's not true for the other applications.

Click to Run breaks automation

Some versions of Office 2010 offer a new kind of installation called "Click to Run." With Click to Run, Office (or the chosen Office application) runs in a virtual machine, downloading features as needed. Office applications installed as Click to Run cannot be automated.

Fortunately, all versions of Office 2010 that offer Click to Run installation also offer standard installation. Be sure to specify that standard installation of Office is required for any application that automates Office.

File format issues

With Office 2007, Microsoft introduced new XML-based file formats for Office files. The new formats use extensions ending in "X," such as "DOCX" for Word, and "XLSX" for Excel. By default, the Office applications save documents in the new formats.

This format change can break existing Automation code, if that code assumes that you're creating the older format, and acts accordingly. It also can break user expectations. For example, suppose you have code that creates a spreadsheet, saves it, and emails it to a customer. Using Office 2003 or earlier, the spreadsheet would be in XLS format; in Office 2007 or later, it will arrive in XLSX format. (In both cases, that's assuming you don't explicitly specific the file format.) That may or may not be a problem.

You can, of course, specify the file format you want. To do so, pass the optional FileFormat parameter to the SaveAs method. When the parameter is omitted, SaveAs uses the default file format for that application. Be aware that the user controls the default, so for example, Word 2010 can be set to use the DOC format rather than the DOCX format. For that matter, if the user has installed the Office 2007 Compatibility Pack, Word 2003 can be set to use the DOCX format by default. The takeaway here is that, if the file format matters, you need to explicitly specify it.

Dealing with XLSX files

The biggest file format issue isn't exactly an automation issue. In Excel 2007 and Excel 2010, if a user saves a workbook in the "Excel 97-2003 Workbook (*.xls)" format, the resulting workbook cannot be imported into VFP using either APPEND FROM or IMPORT. Instead, you get the error message "Microsoft Excel file format is invalid." The same problem occurs if you use Automation to save the file in that format, by passing 56 (xlExcel8) for the file format parameter.

In some cases, the problem can be solved with Automation. Open the file and then use SaveAs and choose to save in Excel 95 format, passing 39 (xlExcel5) for the file format

parameter. Files saved this way work with APPEND FROM and IMPORT. The same approach can be used to convert XLSX files for use with APPEND FROM and IMPORT.

There is one caveat: Excel 95 files are limited to 16,384 rows. Excel 97-2003 raised the limit to 65,535. Excel 2007 and later support over a million rows using the new XML-based format. So before saving in the older format, you'll want to check the number of rows used. If necessary, break the file into a set of smaller files for import. The program in Listing 5 performs this process; just pass the file name for the original workbook; it's included in the materials for this session as ConvertToExcel5.PRG.

Listing 5. This routine opens a specified workbook and resaves it in Excel 95 (Excel 5.0) format. If necessary, it breaks the workbook up into multiple workbooks, each small enough to be saved in that format.

```
* Open a specified workbook, and if it's stored in a format
* later than Excel 95, resave it in the older format.
* If necessary, break it into multiple workbooks.
* Several caveats:
*   1) Works only on the active sheet of the workbook.
*   2) If it's necessary to break the worksheet up, this code
*      assumes there are no formulas working across multiple rows.
* Return the number of resulting workbooks.
* Return 0 if no change is needed.
* Return a negative value to indicate a problem.

LPARAMETERS cFileWithPath

LOCAL oExcel, oWorkbook, nWorkbookCount, cBaseName, cPath

TRY
   oExcel = CREATEOBJECT("Excel.Application")
CATCH
   oExcel = .null.
ENDTRY

IF ISNULL(m.oExcel)
   RETURN -1
ENDIF

TRY
   oWorkbook = oExcel.Workbooks.Open(m.cFileWithPath)
CATCH
   oWorkbook = .null.
ENDTRY

IF ISNULL(m.oWorkbook)
   RETURN -1
ENDIF

* If we get this far, we've opened Excel and the workbook.
* Now figure out whether we need to convert.
cPath = JUSTPATH(m.cFileWithPath)
```

```
cBaseName = JUSTSTEM(m.cFileWithPath) + "XL5"

DO CASE
CASE oWorkbook.FileFormat <= 39 && Excel 5 or earlier
   * Nothing to do.
   nWorkbookCount = 0

CASE oWorkbook.ActiveSheet.UsedRange.Rows.Count <= 16384
   * Just save as.
   cFileName = FORCEPATH(m.cBaseName, m.cPath) && extension is automatic
   oWorkbook.SaveAs(m.cFileName, 39)
   nWorkbookCount = 1

OTHERWISE
   * Need to break up into multiple workbooks.
   LOCAL nTotalRows, nSheetsNeeded, nSheet, oNewBook, nLastColumn, oRangeToCopy
   WITH oWorkbook.ActiveSheet

      nTotalRows = .UsedRange.Rows.Count
      nSheetsNeeded = CEILING(m.nTotalRows/16384)

      nLastColumn = .UsedRange.Columns.Count

      FOR nSheet = 1 TO m.nSheetsNeeded
         oNewBook = oExcel.Workbooks.Add()
         oRange = .Range(.Cells((m.nSheet-1) * 16384 + 1, 1), ;
                         .Cells(m.nSheet * 16384, m.nLastColumn))
         oRange.Copy(oNewBook.ActiveSheet.Range("A1"))

         cFileName = FORCEPATH(m.cBaseName + "_" + TRANSFORM(m.nSheet), m.cPath)
         oNewBook.SaveAs(m.cFileName, 39)
         oNewBook.Close()
      ENDFOR

   ENDWITH

   nWorkBookCount = m.nSheetsNeeded
ENDCASE

oWorkBook.Close()
oExcel.Quit()

RETURN m.nWorkBookCount
```

This approach is appropriate if the workbook is essentially a table. If the workbook contains formulas that need to work across all rows (or large groups of rows), breaking the workbook up will be a problem.

Note that the problem occurs even if an XSLX file is opened in Excel 2003 and resaved from there. The key element appears to be that the file originates in the XML format; this suggests to me that the bug is in the conversion from XLSX to XLS. Microsoft Knowledge Base article #954318 (http://support.microsoft.com/kb/954318) explains that the issue is additional content in the file to avoid losing some Excel 2007 features. In my view, this makes it an

Excel bug; saving to an older format normally does lose newer features. Creating a non-compatible file is not the right answer.

Finally, if you don't have Excel available to resave the file, you can use the Excel ODBC driver to open the file and extract the data. See http://support.microsoft.com/default.aspx?scid=kb;en-us;949529. This approach works whichever format the file is in.

Configuring Office for easier development

While you can't usually control the way Office is installed or set up on your end users' machines, you can configure your own computer to make it easier to develop Office automation code. There are a couple of tweaks worth doing.

Making Help available

The Office documentation includes information on the object model and the VBA dialect for each of the Office applications. This documentation is one of the best sources of information when writing Automation code. However, how you get to it isn't always obvious.

In Office 2003 (and earlier versions), the VBA Help files for Office aren't automatically installed. For Office 2003, by default, they're installed as "on first use"; if that's your situation, you can repair your installation to get the VBA Help files. Follow these steps:

1. From the Control Panel, choose to Uninstall a Program.
2. Choose your version of Office 2003 in the list.
3. Click the Change button.
4. In the Microsoft Office 2003 Setup dialog, choose Add or Remove Features. Click Next.
5. Check Choose advanced customization of applications. Click Next.
6. In the treeview, expand Office Shared Features, then expand Visual Basic for Applications.
7. Change Visual Basic Help to Run from My Computer.
8. Click Update.

If you're initially installing Office 2003, use steps 6-8 as part of a custom installation to get the VBA Help files installed in the first place.

Once you have the Office 2003 VBA Help files installed, you may want to create desktop shortcuts for them. The US English versions of the files are found in the Office11\1033 folder of your Office installation and their names are in the form VBAxxnn.CHM, where xx is an abbreviation for the application and nn is a version number. Although Office 2003 is Office 11, some of the VBA Help files have nn set to 10.

For Office 2007 and 2010, the situation is a little more complicated. The information (no longer called "VBA Help," but "Developer Reference") is installed as part of the Help file. However, these versions default to using Office.COM as the primary source of Help. Whether you're using Office.COM or the local version, you access the information by choosing "Developer Reference" from the Search dropdown in Help for the specified application, as in Figure 1. This brings up the Developer Reference, shown in Figure 2.

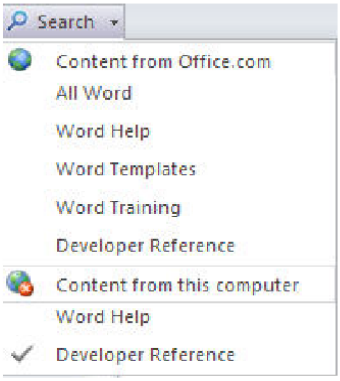

Figure 1. To see VBA Help from an Office 2007 or Office 2010 application, open Help from inside the application, then choose Developer Reference either from Office.com or from the local computer.

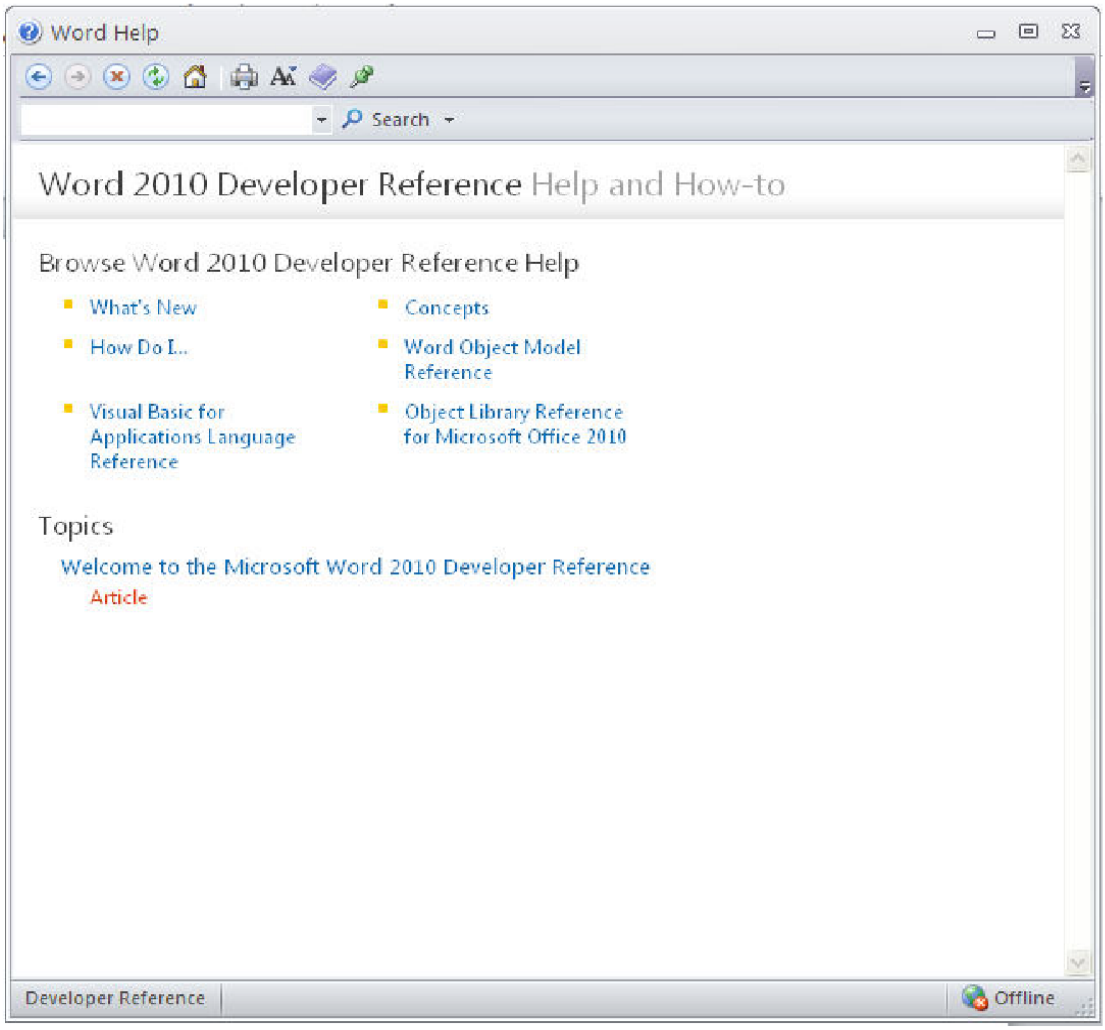

Figure 2. The home page of the Developer Reference for Word 2010 is fairly bare, but does give you access to the entire file.

The object models and VBA for each application are fully documented in both the local file and the website, but in my view, the usability of the information is significantly reduced from earlier versions. The Office 2003 and earlier VBA Help files provided a clickable visual representation of the object model, and the index let you choose whether to look at collections, objects, properties, methods or events. In Office 2007 and 2010, the diagram is gone for some of the applications and well-hidden for the others (try searching inside Help for "<Application > Object Model Reference") and the table of contents shows you only objects. Turning on the Table of Contents (by clicking the "book" button in the toolbar) helps some; Figure 3 demonstrates.

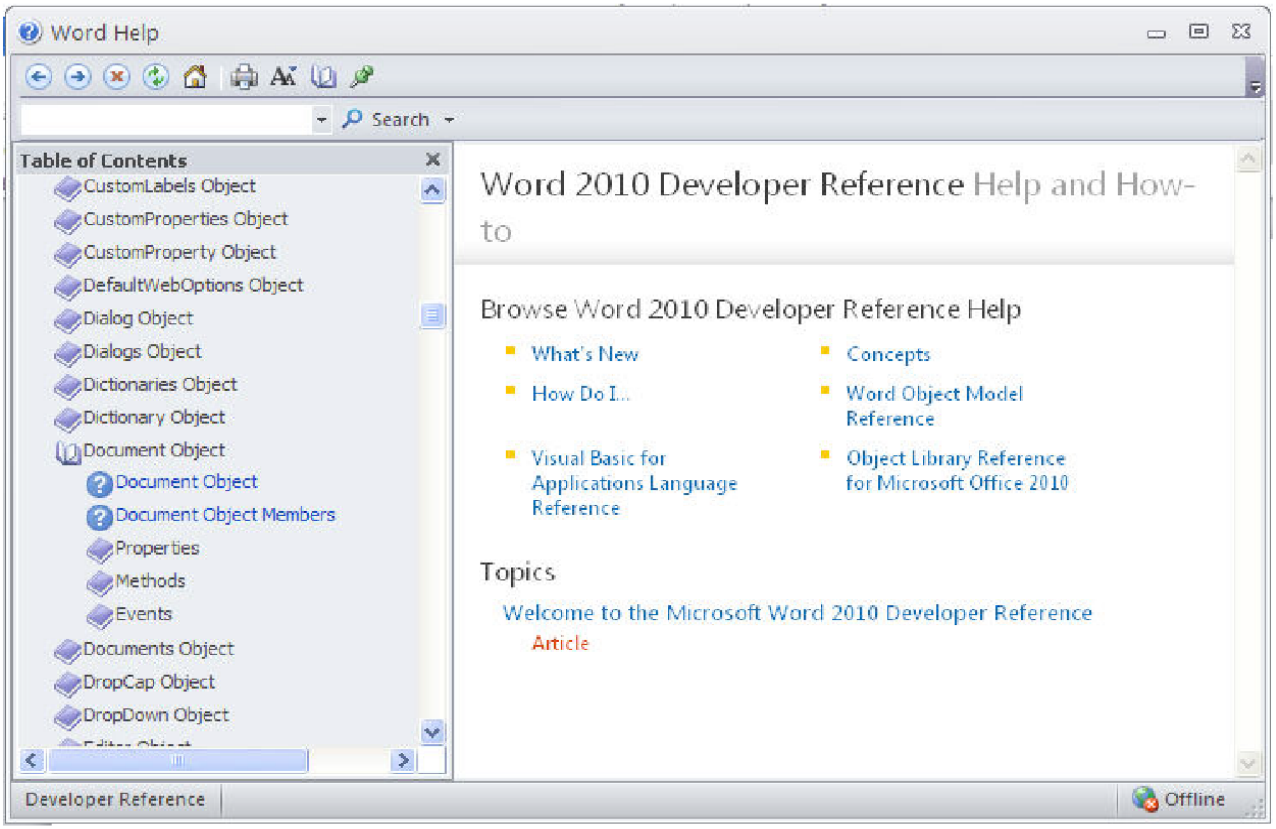

Figure 3. Using the Table of Contents helps to make the Developer Reference easier to use.

Turning on the Developer menu

When working on Automation code, you may want access to the various developer tools available inside Office. In particular, since one of the strategies for writing automation code is recording a macro (though see the next section of this document), getting into the Visual Basic Editor (VBE) to examine and edit your macros is handy.

In Office 2003, you can open the VBE from the main menu (Tools | Macro | Visual Basic Editor) or with the Alt+F11 keystroke. In Office 2007 and 2010, Alt+F11 still works, but the menu option is a little harder to find. On the View tab of the ribbon, choose Macros | View Macros. Then pick a macro and click Edit. (Of course, this means you have to have at least one macro to do this.)

However, there's an optional Developer tab for the ribbon that provides direct access to the VBE as well as to other development-related options like the Add-ins dialogs. The steps to turn on that tab are different for Office 2007 and Office 2010.

In Office 2007, click the Office button and click the "XX Options" button (where "XX" is the application name) at the bottom of the menu. In the dialog, click Popular. (It may already be selected as it's the first option.) Check Show Developer tab in the Ribbon. Click OK.

In Office 2010, choose File | Options. In the dialog, click Customize Customize Ribbon. In the right-hand listbox (Customize the Ribbon), find Developer and check it. Click OK.

What's wrong with recording a macro?

If you ask people how to automate a particular task in Office, often the response is "record a macro and see how it does it." The macro recorder creates VBA code for whatever you do interactively, so it does seem like a logical starting point for automation. (Note that Outlook doesn't have a macro recorder.) Once you know a few tricks for converting VBA to VFP, you can turn a macro into code you can run very fast.

So what's the problem? The code the macro recorder produces often isn't very good code. There are several issues.

Recorded macros do too much

First, the macro recorder code often is quite verbose, including commands and options you don't really need. Figuring out which lines or parameters are essential and which are overkill can be tricky. For example, recording a macro in Word to insert today's date results in the code in Listing 6.

Listing 6. Recorded macros tend to include a lot of extra code.

```
Selection.InsertDateTime DateTimeFormat:="dddd MMMM dd, yyyy", _
        InsertAsField:=False, DateLanguage:=wdEnglishUS, CalendarType:= _
        wdCalendarWestern, InsertAsFullWidth:=False
```

But most of the time, the VFP command in Listing 7 is sufficient to actually insert the date in the same format (assuming oWord is a reference to the Word object).

Listing 7. When translating from a macro to VFP Automation code, you can usually cut away a lot of the extra code.

```
oWord.Selection.InsertDateTime("dddd MMMM dd, yyyy")
```

When omitting parameters, however, you do have to be aware that the user may have changed settings from the defaults.

The verbosity of the macro recorder sometimes results in doing more than you actually intend. Because it records extra settings, if you translate directly, you may be including instructions you don't want.

For example, I recorded a macro in Excel to change the current cell to 12-point bold; it's shown in Listing 8.

Listing 8. Recorded macros may do more than you intend.

```
    Selection.Font.Bold = True
    With Selection.Font
        .Name = "Arial"
        .Size = 12
        .Strikethrough = False
```

```
        .Superscript = False
        .Subscript = False
        .OutlineFont = False
        .Shadow = False
        .Underline = xlUnderlineStyleNone
        .ColorIndex = xlAutomatic
    End With
```

Note that this macro doesn't just change the cell to 12-point bold. It sets the font to Arial and sets a bunch of other characteristics as well. If your intention is just to change whatever's there to 12-point bold, you need to strip away all that extra code. In this case, it's pretty easy to see, but in a more complex macro, or working in an area where you're not so familiar with the options, you need to be careful.

Macros use Selection

Another issue is that the macro recorder, of necessity, works with the Selection object. In Word and Excel, for several reasons, it's almost always better to work with Range objects. First, you can have multiple Range objects, while you can have only a single Selection at any time. Also, even when working in code, selecting a range has interactive effects. While you're unlikely to notice a slowdown with a few calls to the Select method, doing it over and over as is likely to be necessary in any serious automation code can slow things down. There's almost nothing you can do in Word and Excel with a Selection that you can't also do with a Range, and ranges give you more flexibility.

The macro in Listing 8 also demonstrates the issue of working with Selection rather than Range. Putting the two issues together, the Automation code I'd prefer to write is in Listing 9 (assuming I'd already created the range and stored it in oRange).

Listing 9. Working with Range objects is generally a better idea.

```
oRange.Font.Bold = .T.
oRange.Font.Size = 12
```

While you can create a range from the current selection (for example, oRange = oWord.Selection.Range()), in Automation code, your best option is to ignore the Selection object entirely and create ranges based on the content or structure of your document.

To see how much difference working with Ranges makes, I created a document by copying Table 2 from this white paper and saving it. Then, I wrote automation code to go through the lists in the third column of the table and bold the method names only. That is, I didn't want to bold the commas or the parenthetical version information. I wrote four versions of the code, all shown in Listing 10 (included in the session material as SelectionVs.Range.PRG). Version 1a is a brute force approach using the Selection object—inside a loop, it selects the entire cell, then selects a single word from the cell and processes it. Version 2a is the Range equivalent. Rather than reselecting the entire cell, each time through the loop, it stores the specified word as a range and operates on it. The b versions take advantage of the ability to collapse and expand a selection or range. Version 1b collapses the selection and expands it

by a single word each time through the loop. Version 2b does the same thing, but to the range containing the current word.

Listing 10. To test the speed of working with the selection object and range objects, this code tries four approaches to bold only a subset of words in a particular column on a table.

```
LPARAMETERS lVisible
   * Pass .T. to show Word as your work and to see the difference having
   * Word visible makes in these tests.

LOCAL oWord, oDoc, cFile

oWord = CREATEOBJECT("Word.Application")
IF m.lVisible
   oWord.Visible = .T.
ENDIF

* This code assumes the selected document has a table with at least three columns.
* It will fail if any other kind of file is selected.
cFile = GETFILE()

oDoc = oWord.Documents.Add(m.cFile)

LOCAL nStart, nEnd, nRow, nRowCount
LOCAL lInsideParens, nWordCount, nWord

* The task performed in each test is to go through
* column 3 of the table, select each non-parenthesized
* word and bold it.

* Test 1a: Use the selection object. Each time through the
*          loop to process a word, reselect the entire cell,
*          then pick out the word you want.

nStart = SECONDS()
oDoc.Tables[1].Select()

nRowCount = oWord.Selection.Rows.Count

FOR nRow = 2 TO m.nRowCount
   oDoc.Tables[1].Cell(m.nRow, 3).Select()
   nWordCount = oWord.Selection.Words.Count

   FOR nWord = 1 TO m.nWordCount-1
      * Stop before the last word, which is just punctuation
      * Select just one word
      oWord.Selection.Words[m.nWord].Select()
      DO CASE
      CASE oWord.Selection.Text = "("
         lInsideParens = .T.

      CASE oWord.Selection.Text = ")"
         lInsideParens = .F.
```

```
      CASE INLIST(oWord.Selection.Text, ",", CHR(13))
         * Do nothing

      CASE NOT m.lInsideParens
         * Bold it
         oWord.Selection.Font.Bold = .T.

      OTHERWISE
         * Nothing to do
      ENDCASE

      * Reselect the whole cell
      oDoc.Tables[1].Cell(m.nRow, 3).Select()
   ENDFOR
ENDFOR

nEnd = SECONDS()
?"1a: Using brute force selection, bolding non-paren words in column 3 takes", ;
 m.nEnd - m.nStart

* Close the document without saving changes
oDoc.Close(0)

* Test 1b: Use the selection object, but use collapse
*          and expand to modify selection
oDoc = oWord.Documents.Add(m.cFile)

nStart = SECONDS()
oDoc.Tables[1].Select()

nRowCount = oWord.Selection.Rows.Count

LOCAL lInsideParens
FOR nRow = 2 TO m.nRowCount
   oDoc.Tables[1].Cell(m.nRow, 3).Select()
   nWordCount = oWord.Selection.Words.Count

   oWord.Selection.Collapse(1) && Collapse to beginning

   FOR nWord = 1 TO m.nWordCount-1
      * Stop before the last word, which is just punctuation
      * Select just one word
      oWord.Selection.Expand(2)
      DO CASE
      CASE oWord.Selection.Text = "("
         lInsideParens = .T.

      CASE oWord.Selection.Text = ")"
         lInsideParens = .F.

      CASE INLIST(oWord.Selection.Text, ",", CHR(13))
         * Do nothing
```

```
      CASE NOT m.lInsideParens
         * Bold it
         oWord.Selection.Font.Bold = .T.

      OTHERWISE
         * Nothing to do
      ENDCASE

      * Collapse the selection
      oWord.Selection.Collapse(0)
   ENDFOR
ENDFOR

nEnd = SECONDS()
?"1b: Using selection and collapse/expand, " + ;
 "bolding non-paren words in column 3 takes", m.nEnd - m.nStart

* Close the document without saving changes
oDoc.Close(0)


* Test 2a: Use ranges and other object references
oDoc = oWord.Documents.Add(m.cFile)

nStart = SECONDS()
FOR nRow = 2 TO oDoc.Tables[1].Rows.Count
   oRange = oDoc.Tables[1].Cell(m.nRow, 3).Range()

   FOR nWord = 1 TO oRange.Words.Count-1
      oWordRange = oRange.Words[m.nWord]
      cWord = oWordRange.Text
      DO CASE
      CASE m.cWord = "("
         lInsideParens = .T.

      CASE m.cWord = ")"
         lInsideParens = .F.

      CASE INLIST(m.cWord, ",", CHR(13))
         * Do nothing

      CASE NOT m.lInsideParens
         * Bold it
         oWordRange.Font.Bold = .T.

      OTHERWISE
         * Nothing to do
      ENDCASE
   ENDFOR
ENDFOR
nEnd = SECONDS()

?"2a: Using ranges, bolding non-paren words in column 3 takes ", m.nEnd-m.nStart
```

```
* Close the document without saving changes
oDoc.close(0)

* Test 2b: Use ranges and collapse/expand
oDoc = oWord.Documents.Add(m.cFile)

nStart = SECONDS()
FOR nRow = 2 TO oDoc.Tables[1].Rows.Count
   oRange = oDoc.Tables[1].Cell(m.nRow, 3).Range()
   nWordCount = oRange.Words.Count

   oRange.Collapse(1)
   FOR nWord = 1 TO m.nWordCount-1
      oRange.Expand(2)
      cWord = oRange.Text
      DO CASE
      CASE m.cWord = "("
         lInsideParens = .T.

      CASE m.cWord = ")"
         lInsideParens = .F.

      CASE INLIST(m.cWord, ",", CHR(13))
         * Do nothing

      CASE NOT m.lInsideParens
         * Bold it
         oRange.Font.Bold = .T.

      OTHERWISE
         * Nothing to do
      ENDCASE

      oRange.Collapse(0)
   ENDFOR
ENDFOR
nEnd = SECONDS()

?"2b: Using ranges and collapse/Expand, " + ;
 "bolding non-paren words in column 3 takes ", m.nEnd-m.nStart

* Close the document without saving changes
oDoc.close(0)

* Close Word.
oWord.Quit()

RETURN
```

In my tests, version 2b using collapse and expand with ranges was the fastest. The specific times varied depending on the Word version. However, in most of my tests, both 2a and 2b took less than half the time of their 1a and 1b equivalents. 2a took about half again as long as 2b.

It also appears that there have been some performance improvement for ranges in Word 2010. In those tests, 2a took less than one-third the time of 1a and 2b used only a little more than half the time of 2a. In other words, in Word 2010, 2b was the winner by a lot.

Further tests also indicate that, at least for this task, Range's speed advantage over Selection increases dramatically, if there are multiple documents open. In my tests, with two or more documents open, the versions using Selection took about four times as long as with a single document open. The versions using Range took only fractionally longer with multiple documents open.

Macros may lead you down the wrong path

Finally, while the macro recorder doesn't record your actual keystrokes, it is oriented toward capturing the gist of interactive behavior. When you record a macro, you're likely to think in step-oriented terms. But often, there's a better way to perform a task.

Sometimes, that's doing the whole job at once. (Think of the difference between SCANning a table to REPLACE data in each record vs. using REPLACE ALL or SQL UPDATE.) For example, when recording a macro to format data in a table column, you might be inclined to format one cell and then move down, ready to work on the next cell. But both Word's tables and Excel's workbooks let you format an entire column at once. Be sure to think about whether a given task can be performed as a single operation instead of a repetitive one.

In addition, sometimes the most efficient way to do a task is to bring data into VFP and then write it back to the Office application. A macro will never send you in that direction.

I wrote one more version of the example shown in Listing 10. In this version, I copied the text from a cell into VFP, parsed it there, and then updated the document. This version is shown in Listing 11. It's faster even than version 2b, though the improvement is minor.

Listing 11. Sometimes, the most efficient way to perform an Automation task is to do a lot of the work in VFP.

```
* Test 3: Copy entire cell to VFP, parse there, and then
*         process as needed.

oDoc = oWord.Documents.Add(m.cFile)

nStart = SECONDS()
FOR nRow = 2 TO oDoc.Tables[1].Rows.Count
   oRange = oDoc.Tables[1].Cell(m.nRow, 3).Range()
   cCellContent = oRange.Text

   * Break into words
   nWordCount = ALINES(aWords, m.cCellContent, ",")

   * Word counts words differently, so need to keep track.
   LOCAL nWordWordNum, lHasParen
   nWordWordNum = 0
   lHasParen = .F.
```

```
   FOR nWord = 1 TO m.nWordCount
      cWord = aWords[m.nWord]
      nWordWordNum = m.nWordWordNum + 1

      * Check whether this one has parenthetical
      IF "(" $ m.cWord
         lHasParen = .t.
      ELSE
         lHasParen = .F.
      ENDIF

      * Highlight this one
      oRange.Words[m.nWordWordNum].Font.Bold = .T.

      * Update count of words in Word.
      IF m.lHasParen
         * Left paren, year, right paren with comma
         nWordWordNum = m.nWordWordNum + 3
      ELSE
         * Just the comma
         nWordWordNum = m.nWordWordNum + 1
      ENDIF

   ENDFOR

ENDFOR
nEnd = SECONDS()

?"3: Using VFP to parse, bolding non-paren words in column 3 takes ", ;
 m.nEnd-m.nStart
```

For some tasks, especially those where you need to accumulate a lot of data to send to the automation server, doing the work in VFP and sending it to the server only once can save a significant amount of time.

Automating email in Outlook

When I first started writing about automating Office, about a decade ago, automating Outlook was straightforward. You could work with the address book, create and send emails, and all worked very nicely. Then along came the "script kiddies" who figured out how to abuse those abilities, and filled the Internet with spam.

In SP2 for Outlook 2000, Microsoft introduced what many refer to as the "hell patch," which prevents total automation of a variety of activities in Outlook. All subsequent versions of Outlook include this patch, though Outlook 2007 and 2010 give you more control. (For an interesting perspective on this patch, read http://blogs.technet.com/b/kclemson/archive/2004/08/16/215499.aspx.)

The basic idea of the patch is to require human interaction when automating tasks in Outlook that might be abused. So, for example, when you programmatically add a recipient to an email, the message in Figure 4 appears. If you just answer Yes, the current action is allowed.

Checking the checkbox and choosing a length of time permits access to your address book for up to 10 minutes without being prompted again. Interestingly, you get the prompt even if you add a recipient by specifying the exact email address; you don't have to access your Contacts folder.

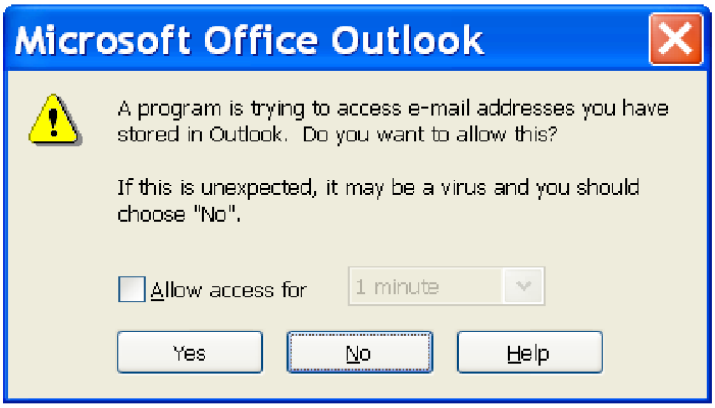

Figure 4. When you add a recipient to an Outlook email programmatically, you see this message.

When you send email programmatically, you see the message in Figure 5. This message applies to a number of operations that send email, not just the Send method. This dialog is designed so that you must wait five seconds before pressing or clicking the Yes button.

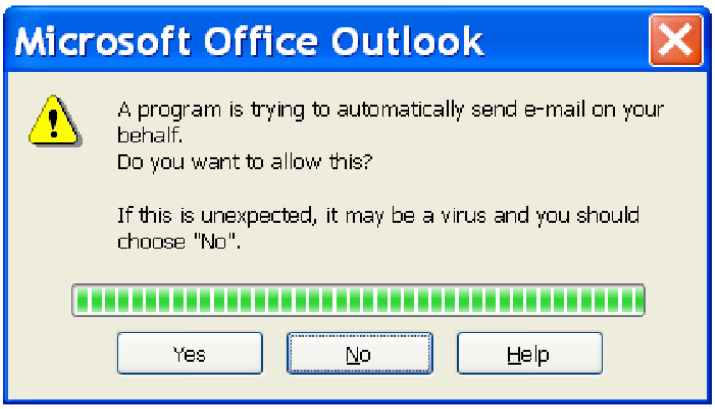

Figure 5. When you call the Send method for an Outlook MailItem, this prompt appears.

In Office 2007 and later, you can control whether these messages appear. The Trust Center gives you three options for programmatic access: Warn me about suspicious activity when my antivirus software is inactive or out-of-date (the default), Always warn me about suspicious activity, or Never warn me about suspicious activity. Choosing the Never warn me option prevents the dialogs from appearing. However, you're unlikely to be allowed to control that option on your client's machines.

Fortunately, programmers are resourceful and several other options emerged to avoid these prompts in your applications. The first option is to avoid automating Outlook for email entirely, and send email via another approach. While there are several good alternatives, those are a topic for another session.

If you need to use Outlook to send email from an application, you still have choices. The options vary widely in how they work.

Using VFPExMAPI

Craig Boyd has made something of an art of wrapping useful functionality to make it easily accessible in VFP. His solution to the "hell patch" is a library that wraps Extended MAPI capabilities in easy-to-use functions. Craig's solution addresses only the issue of creating and sending email. For example, it doesn't provide avoid prompts when accessing the address book other than to set an email recipient.

Nonetheless, if all you need to do is send emails and you'll always know the email addresses of the recipients or you'll know the Outlook contact names for the recipients, this library will do the trick. Since it's free both for development and distribution, that puts it at the top of list of solutions.

The program in Listing 12 (SendViaVFPExMAPI.PRG in the session materials) creates and sends an email, using the VFPExMAPI library.

Listing 12. Craig Boyd's VFPExMAPI library wraps Extended MAPI calls in easy-to-use functions.

```
LOCAL cBody

SET LIBRARY TO d:\fox\utils\vfpexmapi\vfpexmapi.fll
cBody = "When you run this example, no prompts appear. " + ;
        "But there's no way to use Outlook's Find capability."
EMCreateMessage("Demonstrate Outlook security", m.cBody, 1)
IF EMAddRecipient("Tamar Granor", 1)
   * Only send if we found the recipient
   EMSend()
ELSE
   WAIT WINDOW "Recipient not found. Not sending."
ENDIF

RETURN
```

Both the library itself and the documentation are found through Craig's blog. As of this writing, the most recent version of the library is linked to the post at http://www.sweetpotatosoftware.com/spsblog/2007/06/11/AdditionalExtendedMAPIFLLUpdate.aspx. The original version of the library and the full documentation is at http://www.sweetpotatosoftware.com/SPSBlog/PermaLink,guid,baccc84d-4d91-458b-a839-ad03662dfc34.aspx.

Using Outlook Redemption

Outlook Redemption is a COM library that sits between your application and Outlook. You create both Redemption objects and Outlook objects, make the Redemption objects point to the corresponding Outlook objects and then do all your manipulations on the Redemption objects.

The code in Listing 13 (SendViaOutlook.PRG in the session materials) demonstrates a very simple program to build and send an email message. It finds a particular recipient in the Outlook contacts, and sends a message to that person. In my tests, this code shows the message in Figure 4 three times (if I don't allow extended access) and the message in Figure 5 once. Listing 14 (SendWithRedemption.PRG in the session materials) shows the equivalent code using Redemption; when you run this version, you don't see any security prompts.

Listing 13. When you run this program with any version after Outlook 2000 SP1, you're prompted several times, and you have to wait five seconds to send the message.

```
LOCAL oOutlook, oNameSpace, oMailItem
LOCAL oContacts, oRecipient

oOutlook = CREATEOBJECT("Outlook.Application")
oNameSpace = oOutlook.GetNameSpace("MAPI")

oContacts = oNameSpace.GetDefaultFolder(10)
oRecipient = oContacts.Items.Find("[FirstName] = 'Tamar'")
WAIT WINDOW "Found recipient. Sending to " + oRecipient.Email1Address NOWAIT
IF NOT ISNULL(m.oRecipient)
   oMailItem = oOutlook.CreateItem(0)
   WITH oMailItem
      .Recipients.Add(m.oRecipient)
      .Subject = "Demonstrate Outlook security"
      .Body = "When you run this example, many prompts appear."
      .Send()
   ENDWITH
ENDIF
```

Listing 14. Using Redemption, you can avoid the prompts. Redemption uses a set of items that sit between your code and Outlook.

```
LOCAL oOutlook, oNameSpace, oMailItem
LOCAL oContacts, oRecipient

LOCAL oSafeItem, oSafeContact

oOutlook = CREATEOBJECT("Outlook.Application")
oNameSpace = oOutlook.GetNameSpace("MAPI")

oContacts = oNameSpace.GetDefaultFolder(10)
oSafeContact = CREATEOBJECT("Redemption.SafeContactItem")
oRecipient = oContacts.Items.Find("[FirstName] = 'Tamar'")
oSafeContact.Item = m.oRecipient
WAIT WINDOW "Found recipient. Sending to " + oSafeContact.Email1Address NOWAIT
IF NOT ISNULL(m.oRecipient)
   oSafeItem = CREATEOBJECT("Redemption.SafeMailItem")
   oMailItem = oOutlook.CreateItem(0)
   oSafeItem.Item = oMailItem
   WITH oSafeItem
      .Recipients.Add(m.oRecipient)
      .Subject = "Demonstrate Outlook security with Redemption"
```

```
      .Body = "When you run this example, no prompts appear."
      .Send()
   ENDWITH
ENDIF
```

Redemption does include one strange behavior. When you issue oSafeItem.Send(), the message lands in Outlook's Drafts folder rather than in the Outbox. However, it is still sent on the next Send/Receive pass.

The developer version of Redemption is free. However, to distribute it with your apps, you need the distributable version, which costs $199.99. More information and the download are at http://www.dimastr.com/redemption/.

Using Express ClickYes

Express ClickYes from Context Magic takes an entirely different approach to the problem. You install it, it runs in the background, and when the security dialogs appear, it clicks Yes for you. That means that the dialogs still appear, and Express ClickYes has to wait through the delay on the "A program is trying to automatically send mail on your behalf" dialog.

Express ClickYes has a presence in the system tray; from there, you can suspend or resume it or turn it off entirely.

Express ClickYes is free, but is documented to work only through Outlook 2003. You can download it from http://www.contextmagic.com/express-clickyes/.

Using ClickYes Pro

Context Magic offers another approach to the problem with ClickYes Pro 2010. This product provides you with a console to specify what applications can automate Outlook without seeing the security dialogs. Figure 6 shows the console after adding VFP 9.

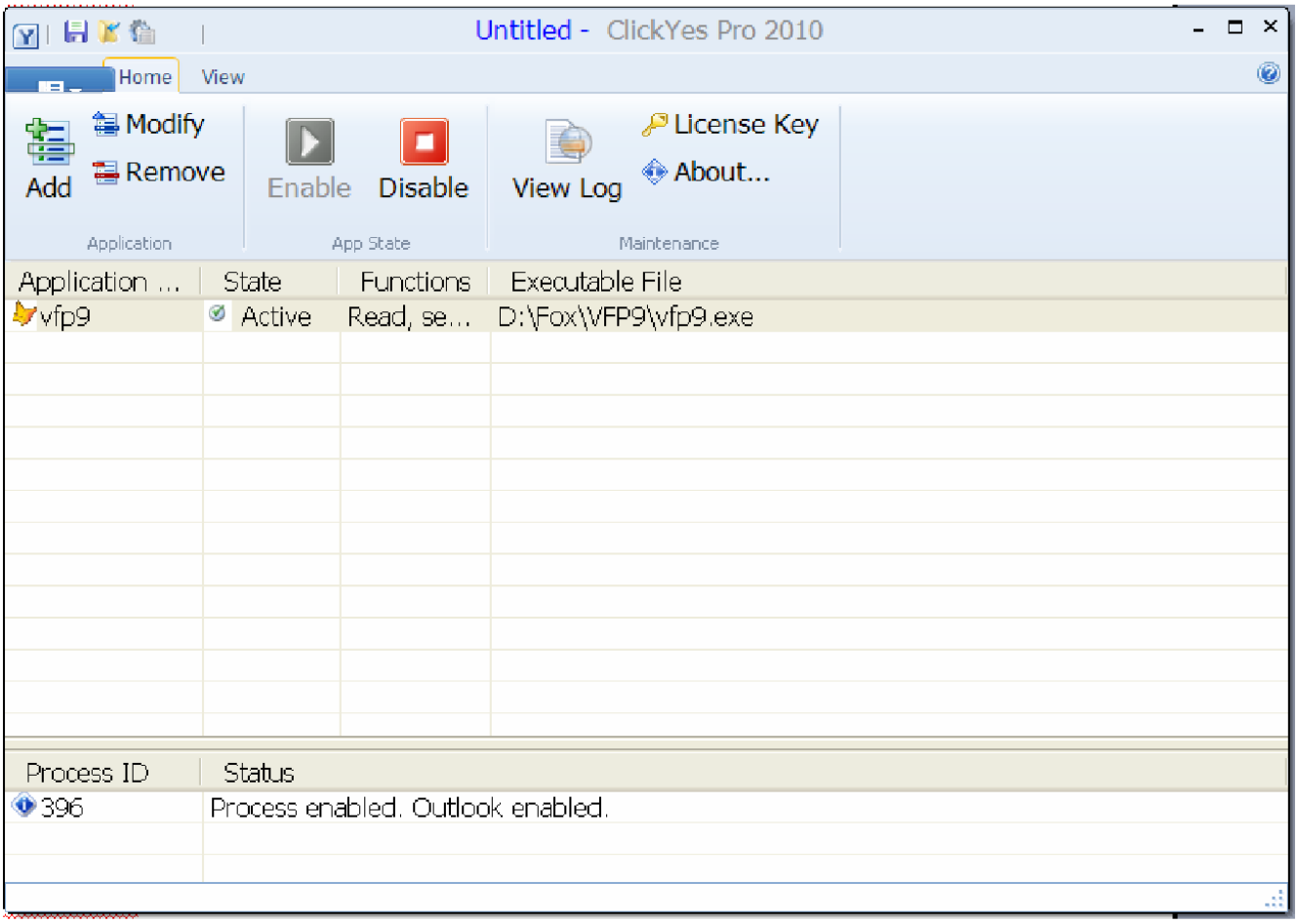

Figure 6. ClickYes Pro 2010 allows you to specify what applications can automate Outlook.

For each application, you can specify what operations it's permitted to automate, as shown in Figure 7. So to allow your application to automate Outlook, you add the EXE to the ClickYes Pro console and specify the appropriate operations.

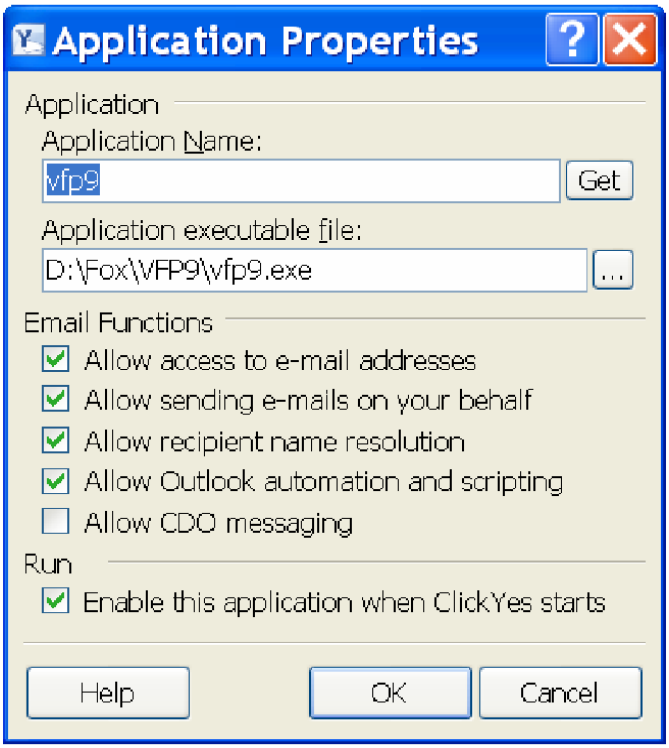

Figure 7. ClickYes Pro lets you control each application in a granular way.

There are a couple of issues in using ClickYes Pro. First, in situations where you can't control the user's Trust Center settings, you also may not be allowed to install an application that provides a security hole.

In addition, there's no way to add applications to the permitted list programmatically (which makes sense since that would allow malware to add itself). That means you can't do something in your application's installation routine to set it up in ClickYes Pro. Context Magic offers a work-around for this issue. You can save a profile from ClickYes Pro that includes the application you want; Context Magic will then create a customized setup for you that includes that profile. (However, according to Context Magic, such a setup would require end-users to install your application to the path you specify in the custom setup.)

ClickYes Pro has a 30-day free trial. A personal use license is $19.95. A commercial license for a single user is $39.90; there are discounts for purchasing multiple licenses. There's also a server edition. Full details of editions and pricing are available at http://www.contextmagic.com/express-clickyes/.

Responding to Office events

While most Automation code simply issues commands to the various Office applications to create, edit, parse or print documents, at times, you need to respond to things that happen in the server application. For example, when automating Outlook, you might present the user with the bare bones of a mail message and allow him to complete it. When the user saves the message, you want your application to do something. In one application I worked on, all

documents were tracked in a VFP database and each time a user finished working with a document, we logged the user's name and the time.

Just like VFP, the Office applications have a set of events that fire when things happen. There are two approaches you can use with VFP to respond to Office events. The first is useful when you know that your VFP application will be running while the user is working in Office. The second can be used whether or not your VFP application is running.

Binding server events with EventHandler()

The first solution uses the EventHandler() function added in VFP 7. To use this approach, you need to create a class that implements the appropriate interface of the server application. An interface is a set of methods defined by a server. Implementing an interface means providing code for those methods. It's similar to inheriting from a class, but the class that implements an interface doesn't execute any code from the original class. In addition, you can implement an interface written in a different language. The IMPLEMENTS keyword was added to DEFINE CLASS in VFP 7.

How do you know what interface you need to implement? VFP's Object Browser lets you explore the interfaces of an Automation server. Open the appropriate type library and expand the Interfaces node to see all the interfaces supported. The type library to look at varies with the Office version. In Office 2003, look for Microsoft <product name> 11.0 Type Library. In Office 2007 and 2010, look for Microsoft.Office.Interop.<product name>, Version=<version>, where <version> is 12.0 for Office 2007 and 14.0 for Office 2010. Figure 8 has the type libraries for Excel, Outlook, PowerPoint and Word 2010 selected.

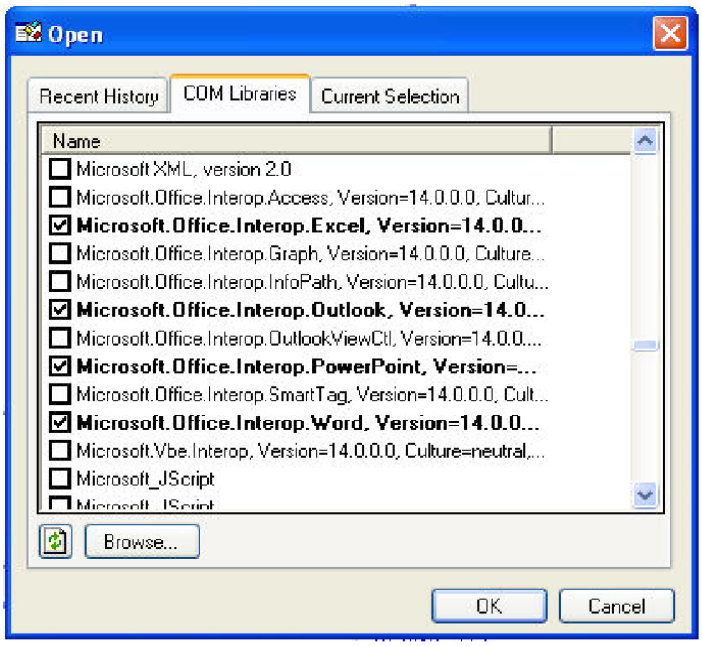

Figure 8. These choices will let you see interfaces (and other information) for the major Office 2010 apps in the Object Browser.

To respond to user activity in the Office apps, you're looking for an interface that includes events. So when you expand the Interfaces node, you're typically looking for one with "events" in its name. Of course, since each of the Office apps was originally created by a different team, no rule is absolute; the principal event interface for PowerPoint doesn't include the word "events." Table 1 shows the main interface that contains events you can respond to for the Office servers discussed in this paper. (Some of the applications have more than one interface containing events. The application where you're most likely to want to implement an additional interface is Outlook, which has separate event interfaces for its explorers and inspectors, the user interface components that represent folders and items, respectively.)

The set of events for which you can write code has changed over time. Some of the teams have added newer interfaces to incorporate the changes, while others simply keep adding to the original interface. Even when new interfaces have been added, sometimes, additional methods get added to those interfaces. For example, the most up-to-date interface for Word is ApplicationEvents4, for example. That's been the case at least since Word 2003, and additional methods were added in Word 2007 and Word 2007.

The list in Table 2 includes all the methods through Office 2010. Methods added in Office 2007 or Office 2010 are marked appropriately.

Table 2. The various Office servers each support one or more interfaces that let you respond to events. The principal interface for each application is shown here.

Server	Interface	Methods
Excel	AppEvents	AfterCalculate (2007), NewWorkbook, ProtectedViewWindowActivate (2010), ProtectedViewWindowBeforeClose (2010), ProtectedViewWindowBeforeEdit (2010), ProtectedViewWindowBeforeEdit (2010), ProtectedViewWindowDeactivate (2010), ProtectedViewWindowResize (2010), SheetActivate, SheetBeforeDoubleClick, SheetBeforeRightClick, SheetCalculate, SheetChange, SheetDeactivate, SheetFollowHyperlink, SheetPivotTableAfterValueChange (2010), SheetPivotTableBeforeAllocateChanges (2010), SheetPivotTableBeforeCommitChanges (2010), SheetPivotTableBeforeDiscardChanges (2010), SheetPivotTableUpdate, SheetSelectionChange, WindowActivate, WindowDeactivate, WindowResize, WorkbookActivate, WorkbookAddinInstall, WorkbookAddinUninstall, WorkbookAfterSave (2010), WorkbookAfterXmlExport, WorkbookAfterXmlImport, WorkbookBeforeClose, WorkbookBeforePrint, WorkbookBeforeSave, WorkbookBeforeXmlExport, WorkbookBeforeXmlImport, WorkbookDeactivate, WorkbookNewChart (2010), WorkbookNewSheet, WorkbookOpen, WorkbookPivotTableCloseConnection, WorkbookPivotTableOpenConnection, WorkbookRowsetComplete (2007), WorkbookSync

Server	Interface	Methods
Outlook	ApplicationEvents_11	AdvancedSearchComplete, AdvancedSearchStopped, AttachmentContextMenuDisplay (2007), BeforeFolderSharingDialog (2007), ContextMenuClose (2007), FolderContextMenuDisplay (2007), ItemContextMenuDisplay (2007), ItemLoad (2007), ItemSend, MAPILogonComplete, NewMail, NewMailEx, OptionsPagesAdd, Quit, Reminder, ShortcutContextMenuDisplay (2007), Startup, StoreContextMenuDisplay (2007), ViewContextMenuDisplay (2007)
PowerPoint	EApplication	AfterNewPresentation, AfterPresentationOpen, ColorSchemeChanged, NewPresentation, PresentationBeforeSave, PresentationClose, PresentationCloseFinal (2010), PresentationNewSlide, PresentationOpen, PresentationPrint, PresentationSave, PresentationSync, ProtectedViewWindowActivate (2010), ProtectedViewWindowBeforeClose (2010), ProtectedViewWindowBeforeEdit (2010), ProtectedViewWindowDeactivate (2010), ProtectedViewWindowOpen (2010), SlideSelectionChanged, SlideShowBegin, SlideShowEnd, SlideShowNextBuild, SlideShowNextClick, SlideShowNextSlide, SlideShowOnNext (2007), SlideShowOnPrevious (2007), WindowActivate, WindowBeforeDoubleClick, WindowBeforeRightClick, WindowDeactivate, WindowSelectionChange
Word	ApplicationEvents4	DocumentBeforeClose, DocumentBeforePrint, DocumentBeforeSave, DocumentChange, DocumentOpen, DocumentSync, EPostageInsertEx, EPostagePropertyDialog, EPostageInsert, MailMergeAfterMerge, MailMergeAfterRecordMerge, MailMergeBeforeMerge, MailMergeBeforeRecordMerge, MailMergeDataSourceLoad, MailMergeDataSourceValidate, MailMergeDataSourceValidate2 (2007), MailMergeWizardSendToCustom, MailMergeWizardStateChange, NewDocument, ProtectedViewWindowActivate (2010), ProtectedViewWindowBeforeClose (2010), ProtectedViewWindowBeforeEdit (2010), ProtectedViewWindowDeactivate (2010), ProtectedViewWindowOpen (2010), ProtectedViewWindowSize (2010), Quit, WindowActivate, WindowBeforeDoubleClick, WindowBeforeRightClick, WindowDeactivate, WindowSelectionChange, WindowSize, XMLSelectionChange, XMLValidationError

Once you're looking at the right interface in the Object Browser, you can expand its Methods node to see the supported events. Figure 9 shows the Word ApplicationEvents4 interface of Word 2010 expanded in the Object Browser.

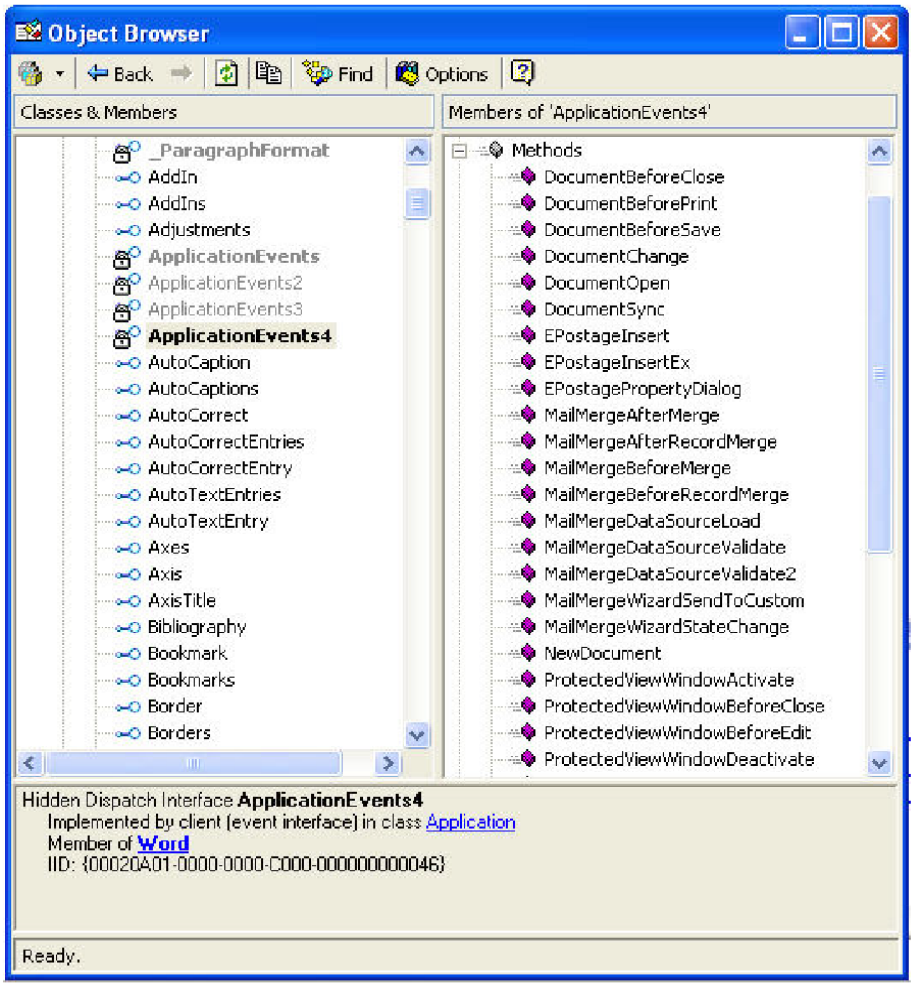

Figure 9. The Object Browser lets you explore the interfaces supported by a server. You can drag an interface to a code window to create a class that implements the interface.

The Object Browser does more than just let you find out what interfaces are available and what methods they support. Drag an interface to an editing window (MODIFY COMMAND) and a class is defined that implements that interface. For example, if you drag Word's ApplicationEvents4 interface to a code window, the code in Listing 15 is generated. (The code has been reformatted as needed to fit the page.)

Listing 15. This code is generated by dragging the ApplicationEvents4 interface from Word 2010 out of the Object Browser into a code window.

```
x=NEWOBJECT("myclass")

DEFINE CLASS myclass AS session OLEPUBLIC

   IMPLEMENTS ApplicationEvents4 ;
      IN "c:\program files\microsoft office\office14\msword.olb"

   PROCEDURE ApplicationEvents4_Quit() AS VOID
   * add user code here
   ENDPROC
```

```
PROCEDURE ApplicationEvents4_DocumentChange() AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_DocumentOpen(Doc AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_DocumentBeforeClose(Doc AS VARIANT, ;
   Cancel AS LOGICAL @) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_DocumentBeforePrint(Doc AS VARIANT, ;
   Cancel AS LOGICAL @) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_DocumentBeforeSave(Doc AS VARIANT, ;
   SaveAsUI AS LOGICAL, Cancel AS LOGICAL @) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_NewDocument(Doc AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_WindowActivate(Doc AS VARIANT, Wn AS VARIANT) ;
   AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_WindowDeactivate(Doc AS VARIANT, Wn AS VARIANT) ;
   AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_WindowSelectionChange(Sel AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_WindowBeforeRightClick(Sel AS VARIANT, ;
   Cancel AS LOGICAL @) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_WindowBeforeDoubleClick(Sel AS VARIANT, ;
   Cancel AS LOGICAL @) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_EPostagePropertyDialog(Doc AS VARIANT) AS VOID
* add user code here
```

```
ENDPROC

PROCEDURE ApplicationEvents4_EPostageInsert(Doc AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_MailMergeAfterMerge(Doc AS VARIANT, ;
   DocResult AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_MailMergeAfterRecordMerge(Doc AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_MailMergeBeforeMerge(Doc AS VARIANT, ;
   StartRecord AS Number, EndRecord AS Number, Cancel AS LOGICAL @) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_MailMergeBeforeRecordMerge(Doc AS VARIANT, ;
   Cancel AS LOGICAL @) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_MailMergeDataSourceLoad(Doc AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_MailMergeDataSourceValidate(Doc AS VARIANT, ;
   Handled AS LOGICAL) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_MailMergeWizardSendToCustom(Doc AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_MailMergeWizardStateChange(Doc AS VARIANT, ;
   FromState AS Integer, ToState AS Integer, Handled AS LOGICAL) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_WindowSize(Doc AS VARIANT, Wn AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_XMLSelectionChange(Sel AS VARIANT, ;
   OldXMLNode AS VARIANT, NewXMLNode AS VARIANT, Reason AS Number) AS VOID
* add user code here
ENDPROC

PROCEDURE ApplicationEvents4_XMLValidationError(XMLNode AS VARIANT) AS VOID
* add user code here
```

```
   ENDPROC

   PROCEDURE ApplicationEvents4_DocumentSync(Doc AS VARIANT, ;
      SyncEventType AS VARIANT) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents4_EPostageInsertEx(Doc AS VARIANT, ;
      cpDeliveryAddrStart AS Integer, cpDeliveryAddrEnd AS Integer, ;
      cpReturnAddrStart AS Integer, cpReturnAddrEnd AS Integer, ;
      xaWidth AS Integer, yaHeight AS Integer, ;
      bstrPrinterName AS STRING, bstrPaperFeed AS STRING, ;
      fPrint AS LOGICAL, fCancel AS LOGICAL @) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents4_MailMergeDataSourceValidate2(Doc AS VARIANT, ;
      Handled AS LOGICAL @) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents4_ProtectedViewWindowOpen(PvWindow AS VARIANT) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents4_ProtectedViewWindowBeforeEdit(PvWindow AS VARIANT, ;
      Cancel AS LOGICAL @) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents4_ProtectedViewWindowBeforeClose( ;
      PvWindow AS VARIANT, CloseReason AS Integer, Cancel AS LOGICAL @) ;
      AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents4_ProtectedViewWindowSize(PvWindow AS VARIANT) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents4_ProtectedViewWindowActivate(PvWindow AS VARIANT) ;
      AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents4_ProtectedViewWindowDeactivate(PvWindow AS VARIANT) ;
      AS VOID
   * add user code here
   ENDPROC

ENDDEFINE
```

To create an event handler, put code in the methods for the events you want to handle. Do not delete the other methods; just leave them empty. For a class to implement an interface, every method of the interface must be included.

Fortunately, it's not a problem for the class that implements an interface to contain extra methods. So, if you need to write event handler code to work with multiple versions of an Office application, implement the interface for the most recent version. If you use that class against an older version, the events that don't exist in the older version are simply ignored.

It's a good idea to change the reference to the class in the IMPLEMENTS line. The Object Browser generates a reference to the type library, including the path. If your event handler will be used on more than one machine, the path may be wrong. It's better to use the application's ProgID (such as "Word.Application"), as in Listing 16.

Listing 16. It's a good idea to replace the type library reference in the IMPLEMENTS line with the application's ProgID.

```
IMPLEMENTS ApplicationEvents4 IN "Word.Application"
```

You probably want to change the name of the class, as well.

Once you've defined the event handler class, you can bind it to an instance of the server using the EventHandler() function. Listing 17 binds the class shown above to a Word instance.

Listing 17. To have your code respond to Office events, use EVENTHANDLER() to bind the automation server to your implementation.

```
oWord = CreateObject("Word.Application")
oHandler = CreateObject("MyClass")
EVENTHANDLER( oWord, oHandler)
```

The bindings last as long as the objects involved stay in scope. You can unbind the handler from the server sooner by calling EventHandler() again and passing .T. for the optional third parameter.

The session materials include a class (OfficeEventHandler.PRG) that implements the principal interfaces of Word, Excel and PowerPoint. For each, it logs opening, closing, saving and creating a new document to a table. The materials also include a form (OfficeEvents.SCX) to open the servers and display the log.

Binding server events with an add-in

The second approach is more complicated. It uses VBA code to create what's known as an *add-in* and a VFP COM object to be used by the add-in. It's useful when you can't be sure that VFP will be running at the time the server event occurs. (You can also create add-ins for Office with Visual Studio .NET; such add-ins are called COM Add-ins. For information about building COM Add-ins for Office using C#, see http://support.microsoft.com/kb/302901. For

information about building COM Add-ins for Office using VB.NET, see http://support.microsoft.com/kb/302896.)

An add-in is a special document in Word, Excel or PowerPoint that contains code. (While Outlook also supports add-ins, the mechanism is different enough that it's not discussed here.) Add-ins can be loaded interactively or via Automation.

An event handler add-in needs two components. There's a class that contains the actual event handling code, plus separate code to connect the event handler class to the application object (just as EventHandler() does in the VFP example). Both of these pieces are written in Visual Basic for Applications (VBA) using the Visual Basic Editor (VBE), which is available in all of the Office applications. While the particulars, especially the event methods available, vary from one Office server to the next, the general structure is the same for each.

Creating an add-in

To create an add-in, create a new, blank document in the server application and then open the VBE (using one of the methods discussed in "Turning on the Developer menu" earlier in this document). The first step in the VBE is to create the event handler class. To do so, make sure the Project Explorer is open. It's normally docked on the left-hand side of the VBE underneath the menu. If it's not open, use View | Project Explorer to open it. Right-click on the project for your document and choose Insert | Class Module. Figure 10 shows the Project Explorer in Excel after adding the new class module.

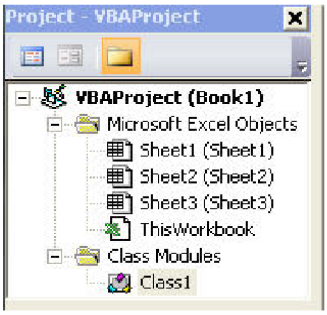

Figure 10. The Project Explorer provides an overview of your VBA project. Here, the class module for the event handler has been added.

When you add the class module, a code window opens to hold code for that module. That's where the event handler code goes.

Before starting to write code, it's a good idea to rename the event class. Click on the new class module in the Project Explorer. Then, in the Properties window (normally docked below the Project Explorer), click into the Name property and give your class a meaningful name, like EventHandler.

To create an event handler for one of the Office servers, you need an object reference to the application object. To add one, declare a property to hold the reference. Switch to the code

window, making sure the dropdowns show "(General)" and "(Declarations)," respectively, and type code like that in Listing 18, using an appropriate variable name, of course.

Listing 18. You need to declare a variable to hold a reference to the application.

```
Public WithEvents XLApp As Application
```

The WithEvents keyword lets the object respond to events. Once you've completed this line and pressed Enter, the dropdowns at the top of the code window change. The left-hand dropdown, which shows the available objects, now includes your application property as in Figure 11. When you choose the property in the left-hand dropdown, the right-hand dropdown (Figure 12) is populated with the events to which your code can respond.

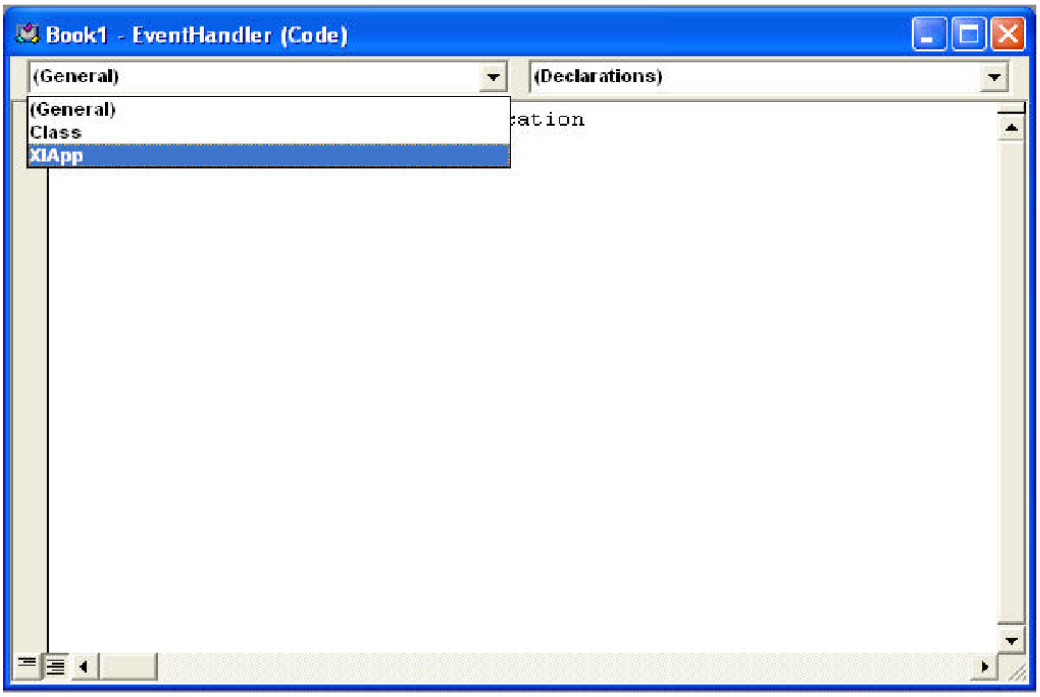

Figure 11. Once you declare a variable to hold the application object, that object is accessible in the code window.

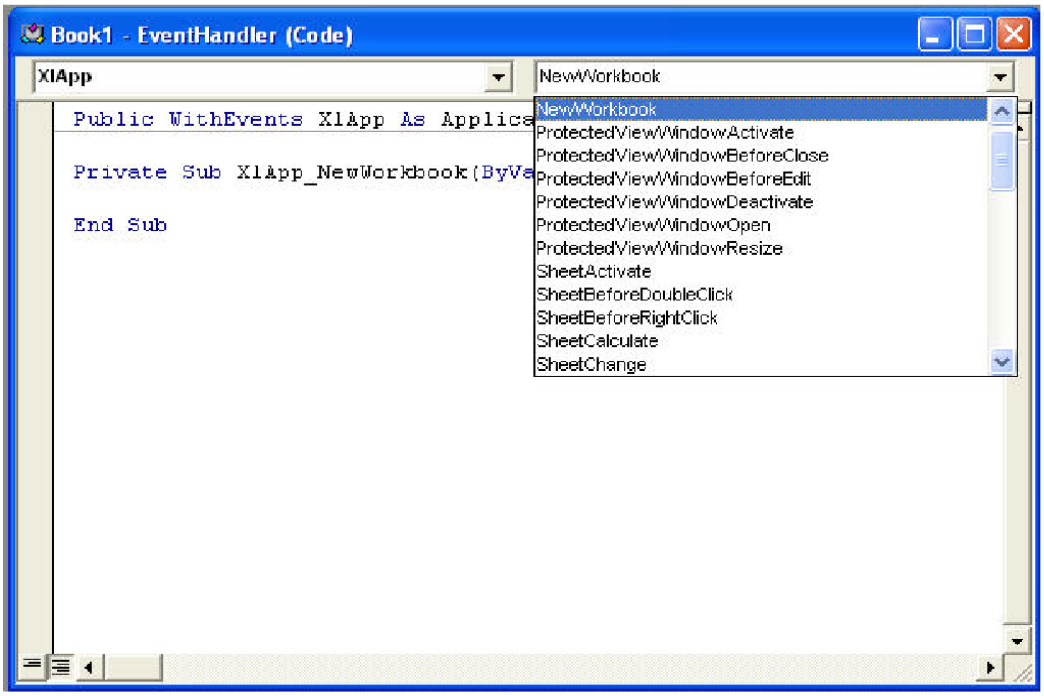

Figure 12. When you choose the application property from the objects dropdown, the right-hand dropdown shows the list of available events.

When you choose an event from the right-hand dropdown, stub code for that event is inserted into your code window. (In Figure 12, you can see the stub for the NewWorkbook event.)

Choose each event for which you want to write code from the events dropdown and add the VBA code that should run in response to that event. (Unlike the previous approach using IMPLEMENTS, you do not need to include all event methods, only those for which you have code.) For example, this code for an Excel add-in responds to the NewWorkbook and WorkbookBeforeClose events, in both cases displaying a message box that tells the user what's going on.

```
Public WithEvents XlApp As Application

Private Sub XlApp_NewWorkbook(ByVal Wb As Workbook)
MsgBox ("About to open a new workbook")
End Sub

Private Sub XlApp_WorkbookBeforeClose(ByVal Wb As Workbook, Cancel As Boolean)
MsgBox ("About to close workbook " & Wb.FullName)
End Sub
```

Next, you need to connect the actual application object to the defined object reference property (XLApp, in the example). Add a module (not a class module) to the project by right-clicking on the project in the Project Explorer and choosing Insert | Module. As before, rename the new module by choosing it in the Project Explorer and changing the Name property in the Properties window. (I call the one for Excel "CatchXL".)

In this new module, declare a variable as a new instance of the event handler class you just created:

```
Dim XLHandler As New EventHandler
```

Add a method whose sole purpose is to connect the application property of the event handler to the application object. I call it HandleEvents. Here's the version of the code I use for Excel:

```
Sub HandleEvents()
Set XLHandler.XLApp = Application
End Sub
```

Next, we need a way to make this code run every time the add-in is loaded. While each of the applications has an event that fires automatically when a document or the application is opened, those events behave differently in different circumstances. (For example, Word doesn't run its AutoExec method when Word was started by automation.) So it's better to use code to explicitly run HandleEvents after loading the add-in. That code is included in "Loading the add-in" below.

The document now has everything it needs to handle events. Next, save it as an add-in. The technique varies with the application. In all cases, though, switch back to the main application (rather than the VBE) before choosing Save As.

For Word, simply save the document as a template; in Word 2007 and later, you'll need to make it a "macro-enabled template." If you store it in Word's startup folder (as specified in File Locations in Word's Options dialog), the add-in loads automatically each time you start Word. Alternatively, you can store it elsewhere and load it manually or via Automation each time you want it.

In Excel and PowerPoint, you need to save the document twice because opening an add-in directly for editing is difficult or impossible. First, save it as a (macro-enabled) workbook or presentation, then use Save As to save it again as an add-in. If you need to modify it later, open the native format version, make the changes, then save it both in the native format and as an add-in.

To save as an add-in in Excel, choose Excel add-in (the extension is .XLA for Excel 2003, and .XLAM for later versions) from the "Save as type:" dropdown. Once you make this choice, the dialog switches to the AddIns directory for the current user. Saving your add-in in that directory puts it on the list of add-ins available from Excel's Add-Ins dialog, but doesn't automatically enable it. You have to enable it in that dialog or via Automation. If you save it elsewhere, you can load it using the Add-Ins dialog or via Automation. (If you save it in the user's XLStart directory, it's loaded automatically.)

Saving an add-in in PowerPoint is pretty much the same as for Excel. Once you've saved the presentation, use File | Save As and choose PowerPoint Add-In (.PPA for PowerPoint 2003; .PPAM for 2007 and 2010). Again, the dialog switches to the AddIns directory. However,

saving your add-in there doesn't have any special advantages except that the dialog for loading an add-in defaults to looking in that directory. Regardless of where it's saved, you can load an add-in through the Add-Ins dialog or via Automation.

Loading the add-in

There are several ways to get your add-in running. In general, it's a two-step process. First, add-ins need to be "registered" to make the application aware of them. Once registered, an add-in can be loaded to get it running. In some cases, you can do both steps at once.

You get the most control by using Automation code to load your add-ins. Each of the servers has an AddIns collection with an Add method. The exact behavior of the Add method varies, however, and, in Excel, using AddIns.Add isn't the best choice.

The Add method of Word's AddIns collection both registers and loads the add-in. All you need to do afterward is run the HandleEvents method to create the connection to the application object, as in Listing 19.

Listing 19. A few lines of VFP code can register and load your Word add-in.

```
oWord = CreateObject("Word.Application")
oWord.AddIns.Add("WordEventHandler.DOT") && Add path
oWord.Run("HandleEvents")
```

In PowerPoint, the Add method registers the add-in, and you set the Loaded property to True to load it. Then run the HandleEvents method to create the connection.

Listing 20. Registering and loading the add-in in PowerPoint requires one additional step.

```
oPPT = CreateObject("PowerPoint.Application")
oAddIn = oPPT.AddIns.Add("PPTEventHandler") && Add path
oAddIn.Loaded = .T.
oPPT.Run("HandleEvents")
```

Excel doesn't let you register and load an add-in with AddIns.Add unless there's an open workbook. In addition, when you leave an add-in loaded when Excel closes, the next time you run Excel, the add-in shows as installed, but doesn't actually get properly loaded. A better approach with Excel is to open the add-in with the Workbooks.Open method. That registers and loads the add-in cleanly. As with the others, you then need to run the HandleEvents method.

Listing 21. In Excel, the easiest way to registering and loading an add-in is to open it as a workbook.

```
oXL = CreateObject("Excel.Application")
WITH oXL
   * Add path in next line
   oWorkbook = .Workbooks.Open("ExcelEventHandler.XLAM")
   .Run("HandleEvents")
ENDWITH
```

In the more recent versions of the Office apps, when you open your add-in for editing, you're likely to get some kind of security warning and need to click a button to allow you to actually see and use the add-in code. This is part of Microsoft's increased security awareness. Other than the annoyance of an extra step, it doesn't cause any real problems.

Add-ins for Word, Excel and PowerPoint that log creation, opening, saving and closing of documents to a VFP table are included in the session materials. The add-ins are provided in both Word 2003 (.DOT, .XLS/.XLA, .PPT/.PPA) and Word 2007/2010 format (.DOTM, .XLSM/.XLAM, .PPTM/.PPAM).

Talking to VFP via a COM object

The final piece of this approach is a way to communicate with FoxPro. Depending what you want, there are a couple of possibilities. The VBA code can use ADO to update VFP data directly. Alternatively, the VBA code can instantiate a VFP COM object. We'll look at the second method here.

To create a COM object in VFP, you need a class defined as OLEPUBLIC, and a project containing the class. In the class, put methods for whatever should be done when the Office events fire. My experience is that it's best to have very granular methods, each performing a single, fairly simple task. In one application, I used one method to update a document log to indicate that the specified document was edited. That application used a semaphore locking scheme for the documents (to allow only one user to edit a document at a time), so another method released the semaphore lock when the document was closed.

The Session class is designed to be used for COM objects; it's extremely lightweight. The downside is that Session classes can't be created in the Class Designer; you have to write code in a PRG file.

One thing you probably want in all COM objects is some kind of error handler. Since most COM objects can't have a user interface, a good way to deal with errors is to log all relevant information to a text file (or a table). In the example here, the Error method handles any errors. If your COM object calls on other objects or outside code, and you want unified error handling, you're better off using ON ERROR to set up a global error handler or setting all the objects up to pass errors to the error handler of a single object. (Of course, an ON ERROR handler won't be called by any object that has its own error handling code.)

Listing 22 shows a simple class (OfficeResponder.PRG in the session materials) that has one method to update a log table. The Init method ensures that the log table exists.

Listing 22. This class, designed to be used as a COM object, updates a log table. It creates the table, if it doesn't already exist.

```
DEFINE CLASS OfficeResponder AS Session OLEPUBLIC
* COM server to be called in response to Office events.

PROTECTED cLogTable, cLogPath, cLogFullPath
cLogTable = "OfficeLogFromAddIn.DBF"
```

```
PROTECTED PROCEDURE Init
* Set up

This.cLogPath = SYS(2023)
This.cLogFullPath = FORCEPATH(This.cLogTable, ;
                              This.cLogPath)

* Make sure the log table exists
IF NOT FILE(This.cLogFullPath)
    CREATE TABLE (This.cLogFullPath) ;
        (cDocument C(150), cAction C(12), tModified T)
    CLOSE TABLES
ENDIF

SET EXCLUSIVE OFF
RETURN

ENDPROC

PROCEDURE UpdateLog( cDocument as String, cAction as String) as Boolean
* Update the activity log

IF VARTYPE(cDocument) <> "C" OR EMPTY(cDocument)
    ERROR 11
    RETURN .F.
ENDIF

IF VARTYPE(cAction) <> "C" OR EMPTY(cAction)
   ERROR 11
   RETURN .f.
ENDIF

INSERT INTO (This.cLogFullPath) ;
   VALUES (m.cDocument, m.cAction, DATETIME())

RETURN .t.

ENDPROC

PROCEDURE Error(nError, cMethod, nLine)

LOCAL lcErrorMsg, lcFileName

lcErrorMsg = "Error " + TRANSFORM(nError) + SPACE(1) + ;
             CHR(13) + CHR(10) + ;
             "Method " + cMethod + SPACE(1) + ;
             CHR(13) + CHR(10) + ;
             "Line " + TRANSFORM(nLine) + SPACE(1) + ;
             CHR(13) + CHR(10)+ ;
             "At " + TRANSFORM(DATETIME())

**********************************************
* Dump an error log into the specified directory
```

```
*********************************************
lcFileName = FORCEPATH("OfficeResponder.ERR", ;
                       This.cLogPath)
STRTOFILE(lcErrorMsg, lcFileName, .T.)
LIST MEMORY TO FILE (lcFileName) ADDITIVE noconsole
LIST STATUS TO FILE (lcFileName)  ADDITIVE noconsole
RETURN
ENDPROC

ENDDEFINE
```

To turn the class into a COM object, create a project (also called OfficeResponder in the example and included in the session materials), add the class to it, and build a .DLL COM server. The resulting .DLL file contains the COM object.

To use the COM object from the Office event handler code (that is, in VBA code), declare an appropriate variable, then use CreateObject() to instantiate the server. Call its methods as needed, and when you're done, set the object variable to Nothing (VBA's version of .null.) to release the server.

Listing 23 shows an example from WordLogger.DOT in the session materials; Word's DocumentBeforeClose method instantiates the OfficeResponder object and calls the UpdateLog method.

Listing 23. This code, in Word's DocumentBeforeClose event, uses the VFP COM object to log the closing of a document.

```
Private Sub wordapp_DocumentBeforeClose(ByVal Doc As Document, Cancel As Boolean)

Dim oVFPResponder, lResult As Boolean

Set oVFPResponder = CreateObject("OfficeResponder.OfficeResponder")
lResult = oVFPResponder.UpdateLog(Doc.FullName, "Closed")
Set oVFPResponder = Nothing

End Sub
```

In Excel, if you load the add-in as described above, the WorkbookBeforeClose code fires when you close the workbook that contains the add-in (such as when you close Excel), so you'll get a record in the log for the add-in itself.

Make sure that your main VFP application registers and loads the appropriate add-in at the same time that it instantiates each of the Office applications.

Final Thoughts

The more you automate Microsoft Office, the more ways you think of to automate it. This paper has covered some of the roadblocks you might run into when doing so, as well as looking at ways to set up tighter integration of your applications with Office.