Talking to Microsoft Office

Office 2007 and 2010 brought some changes in working with Office applications from VFP.

Tamar E. Granor, Ph.D.

While most Office Automation code moves smoothly from older versions to the latest, some changes in recent versions do have an impact on the process of automating the Office apps. One change in Office 2010 has major implications for applications that need to use Office's applications programmatically.

I learned how to automate the Office applications (and wrote a book about it) using Office 2000. Almost everything I knew made the transition to the next two versions of Office (Office XP and Office 2003). After that, the transition wasn't quite as smooth.

With Office 2007, Microsoft changed the user interface and the file format. Office 2010 introduced a new way of installing that breaks automation.

In this article, I'll take a look at changes in Office 2007 and 2010 that have an impact on Automation.

Object models

Each new version of Office has brought changes to the object model, with some properties, events and methods (PEMs) being deprecated (Microsoft-ese for "discouraged") or removed and new ones being introduced. Table 1 provides links to "What's New" pages in MSDN for the core Office applications (Word, Excel, PowerPoint and Outlook). 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

Getting Help

One of the tricky things about automating Office has always been finding the documentation. In Office 2003 and earlier, the VBA Help files for Office aren't automatically installed. For Office 2003, by default, they're installed as "on first use." But once you get them installed, each exists as a separate Help file (.CHM). I find them so useful that I actually keep a shortcut to each on my desktop.

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.

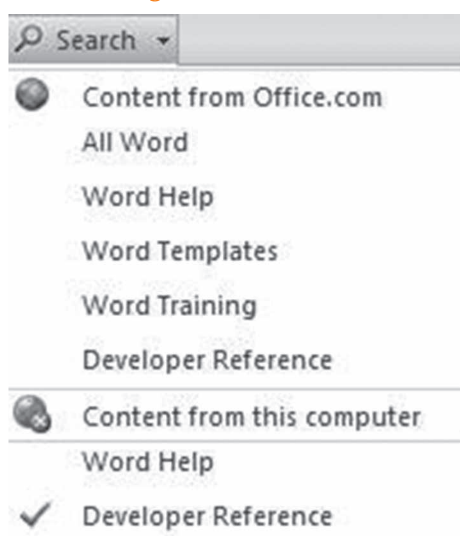

Figure 1. To see VBA Help from an Office 2007 or Office 2010 application, open Help from inside the application, then choose Developer Reference either from Office.com or from the local computer.

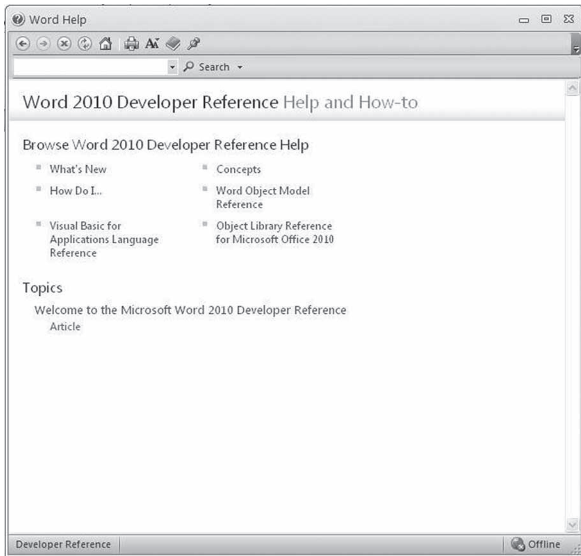

Figure 2. The home page of the Developer Reference for Word 2010 is fairly bare, but does give you access to the entire file.

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", substituting the actual application name for "<Application>") 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.

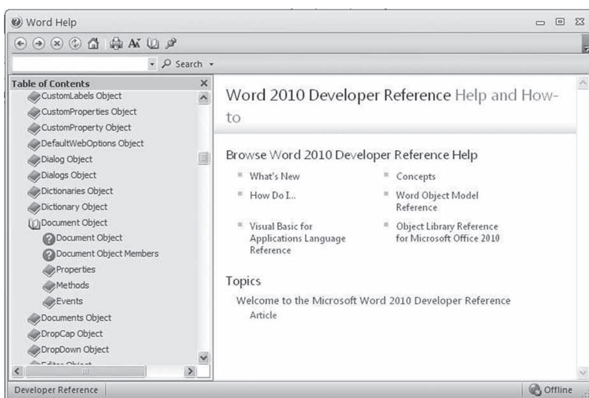

Figure 3. Using the Table of Contents helps to make the Developer Reference easier to use.

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, 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.

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 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, but one solution to the problem uses automation. 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 the file was created using Automation, saving it in the "Excel 97-2003 Workbook (*.xls)" format by passing 56 (xlExcel8) for the file format parameter. In fact, the problem even occurs if an XSLX file is opened in Excel 2003 (using the Office Compatibility Pack) and resaved in the older format from there.

The key element is that the file originates in the XML format; the problem 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.

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 1 performs this process; just pass the file name for the original workbook; the code is included in the downloads for this article as ConvertToExcel5.PRG.

Listing 1. 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. Extension is automatic.
   cFileName = FORCEPATH(m.cBaseName, m.cPath)
   oWorkbook.SaveAs(m.cFileName, 39)
   nWorkbookCount = 1

OTHERWISE
   * Need to break up into multiple workbooks.
   LOCAL nTotalRows, nSheetsNeeded, nSheet
   LOCAL 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 data 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.

If you don't have Excel available to resave the file or you have formulas that work across large groups of rows, there are a couple of other solutions. One option is to export the data from Excel in the CSV (comma-separated values) format, and append that in VFP, as in **Listing 2**.

Listing 2. One way to handle XLSX files is to save them as CSV and then use APPEND FROM or IMPORT on the result.

```
* Assumes you've already created the CSV file
* and that the filename is stored in cFileName
APPEND FROM (m.cFileName) TYPE CSV
```

If you can't control the format you receive and can't count on having Excel available, the final choice is to use the Excel ODBC driver to open the file and extract the data. To use this approach, you must have the driver installed. It's available for download from Microsoft (search for "2007 Office System Driver: Data Connectivity Components"). **Listing 3** shows some generic code to do the import; you need to specify the fully-pathed filename for the Excel file and the alias for the cursor or table to which you want to append the results. It's included in the downloads for this article as ExcelAppendODBC.PRG

Listing 3. Another way to handle Excel files is to use the Excel ODBC driver to open them.

```
* Open and import an Excel file using
* the Excel ODBC driver. Put the results
* in the specified table/cursor.
LPARAMETERS cFileName, cDestinationAlias

LOCAL cConnStr, nHandle, cSQL

DO CASE
CASE VARTYPE(m.cFileName) <> "C" OR ;
   EMPTY(m.cFileName)
   * File name not specified
   ERROR 11

CASE NOT FILE(m.cFileName)
   * Specified file doesn't exist
   ERROR 1, m.cFileName

OTHERWISE
   * All is well with file. Proceed.
   cConnstr = [Driver=] + ;
     [{Microsoft Excel Driver ] + ;
     [(*.xls, *.xlsx, *.xlsm, *.xlsb)};] + ;
     [DBQ=] + m.cFileName

   nHandle = SQLSTRINGCONNECT(m.cConnStr)
   IF m.nHandle > 0
      TEXT TO m.cSQL NOSHOW
         SELECT * FROM "Sheet1$"
      ENDTEXT

      IF SQLEXEC(m.nHandle, m.cSQL, ;
                 "csrResult") > 0
         SELECT (m.cDestinationAlias)
         APPEND FROM DBF("csrResult")
      ELSE
         ERROR ;
           "Unable to extract data from " + '
           m.cFileName
      ENDIF

      * Disconnect
      SQLDISCONNECT(m.nHandle)
   ELSE
      ERROR "Unable to connect to Excel"
   ENDIF
ENDCASE

RETURN
```

The one complication with this approach is that the field names in the destination alias must match the column names in the Excel file.

The bottom line

If you've been automating older versions of Office, the more recent versions can introduce some new issues. ("Click to Run" installation seems to be the most frequently encountered problem.) But there's nothing that can't be overcome with either some code or some user training.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. She currently focuses on working with other developers through consulting and subcontracting. Tamar is author or co-author of nearly a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with VisualFoxPro and Taming Visual FoxPro's SQL . Her latest collaboration is Making Sense of Sedna and SP2. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Support Most Valuable Professional and one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.