Excel Automation

For clarity's sake, the full object hierarchy is used for each command. Using a 'With oExcel' will tend to make your code more readable.

Connecting to Excel
We'll assume throughout this page that you named your Excel object oExcel and your workbook oWorkbook.

```vba
oExcel = CreateObject("Excel.Application")
if vartype(oExcel) != "O"
    * could not instantiate Excel object
    * show an error message here
    return .F.
endif
```

Constants for Excel
Excel Constants documents the constants that Excel's macros use (xlThis, xlThat etc)
2007/08/30 Mike Mohr - Excel 2003 Constants MSDN Library
2007/08/30 Mike Mohr - Excel 2007 (office 12) Constants MSDN Library

Opening an existing XLS

```vba
oWorkbook = oExcel.Application.Workbooks.Open("C:\temp\test.xls")
```

Creating a blank workbook:

```vba
oWorkbook = oExcel.Application.Workbooks.Add()
```
Creating a new workbook based on a template:

```vba
oWorkbook = oExcel.Application.Workbooks.Add("C:\temp\template.xlt")
```

You can turn off Excel's alerts with the Display Alerts property: when set to False, Excel automatically chooses the default message:

```vba
oExcel.DisplayAlerts = .F.
oWorkbook.Close() && Unsaved changes will be discarded
oExcel.DisplayAlerts = .T.
```

Saving a workbook as Excel 97/03 from Excel 2007

Using `SaveAs` while automating Excel 2007 creates a 2007 style workbook with an XLS extension (regardless of compatibility file format is specified):

```vba
if val(oExcel.Version) > 11
    oWorkbook.SaveAs("C:\temp\foobar.xls", 56) && xlExcel8
else
    oWorkbook.SaveAs("C:\temp\foobar.xls")
endif
```

Controlling visibility

If the Excel window is not visible it is harder for the user to interact with Excel. This makes it slightly safer likely to issue commands in the middle of your automation.

```vba
oExcel.visible = .T.
oExcel.visible = .F.
```
**Controlling Interaction**

Also, if it is preferred that Excel be seen during automation set these two properties to .F.

```vba
oExcel.Application.UserControl=.F.
oExcel.Application.Interactive=.F.
```

After completing automation, return their value to .T. to allow the user to start interaction

```vba
oExcel.Application.UserControl=.T.
oExcel.Application.Interactive=.T.
```

The **Interactive** property is the one that controls whether the user is allowed to interact with Excel. The hourglass icon when they hover over Excel, and mouse clicks on the Excel application are ignored.

The **UserControl** property does NOT prevent the user from interacting with Excel. That property indicates whether the Excel application was opened programmatically via `CREATEOBJECT("Excel.Application")` to get a reference to the Excel application, you can use this property to determine if you are sharing with the user or not. One cool thing is that this property is automatically updated if the user closes Excel. See UserControl Property [Excel 2007 Developer Reference]

Bottom line: even though you CAN change UserControl, I recommend that you NOT do that.

Mike Potjer

**Storing data to a cell**

```vba
oExcel.Range("b2").Value = "Hello world"
```

**Set the font color and style**

```vba
oExcel.Range("B6").font.bold = .t.
oExcel.Range("B6").font.colorindex = 3 && red
```
or

```vfp
oExcel.Range("B6").Select()
oExcel.Selection.font.colorindex = 3 && red
oExcel.Selection.font.bold = .t.
```

-- David Fung

**Set Excel Cell to Text Format**
If your app exports user-entered text data from a field into an Excel cell, you always convert the cell's number value. I forgot this in an app once, and a user entered a long string of equal signs into a notes field in VF setting the cell's value (Excel doesn't like "=======" as a value, thinks it's an incorrectly written formula)

```vfp
oExcel.Range("A1").NumberFormat = "@"
oExcel.Range("A1").Value = cursorA.MemoField
```

-- Bryan Palmer

**Getting data into Excel**
Assuming your table is selected. First the easy way. You can make a new file in an old Excel format which

```vfp
copy to c:\mytable.xls type x15
```

Or if you have a pre-formatted template (.XLS or .XLT) that you want to paste into. Note that this method

```vfp
_VFP.DataToClip(,,3) && current table onto the clipboard, delimited with
oExcel.Range("A1").Select
oExcel.ActiveSheet.Paste() && from clipboard. since delimited with
```

**Selecting a range using the Cells collection**
oExcel.Range(oExcel.Cells(1, 1), oExcel.Cells(3, 3)).Select

Resize all columns

oExcel.ActiveSheet.UsedRange.EntireColumn.Autofit

Insert two rows before the first row

oExcel.Rows("1:2").Insert(-4121) && xlDown

-- David Fung

Closing Excel
You'll still need to handle closing questions like saving changes and file format changes. And you'll need to handle the instance of Excel remaining after quitting. To clear the issue:

oExcel = .Null.

Closing Excel with all changes discarded - no question asked

oExcel.DisplayAlerts = .F.
oExcel.ActiveWindow.Close(.f.) && assuming only one active window
oExcel.quit()

-- David Fung

**Iterate through Excel's Worksheets**

```vba
For i=1 To oExcel.ActiveWorkbook.Sheets.Count
   ? oExcel.ActiveWorkbook.Sheets(i).Name
Endfor
```

**Finding text in Excel**

**Searching for "Tax"**

```vba
oExcel.Range("A1:H250").Find("Tax").Select && simple default search
* Alternately
* Range.Find(What, After, LookIn, LookAt, SearchOrder, SearchDirection, MatchCase, MatchByte, Se
```

Range("A1:H250") specifies that we're searching columns A to H (inclusive) and rows 1-250.
oExcel.ActiveCell is where to start searching, and -4123 is the constant for xlFormulas. I theorize that this means 'if there is a code rather than its output.' 1 is the constant for xlWhole, meaning match against all the text in the cell. 2 matches.

-- Tom Cerul

You have to be careful when specifying the extra parameters to Find as they persist between searches, as:

The settings for LookIn, LookAt, SearchOrder, and MatchByte are saved each time you don't specify values for these arguments the next time you call the method. Setting these arguments changes the settings in the Find dialog box, or the Find dialog box changes the saved values that are used if you omit the arguments explicitly each time you use this method.

-- Stuart Dunkeld
Developing new code
Sometimes the easiest way to figure out how to code an automation routine is this: open Excel, tell it to automate. Stop the recording and look at the code that it generated.

Putting it all together, a runnable example
First, COPY TO all fields (or some) in Excel Format

```vfp
#define xlLastCell 11
#define xlMaximized -4137
#define xlRangeAutoformatClassic2 2
#define xlPortrait 1

use MyTable && or SELECT * INTO MyCursor

cFileName = "MyXLSFile" && or whatever, including path
*copy to (cFileName) fields (cFields) TYPE xls
copy to (cFileName) TYPE xls

* then open excel and make the data look good, like this
oExcel = CreateObject("Excel.Application")
if vartype(oExcel) != "O"
* could not instantiate Excel object
* show an error message here
    return .F.
endif

* make excel visible during development
*oExcel.visible = .T.

* open the workbook you just created
oExcel.SheetsInNewWorkBook = 1
oWorkbook = oExcel.Workbooks.Open(cFileName)

* rename the Sheet to whatever you like
oActiveSheet = oExcel.ActiveSheet
oActiveSheet.Name = "MyData"

oExcelApp = oExcel.Application
oExcelApp.WindowState = xlMaximized

* find address of last occupied cell
```
lcLastCell = oExcel.ActiveCell.SpecialCells(xlLastCell).Address()

* resize all columns
lnMarker1 = at("$",lcLastCell,1)  && i.e. 1 when lcLastCell = "$AF$105"
lnMarker2 = at("$",lcLastCell,2)  && i.e. 4 when lcLastCell = "$AF$105"
lnStartPos = lnMarker1 + 1
lnStrLen = lnMarker2 - lnStartPos
oExcel.Columns("A:" + substr ;
  (lcLastCell,lnStartPos,lnStrLen)).EntireColumn.AutoFit

* you can even add a nice autoformat
oExcel.Range("A" + alltrim(str(nTOPBLANKROWS+1)) + ":" + lcLastCell).Select
oExcel.Selection.AutoFormat(xlRangeAutoformatClassic2,.t.,.t.,.t.,.t.,.t.,.t.)

* set Excel Print Area
oActiveSheet.PageSetup.PrintArea = "$A$1:" + lcLastCell

* define printed page footer
With loActiveSheet.PageSetup
  *.LeftHeader   = ""
  *.CenterHeader = ""
  *.RightHeader  = ""
  .LeftFooter   = "&BMy Footer goes here&B"
  .CenterFooter = "&D"
  .RightFooter  = "Page &P"
  *.PrintHeadings = .F.
  .PrintGridlines = .F.
  .CenterHorizontally = .T.
  .CenterVertically = .F.
  .Orientation = xlPortrait
endwith

* save Excel file in new Excel format (COPY TO XLS uses old format)
oWorkbook.Save()

* display finished product to the user
oExcel.visible = .T.

--- Alex Feldstein
- Sometimes the Last Cell is not up-to-date after deleting a row in Excel,
- Calling ActiveSheet.UsedRange after deleting a row will keep Last Cell
  up-to-date.
**loExcel = createobject('Excel.Application')**

**loExcel.Workbooks.Open(tcFile)**

**loExcel.Rows("1").Delete(xlUp)**

**lnLastRowIncorrect = loExcel.Cells.SpecialCells(xlCellTypeLastCell).Row**

**loExcel.ActiveSheet.UsedRange && add this line**

**lnLastRowCorrect = loExcel.Cells.SpecialCells(xlCellTypeLastCell).Row**

---

**David Fung**

Having worked with excel a lot a few notes:

Office applications are not as much backward compatible as VFP is. For that reason I suggest not to use `\` like:

`copy to myExcel type xls` & `or xl5,xl8...`  
`import ... type xl8`

etc. These commands are version dependant directly within VFP themself. You immediately lose data with:

- The number of rows you can copy is limited for example (VFP5 copied 16384 max while VFP9 copies 65536 max come into market those limits are not sufficient anymore).
- Memo fields are immediately dropped as with any 'copy to' command

**Data To Clip()** - Do not use Data To Clip() for Excel transfers.  
- You lose Memo fields but worse you get a `transform()` version of memo fields.  
- You're limited on rowcount that you can transfer with this method. There is no exact number and sounds available memory. It's possible you end with much fewer rows than you could transfer using 'copy to'. Like

Instead you can use:

**Copy to myExcel.xls type fox2x** & actually creating a dBaseIII compatible file. Excel recognizes internally  
**Copy to myExcel.csv type csv** & CSV files are natively recognized

Both fail to transfer memo fields and CSV might have problems with datatypes converted correctly (most better than Data To Clip() and 'copy ... type xls'.

Similar to Data To Clip() you can copy to a tab delimited file, read it into clipboard with `FileToStr()` and pastData To Clip() but it again falls short of transferring memo fields.

Excel (especially newer versions) also recognizes XML and HTM (table tags).

My best preference is to transfer data using ADO instead. Passing with ADO uses Excel's own VBA comm:sample sending data to Excel and putting data starting at A10:

**LOCAL oExcel**
oExcel = CreateObject("Excel.Application")
With oExcel
    .WorkBooks.Add
    .Visible = .T.
VFP2Excel(_samples+'data\testdata.dbc','select * from employee',.ActiveSheet.Range('A10')
Endwith

function VFP2Excel
 tparameters tcDataSource, tcSQL, toRange
Local loConn As AdoDB.Connection, ;
    loRS As AdoDB.Recordset,;
    ix
loConn = Createobject("Adodb.connection")
loConn.ConnectionString = "Provider=VFPOLEDB;Data Source=\"+m.tcDataSour
loConn.Open()
loRS = loConn.Execute(m.tcSQL)

FOR ix=1 TO loRS.Fields.Count
    toRange.Offset(0,m.ix-1).Value = PROPER(loRs.Fields(m.ix-1).Nam
    toRange.Offset(0,m.ix-1).Font.Bold = .t.
ENDFOR

toRange.Offset(1,0).CopyFromRecordSet( loRS )
loRs.Close
loConn.Close

Note that .Visible = .T. is very early in code just after adding the workbook. Having that "later after you'r things faster" is a myth. Surprisingly having it early in code makes your code faster in many cases.

My suggestions:
Working with Excel means you're doing COM calls using VBA which by nature is slow. Therefore, whenever call Excel automation commands as few as you can. ie:

Instead of this:

for ix = 1 to 5000
    for jx=1 to 10
        .Cells(m.ix,m.jx).Value = m.ix*100+m.jx
    endfor
endfor
Do this:

```vfp
Do this:

dimension aExcelData[5000,10]
for ix = 1 to 5000
    for jx=1 to 10
        aExcelData[m.ix,m.jx] = m.ix*100+m.jx
    endfor
endfor

WITH oExcel.ActiveWorkBook.ActiveSheet
    .Range(.Cells(1,1), .Cells(5000,10)).Value = GetArrayRef('aExcelData')
endwith

PROCEDURE GetArrayRef(tcArrayName)
    RETURN @@tcArrayName

Above code also shows another alternative to transferring data to Excel via array instead of pasting.

Cetin Basoz
```

Getting Data from Excel

Using automation, you can write loops which go through a worksheet cell-by-cell reading the data. Although it appears to be universally acknowledged as being very slow. Another technique which is promoted in different places is to use ADO or ODBC/OLEDB, which is quite fast. However, it was Cetin's example above, which uses an array to populate a worksheet, which helped me to discover a third option that it is very simple, and also very fast, to use automation to read data from Excel into an array.

```vfp
#DEFINE xlLastCell 11

LOCAL loLastCell AS Excel.Range, ;
    lnFieldCount, ;
    lnImportedColumns, ;
    lnField, ;
    xx
LOCAL ARRAY laData[1,1], ;
    laFieldInfo[1]

* I'm skipping the code to create the oExcel object, load a workbook, and selec
* a worksheet.

WITH oExcel.ActiveWorkBook.ActiveSheet
  * This is just asking Excel to give us the last cell for the range of data,
  * but you can use whatever technique you want to determine the range to use.
  loLastCell = .Cells.SpecialCells( xlLastCell )

  * This command re-dimensions and populates the array with all the data from
  * the specified Range.
  * Note that in this example I assume that the first row contains captions
  * or field names.
  laData = .Range( .Cells(2,1), m.loLastCell ).Value
ENDWITH

*-- MJP -- Begin Update 2011/09/13
lnFieldCount = AFIELDS( laFieldInfo, "MyAlias" )

* We need to check how many columns are actually being imported, since the user
  * could have omitted some.
  lnImportedColumns = MIN( m.lnFieldCount, ALEN( laData, 2 ) )

* Check the data being imported, and convert data types as needed.
FOR lnField = 1 TO m.lnImportedColumns
  * For now, assume that any data not being imported into a Character or Varcha
  * field is either of a compatible data type, or is invalid and should be igno
  IF NOT INLIST( laFieldInfo[m.lnField,2], "C", "V" )
   LOOP
ENDIF
ENDFOR

FOR xx = 1 TO ALEN( laData, 1 )
  * The situation we are most likely to encounter is a Numeric value being
  * imported into a Character field.  INSERT .. FROM ARRAY does not perform
  * any kind of CAST, but simply ignores the numeric value, leaving the field
  * blank.  Converting the number to a string before the INSERT allows it to
  * be imported properly.
  IF VARTYPE( laData[m.xx,m.lnField] ) = "N"
    laData[m.xx,m.lnField] = TRANSFORM( laData[m.xx,m.lnField] )
  ENDIF
ENDFOR

*-- MJP -- End Update 2011/09/13

* You can now process the data directly in the array, or dump it into a cursor:
  * INSERT INTO MyAlias FROM ARRAY laData
Using what I consider a fairly average modern PC (dual core, 2GB of RAM), I used the above code to read populate a cursor with the data in under 0.15 seconds. Obviously, your mileage may vary, but you should most situations.

Note that this technique allows you to populate memo fields in a cursor. If a column contains cells with more than 254 characters, the entire text is copied to the array, and from there can dumped directly into a memo field using either the MEMO FORMULA or APPEND FROM .. XLS|XL5|XL8.

Note that a data cell in Excel must be compatible with the data type of the VFP cursor field in which you are inserting data. In Excel, which is stored as numbers will be ignored when you try to copy them into a VFP character field, I stored as text in Excel are normally differentiated by a little green triangle in the upper left corner of the spreadsheet to correct this problem, but a far simpler solution is to modify your VFP code to transform them into the VFP cursor. The sample code above has been updated to include this solution.

Mike Potjer

If you are using VFP 8 or earlier, arrays are limited to 65,000 elements, so you will get an error if you attempt to read data from Excel into an array. However, VFP 9 raised the limit on arrays to 2GB, which should be more than enough.

Notes/Caveats on the above excel->array->table method:
#1. Call UsedRange before determining the last cell. It's not core to the method, but it helps.
#2. Whilst Memo fields do transfer to the array, they are not transferred to the table with APPEND FROM REPLACE afterwards.
#3. I have had better results using the Value2 property of the range for the transfer to the array. The Value property includes localization options and rounding, whilst the Value2 property is just the underlying value.

Brian Marston

Contributors: Alex Feldstein Tom Cerul Stuart Dunkeld David Fung Cetin Basoz Mike Potjer

Category Code Samples Category Automation