Working with Microsoft Excel, Part 2

Doug Hennig

Microsoft Excel is one of the most widely-used applications ever. Because of its popularity, other applications often need to read from or write to Excel documents. The previous issue's article discussed several mechanisms for outputting VFP data to Excel. This article looks at ways to read Excel documents into VFP.

Most applications I've worked on had to interact with Microsoft Excel documents in one way or another. Outputting data to Excel is extremely common, as most people are comfortable working with Excel and often want to analyze or massage the data in their applications in ways a developer can't foresee. But importing data from Excel is also popular, both because it can be a common interchange format between different applications and because people often treat Excel like a database, with its row and column layout making data entry fast and easy.

In the previous issue of FoxRockX, I discussed several mechanisms for output data from VFP to Excel, including the built-in COPY TO and EXPORT TO commands, using OLE Automation, using ADO, and writing directly to Excel XML files. In this issue, we'll look at the opposite job: reading from Excel documents into VFP cursors and tables.

Built-in commands

The VFP APPEND FROM and IMPORT FROM commands support reading from Excel version 2.0 (using the "TYPE XLS" clause), version 5.0 (using "TYPE XL5"), and Excel 97 (using "TYPE XL8") files (all versions are XLS files). APPEND FROM appends records into an existing table while IMPORT FROM creates a new table, either a free table or one in a database container, depending on keywords you supply with the command (see VFP help for details).

There are a few issues:

- Reading from some XLS files saved from Excel 2007 or later causes errors; for example, importing from EmployeesFromXLSX.xls included with the sample files for this article causes VFP to crash.
- The columns in the table created with IMPORT FROM are named A, B, C, etc. You have to use ALTER TABLE or MODIFY STRUCTURE to give them more meaningful names.
- Column headings in the first row are treated as a record so you have to delete that record after reading from the Excel file.
- Memo fields are ignored. For example, USE EMPLOYEES to open Employees.DBF (included with the sample files for this article), then APPEND FROM Employees.XLS TYPE XLS. The last column in Employees.XLS is a 254-character text field but after the APPEND FROM command, you'll find the NOTES memo field, the last field in the table, is empty.
- There is no support for reading from XLSX files (Excel 2007 and later).

If you can live with these issues, especially the last one, then these commands are your best solution since they're the fastest way to import from Excel and take only one line of code.

Using ODBC

There are a couple of ODBC drivers for Excel: Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb), which works with files created by any version, and Microsoft Excel Driver (*.xls), which works with Excel 2003 and earlier. Craig Boyd uses these drivers to read from Excel files as if they're databases in AppendXLSX.PRG, which you can download from http://tinyurl.com/jzolom3. This function doesn't require Excel to be installed.

The AppendFromExcel function accepts the following parameters:

- The name of the Excel file to read from.
- The name of the worksheet to read from. You can also specify a range, such as Sheet1$A1:C20 (the "$" is required because the ODBC driver needs it). If you don't specify it, the default "Sheet1$" is used.
- The alias, workarea, or name of the table to append records to (if not specified, the current workarea is used). This can be a cursor or a table.
- A comma delimited list of columns to read from the worksheet (if not specified, all columns are read).
- A comma delimited list of fields to insert values into (if not specified, all fields are used).
- A WHERE clause used when querying the cursor for data to be read from the worksheet.
- .T. if the first row of the worksheet is not a header row but instead should be treated as the first record.

AppendXLSX uses the appropriate Excel ODBC driver to connect to the specified document, uses SQLEXEC to read from the specified worksheet as if it's a table, and then goes through the resulting cursor and copies the values to the specified table. I won't list the code here because it's fairly long; feel free to examine AppendXLSX.PRG yourself. I fixed a couple of bugs in the code; the fixed version is included with this month's downloads.

The code in **Listing 1** imports Employees.XLSX into Employees.DBF.

Listing 1. Test code for AppendFromXLSX.

```
use Employees exclusive
zap
set procedure to CraigBoyd\AppendXLSX
lnRecords = AppendFromExcel('Employees.xlsx')
messagebox(lnRecords)
browse
```

In my experience, ODBC works well and it's a mechanism I use often to import Excel data. It's also the second fastest mechanism I tested.

Reading directly from XLSX files

A XLSX file is basically a zip file containing several XML files that together make up the Excel document. Reading directly from an XLSX file means unzipping the file and reading the content from the appropriate XML file. Since the file is accessed directly, neither Excel nor its ODBC drivers have to be installed.

I looked at a couple of tools available to do this:

- XLSXWorkbook, a VFPX project (http://vfpx.codeplex.com), by Greg Green (I discussed this project's ability to write to XLSX files in the previous article).
- ImportFromXLSX and AppendFromXLSX by Vilhelm-Ion Praisach, which you can download from his blog http://praisachion.blogspot.com.

Let's look at XLSXWorkbook first. You can download it from http://tinyurl.com/zr6vxc5. The download includes a number of test programs and forms, but all of the actual code for the utility is in a single VCX, VFPxWorkbookXLSX.VCX. The download also includes extensive documentation in a PDF file.

To use XLSXWorkbook, start by instantiating the VFPXWorkbookXLSX class in VFPXWorkbookXLSX.VCX. XLSXWorkbook doesn't have a specific method to read all the rows from a worksheet into a table, but it does have a method, GetSheetRowValues, that reads all columns from the specified row in the specified worksheet, so you just need a loop to process all rows. Pass GetSheetRowValues a handle to a document you opened with OpenXlsWorkbook, the name or number of a worksheet in the document, and the row number to read from. GetSheetRowValues returns an object with a Count property, containing the number of columns read, and a Values array, with one row per column read. The first column in the array contains the cell value and the second column the data type of the value.

The code in **Listing 2** imports Employees.XLSX into Employees.DBF. The results are identical to that for Craig Boyd's AppendFromXLSX.

Listing 2. Test code for XLSXWorkbook.

```
use Employees exclusive
zap
loExport = newobject('VFPXWorkbookXLSX', ;
  'VFPXWorkbookXLSX\VFPXWorkbookXLSX.vcx')
lnWB    = ;
  loExport.OpenXlsxWorkbook('Employees.xlsx')
llDone = .F.
lnRow  = 2
do while not llDone
  loRow = loExport.GetSheetRowValues(lnWB, ;
    1, lnRow)
  lnRow = lnRow + 1
  if not isnull(loRow) and ;
    not isnull(loRow.Values[1, 1])
    append blank
    for lnI = 1 to loRow.Count
      lcField = field(lnI)
      replace &lcField with ;
```

```
        loRow.Values[lnI, 1]
    next lnI
  else
    llDone = .T.
  endif not isnull(loRow) ...
enddo while not llDone
browse
```

ImportFromXLSX and AppendFromXLSX are easier to use because they do all the work of reading from the XLSX file and creating the table (in the case of ImportFromXLSX) or adding records to the table (AppendFromXLSX). The downloads from Vilhelm-Ion's blog include a number of test programs, but the utilities themselves are in a couple of PRGs: ImportFromXLSX and AppendFromXLSX.

Call ImportFromXLSX with these parameters:

- The name of the Excel file to read from.
- The starting row to read from; if it isn't specified, all rows are read. Pass 2 if the first row contains column headers.
- The name or number of the worksheet to read from. If you don't specify it, the first worksheet is used.
- .T. if the result should be a cursor or .F. for a table.
- .T. if the worksheet contains empty cells. This slows down the import but handles empty cells better.
- The row number that contains column headers. If you don't specify the row number, ImportFromXLSX uses field names like MFIELD1, MFIELD2, and so on for the field names in the table or cursor. If you do specify the row number, it uses the column headers (which it may adjust to be valid VFP names) for the field names.
- The name of the table or cursor to create; you can specify "?" to display a Save As dialog.

For AppendFromXLSX, the parameters are:

- The name of the Excel file to read from.
- The name of the table to append records to (if not specified, the current workarea is used). This can be a cursor or a table.
- A comma delimited list of fields to insert values into (if not specified, all fields are used).
- The starting row to read from; if it isn't specified, all rows are read. Pass 2 if the first row contains column headers.
- The name or number of the worksheet to read from. If you don't specify it, the first worksheet is used.
- .T. if the worksheet contains empty cells.

Both programs have a couple of constants that determine their behavior. ERRLANG is the language to use for error messages; set it to "Ro" for Romanian, "Fr" for French, "Nl" for Dutch, or anything else for English. If you want to use WinRAR for unzipping the Excel file rather than the default Window file extractor, uncomment the #DEFINE archiveWinRAR .T. statement.

The code in **Listing 3** creates EmployeesImport.DBF from the content of Employees.XLSX. The results are very similar to that for the other mechanisms except the field names come from the column headings since we're creating a table rather than appending into an existing one.

Listing 3. Test code for ImportFromXLSX.

```
set path to ImportFromXLSX
ImportFromXLSX('Employees.xlsx', 2, '', .F., ;
  .F., 1, 'EmployeesImport.dbf')
use EmployeesImport
browse
```

The code in **Listing 4** imports Employees.XLSX into Employees.DBF. The results are identical to that for the other append mechanisms I discussed in this article.

Listing 4. Test code for AppendFromXLSX.

```
set path to AppendFromXLSX
use Employees exclusive
zap
AppendFromXLSX('Employees.xlsx', alias(), ;
  '', 2)
browse
```

Both Greg's and Vilhelm-Ion's utilities worked well with the Excel files and tables I tested them with, and both take about the same amount of time to execute (ImportFromXLSX takes a little longer because it has to determine the field names and data types for the table it has to create). However, they're significantly slower than either the built-in commands or ODBC.

Summary

Because of its popularity, being able to read from or write to Microsoft Excel documents is often an important requirement for a modern application. The past two articles discussed several mechanisms for working with Excel documents. Select the one that best suits your needs, including whether you can work with XLS or XLSX files and whether Excel has to be installed on the machine or not.

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of "VFPX: Open Source Treasure for the VFP Developer," "Making Sense of Sedna and SP2," the "What's New in Visual FoxPro" series (the latest being "What's New in Nine"), "Visual FoxPro Best Practices For The Next Ten Years," and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He was a Microsoft Most Valuable Professional (MVP) from 1996 to 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).