anguage Enhancements in VFP 7, Part IV

Doug Hennig

This fourth article in a series on language enhancements in VFP 7 article covers six new functions. One of them, EDITSOURCE(), is used in a project text search utility, making it easy to find all occurrences of a string in all files in a project and bring up the found locations in the appropriate editor.

Writing this series is like in some ways trying to hit a moving target. When I started, I was using the Tech Preview version released at DevCon in September 2000. Now, the beta 1 version is available (go to http://msdn.microsoft.com/vstudio/nextgen/beta.asp for information about how to get your copy) and with it, there are more changes to some existing functions. ADIR() now accepts a fourth parameter: set bit 0 to preserve filename case (rather than forcing it to uppercase as it does in previous versions of VFP) and bit 1 to show names in DOS 8.3 format. These values can be added, so 1 will preserve case and 3 will preserve case and use 8.3 format. STRTOFILE() now accepts a third parameter: if it's .T., the file won't be overwritten but instead the string will be appended to it. Now, back to our regularly schedule program.

ASESSIONS()

This new function fills an array with the DataSessionID values of all existing data sessions. TESTASESSIONS.PRG (included with this month's Subscriber downloads) shows how it works by creating two Session objects, using ASESSIONS() to fill an array, and then displaying the contents of the array.

In VFP 6 and earlier, a common technique used when an application is being shutdown due to an error is to revert any data changes in all data sessions. One way to do this is to spin through all open forms, SET DATASESSION to their DataSessionID, and use TABLEREVERT() on all open tables:

```
local lnI

* Rollback all transactions.

do while txnlevel() > 0
  rollback
enddo while txnlevel() > 0

* Go through all forms and revert all tables in their
* data sessions.

for lnI = 1 to _screen.FormCount
  if _screen.Forms[lnI].DataSessionID <> 1
    set datasession to _screen.Forms[lnI].DataSessionID
    RevertTables()
  endif _screen.Forms[lnI].DataSessionID <> 1
next lnI

* Revert all cursors in the default data session.

set datasession to 1
RevertTables()

function RevertTables
local laCursors[1], ;
  lnCursors, ;
  lnI, ;
  lcAlias

* Revert all changes in all buffered tables.

lnCursors = aused(laCursors)
```

```
for lnI = 1 to lnCursors
  lcAlias = laCursors[lnI, 1]
  if cursorgetprop('Buffering', lcAlias) > 1
    tablerevert(.T., lcAlias)
  endif cursorgetprop('Buffering', lcAlias) > 1
next lnI
return
```

One problem with this code: it won't handle any private data sessions created by Session objects. So, here's the VFP 7 version of this routine (REVERTALLTABLES.PRG):

```
local laSessions[1], ;
  lnI, ;
  laCursors[1], ;
  lnJ, ;
  lcAlias

* Rollback all transactions.

do while txnlevel() > 0
  rollback
enddo while txnlevel() > 0

* Go through all data sessions and revert all tables in
* them.

for lnI = 1 to asessions(laSessions)
  set datasession to laSessions[lnI]
  for lnJ = 1 to aused(laCursors)
    lcAlias = laCursors[lnJ, 1]
    if cursorgetprop('Buffering', lcAlias) > 1
      tablerevert(.T., lcAlias)
    endif cursorgetprop('Buffering', lcAlias) > 1
  next lnJ
next lnI
```

ASTACKINFO()

This new function will be a welcome addition to any error handling scheme or any other code using the SYS(16) function (which returns the name of the executing program at any level in the call stack). ASTACKINFO() fills an array with information about the entire call stack: the stack level, the executing filename, the module or object method name, the source filename, the line number, and the source code (if it's available). Some of this information isn't available any other way. For example, there's no other way to get the line number and source code for different levels of the call stack, and the information SYS(16) gives for PRGs in an EXE is incomplete.

To see ASTACKINFO() in action, run TEST.EXE from the Windows Explorer (so it's running under the VFP runtime, not the development environment). The main program is TESTASTACKINFO.PRG. It calls the ShowForm function (also in TESTASTACKINFO.PRG), which runs the TEST form. Clicking on the "click me" button instantiates a form class. Clicking on the "click me too" button in that form displays the contents of the array filled by ASTACKINFO(). Here are the results:

(1,1) 1
(1,2) "f:\writing\articles\apr01\test.exe"
(1,3) "testastackinfo"
(1,4) "f:\writing\articles\apr01\testastackinfo.prg"
(1,5) 3
(1,6) "do ShowForm"

(2,1) 2
(2,2) "f:\writing\articles\apr01\test.exe"
(2,3) "showform"
(2,4) "f:\writing\articles\apr01\testastackinfo.prg"
(2,5) 9

(2,6) "do form test"

(3,1) 3
(3,2) "f:\writing\articles\apr01\test.sct"
(3,3) "frmtest.cmdclickme.click"
(3,4) "f:\writing\articles\apr01\test.sct"
(3,5) 2
(3,6) "loForm.Show()"

(4,1) 4
(4,2) "f:\writing\articles\apr01\test.vct"
(4,3) "testform.cmdclickme.click"
(4,4) "f:\writing\articles\apr01\test.vct"
(4,5) 1
(4,6) "local laStack[1]"

The third column contains the module or object method name. Notice the difference in this column in each row: the first row is from code in the main program, so it shows the main program name as the module; the second row is from the ShowForm function in TESTASTACKINFO.PRG, so it shows that function name as the module; and the other two rows show the complete object method hierarchy for the executing code. The fact that column 2 doesn't show the EXE name for rows 3 and 4 is a bug in the current beta version.

We can't create a complete replacement for ASTACKINFO() in VFP 6 since, as I mentioned above, there's no way to get line numbers or source code for different levels of the call stack. Also, in an EXE, SYS(16) gives the name of the EXE for code in the main PRG rather than the PRG name and no path information for all other PRGs. However, given these limitations, here's ASTACKINFO.PRG, which you may find useful in an error handling scheme where you want information about the entire call stack. Remember to pass the array by reference by prefixing it with @.

```
lparameters taArray
local lnLevels, ;
  lnI, ;
  lcFileName, ;
  lnPos, ;
  lcModule, ;
  lcSourceFile
lnLevels = program(-1) - 1
dimension taArray[lnLevels, 6]
for lnI = 1 to lnLevels
  lcFileName = sys(16, lnI)
  if left(lcFileName, 10) = 'PROCEDURE '
    lnPos      = at(' ', lcFileName, 2)
    lcFileName = lower(substr(lcFileName, lnPos + 1))
  else
    lcFileName = lower(lcFileName)
  endif left(lcFileName, 10) = 'PROCEDURE '
  lcModule = lower(program(lnI))
  if justext(lcFileName) = 'fxp'
    lcSourceFile = forceext(lcFileName, 'prg')
  else
    lcSourceFile = lcFileName
  endif justext(lcFileName) = 'fxp'
  taArray[lnI, 1] = lnI
  taArray[lnI, 2] = lcFileName
  taArray[lnI, 3] = lcModule
  taArray[lnI, 4] = lcSourceFile
  taArray[lnI, 5] = 0
  taArray[lnI, 6] = ''
next lnI
return lnLevels
```

ATAGINFO()

ATAGINFO() does in a single function what in earlier versions of VFP you had to use nine functions and about 40 lines of code to do: getting all of the information about the tags for a table. This function fills the specified array with the name, type, expression, filter, order, and collate sequence for each tag and returns the number of tags. This information is essential for creating meta data that can later be used to recreate the indexes for a table if they become corrupted. To see how ATAGINFO() works, run TESTATAGINFO.PRG.

ATAGINFO.PRG is a VFP 6 version of this function. Remember to pass the array to fill by reference by prefixing it with @. Here's the code:

```
lparameters taArray, ;
  tcCDXName, ;
  tuWorkarea
local lcCDXName, ;
  lcAlias, ;
  lnTags, ;
  lnI, ;
  lcTag, ;
  lcType, ;
  lcExpr, ;
  lcFilter, ;
  lcOrder, ;
  lcCollate

* If the CDX was specified, use it. Otherwise, we'll use
* the structural CDX.

lcCDXName = iif(vartype(tcCDXName) = 'C' and ;
  not empty(tcCDXName), tcCDXName, '')

* If an alias or workarea was specified, use it.
* Otherwise, use the current workarea.

if pcount() = 3
  lcAlias = alias(tuWorkArea)
else
  lcAlias = alias()
endif pcount() = 3

* Ensure we have a valid alias.

if empty(lcAlias) or not used(lcAlias)
  error 13, lcAlias
endif empty(lcAlias) ...

* Get the number of tags, dimension the array, and get
* the information for each tag.

lnTags = tagcount(lcCDXName, lcAlias)
dimension taArray[lnTags, 6]
for lnI = 1 to lnTags
  lcTag = tag(lcCDXName, lnI, lcAlias)
  do case
    case primary(lnI, lcAlias)
      lcType = 'PRIMARY'
    case candidate(lnI, lcAlias)
      lcType = 'CANDIDATE'
    case unique(lnI, lcAlias)
      lcType = 'UNIQUE'
    otherwise
      lcType = 'REGULAR'
  endcase
  lcExpr    = key(lcCDXName, lnI, lcAlias)
  lcFilter  = for(lnI, lcAlias)
  lcOrder   = iif(descending(lcCDXName, lnI, lcAlias), ;
    'DESCENDING', 'ASCENDING')
  lcCollate = idxcollate(lcCDXName, lnI, lcAlias)
  taArray[lnI, 1] = lcTag
  taArray[lnI, 2] = lcType
```

```
  taArray[lnI, 3] = lcExpr
  taArray[lnI, 4] = lcFilter
  taArray[lnI, 5] = lcOrder
  taArray[lnI, 6] = lcCollate
next lnI
return lnTags
```

CURSORTOXML() and XMLTOCURSOR()

Although CURSORTOXML() follows next in alphabetical order, I want to cover it and XMLTOCURSOR() together. However, XMLTOCURSOR() is currently a "work-in-progress" (in other words, it doesn't work <g>), so we'll cover these functions in a future article.

DISPLAYPATH()

This function is used when you want to display a filename but don't have a lot of space to do so. You specify the filename and maximum length, and DISPLAYPATH() will return a string up to that many characters with an ellipsis (…) in place of some of the characters if necessary (such as "d:\…\myfile.txt"). Run TESTDISPLAYPATH.PRG to see an example.

EDITSOURCE()

This is one of those functions being added to the language to support tools that come with VFP rather than directly for our benefit. EDITSOURCE() brings up the appropriate editor for something, whether it's a class, a program, a report, stored procedures in a database, etc. It has several advantages over using the appropriate MODIFY command: you don't have to worry about which MODIFY command to use, you can specify a line number to position the cursor to, and you can specify the ID of an editor shortcut (this is the real reason this function is being added: to support editor shortcuts in the new Task List).

Obviously, this isn't the kind of function you'll use in a typical data entry application. However, tool builders can use it as a hook into VFP's source editors. Here's an example: in my September 1998 column ("The Happy Project Hooker"; you just knew I had to find a way to mention that title again, didn't you? <g>), I presented a class for searching for text in files in a project. The results were presented in a grid, and you could edit the appropriate source by simply double-clicking its row in the grid. Unfortunately, the best I could do then was bring up the editor for the source using the appropriate MODIFY command, so there was a good-sized CASE structure to handle each of the different file types, and there was no way to tell the editor to move to the particular line where the text was found. In VFP 7, this is way easier: a single EDITSOURCE() function eliminates the CASE statements and puts the cursor on the first line where the text was found.

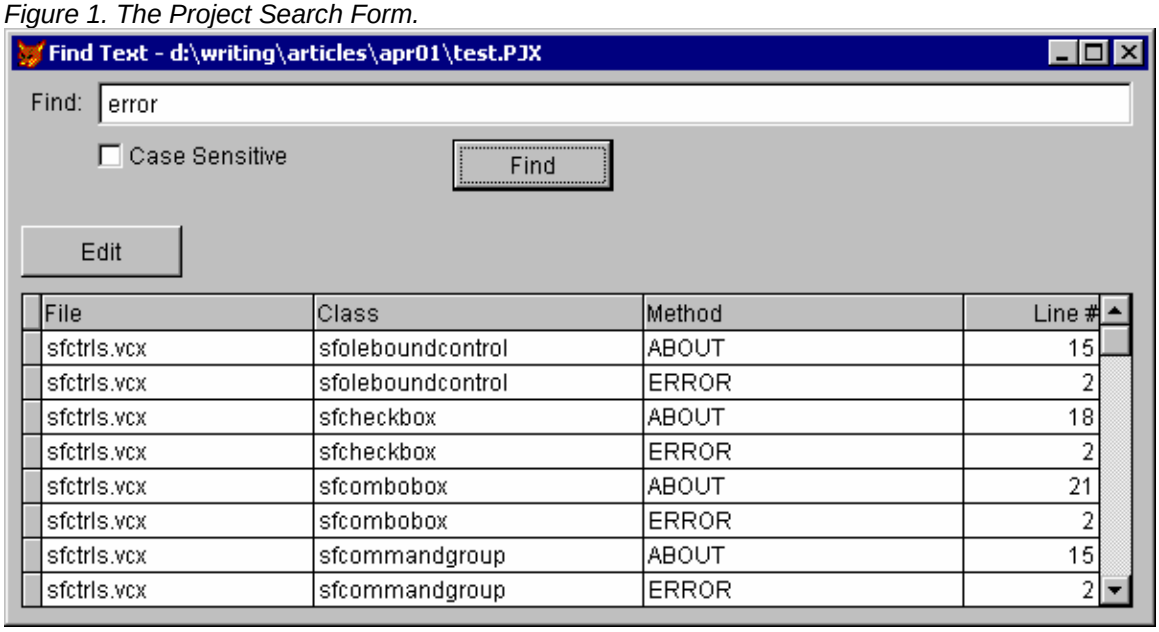
SFPROJUTILS.VCX contains two classes: SFProjectFind, which does the actual job of searching through all the files in a project for the specified text, and SFFindText, a form which allows you to enter the text to search for, displays the results in a grid, and brings up the appropriate editor when a result is selected. SFFINDTEXT.SCX is a form based on the SFFindText class. We won't look at the code here, but the key line related to this topic is in the EditFile method of SFFindText, called when you double-click on a row in the grid:

```
editsource(alltrim(FILES.FILENAME), FILES.LINENO, ;
  alltrim(FILES.CLASS), alltrim(FILES.METHOD))
```

This tells VFP to bring up the editor for the specified file (and class and/or method in the case of VCX or SCX files) and put the cursor on the line where the text was found (this information is all found in fields of a cursor called FILES).

Before trying this out, COMPILE CLASSLIB SFPROJUTIL.VCX and COMPILE FORM SFFINDTEXT.SCX to ensure they'll work in the version of VFP you're using. To use the project search utility, either open a project and DO FORM SFFINDTEXT to search that project or DO FORM SFFINDTEXT WITH <name of the project to search>. Enter the text to search for and click on the Find button to perform the search. To edit a result, double-click on it in the grid. In VFP 7, the cursor will be on the line where the text was found; in VFP 6, the cursor will either be on line 1 or the line you were on when you last edited that source.

Figure 1. The Project Search Form.



EXECSCRIPT()

We discussed this new function, which accepts a string containing VFP code as a parameter and executes it, just a few months ago in my December 2000 column. I presented a runtime command interpreter that uses EXECSCRIPT() to execute the code entered into an editbox; you can add this utility to your applications so you have the equivalent of the VFP Command window in a runtime environment. I also presented EXECSCRIPT.PRG, which provides the VFP 6 equivalent by writing the string to a file, compiling it, executing it, and then deleting it.

Conclusion

Next month, we'll continue looking at new commands and functions. After we've finished with those, we'll look at OOP, COM, and UI improvements in VFP 7.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's award-winning Stonefield Database Toolkit. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the Microsoft FoxPro Developers Conferences (DevCon) since 1997, as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP). He can be reached at dhennig@stonefield.com.