New Language Features in Visual FoxPro 7.0

By Doug Hennig

Introduction

Every new version of Visual FoxPro introduces changes to the programming language and Visual FoxPro 7.0 is no exception. This document highlights new and changed commands, functions, properties, events, and methods. Due to time constraints, we won't discuss database events, COM enhancements, IntelliSense, or IDE enhancements, all of which introduce some changes to commands and functions.

String and Array Functions

VFP has always had powerful and extremely fast string and array processing functions. VFP 7 adds several new functions and greatly enhances the features of a number of existing commands and functions.

ALINES()

ALINES() has been a favorite function of mine since it was introduced. It parses a string (which may be in a variable or memo field) into lines terminated with a carriage return (CR), linefeed (LF), or carriage return-linefeed combination (CRLF), and puts each line into its own element in an array. In VFP 7, ALINES() can now accept one or more parse characters so the "lines" can be delimited with something other than CR and LF. For example, a comma-delimited field list can easily be converted to an array. Why would you want to do this? Because it's easier to process an array than a comma-delimited string. Here's a brute-force way of processing the items in such a string:

```
lnItems = occurs(',', lcString) + 1
lnOld   = 1
for lnI = 1 to lnItems
  lnPos  = iif(lnI = lnItems, len(lcString) + 1, ;
    at(',', lcString, lnI))
  lcItem = substr(lcString, lnOld, lnPos - lnOld)
  lnOld  = lnPos + 1
* do something with lcItem here
next lnI
```

This is ugly code to both read and write. Here's a more elegant approach that converts commas to CR, and then uses ALINES():

```
lcString = strtran(lcString, ',', chr(13))
lnItems  = alines(laItems, lcString)
for lnI = 1 to lnItems
  lcItem = laItems[lnI]
* do something with lcItem here
next lnI
```

In VFP 7, the first two lines in this code can be reduced to the following:

```
lnItems = alines(laItems, lcString, .F., ',')
```

TESTALINES.PRG shows an example of this new feature.

ASCAN()

Two things I've wished for a long time that Microsoft would add to ASCAN() are the ability to specify which column to search in and to optionally return the row rather than the element (to avoid having to subsequently call ASUBSCRIPT() to get the row). My wish was granted in VFP 7, plus ASCAN() gains the ability to be exact- or case-insensitive. The new fifth parameter specifies the column to search in and the new sixth parameter is a "flags" setting that determines if the return value is the element or the row and if the search is exact- or case-insensitive.

Because I always want the row and never want the element number, normally want a case-insensitive but exact search, and have a difficult time remembering exactly which values to use for the flags (no wise cracks from younger readers <g>), I created ARRAYSCAN.PRG, which accepts an array, the value to search for, the column number to search in (the default is column 1), and logical parameters to override the exact and case-insensitive settings. Here's the code in ARRAYSCAN.PRG; note the comment about a workaround for a bug in the current version of VFP 7.

```
lparameters taArray, ;
  tuValue, ;
  tnColumn, ;
  tlNotExact, ;
  tlCase
external array taArray
local lnColumn, ;
  lnFlags, ;
  lnRows, ;
  lnColumns, ;
  lnRow
#define cnCASE_SENS   0
#define cnCASE_INSENS 1
#define cnEXACT_OFF   4
#define cnEXACT_ON    6
#define cnRETURN_ROW  8
lnColumn  = iif(vartype(tnColumn) = 'N', tnColumn, 1)
lnFlags   = iif(tlNotExact, cnEXACT_OFF, cnEXACT_ON) + ;
  iif(tlCase, cnCASE_SENS, cnCASE_INSENS) + cnRETURN_ROW
lnRows    = alen(taArray, 1)
lnColumns = alen(taArray, 2)
lnRow     = ascan(taArray, tuValue, -1, -1, lnColumn, lnFlags)
*** WORKAROUND FOR BUG IN VFP: with return row flag set, return value is 1 too
*** high if the search column is the last one
if lnRow > 0 and lnColumn > 1 and lnColumn = lnColumns and (lnRow > lnRows or ;
  taArray[lnRow, lnColumn] <> tuValue)
  lnRow = lnRow - 1
endif lnRow > 0 ...
return lnRow
```

TESTARRAYSCAN.PRG demonstrates how ARRAYSCAN.PRG works.

ASORT()

VFP 7 adds one new feature to ASORT(): a case-insensitivity flag (in VFP 6, ASORT() is always case-sensitive).

EXECSCRIPT()

This new function accepts a string, which should be VFP code, as a parameter and executes it. Why is this capability useful? That's like asking why macro expansion in VFP is useful—it gives you the ability to define behavior at run time rather than development time. You can generate some code that depends on certain run-time conditions, and then execute that code. One use is giving more advanced users the ability to script new behavior in your application. For example, consider an accounting system that gives its users the ability to perform additional tasks for key operations. Perhaps a user wants to send an e-mail to a client when an order from that client is posted. He could write some code to send the e-mail and tell the accounting system to execute that code when an order is posted.

Even if you don't provide such advanced capabilities in your applications, here's a simple one you might want to use: a run-time command interpreter you can hook into your application so you can have the equivalent of the VFP Command Window in a run-time environment. Such a utility is simple to build: It's just a form with an editbox for entering code and a command button to execute the contents of the editbox using EXECSCRIPT().

TESTEXECSCRIPT.PRG shows how EXECSCRIPT() works. Enter some code in the memo window that appears and that code will be executed.

GETWORDCOUNT() and GETWORDNUM()

As was done with several filename processing functions (JUSTPATH(), JUSTSTEM(), JUSTFNAME(), etc.) in VFP 6, these two functions are being taken from FOXTOOLS.FLL and added as native functions in VFP 7. GETWORDCOUNT() returns the number of words found in a string and GETWORDNUM() returns the specified word (by index) from a string. Although the default delimiters are space, tab, and CR, you can specify different delimiters for processing different types of strings (for example, you could specify a comma and carriage return to process comma-delimited files).

My first thought when reading about these functions was "so what?"; since I never used these functions in FOXTOOLS, why would I need them in the language? Then one use hit me: finding fields in VFP expressions. Because of the types of tools I develop, I frequently need to determine which fields are involved in an expression. For example, I might want to know if a particular field has an index on it so if someone wants to remove that field, the index needs to be removed as well. This is more complicated than it seems. Just using the $ or AT() functions is prone to being fooled; for example, "TAX" $ "UPPER(CUSTOMER.TAXSTATUS)" returns .T. so it incorrectly appears that the TAX field is used in the expression. So, to do this previously, I had to do a lot of careful (and ugly) string parsing to check exactly how a field name was used in an expression. Once I started thinking about GETWORDCOUNT() and GETWORDNUM(), a much simpler approach made itself clear.

FIELDSINEXPR.PRG is the new version of this code. The first parameter is an array (passed by reference using @) that it'll fill with all the fields involved in the expression passed as the second parameter. The optional third parameter is the alias to add to each field name if the field isn't aliased in the expression; if it isn't passed, unaliased fields are left unaliased. This function uses GETWORDCOUNT() to determine how many "words" there are in the expression; the delimiter for these "words" are punctuation characters that can appear in VFP expressions, such as #, <, >, (,), etc. It then uses GETWORDNUM() in a loop to get each "word". Numbers are ignored, as are functions (the character following the word is an opening parenthesis) and strings (the character at the start of the word is a quote or opening square bracket). The "word" is then aliased if it isn't already and an alias was passed, and then if it isn't already in the array, it's added. Here's the part of the code in this routine that uses these functions:

```
#define ccDELIMITERS ' !#%^*()-+/,<>[]{}'
lnWords = getwordcount(lcExpression, ccDELIMITERS)
for lnI = 1 to lnWords
  lcWord  = getwordnum(lcExpression, lnI, ccDELIMITERS)
```

TESTFIELDSINEXPR.PRG tests this using several different expressions. Some of them are slightly different from others to show that FIELDSINEXPR.PRG can handle differently formatted expressions.

Another place where these functions can be used is processing HTML. The FoxWiki (an excellent VFP resource developed by Steven Black; http://fox.wikis.com), for example, automatically hyperlinks a word in "camel notation" (an upper-cased letter in the middle of a word, such as "FoxWiki") to another topic with that word as the title. This allows someone to enter the text of a document and automatically link it to other documents without having to enter <A HREF> tags. GETWORDCOUNT() and GETWORDNUM() would make the text parsing this kind of ability requires simpler.

RETURN

VFP 7 provides the ability to return arrays from functions and methods. Here's an example taken from TESTRETURNARRAY.PRG:

```
dimension laTest[1]
local laArray[1]
laArray = TestArray()
clear
display memory like la*

function TestArray
dimension laTest[2]
laTest[1] = 'howdy'
laTest[2] = 'doody'
return @laTest
```

There's something interesting (OK, weird) about this code. Even though it's not receiving the return array, laTest must be dimensioned in the calling routine because that's the name of the array in the TestArray function and in order for this to work, it must be in scope in the calling routine. Also, laTest can't be declared LOCAL anywhere or scoping problems occur. As you can tell, using this feature requires a lot of coupling between the caller and called routines, and coupling is normally something to be avoided like the plague.

Returning property arrays from object methods works a little better, because you don't have to define the array in the calling code. However, the property array can't be PROTECTED, or the same scoping issues occur. Here's an example, also taken from TESTRETURNARRAY.PRG:

```
oTest = createobject('Test')
local laNew[1]
laNew = oTest.Test()
display memory like laNew

define class Test as custom
  dimension aArray[1]
  function Test
    dimension This.aArray[3]
    This.aArray[1] = '1'
    This.aArray[2] = '2'
    This.aArray[3] = '3'
    return @This.aArray
  endfunc
enddefine
```

The main reason this feature was added in VFP 7 is for COM servers. I really doubt I'll use it, because it makes more sense to me to do things the way we do now: create an array in the calling code, pass it by reference to a function, and have the function work on the passed array. This requires little coupling between routines.

SET TEXTMERGE, TEXT, TEXTMERGE(), and SET('TEXTMERGE')

SET TEXTMERGE is a useful command for creating text files where the output has to include the results of VFP expressions; strings between << and >> (you can change the delimiters to something else using SET TEXTMERGE DELIMITERS) are assumed to be expressions to be evaluated if SET TEXTMERGE ON is used. For example, the following code creates an HTML file called TEST.HTML that shows company and contact names in a table (this is a truncated version of TESTTEXTMERGE.PRG included in the source code files accompanying this document):

```
use (_samples + 'data\customer')
set textmerge on to TEST.HTML noshow
\\<html>
\<body>
\<table border=1>
\<tr><th>Company</th><th>Contact</th></tr>
scan
  \<tr>
  \<td><<trim(COMPANY)>></td>
  \<td><<trim(CONTACT)>></td>
  \</tr>
endscan
\</table>
\</body>
\</html>
set textmerge to
```

One problem: what if you really want the output in a variable rather than a file (for example, you want to return the HTML as a string from a VFP COM object)? You'd have to use FILETOSTR() to read the contents of the file back into a variable and then delete the file.

Fortunately, VFP 7 adds the ability to send the text merge directly to a variable rather than to a file by using the TO MEMVAR clause. For example, in the above code, changing the SET TEXTMERGE command to:

```
set textmerge on to memvar lcHTML noshow
```

results in lcHTML containing the HTML and no file being created.

When was the last time you used the TEXT command? Back in FoxPro 1.0? Never? That's because this little-used command originally sent a block of hard-coded text between TEXT and ENDTEXT statements to the screen or the printer, and who needs to do that anymore? Although it's been enhanced over the years to support evaluation of expressions (if SET TEXTMERGE ON is used) and output to a file, few developers I know ever used it. In VFP 7, however, this command is much more useful because, like SET TEXTMERGE, you can now send the block of text to a variable. In addition, the new ADDITIVE, TEXTMERGE, and NOSHOW commands control whether existing variable contents are overwritten or not, whether expressions are evaluated or not (so you don't have to use SET TEXTMERGE ON as a separate command), and whether output is echoed to the screen or not (so you don't have to use SET CONSOLE OFF to suppress it). The following code (taken from TESTTEXT.PRG) does the same thing as the previous program:

```
use (_samples + 'data\customer')
text to lcHTML textmerge noshow
<html>
<body>
<table border=1>
<tr><th>Company</th><th>Contact</th></tr>
endtext
scan
  text to lcHTML additive textmerge noshow
  <tr>
  <td><<trim(COMPANY)>></td>
  <td><<trim(CONTACT)>></td>
  </tr>

  endtext
endscan
text to lcHTML additive textmerge noshow
</table>
</body>
</html>
endtext
```

Notice the difference in techniques: SET TEXTMERGE permits lines of code to be interspersed with the output because you have to prefix output with \ or \\, while everything between TEXT and ENDTEXT is considered part of the output so prefixes aren't needed but you can't mix code and output. That means TEXT is more useful when no processing code is needed and SET TEXTMERGE is more useful when code is needed.

The new TEXTMERGE() function accepts a string and returns that string with any expressions evaluated. It can accept three other parameters: .T. to perform the expression evaluation recursively (so expressions with expressions are evaluated) and the left and right expression delimiter strings. Here's an example:

```
lcText = 'Today is <<date()>>'
? textmerge(lcText)
```

SET('TEXTMERGE') has been enhanced to return the output filename (but not the variable name), SHOW or NOSHOW setting, and the nesting level for evaluations in TEXT statements.

STREXTRACT()

STREXTRACT() is a new function that returns the text between delimiters. Hmm, let me think, what type of text do we know of that's littered with delimiters? Can you spell HTML and XML?

STREXTRACT() accepts up to five parameters: the string to search, the beginning and ending delimiters, the occurrence number to retrieve, and a case-sensitivity flag. You can leave out the beginning delimiters to return text from to the start of the string to the ending delimiter or the ending delimiter to return text from the beginning delimiter to the end of the string.

Here's an example: TESTSTREXTRACT.PRG reads FOXWEB.HTML into a variable, then finds and prints all hyperlinks in the document. STREXTRACT() is used for three tasks in this code. First, it returns the text between "<A HREF" and "</A>" (in the FindLink subroutine), which is the URL and text associated with it. Then, in the first lcHRef assignment statement, it returns the text between "=" and ">", which is the URL to jump to. Finally, in the lcText assignment, it retrieves the string starting from ">", which is the text associated with the URL.

```
clear
lcHTML  = filetostr('FoxWeb.html')
lcLink  = FindLink(lcHTML, 1)
lnOccur = 1
do while not empty(lcLink)
  lcHRef = strextract(lcLink, '=', '>')
  lcHRef = alltrim(strtran(lcHRef, '"'))
  lcText = alltrim(strextract(lcLink, '>'))
  ? 'Link ' + transform(lnOccur) + ': ' + lcText + ;
    ' (' + lcHRef + ')'
  lnOccur = lnOccur + 1
  lcLink  = FindLink(lcHTML, lnOccur)
enddo while not empty(lcLink)
```

```
function FindLink(tcHTML, tnOccur)
return alltrim(strextract(tcHTML, '<a href', '</a>', ;
  tnOccur, 1))
```

For the first lcHRef assignment statement, here's the code you'd have to use without STREXTRACT():

```
lnPos1 = at('=', lcLink)
lnPos2 = at('>', lcLink)
lcHRef = substr(lcLink, lnPos1 + 1, lnPos2 - lnPos1 - 1)
```

I don't know about you, but I always have to spend time figuring out what that last parameter in SUBSTR() should be ("do I subtract 1 here or not?"). So, not only will STREXTRACT() save two lines of code, it'll also help me avoid the infamous "boundary" problem.

STRTOFILE()

The third parameter in STRTOFILE() can now be a numeric flags value instead of logical (which in VFP 6 indicated whether the file is overwritten or appended to). This allows you to specify whether UTF-8 or Unicode Byte Order Marks are added to the start of the file.

STRTRAN()

STRTRAN() now supports a new sixth parameter to permit searches to be case-insensitive and to determine what case to use for the replacement expression. Here's an example of the use of this new parameter (taken from TESTSTRTRAN.PRG):

```
clear
lcString   = 'The quick BROWN fox'
lcOldValue = 'brown'
lcNewValue = 'green'
? strtran(lcString, lcOldValue, lcNewValue)
&& displays The quick BROWN fox
? strtran(lcString, lcOldValue, lcNewValue, -1, -1, 1)
&& displays The quick green fox
? strtran(lcString, lcOldValue, lcNewValue, -1, -1, 2)
&& displays The quick BROWN fox
? strtran(lcString, lcOldValue, lcNewValue, -1, -1, 3)
&& displays The quick GREEN fox
?
lcOldValue = 'BROWN'
? strtran(lcString, lcOldValue, lcNewValue)
&& displays The quick green fox
? strtran(lcString, lcOldValue, lcNewValue, -1, -1, 1)
&& displays The quick green fox
? strtran(lcString, lcOldValue, lcNewValue, -1, -1, 2)
&& displays The quick GREEN fox
? strtran(lcString, lcOldValue, lcNewValue, -1, -1, 3)
&& displays The quick GREEN fox
```

Notice the following interesting things:

- Use 1 as the sixth parameter for case-insensitivity. The first STRTRAN() fails to do anything because "brown" isn't found in lcString, but the second succeeds because it's case-insensitive.
- Specifying 2 or 3 as the sixth parameter tells VFP to use the case of the found text for the case of the replacement text. Notice in the fourth, seventh, and eighth examples above that "GREEN" appears in the result even though "green" was specified as the replacement text. That's because the text being replaced ("BROWN") is all uppercased.

Resource Management Commands and Functions

A lot of the commands and functions in VFP fall in the category of "resource management." Resources can be anything an application needs to access, such as file system resources (files, directories, and drives, whether local or network resources), printers, functions in Windows or other DLLs, dialogs, Help files, language support, fonts, memory, and so on. This section describes the changes made in commands and functions in VFP 7 related to resource management.

_DBLCLICK and _INCSEEK

There are two changes to _DBLCLICK in VFP 7. First, rather than a default value of 0.5 second, it now respects the double-click speed setting in the Mouse applet of the Windows Control Panel, a welcome change to those developers trying to make their applications respect the user's Windows settings. Second, _DBLCLICK is now used only as the double-click rate and not for incremental searching in controls such as combo boxes and list boxes. The new _INCSEEK system variable, which supports the same range of values as _DBLCLICK (0.05 to 5.5 seconds), is used as the incremental search interval

ADIR()

ADIR() now accepts a fourth parameter: pass 0 to force filenames to uppercase (the behavior of previous versions of VFP and the default behavior if the fourth parameter isn't passed), 1 to preserve filename case, and 2 to show names in DOS 8.3 format (so MyLongFileName.Txt would appear as MYLONG~1.TXT).

ADLLS(), CLEAR DLLS, and DECLARE DLL

The new ADLLS() function fills an array with declared DLL functions. The array contains the name of the function, the alias name it was given, and the DLL in which it exists. TESTADLLS.PRG shows an example of this, showing the DLL functions loaded by the FoxPro Foundation Classes Registry class:

```
clear
oRegistry = newobject('Registry', ;
  home() + 'FFC\Registry.vcx')
oRegistry.LoadRegFuncs()
adlls(laDLLs)
display memory like laDLLs
```

Here are the contents of the first row of the array created by this code:

```
laDLLs[1, 1]    RegCloseKey
laDLLs[1, 2]    RegCloseKey
laDLLs[1, 3]    C:\WINNT\system32\ADVAPI32.DLL
```

CLEAR DLLS can now accept a list of function aliases to clear from memory; in earlier versions, it cleared all DLL functions from memory. Combining this new capability with ADLLS() makes unloading only certain sets of functions much easier in VFP 7.

In addition to the usual data types (such as INTEGER and STRING), DECLARE now accepts a new one, OBJECT, both for parameters and the function return value. This was primarily added for Active Accessibility support.

ANETRESOURCES()

This function, which fills an array with the share or printer resources on a specific server, now accepts 0 for the third parameter to fill the array with both share and printer resources. Previously, you had to call it twice, first passing 1 for shares and then 2 for printers, to get both types of resources. Also, you can now pass a domain name as well as a server name for the second parameter. Here's an example (substitute your server or domain name for "MyServer"):

```
anetresources(laAll, '\\MyServer', 0)
display memory like laAll
```

ASTACKINFO()

This new function is 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 run time, 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 on my system (the paths on yours will obviously be different):

```
(1,1)   1
(1,2)   "d:\writing\sessions\new language vfp 7\test.exe"
(1,3)   "testastackinfo"
(1,4)   "f:\writing\articles\apr01\testastackinfo.prg"
(1,5)   4
(1,6)   "do ShowForm"

(2,1)   2
(2,2)   "d:\writing\sessions\new language vfp 7\test.exe"
(2,3)   "showform"
(2,4)   "f:\writing\articles\apr01\testastackinfo.prg"
(2,5)   10
(2,6)   "do form test"

(3,1)   3
(3,2)   "d:\writing\sessions\new language vfp 7\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)   "d:\writing\sessions\new language vfp 7\test.vct"
(4,3)   "testform.cmdclickme.click"
(4,4)   "f:\writing\articles\apr01\test.vct"
(4,5)   3
(4,6)   "astackinfo(laStack)"
```

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. The other two rows show the complete object method hierarchy for the executing code. The fact that the second column doesn't show the EXE name for rows three and four is a bug that hopefully will be fixed in a service pack.

COMPILE

In VFP 7, the COMPILE commands (COMPILE, COMPILE CLASSLIB, COMPILE REPORT, and so forth) respect the setting of SET NOTIFY. With SET NOTIFY OFF, no "compiling" dialog is displayed. This obviously isn't an issue in a development environment, but VFP 6 Service Pack 3 added the ability to compile files in a run-time environment. In that case, this improvement is important for two reasons: In-process COM servers can't display any user interface, and you likely don't want your users to see such a dialog.

Another improvement is that relative paths in #INCLUDE statements in include and PRG files are now considered relative to the location of the source file, not the current directory, when they're compiled. In addition, VFP 7 now searches more extensively to find an Include file. For a VCX or SCX, any relative path in the Include file specification is applied in the following search order:

1. The directory the VCX or SCX is in
2. The current directory
3. The VFP path

For PRGs, the search order is:

1. The current directory (including relative path)
2. The VFP path (including relative path)
3. For compilation done from a project build, the project directory (including relative path)
4. For compilation done from a project build, the directory the PRG is in (including relative path)
5. The directory the PRG is in (no relative path)
6. VFP home directory (no relative path)

DISKSPACE()

In previous versions of VFP, DISKSPACE() returned incorrect values when there was more than 2GB of free disk space (this wasn't a bug in VFP, but in the Windows API function DISKSPACE() calls). This has been fixed in VFP 7, as long as the operating system is newer than the first release of Windows 95 (Win95 OSR2 or later). In addition, you can pass a new second parameter to determine whether DISKSPACE() should return the total amount of space on the volume, the amount of free space, or the amount of free space available to the current user.

DISPLAYPATH()

This new 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.

GETDIR() and CD ?

GETDIR() has three new parameters: the caption of the dialog (the default is "Browse for Folder"), flags to indicate the behavior of the dialog, and .T. to treat the starting directory as the root so the user can't navigate above it. If any of the new parameters are passed, this function uses the SHBrowseForFolder Windows API function, so the dialog has the same user interface as other Windows applications.

The flags parameter supports all the flags SHBrowseForFolder supports (such as displaying an editbox where the user can type a folder name or including files as well as folders in the dialog). In Windows 2000, there's also support for drag and drop, reordering, context menus, new folders, and other functionality. The VFP Help topic for GETDIR() includes a list of some of the values for this parameter; the SHBrowseForFolder topic in the MSDN Help file has a complete list. Like other functions that support a similar parameter (such as the MESSAGEBOX() function), you can add together different values to provide the behavior you want. Here's an example:

```
lcDir = getdir('', 'Select directory', 'Import From', ;
  16 + 1, .T.)
```

The resulting dialog has "Import From" as the caption, displays "Select directory" and an edit box (the "16" in the flags parameter), only allows physical locations to be selected (the "1" in the flags parameter), and doesn't allow the user to navigate above the current directory.

The dialog displayed when you type CD ? is the same one you see if you specify 64 (use the Windows 2000 interface) as the flags parameter for GETDIR().

GETFONT()

GETFONT() has a new optional fourth parameter to specify the default language script. Passing a value for this parameter enables the Script combobox in the dialog and includes the selected language script code in the return string.

HOME()

To meet Windows 2000 requirements, applications should store user-specific files in a user application data directory. In VFP 7, this directory is something like C:\Documents and Settings*user name*\Application Data\Microsoft\Visual FoxPro. By default, VFP stores tables such as the FoxUser resource file, the IntelliSense FoxCode table, and the Task List FoxTask table in this directory. You can now determine the location of this directory by passing 7 to the HOME() function.

INPUTBOX()

This new function displays a dialog in which the user can enter a single string. Interestingly, this dialog has existed in VFP since the beginning but wasn't accessible before: it's the same dialog displayed to get values for parameterized views. You can specify the prompt, the dialog caption, a default value, a timeout value, and a value to be returned if the dialog times out. It uses the current _Screen icon for its icon. The following code (TESTINPUTBOX.PRG in the sample files) opens the VFP sample CUSTOMER table, asks you for a city (the default is "San Francisco"), and shows how many customers were found in that city:

```
use _samples + 'data\customer'
```

```
lcCity = inputbox('Enter the city to search for:', 'Find City', ;
  'San Francisco', 5000, 'San Francisco')
if not empty(lcCity)
  select count(*) from customer ;
    where city = lcCity ;
    into array laFind
  wait window transform(laFind[1]) + ;
    ' customers were found in ' + lcCity
endif not empty(lcCity)
```

Because we have no control over the appearance of the dialog other than the prompt, caption, and icon, it's hard to say whether this function will be used much in real applications. However, for developer tools or simple applications, it saves you having to create a form just so the user can enter a single string.

MESSAGEBOX()

MESSAGEBOX() has two new features: a new fourth parameter to specify the timeout value in milliseconds (after the timeout expires, the dialog automatically closes and the function returns -1) and auto-transformation of non-character values. Here's an example that displays the current date and automatically closes after two seconds:

```
messagebox(date(), 'Current Date', 0, 2000)
```

OS()

OS() now accepts a whole ton of new values to determine different things about the operating system, such as the major and minor version numbers and build number, which service pack is installed (including major and minor version numbers), and which additional products may be installed (such as BackOffice or Terminal Server). For example, on my system (Windows 2000 Server with Service Pack 1 installed), OS(1) returns "Windows 5.00", OS(7) returns "Service Pack 1", and OS(11) returns 3 (indicating Windows 2000 Server).

SET('CENTURY')

Passing 3 as the second parameter returns the rollover date setting in the Regional Options Control Panel applet of Windows 98, ME, and Windows 2000 (for example, on my system, it returns 2029). In Windows 95 and NT, it returns -1.

WDOCKABLE()

VFP 7 supports "dockable" system windows. A dockable window is like a toolbar in that it can be docked at any edge of the screen. The Command, Document View, Data Session, Properties, and various Debugger windows are all dockable. To visually change a window's current dockable state, right-click in its title bar and choose Dockable. If Dockable is checked, the window can be docked. The new WDOCKABLE() function returns the current dockable state of the specified window and can optionally change it.

Data-Related Commands and Functions

The defining characteristic of VFP is its built-in database engine. It's been a while since Microsoft made improvements in the engine, but VFP 7 has several.

ASESSIONS()

This new function fills an array with the DataSessionID values of all existing data sessions. TESTASESSIONS.PRG 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. One problem with this technique: it won't handle any private data sessions created by Session objects. So, here's the VFP 7 version of this routine:

```
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
```

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: get all of the information about the tags of a table. It even works with non-structural indexes as long as they're open. 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.

BROWSE

At long last, we get a NOCAPTION option for BROWSE that displays the actual field names rather than the captions defined for the fields in the database container.

CURSORTOXML(), XMLTOCURSOR(), and XMLUPDATEGRAM()

The inclusion of several new string processing capabilities (such as STREXTRACT() and TEXTMERGE()) might lead you to believe that VFP 7 is more XML-oriented than previous versions, and you'd be correct. Three new XML functions have been added in VFP 7. These functions make it easy to transfer data between XML and VFP cursors.

CURSORTOXML() creates XML from a cursor. You can specify the alias or workarea of the cursor, the name of a file or memory variable to send the XML to, the output format (element-centric, attribute-centric, or generic attribute-centric), the type of XML schema to create, the type of encoding to use, how many records to output, a schema location, and a namespace to use. Here's an example (taken from TESTXML.PRG):

```
cursortoxml('customer', 'CUSTOMER.XML', 1, 512, 0, '1')
```

This creates a file called CUSTOMER.XML (the second and fourth parameters specify this) from the CUSTOMER cursor (the first parameter) with element-centric XML (the 1 parameter), all records (the 0 parameter), and an in-line schema (the '1' parameter).

XMLTOCURSOR() does the opposite: it creates a VFP cursor from XML. Specify the XML data or the name of a file containing the XML, the name of the cursor to create ("XMLRESULT" is used if none is specified), and a flag indicating how to treat the XML. Here's an example (also taken from TESTXML.PRG):

```
xmltocursor('CUSTOMER.XML', 'MyCustomers', 512)
```

This creates a cursor called MyCustomers from CUSTOMER.XML (512 indicates that the first parameter is a filename).

XMLUPDATEGRAM() creates an XML updategram (before and after snapshots of the changes in data) for one or more cursors. The resulting XML can be used to commit these changes to a database, such as submitting it to SQL Server. XMLUPDATEGRAM() accepts a comma-delimited list of aliases or workareas and a flag indicating how the XML should be formatted and encoded. This function requires table buffering in order to work. Before calling XMLUPDATEGRAM(), use CURSORSETPROP() to specify the key fields for the cursors if you want to minimize the size of the XML (the updategram will consist of only the key and changed fields in that case; otherwise, it'll consist of all fields). Here's an example that creates an XML updategram from the cursor in the current workarea:

```
lcUpdate = xmlupdategram()
```

Run TESTXML.PRG to see how these functions work.

GETNEXTMODIFIED()

GETNEXTMODIFIED() returns the record number of the first or next modified record (depending on what parameters are passed) in a table-buffered cursor. This can be used, for example, to determine if Save and Revert buttons in a toolbar should be enabled. The following routine, DATACHANGED6.PRG, is an example:

```
local llChanged, ;
  laTables[1], ;
  lnTables, ;
  lnI, ;
  lcAlias, ;
  lcState

* Check each open table to see if something changed.

llChanged = .F.
lnTables  = aused(laTables)
```

```
for lnI = 1 to lnTables
  lcAlias = laTables[lnI, 1]
  do case

* This cursor isn't buffered, so we can ignore it.

    case cursorgetprop('Buffering', lcAlias) = 1

* This is a row-buffered cursor, so check the current
* record.

    case cursorgetprop('Buffering', lcAlias) < 4
      lcState   = getfldstate(-1, lcAlias)
      llChanged = not isnull(lcState) and ;
        lcState <> replicate('1', fcount(lcAlias) + 1)

* This is a table-buffered cursor, so look for any
* modified record.

    otherwise
      llChanged = getnextmodified(0, lcAlias) <> 0
  endcase
  if llChanged
    exit
  endif llChanged
next lnI
return llChanged
```

There's one major problem with this routine: GETNEXTMODIFIED() fires the rules (field, table, and index uniqueness) for the current record before figuring out what the next modified record is. This prevents the routine from working if there's anything wrong with the current record.

In VFP 7, GETNEXTMODIFIED() can now accept .T. as a third parameter to prevent rules from firing. DATACHANGED7.PRG has the same code as above except .T. is specified as the third parameter to GETNEXTMODIFIED(). To see the difference in behavior, run TESTGETNEXTMODIFIED.PRG.

IN Clause

This isn't really a command or function itself, but the IN clause was added to several existing commands: BLANK, CALCULATE, PACK, RECALL, and SET FILTER. This eliminates the need to save the current workarea, perform the action, and then restore the workarea as you do in previous versions of VFP. For example, instead of:

```
lnSelect = select()
select CUSTOMER
set filter to CITY = 'New York'
select (lnSelect)
```

you can simply use:

```
set filter to CITY = 'New York' in CUSTOMER
```

ISREADONLY()

You can now determine if the current database is read-only by passing 0 as the parameter.

SELECT READWRITE

As you probably know, a cursor created with a SQL SELECT statement is read-only. While this isn't often an issue, since cursors are typically used for reports or other read-only processing, it can be if further processing has to be performed on the cursor before it's used, such as adding summary records or performing calculations on the cursor. Before VFP 7, the way you made a cursor read-write was to open it in another workarea, using code similar to the following:

```
select ... into cursor TEMP1
select 0
use dbf('TEMP1') again alias TEMP2
use in TEMP1
* TEMP2 is a read-write copy of the cursor
```

In VFP 7, you can cut this down to a single command by adding the new READWRITE clause to the SELECT statement:

```
select ... into cursor TEMP1 readwrite
```

SET REPROCESS

The new SYSTEM clause for this command allows you to control record locking in the "system" data session, which is used for internal access to tables such as databases, the resource file, SCX and VCX files, etc. To see an example of this, fire up two instances of VFP 7, type SET REPROCESS TO AUTOMATIC SYSTEM in the Command window of each one, then modify a class in a VCX in one instance and try to modify the same class in the same VCX in the second. Notice that VFP will wait to get a lock on the class' records in the VCX until you either close the class in the other instance or press Esc. Press Esc to cancel the attempt. Now type SET REPROCESS TO 2 SECONDS SYSTEM and try again. This time, VFP tries for two seconds to lock the records, and gives up when it can't.

To determine the value of this setting, use SET('REPROCESS', 'SYSTEM').

SYS(3054)

This indispensable function, which tells you the Rushmore optimization levels for SQL SELECT commands (including those executed when a view is opened), has been made even better in VFP 7. Passing 2 or 12 for the first parameter tells VFP to include the SQL SELECT statement in the output (making it easier to figure out which of several SQL SELECT statements the output applies to). A new third parameter sends the results to a variable rather than _SCREEN, which allows you to control the output (for example, you could log it to a file) or to automate analysis (for example, warning the developer if "none" or "partial" appears in the results but not if "full" does). You must specify the variable name as a string rather than a name (that is, as "VariableName" instead of VariableName). The following, taken from TESTSYS3054.PRG, shows the use of these new features:

```
close tables all
use (_samples + 'data\customer')
use (_samples + 'data\orders') in 0
sys(3054, 12, 'lcRushmore')
select CUSTOMER.COMPANY, ;
    max(ORDERS.ORDER_DATE) ;
  from CUSTOMER ;
    join ORDERS ;
      on CUSTOMER.CUST_ID = ORDERS.CUST_ID ;
  group by CUSTOMER.CUST_ID ;
  into cursor TEST
clear
? lcRushmore
```

```
sys(3054, 0)
```

USE

Whether you use remote views in production applications or not, they are still useful tools. For example, although I normally use SQL Passthrough or ADO when accessing non-VFP data in production, in a development environment, I sometimes find it handy to create a DBC and some remote views so I can quickly browse the data. One downside of remote views compared to SQL Passthrough, though, is that the connection information is hard-coded (either in a connection stored in the database or in an ODBC datasource). This means this information can't be changed, so you can't use the same remote view to access data on a different server, for example. Also, user names and passwords aren't encrypted in the DBC, so that's a potential security risk.

VFP 7 adds the ability to specify a connection string in the USE command for a remote view. If one is specified, it overrides the defined connection information for the view. If an empty string is specified, VFP displays the Select Data Source dialog.

To see an example of this, do the following steps. First, create an ODBC datasource called "Northwind" for the Northwind database that comes with Access using the Access ODBC driver (if you don't have Access installed, used the SQL Server Northwind database and the SQL Server driver instead). Next, type the following in the Command window (the TEST database referenced in this code is included in the source code files accompanying this document):

```
open database TEST
use rv_customers connstring 'dsn=Northwind'
```

What's so remarkable about this? Well, the RV_CUSTOMERS view is defined as:

```
create sql view "rv_customers" ;
  remote connect "NorthwindConnection" ;
  as select * from customers customers
```

The NorthwindConnection connection specified here uses the SQL Server driver to access the Northwind database on my server named DHENNIG. Specifying CONNSTRING in the USE command allows you to create a cursor from the CUSTOMERS table in the Access database on your system instead.

The benefit of this enhancement is that you can store the server, user name, and password in a table, build a connection string, and then use it in the USE command for your remote views. This gives you both flexibility (you can easily change the server, user name, and password in the table) and security (the password can be encrypted in the table and decrypted when the connection string is built).

VALIDATE DATABASE RECOVER

Formerly, this command could only be used in "interactive mode" (that is, from the VFP Command window). In VFP 7, this command can now be used in code, allowing you to use it in a run-time environment. Of course, since this is a potentially dangerous command (it can remove tables from a DBC, for example) that displays dialogs that are confusing to the typical end-user ("What *is* a backlink and why would I want to update or remove it?"), you'll want to only use it in "administrator" functions.

SYS() Functions

What would a new release of VFP be without a few new SYS() functions? Half of them provide features to COM servers, so they won't be discussed here.

VFP 6 Service Pack 4 added a new setting configurable only in CONFIG.FPW: BITMAP. Normally, VFP updates the screen by drawing a bitmap in memory, then moving the entire bitmap to the screen at once. While this provides better video performance for normal applications, it slows down applications running under Citrix or Windows Terminal Server. Those environments look for changes to the screen and send them down the wire to the client system. Since every change causes the entire screen to be sent to the client, rather than just the changes made, this results in a lot of data being sent. Adding BITMAP=OFF turns off this screen drawing behavior. OK, that's the background. What's new to VFP 7 is SYS(602), which allows you to determine the value of this setting.

Technically, SYS(1104) isn't new (I'm not sure which release it was added in), but in VFP 7, it's finally documented. This function purges memory cached by programs and data. Since the amount of memory available to VFP can make a huge difference in performance, calling this function after executing commands that make extensive use of memory buffers (such as SQL SELECT statements) can help your application's speed.

Normally, VFP won't let you use the debugger in "system component" code, such as the Class Browser. According to the help file, SYS(2030) allows you to enable or disable debugging features within this code. I was really hoping this would allow us to use the debugger to trace code in or called from a report, but that doesn't appear to be the case.

VFP has an internal list of code pages it supports, called the "national language support," or NLS, list. The following code pages are in the default list: 874 (Thai Windows), 932 (Japanese Windows), 936 (Chinese [PRC, Singapore] Windows), 949 (Korean Windows), 950 (Chinese [Hong Kong SAR, Taiwan] Windows), 1255 (Hebrew Windows), and 1256 (Arabic Windows). SYS(2300) is a new function that allows you to add or remove support for code pages. Pass it the code page number and either 0 to remove the code page from the NLS list or 1 to add it. If you omit the third parameter (0 or 1), the function just tells you whether the code page is in the list (1, if it is; 0, if not).

SYS(2600) either copies the contents of a specified range of memory addresses to a string or copies the contents of a string to memory. Readers familiar with BASIC will recognize these features as the equivalent of the PEEK() and POKE() functions (honestly, I didn't make those names up!). To see an example of this function, type GETENV and an open parenthesis in the Command window and notice the list of environment variables that appears. That list is read from memory using SYS(2600) (see the DATA memo field for the GETENV record in FOXCODE.DBF to see how it was done).

SYS(2800) enables or disables support for Microsoft Active Accessibility.

SYS(2801) extends event tracking to include Windows mouse and keyboard events. You can configure it to track VFP events only (the current behavior and the default setting), Windows events only, or both.

Miscellaneous Commands and Functions

ALANGUAGE()

This function fills an array with VFP language elements. The first parameter is the array to fill and the second is the type of language elements desired: commands, functions, base classes, or DBC events. In the case of functions, the array will have two columns: one for the name of the function and one showing a range of the minimum required and total number of parameters plus a flag indicating if the entire function name is required in code.

Why did Microsoft add this function to the language? Mostly, I think, because of IntelliSense. I suspect that they needed something like this themselves for IntelliSense or the IntelliSense Manager, so we get it for free. Would you actually use this function yourself? I doubt it, unless you need to parse FoxPro code for some purpose (such as a documenter or Beautify-like utility).

AMEMBERS()

This is one of those functions you likely don't use very often, but is very useful for those times you need it. AMEMBERS() fills an array with the properties, events, and methods (PEMs) of an object or class. There are two changes to AMEMBERS() in VFP 7: you can now get the PEMs of a COM object and you can specify a filter for the PEMs.

Passing 3 for the third parameter indicates the second parameter is a COM object. Here's an example that puts the PEMs for Excel into laPEMs:

```
oExcel = createobject('Excel.Application')
? amembers(laPEMs, oExcel, 3)
```

A new fourth parameter allows you to filter the list of PEMs so the array contains only a desired subset. For example, you may want only protected, hidden, or public PEMs, native or user-defined PEMs, changed PEMs, etc. This is handy, because prior to VFP 7, the only way to filter a list of PEMs was to use PEMSTATUS() on each one. For example, imagine a form that you pass an object to and have the user modify the properties of the object in the form's controls. What if the user wants to press a Cancel button to undo their changes? I decided to copy the object's properties to another object of the same class, then have the form work on the new object, and if the user chooses OK, copy the properties of the edited object to the original object. If the user chooses Cancel instead, the original object isn't touched. So, the form creates another instance of the passed object's class and then calls a utility function named CopyProperties to copy the properties of the original object to the new instance. COPYPROPERTIES6.PRG has the VFP 6 version of this function.

The VFP 7 version is a bit simpler. First, it uses the new "G" flag so the array only contains the public properties of the source object, so we don't have to use PEMSTATUS() to later ignore protected or hidden properties. Next, although there's currently a bug that prevents it from working with array properties, the "C" flag ensures the array only contains properties that have changed from their default values; when this bug is fixed (notice I'm being optimistic and didn't say "if" <g>), we'll be able to use that flag and remove the CASE that checks for changed properties. Finally, although there isn't a flag to include only read-write properties in the array (there is one for read-only properties), the # flag tells AMEMBERS() to include all attributes of each property in an additional column of the array, so we can skip those that have "R" (for read-only) in that column. Thus, the VFP 7 version is simpler and faster than its VFP 6 counterpart. Here's the code for COPYPROPERTIES7.PRG:

```
lparameters toSource, ;
  toTarget
local lcFlags, ;
  laProperties[1], ;
  lnProperties, ;
  lnI, ;
  lcProperty, ;
  luValue

* Get an array of public, changed properties and process each one.
* Note: due a bug in the current version of VFP 7 that doesn't put changed
* array properties into the array, we'll avoid the "C" flag for now and test
* for a changed property later.

*!*     lcFlags      = '#G+C'
```

```
lcFlags      = '#G'
lnProperties = amembers(laProperties, toSource, 0, lcFlags)
for lnI = 1 to lnProperties
  lcProperty = laProperties[lnI, 1]
  do case

* Ensure the property exists in the target object and is not read-only.

    case 'R' $ laProperties[lnI, 2] or ;
      not pemstatus(toTarget, lcProperty, 5)

* If this is an array property, use ACOPY (the check for element 0 is a
* workaround for a VFP bug that makes native properties look like arrays; that
* is, TYPE('OBJECT.NAME[1]') is not "U").

    case type('toTarget.' + lcProperty + '[0]') = 'U' and ;
      type('toTarget.' + lcProperty + '[1]') <> 'U'
      acopy(toSource.&lcProperty, toTarget.&lcProperty)

* We normally wouldn't need this case to test for an unchanged property, but we
* need it for now due to the array properties bug mentioned above.

    case not 'C' $ laProperties[lnI, 2]

* Copy the property value to the target object.

    otherwise
      luValue = evaluate('toSource.' + lcProperty)
      store luValue to ('toTarget.' + lcProperty)
  endcase
next lnI
return
```

TESTCOPYPROPERTIES.PRG shows how CopyProperties can be used.

APROCINFO()

Like ALANGUAGE(), I suspect this new function was added because new tools that come with VFP require it rather than because VFP developers requested it. It fills an array with information from a PRG file, including things like PROCEDURE and FUNCTION statements, compiler directives, and class definitions. Run TESTAPROCINFO.PRG to see how it works. As with ALANGUAGE(), it might be useful if you're creating a documenter or some other utility that works with PRGs.

By the way, if you haven't already done so, you might want to download a free utility written by Michael Emmons called Class Navigator that provides a Class Browser-like tool for PRG files. It's available at http://www.comcodebook.com. I bet if Michael were to write this program again, he'd use this new function because it seems perfectly suited to what his utility does.

BITOR(), BITXOR(), and BITAND()

These functions can now accept more than the two parameters they do in VFP 6; they'll accept up to 26 parameters in VFP 7. This is useful in cases (such as some API functions and COM objects) where several flags have to be ORed together; in VFP 6, you have to use something like BITOR(BITOR(BITOR(expr1, expr2), expr3), expr4) to do this.

CTOT()

In VFP 7, CTOT() supports the XML date format YYYY-MM-DDTHH:MM:SS.SSS (such as 2001-03-22T14:35:49), which is returned by XML queries against SQL Server and other databases that support it. One quirk, however, is that you have to SET DATE YMD for this to work. Hopefully, that'll be fixed in a service pack.

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).

MODIFY PROCEDURE and MODIFY VIEW

These commands now have an optional NOWAIT clause to bring them in line with other MODIFY commands.

QUARTER()

This new function returns the quarter the specified Date or DateTime expression falls in. You can specify the starting month for the year (the default is 1) so it can be used for either calendar or fiscal years. TESTQUARTER.PRG demonstrates this function.

VERSION()

The build number for VFP 7 now appears in the final section of the version number (for example, "07.00.0000.9147") rather than the third section (such as "06.00.8862.00"). This, of course, will break any code you wrote that checks the build number, so be sure to update any such code.

Object PEMs

While commands and functions have the majority of the improvements in VFP 7, several new PEMs were added to various objects as well. Some of these improvements allow you to create applications with more modern interfaces.

Form.ShowInTaskBar

The new ShowInTaskBar property of Form indicates whether top-level forms appear in the Windows taskbar. The default is .T.; setting it to .F. means the form doesn't appear in the taskbar and it minimizes to the desktop rather than to the taskbar.

Form/Toolbar.hWnd

Forms and toolbars now have an hWnd property which contains the handle of the appropriate window. Window handles are used by many Windows API functions to do something with a window, such as sending a message to it. Obtaining the window handle for VFP in previous versions meant calling a Windows API function such as GetDesktopWindow(). Now, you simply refer to the hWnd property.

Grid.HighlightRowLineWidth

This new property specifies the thickness of the border around the selected grid row in pixels. This setting is only used when HighlightRow is set to .T. Unfortunately, this still doesn't provide what VFP developers really want: an easy way to completely highlight the selected row. Run TESTHIGHLIGHTROWLINEWIDTH.SCX to see the effect of setting this property.

Grid.RowColChange

This new property indicates what caused the BeforeRowColChange and AfterRowColChange events to fire. A value of 1 means the row changed, 2 means the column changed, 3 means both changed, and 4 means neither changed.

Header.WordWrap

Setting this new property to .T. allows column headers to wrap over multiple lines. The Customer ID column in TESTHIGHLIGHTROWLINEWIDTH.SCX has this property set to .T.

MouseEnter and MouseLeave Events

All visible controls have these new events, which (as you likely guessed from their names) fire when the mouse enters or leaves the borders of a control. Some uses of this are to change the foreground color of the text in a textbox or editbox or to make a checkbox appear in bold when the mouse is over it. For an example of these events, run TESTHOTTRACKING.SCX and notice that the text in the textbox becomes bold when you move the mouse over it. Look at the code in the MouseEnter and MouseLeave events of this textbox to see how the behavior is implemented.

Objects Collection

One non-polymorphic aspect of VFP is that every container class has a different way of accessing its members: OptionGroup has a Buttons collection, PageFrame has a Pages collection, Grid has a Columns collection, and so on. This makes writing generic code to drill down through the members of an object more complex; you have to use CASE statements to handle different container types. In VFP 7, all container classes now have an Objects collection. Objects.Count returns the number of members and Objects[*index*] returns a reference to the specified member.

ProjectHook.QueryNewFile, Activate and Deactivate

These events fill some gaps in the ProjectHook object model. QueryNewFile fires when you click the New button in the Project Manager. This could be used to run a wizard to create the new file. Activate and Deactivate fire when the project becomes active or loses focus. These events are handy for setting up the appropriate environment for a project, such as changing to the project's directory, setting the VFP PATH, changing field mappings and _INCLUDE, and so on.

Open the DEMO project accompanying this document. It has TestProjectHook in TESTPROJECTHOOK.VCX as its ProjectHook. Notice the WAIT WINDOW that displays, indicating the Activate event fired. Click on the Command window; a WAIT WINDOW indicates the Deactivate event fired. Click on the project window; Activate will fire again. Click on the New button; QueryNewFile will fire with the parameter indicating the type of file being created.

Separators.Style

Set this new property to 1 to create Office 2000-like toolbars: the separators will appear as a vertical rule. Run TESTTOOLBAR.PRG, which instantiates the TestToolbar class, to see how a separator with Style set to 1 looks.

SpecialEffect and CommandButton.VisualEffect

Setting the SpecialEffect property of CheckBox, ComboBox, CommandButton, EditBox, ListBox, OptionButton, Spinner, and TextBox controls to the new 2-Hot Tracking value makes them appear flat except when the mouse is over them, in which case they appear raised (CommandButton, as well as CheckBox and OptionButton if Style is set to 2-Graphical) or sunken (the rest of the controls). Run TESTHOTTRACKING.SCX to see how hot tracking works.

CommandButton has a new VisualEffect property (only available at run-time) that programmatically changes the appearance of the button. Set it to 0 for no effect, 1 to make it appear raised, or 2 to make it appear sunken. This is useful, for example, when clicking on one control should change the appearance of another. For an example, DO (_SAMPLES + 'SOLUTION\SOLUTION') to run the SOLUTION application that comes with VFP, expand the New Features for VFP 7.0 node, run the Use New Style Toolbars sample, and click on the Show Toolbar button. Move the mouse over the down arrow beside the table button and notice that they both appear raised; this behaves just like the Back button and associated down arrow button in Microsoft Internet Explorer. To see how this was done, close the toolbar, click on the Toolbar Source button, and look at the MouseEnter and MouseLeave events of Command2 (the down arrow button). They change Command1.VisualEffect appropriately.

WriteMethod Method

VFP 7 adds a third parameter to this method, which allows you to programmatically add code to a method; pass .T. to create the specified method if it doesn't exist. This would be useful, for example, for builders to both create a new method and add code to it.

_SCREEN and _VFP

In previous versions of VFP, there was no way to determine the exact coordinates of the main VFP window, _Screen. While menu bars and docked toolbars obviously reduce the amount of usable space in _Screen, its Top, Left, Height, and Width properties reflected the values of the entire VFP application window. In VFP 7, these properties of _Screen now apply only to the VFP "client" area, while those in _VFP apply only to the entire VFP application window.

For example, the following table shows the values of these properties in VFP 6 and 7 on a system using 1280x1024 resolution and running VFP in a maximized window with the Standard toolbar docked at the top and the status bar turned on. The reason the Top and Left properties of _VFP are negative is that the border of the window is by default moved outside the viewable area of the screen. Notice in VFP 7 that the Top and Height properties of _Screen show the effects of the menu bar, docked toolbar, and status bar. Removing any of these or docking additional toolbars or windows at any edge alters the size of the client area, so the properties of _Screen change appropriately but those of _VFP do not.

	VFP 6	VFP 7

Property	_Screen	_VFP	_Screen	_VFP
Top	-4	-4	69	-4
Left	-4	-4	0	-4
Height	899	899	902	1003
Width	1280	1280	1280	1288

Another change is that both of these system variables now have an hWnd property; this property was described earlier. Like the Form base class, _Screen has a new ShowInTaskBar property; because it's read-only for _Screen, it doesn't really serve a useful purpose.

You can configure some IntelliSense and editor window settings using the new EditorOptions property of _VFP. This property contains a string in which each character represents a setting; an empty string completely disables IntelliSense. Include "L" and "Q" in the string to enable Auto List Member and Auto Quick Info, or "l" and "q" to disable the automatic capability of these features but permit manual activation. "T" enables the tip window displayed in lists. "K" enables the display of hyperlinks in editor windows, and "W" enables drag and drop between words.

_VFP.VFPXMLProgID provides a way to override the functionality of the new XML functions. Specify the ProgID of a COM component to use instead of the built-in VFP functions. See the VFP 7 help topic for this property for details.

Using undeclared memory variables can lead to some of the thorniest bugs in VFP. A variable can suddenly take on an unexpected value after a subroutine has been called if that subroutine uses the same variable. This often doesn't cause an error, which would be easy to debug, but incorrect behavior instead, which is much harder to detect. To help track this problem down, _VFP has a new LanguageOptions property. Setting this property to 1 sends information to the debugger output (either the Debug Output window or a file if you use the SET DEBUGOUT command) when any code references a variable that hasn't been declared as PUBLIC or LOCAL. Since PRIVATE doesn't actually create a variable, any variables declared as PRIVATE are considered undeclared. Also, variables created on the fly by a command or function, such as arrays created with functions like ADIR() or variables created with SCATTER, are considered PRIVATE and therefore undeclared. Note that this only works in the development environment (it's ignored at run time) and only when you run the code (it's not handled at compile time, which would have been even better).

To see this setting in action, run TESTLANGOPTIONS.PRG:

```
_vfp.LanguageOptions = 1
activate window 'Debug Output'

a = 1
private b
b = 1
public c
c = 1
local d
d = 1
adir(e, '*.*')
do TestVars

_vfp.LanguageOptions = 0
```

```
procedure TestVars
f = 2
```

It produces the following in the Debug Output window (the path for the file shown in the fifth item in each line has been removed for brevity):

```
LangOptionsErr,4/7/2001 9:58:15 AM,4,TESTLANGOPTIONS,TESTLANGOPTIONS.FXP,A
LangOptionsErr,4/7/2001 9:58:15 AM,6,TESTLANGOPTIONS,TESTLANGOPTIONS.FXP,B
LangOptionsErr,4/7/2001 9:58:15 AM,11,TESTLANGOPTIONS,TESTLANGOPTIONS.FXP,E
LangOptionsErr,4/7/2001 9:58:15 AM,17,TESTVARS,PROCEDURE TESTVARS
  TESTLANGOPTIONS.FXP,F
```

The first and second comma-delimited items indicate what the output is for and when the code was executed. The third, fourth, and fifth items are the line number, routine (method, function, or procedure) name, and source code file where the variable was used. The sixth item is the name of the undeclared variable.

Summary

VFP 7's new and improved commands, functions, and PEMs make it easier to write fast, powerful code that processes strings or arrays, allow you to create applications with more modern interfaces, permit exchanging data with other applications via XML, and make it easier to detect problems like undeclared variables. Spend time going through the VFP help file to learn more about the new capabilities of our favorite language.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of the award-winning Stonefield Database Toolkit (SDT). He is co-author, along with Tamar Granor and Kevin McNeish, of "What's New in Visual FoxPro 7.0", available from Hentzenwerke Publishing (www.hentzenwerke.com). He writes the monthly "Reusable Tools" column for FoxTalk. Doug has spoken at 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).

Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341 Fax: (306) 586-5080
Email: dhennig@stonefield.com
Web: www.stonefield.com