anguage Enhancements in VFP 7, Part II

Doug Hennig

This article continues the series on language enhancements in VFP 7, discussing enhancements to existing commands and functions.

This is the second in a series of articles on language enhancements in VFP 7. The first three articles cover improvements to existing commands and functions, going through them in alphabetical order. Once we're finished with these, we'll move to new commands and functions. Last month, I discussed improvements to ALINES(), AMEMBERS(), ASCAN(), ASORT(), the "bitwise" functions, BROWSE, and COMPILE. This month, we'll cover from CLEAR DLLS (yeah, that comes before COMPILE; I missed it last month <s>) to SELECT. Just as a reminder, we'll keep three things in mind:

- We'll briefly discuss the purpose of each command and function without getting too bogged down in details (for example, we won't discuss parameters or return values in detail; you can get that from the VFP 7 help).
- We'll look at practical uses for the commands and functions.
- If possible, we'll discuss a way to get the functionality now in VFP 6.

Also, remember that, as of this writing, VFP 7 isn't even in beta yet (it's considered a Tech Preview version) and these articles are based on the version distributed at DevCon in September 2000. So, commands and functions may be added, removed, or altered.

CLEAR DLLS

This command can now accept a list of function aliases to clear from memory; in earlier versions, it cleared all DLL functions from memory. You can use the new ADLLS() function, which we'll cover in a future issue, to determine which functions have been declared.

DECLARE DLL

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.

DISKSPACE()

In previous versions of VFP, DISKSPACE() returned incorrect values when there was more than 2 GB of free disk space; this has been fixed in VFP 7, as long as the operating system is newer than the early release of Windows 95 (Win95 OSR2 or later). In addition, you can pass a new second parameter to determine what 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 user.

In VFP 6, I use GetDriveSpace, a routine based on code posted by George Tasker on the Universal Thread (www.universalthread.com). GetDriveSpace accepts two parameters: any directory on the drive to check space for (if it isn't specified, the current drive is used), and .T. to return the total space on the drive or .F. to return the free space. It uses the GetVersionEx API function to determine if the operating system supports the GetDiskFreeSpaceEx API function, which returns the correct amount of disk space; if it can't use this function, it resorts to using DISKSPACE(). GetDriveSpace uses two other functions, Decimal2Hex and Hex2Decimal, to handle converting between hex and decimal values (the code for these two routines isn't shown here). Here's the code for GetDriveSpace.prg:

```
lparameters tcDirectory, ;
  tlTotal
local lcCaller, ;
  lcTotal, ;
  lcFree, ;
  lcStruct, ;
  lcDir, ;
  lnResult

* Declare a Win32 API function to determine the version.

declare integer GetVersionEx in Win32API ;
  string @lpVersionInformation

* Create the version information structure.

store replicate(chr(0), 8) to lcCaller, lcTotal, lcFree
lcStruct = Decimal2Hex(148) + replicate(chr(0), 144)

* If the path wasn't specified, use the current drive.
* Otherwise, get the drive for the specified path,
* handling UNC paths specially.

do case
  case vartype(tcDirectory) <> 'C' or empty(tcDirectory)
    lcDir = sys(5)
  case left(tcDirectory, 2) = '\\'
    lcDir = addbs(tcDirectory)
    lcDir = left(lcDir, at('\', lcDir, 4))
  otherwise
    lcDir = addbs(justdrive(tcDirectory))
endcase

* If this is VER_PLATFORM_WIN32_WINDOWS and OSR2 or
* later, use the GetDiskFreeSpaceEx API function to get
* the drive space.

if GetVersionEx(@lcStruct) <> 0 and ;
  inlist(Hex2Decimal(substr(lcStruct, 17, 4)), 1, 2) and ;
  Hex2Decimal(substr(lcStruct, 13, 4)) > 1000
  declare short GetDiskFreeSpaceEx in Win32API ;
    string @lpDirectoryName, ;
    string @lpFreeBytesAvailableToCaller, ;
    string @lpTotalNumberOfBytes, ;
    string @lpTotalNumberOfFreeBytes
  if GetDiskFreeSpaceEx(@lcDir, @lcCaller, @lcTotal, ;
    @lcFree) <> 0
    lnResult = Hex2Decimal(iif(tlTotal, lcTotal, lcFree))
  endif GetDiskFreeSpaceEx(@lcDir, ...

* Since we can't use GetDiskFreeSpaceEx, just use
* DISKSPACE.

else
  lnResult = diskspace(lcDir)
endif GetVersionEx(@lcStruct) <> 0 ...
return lnResult
```

GETDIR()

GETDIR() has several new parameters: the caption of the dialog, 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. This function and the CD command now use the SHBrowseForFolder Win32 API function, so the dialog has the same UI as other Windows applications and the flags parameter supports all the flags this function 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. Here's an example:

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

The resulting dialog has “Import From” as the caption, displays “Select directory” and an editbox, and doesn’t allow the user to navigate above the current directory.

GETFONT()

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

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 the following program (TestGetNextModified.prg):

```
* Set up an error handler.

on error llError = .T.
llError = .F.
```

```
* Open the CUSTOMER table and table buffer it.

open database (_samples + 'DATA\TESTDATA')
use CUSTOMER
set multilocks on
cursorsetprop('Buffering', 5)

* Do some data updates; the second one duplicates the
* primary key of the first record.

replace COMPANY with 'Howdy'
skip
replace CUST_ID with 'ALFKI'

* Try using GETNEXTMODIFIED; an error will occur.

llChanged = DataChanged6()
if llError
  aerror(laError)
  wait window laError[2]
  llError = .F.
else
  wait window 'Data ' + iif(llChanged, 'was', ;
    'was not') + ' changed'
endif llError

* Try again with the lNoFire parameter; no error should
* occur.

llChanged = DataChanged7()
if llError
  aerror(laError)
  wait window laError[2]
  llError = .F.
else
  wait window 'Data ' + iif(llChanged, 'was', ;
    'was not') + ' changed'
endif llError

* Clean up and exit.

tablerevert(.T.)
on error
```

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.

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)
```

MODIFY PROCEDURE and MODIFY VIEW

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

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).

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.

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
```

Conclusion

Next month, we'll finish examining improved commands and functions in VFP 7, and then move on to new commands and functions.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author 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.