anguage Enhancements in VFP 7, Part I

Doug Hennig

Want an early peek at what's new in VFP 7? This is the first article in a series discussing the language enhancements Microsoft is implementing and how to take advantage of some of them now.

In last month's article, I discussed "version envy": wanting to use features planned in an upcoming version of a product in the current version. In that article, I discussed several new functions in VFP 7 and showed PRGs that can provide their functionality in VFP 6. After reading this article, FoxTalk editor Whil Hentzen thought it'd be a great idea to extend this to covering all the new commands and functions in VFP 7, both to familiarize you with them and to enable you, where possible, to use them in VFP 6.

So, we'll start the new year off with this idea and spend several months looking at language enhancements in VFP 7. We won't cover everything new in VFP 7 (such as Intellisense, editor improvements, DBC events, etc.); it would take a whole book to cover a topic that large. Instead, we'll focus on functions and commands with 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.

Keep in mind 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.

We'll start off with 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.

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. 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, 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, , ',')
```

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, I use the following routine, CopyProperties, in VFP 6 to copy the properties of one object to another instance of the same class. Why would you want to do that? 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 CopyProperties to copy the properties of the original object to the new instance. Here's the code for CopyProperties (in COPYPROPERTIES6.PRG in this month's Source Code Downloads):

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

* Get an array of properties and process each one.

lnProperties = amembers(laProperties, toSource)
for lnI = 1 to lnProperties
  lcProperty = laProperties[lnI]
  do case

* Ensure the property exists in the target object, is not
* read-only, and is not protected. Note: having separate
* statements for PEMSTATUS is a workaround because VFP
* has a problem with two such commands within one line of
* code.

    case not pemstatus(toTarget, lcProperty, 5)
    case pemstatus(toTarget, lcProperty, 1)
    case pemstatus(toTarget, lcProperty, 2)

* 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)

* If the property wasn't changed, we can ignore it. Note:
* due to a bug in PEMSTATUS(), which returns .F. for
* array properties even if they've changed, we have to do
* this test after handling arrays above.

    case not pemstatus(toSource, lcProperty, 0)

* Copy the property value to the target object.

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

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, we’ll be able to use the “C” flag so 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 remove the PEMSTATUS() check for changed properties. Finally, I’ve submitted an enhancement request (ER) to Microsoft to provide a flag for read-write properties. If this ER is implemented, we’ll be able to remove the PEMSTATUS() check for read-only properties. Thus, the VFP 7 version will be simpler and faster than its VFP 6 counterpart. Here’s the code for COPYPROPERTIES7.PRG (I removed a few comments that duplicate those in the VFP 6 version for space reasons):

```
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. Note: if a flag is added
* so we can filter out read-only properties, we'll add
* that too.

*!* lcFlags = 'G+C'
lcFlags = 'G'
lnProperties = amembers(laProperties, toSource, 0, ;
  lcFlags)
for lnI = 1 to lnProperties
  lcProperty = laProperties[lnI]
  do case

* Ensure the property exists in the target object and is
* not read-only.
* Note: we can remove the second case if we can filter
* out read-only properties in a later version.

    case not pemstatus(toTarget, lcProperty, 5)
    case pemstatus(toTarget, lcProperty, 1)

* Copy an array property.

    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 pemstatus(toSource, lcProperty, 0)

* Copy the property value to the target object.

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

By the way, if you examine COPYPROPERTIES7.PRG, you'll see the header comments includes my email address and Web site, and that they appear blue and underlined, just like a hyperlink in your browser. Clicking on either of these gives the expected action (a Send Message dialog with my email address already filled in or my Web site in your browser). This editor enhancement makes it simple to direct other developers to your Web site for support, more information or documentation, updates, etc.

TESTCOPYPROPERTIES.PRG shows how CopyProperties can be used. Change the statement calling CopyProperties6 to CopyProperties7 to see how the VFP 7 version works.

Here's another use of AMEMBERS() that's somewhat similar. PERSIST.PRG provides a way to persist the properties of an object so they can be restored at another time (for example, the next time the user runs the application). It creates a string that can be stored, for example, in a memo field in a table. This string contains code that can be used to restore the properties of an object. For example, the string might look like this:

```
.cType = 'C'
dimension .aTest[3]
.aTest[1] = 'string1'
.aTest[2] = 'string2'
.aTest[3] = 'string3'
```

After retrieving this string from wherever it's stored, you'd then do something like this to restore the saved properties (in this example, lcPersist contains the string):

```
oObject = createobject('MyClass')
with oObject
  execscript(lcPersist)
endwith
```

This example uses the new VFP 7 EXECSCRIPT() function, which I discussed last month.

I won't show the code for PERSIST.PRG here, both because of space limitation and because it's quite similar to COPYPROPERTIES7.PRG. To see this routine in action, run TESTPERSIST.PRG.

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, 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 ARRAYSCAN7.PRG (I omitted header comments and ASSERT statements for brevity):

```
lparameters taArray, ;
  tuValue, ;
  tnColumn, ;
  tlNotExact, ;
  tlCase
external array taArray
local lnColumn, ;
  lnFlags, ;
  lnRow

* Determine how we're going to do things based on
* parameters.

#define cnEXACT_OFF   4
#define cnEXACT_ON    5
#define cnCASE_SENS   0
#define cnCASE_INSENS 1
#define cnRETURN_ROW  8
lnColumn = iif(type('tnColumn') = 'N', tnColumn, 1)
lnFlags  = iif(tlNotExact, cnEXACT_OFF, cnEXACT_ON) + ;
  iif(tlCase, cnCASE_SENS, cnCASE_INSENS) + ;
  cnRETURN_ROW

* Do the search and return the row it was found in.

lnRow = ascan(taArray, tuValue, -1, -1, lnColumn, ;
  lnFlags)
* Note: a bug in the current version of VFP 7 returns a
* row value 1 too high if the search column is the last
* one
lnRow = iif(lnRow > 0 and lnColumn = alen(taArray, 2), ;
  lnRow - 1, lnRow)
return lnRow
```

In VFP 6, we can do something similar, but since we don't have the new capabilities of ASCAN(), we have to use a different approach: we'll use ASCAN() to find the value anywhere in the array, then determine if it's in the correct column. If not, we'll change the starting element number and try again. ARRAYSCAN6.PRG has almost the same functionality as ARRAYSCAN7.PRG (albeit more complex code and slower) except support for case-insensitivity; to implement that feature, you'd have to do it the brute force method of going through each row in the array and looking for a case-insensitive match in the desired column. Here's the code for ARRAYSCAN6.PRG:

```
lparameters taArray, ;
  tuValue, ;
  tnColumn, ;
  tlNotExact
external array taArray
local lnColumn, ;
  lnRow, ;
  lnRows, ;
  lnColumns, ;
  lnElement, ;
  lnStartElement, ;
  lnCol
lnColumn  = iif(vartype(tnColumn) = 'N', tnColumn, 1)
lnRow     = 0
lnRows    = alen(taArray, 1)
lnColumns = alen(taArray, 2)
lnStartElement = 1
do while .T.
  lnElement = ascan(taArray, tuValue, lnStartElement)
  if lnElement <> 0
    lnCol = iif(lnColumns > 1, ;
      asubscript(taArray, lnElement, 2), 1)
    if lnCol = lnColumn and (tlNotExact or ;
      taArray[lnElement] == tuValue)
      lnRow = iif(lnColumns > 1, ;
        asubscript(taArray, lnElement, 1), lnElement)
      exit
    endif lnCol = lnColumn ...
    lnStartElement = lnElement + 1
  else
    exit
  endif lnElement <> 0
enddo while .T.
return lnRow
```

TESTARRAYSCAN.PRG demonstrates how both ARRAYSCAN6.PRG and ARRAYSCAN7.PRG work.

ASORT()

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

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.

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. You can get this behavior in VFP 6 by running BROWSEIT.PRG instead of using BROWSE. This relies on the fact that a browse window is really a grid, so we can change the caption of each column to the actual field name. Here's the code for BROWSEIT.PRG:

```
browse name oBrowse nowait
for each loColumn in oBrowse.Columns
  loColumn.Header1.Caption = ;
    justext(loColumn.ControlSource)
next loColumn
```

COMPILE

In VFP 7, the COMPILE commands (COMPILE, COMPILE CLASSLIB, COMPILE REPORT, etc.) respect the setting of SET NOTIFY. With SET NOTIFY OFF, no "compiling" dialog is displayed. This 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. In VFP 6, we can suppress the dialog using the Window API LockWindowUpdate function, which prevents updates to a window similar to VFP's LockScreen property (although this won't help in-process COM servers since the dialog is still being called). This month's Source Code downloads includes LOCKWINDOW.PRG, which accepts .T. to prevent window updates and .F. to restore window updates. Here's the code for this PRG:

```
lparameters tlLock
local lnHWnd
declare integer LockWindowUpdate in Win32API ;
  integer nHandle
if tlLock
  declare integer GetDesktopWindow in Win32API
  lnHWnd = GetDesktopWindow()
else
  lnHWnd = 0
endif tlLock
LockWindowUpdate(lnHWnd)
return
```

To prevent the “compiling” dialog from being displayed, use code similar to the following:

```
LockWindow(.T.)
compile MyProgram.prg
LockWindow()
```

Conclusion

Next month, we’ll carry on examining improved commands and functions in VFP 7. When we’re finished with them, we’ll 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.