Have Your Cake and Eat it Too

Doug Hennig

Visual FoxPro 7.0 has several new functions that can make certain tasks easier, such as executing script code or performing text merges. However, since it's still a long way from release, this month's article presents several routines that emulate these new functions in VFP 6 so you can start using these capabilities today.

Any time Microsoft releases details of an upcoming version, we go through "version envy": we wish we had some of those new functions *now*. Some times, new functions added to the language don't do anything you couldn't do before, but do it in a single function call rather than a lot of code. In this case, we can take advantage of these functions before the new version is released by creating PRGs with the same names as the desired functions that implement the desired behavior. In VFP 5.0, for example, I created several routines that provide the same behavior as new functions added in VFP 6: NEWOBJECT.PRG, VARTYPE.PRG, JUSTSTEM.PRG, JUSTPATH.PRG, etc. These PRGs were discussed in my November 1999 FoxTalk column. Once you start using the new version of VFP, you simply discard the PRGs and their new function counterparts automatically kick in.

ExecScript

At the Miami DevCon in September 2000, Ken Levy said his favorite new function in VFP 7.0 is EXECSCRIPT(). This 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 runtime rather than development time. You can generate some code that depends on certain runtime conditions, then execute that code. It can give your more advanced users the ability to script new behavior in your application. An example is an accounting system that provides its users with the ability to perform additional tasks for key operations. Perhaps a user wants to send an email to their client when an order from that client is posted. They could write some code to send the email 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 I'll bet you'll want to use: a runtime command interpreter you can hook into your application so you can have the equivalent of the VFP Command window in a runtime environment.

Because I expect to use EXECSCRIPT() extensively in VFP 7.0, I decided to create EXECSCRIPT.PRG so I can begin using its functionality in VFP 6. EXECSCRIPT.PRG accepts a string parameter, the code to execute, and returns either the return value from that code (or .T. if the code doesn't return a value) or .NULL. if the code couldn't execute (such if there's an error in the code or if this PRG is used in a version of VFP prior to VFP 6 Service Pack 3). It works by copying the code to a file, compiling that file (which is why it requires VFP 6 Service Pack 3; before that version, it wasn't possible to use the COMPILE command in a runtime environment), then executing it. Here's the code (this listing doesn't show the comments or ASSERT statements that appear in the PRG file):

```
lparameters tcCode
local luReturn, ;
  lcVersion, ;
  lcFile, ;
  lcName, ;
  luReturn
luReturn = .NULL.
#if type('version(4)') <> 'U'
  lcVersion = version(4)
  lcVersion = substr(lcVersion, ;
    at('.', lcVersion, 2) + 1)
  if lcVersion >= '8492'
    lcFile = 'A' + sys(3) + '.PRG'
    strtofile(tcCode, lcFile)
    compile (lcFile)
    lcName   = juststem(lcFile) + '()'
    luReturn = &lcName
    erase (lcFile)
    erase forceext(lcFile, 'FXP')
  endif lcVersion >= '8492'
#else
  error 'ExecScript requires VFP 6 SP 3 or later'
#endif
return luReturn
```

SFConsoleForm (in SFCONSOLE.VCX) is a simple command interpreter that acts like the VFP Command window but in a runtime environment. It's just a form with an editbox for entering code and an command button to execute the code in the editbox using EXECSCRIPT. However, I added a few bells and whistles, including the ability to load code from a file and to scroll through a code "history" (similar to the history of commands the VFP Command window displays) so you can re-execute code. This class can be called from an error handler to assist with debugging or from a menu item only available to developers at runtime to allow you to perform tasks that would normally require a development copy of VFP at the client site. To see SFConsoleForm in action, run TESTCONSOLE.PRG.

TextMerge

Text merging is a much more important function these days than it was in the past. For example, a lot of applications today output to HTML or XML for reporting purposes. Text merging gives you the ability to mix text (such as HTML tags) and expressions (such as VFP field names) and have the expressions evaluated at runtime to produce some output. Expressions are distinguished from text by delimiters; << and >> are the defaults but they can be changed with the SET TEXTMERGE DELIMITERS command.

VFP 7.0 includes a new TEXTMERGE() function that makes creating strings of mixed text and expressions much easier. This 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.

So I can start using the capabilities of this function now, I created TEXTMERGE.PRG. It works by creating some code that performs the text merge of the specified string to a file, executing that code using EXECSCRIPT.PRG, then reading the contents of the file back into a variable; this process is repeated if necessary until there are no more expressions in the string. You may be wondering why TEXTMERGE uses EXECSCRIPT on some generated code rather than just text merging the string to a file directly using code similar to the following:

```
\\<<lcString>>
```

This won't work for two reasons. First, if the expression delimiters specified to this routine are something other than << and >>, this code won't work because the SET TEXTMERGE DELIMITERS command has changed the delimiters. Second, the text merge commands (\ and \\) aren't recursive, so they can't evaluate the expressions inside the string, which is the whole point of this routine in the first place.

Here's the code for this routine; this listing doesn't show the header comments or ASSERT statements that appear in the PRG file:

```
lparameters tcString, ;
  tlRecurse, ;
  tcLeft, ;
  tcRight
local lcDelimiters, ;
  lnDelimiters, ;
  lcDefaultLeft, ;
  lcDefaultRight, ;
  lcLeft, ;
  lcRight, ;
  lcString, ;
  lcTempFile, ;
  lcCode

* If the left and right delimiters weren't specified,
* use the defaults.

lcDelimiters   = set('textmerge', 1)
```

```
lnDelimiters   = len(lcDelimiters)/2
lcDefaultLeft  = left(lcDelimiters, lnDelimiters)
lcDefaultRight = right(lcDelimiters, lnDelimiters)
lcLeft         = iif(pcount() > 2, tcLeft,  ;
  lcDefaultLeft)
lcRight        = iif(pcount() > 2, tcRight, ;
  lcDefaultRight)

* If the specified string contains the expression
* delimiters, let's process it. First, create a temporary
* filename, then set the textmerge delimiters to the
* desired values.

lcString = tcString
if lcLeft $ lcString
  lcTempFile = sys(2015) + '.txt'
  set textmerge delimiters to lcLeft, lcRight

* As long as the delimiters exist in the string, create
* some code that performs the text merge, execute it,
* then read the file back into the string.

  do while lcLeft $ lcString
    lcCode = 'set textmerge on to ' + lcTempFile + ;
      ' noshow' + chr(13) + '\\' + lcString + chr(13) + ;
      'set textmerge to' + chr(13)
    execscript(lcCode)
    lcString = filetostr(lcTempFile)

* If we're not supposed to recurse, we're done.

    if not tlRecurse
      exit
    endif not tlRecurse
  enddo while lcLeft $ lcString

* Erase the temporary file and restore the delimiters.

  erase (lcTempFile)
  set textmerge delimiters to lcDefaultLeft, ;
    lcDefaultRight
endif lcLeft $ lcString
return lcString
```

To make TEXTMERGE.PRG (and its VFP 7.0 counterpart) even more useful, I created an SFTextMergeProcessor class (in SFTEXTMERGE.VCX) that provides not just expressions inside a string but script code as well. Why would this be useful? Think about an Active Server Page (ASP) document; it's a mixture of HTML and script code. Wouldn't it be nice if you could do something similar in VFP? That would allow you to quickly create an HTML document from some VFP data without having to hard-code the HTML into your application. You could provide an HTML "template" and allow the user to change the template if they desire. Here's an example of such a template:

```
<html>
<body>
<h1><%= dbf() %></h1>
<table border=1>
<tr><th>Company</th><th>Contact</th></tr>
<% scan
lcHTML = '<tr><td>' + trim(COMPANY) + '</td><td>'
lcHTML = lcHTML + trim(CONTACT) + '</td></tr>' + chr(13)
Response.Write(lcHTML)
endscan %>
</table>
</body>
</html>
```

Notice that while this looks like an ASP document, the script code between the <% and %> tags is actually VFP code! The first script code (on the third line) outputs the file name of the current table and the second script code processes every record in the table, outputting the company and contact names into an HTML table.

TESTTEXTMERGE.PRG is a sample program that uses the template shown above (in TEMPLATE.HTML) against the CUSTOMER table in the TESTDATA database. This program creates an HTML file, TEST.HTML, from the processed template and displays the file in your default browser. Note that it has no idea of what the HTML should look like; that's completely handled in the template file.

```
open database _samples + 'data\testdata'
use customer

* Load an HTML template to output company and contact
* name in a table.

lcStr = filetostr('template.html')

* Create the textmerge processor object and process the
* HTML template.

oOutput  = newobject('SFTextMergeProcessor', ;
  'SFTextMerge')
lcOutput = oOutput.Output(lcStr)

* If everything worked, create the HTML file and display
* it.

if empty(oOutput.cErrorMessage)
  strtofile(lcOutput, 'test.html')
  declare integer ShellExecute in SHELL32.DLL ;
    integer nWinHandle, ;
    string cOperation, ;
    string cFileName, ;
    string cParameters, ;
    string cDirectory, ;
    integer nShowWindow
  ShellExecute(0, 'Open', 'test.html', '', '', 1)

* Display the error that occurred.

else
  messagebox(oOutput.cErrorMessage)
endif empty(oOutput.cErrorMessage)
return
```

SFTextMergeProcessor is a simple class. Its Output method processes the specified string and returns the result, evaluating any expressions and executing any script code. Output creates a variable called Response that can be used in the script code to add to the output stream by calling the Write method (which simply adds the passed string to the protected cOutput property); this variable is called Response so it looks like the ASP Response object. Rather than using TEXTMERGE on the entire string, it processes the string one line at a time so it can look for and handle script code (anything between <% and %> delimiters). The result is built up in the cOutput property, and at the end of this method, that property is returned as the result. Here's the code for the Output method:

```
lparameters tcString
local lcCodeLeft, ;
  lcMergeLeft, ;
  laLines[1], ;
  lnLines, ;
  lnI, ;
  lcLine, ;
  lcCode, ;
  lnJ
private Response
with This
```

```
* Create a reference to this object called "Response" so
* script code can call methods.

  Response = This

* Create variables for the left delimiters for code and
* for merge strings.

  lcCodeLeft  = .cLeftDelimiter + ' '
  lcMergeLeft = .cLeftDelimiter + '='

* Initialize our error message to blank.

  .cErrorMessage = ''

* Initialize the output to blank, then process each line
* in the string.

  .cOutput = ''
  lnLines  = alines(laLines, tcString)
  for lnI = 1 to lnLines
    lcLine = laLines[lnI]

* If this line is the start of script code, get all text
* up to the end of the code and execute it.

    if lcLine = lcCodeLeft
      lcCode = ''
      for lnJ = lnI to lnLines
        lcLine = laLines[lnJ]
        lcCode = lcCode + ;
          alltrim(strtran(strtran(lcLine, lcCodeLeft), ;
          .cRightDelimiter)) + chr(13)
        if This.cRightDelimiter $ lcLine
          lnI = lnJ
          exit
        endif This.cRightDelimiter $ lcLine
      next lnJ
      execscript(lcCode)

* Use TEXTMERGE() to process the line and add it to the
* output stream. Because "=" in the left delimiter causes
* TEXTMERGE() to send the result to the bit bucket, let's
* strip it off.

    else
      lcLine = strtran(lcLine, lcMergeLeft, ;
        .cLeftDelimiter)
      .Write(textmerge(lcLine, .T., .cLeftDelimiter, ;
        .cRightDelimiter) + chr(13))
    endif lcLine = lcCodeLeft

* Exit if we had an error.

    if not empty(.cErrorMessage)
      exit
    endif not empty(.cErrorMessage)
  next lnI
endwith
return This.cOutput
```

Conclusion

You *can* have your cake and eat it too! Even though VFP 7.0 is months away from release, you can start using some of the new functionality today using the routines discussed in this article. Hopefully, these PRGs will relieve your version envy a little bit.

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.