New Language Features in Visual FoxPro 7

Session Number
Tamar E. Granor, Ph.D.
Voice: 215-635-1958
Email: tamar@thegranors.com

Overview

Every new version of VFP introduces changes to the programming language and VFP 7 is no exception. This session highlights new and changed commands, functions, properties, events and methods.

The changes in VFP 7 affect many different areas from string manipulation to building COM servers to data handling to the Interactive Development Environment (IDE) to just about every corner of the language. A number of the changes provide features that have been requested by the Visual FoxPro community, some of them for many years.

This session assumes familiarity with Visual FoxPro 6.0.

What's Not in this Session

This session doesn't cover some of the major new language features in VFP 7 because they're big enough to require entire sessions or are covered in other sessions. In particular, VFP 7 adds events to the Database Container, definitely too large a topic to cover here. A number of new functions and changes to existing functions are designed to improve VFP's position in building COM Servers; again, this topic needs more room than would be available here. Finally, some new language elements support changes to the IDE; they're covered only lightly in this session.

In addition, these notes are not meant to cover every single changed item, but rather to look at the major changes to the language.

String Handling

Working with strings has become increasingly important in the last few years, thanks to the World Wide Web. Since HTML and XML are just text with tags, the ability to manipulate text rapidly and easily has taken on new significance. FoxPro has always had good string-handling tools, but with VFP 7, they continue to improve.

Textmerge improvements

FoxPro's textmerge facility is one of those tools whose time came and went and has now come back. It was added in FoxPro 2.0 to make code generation easier for GENSCRN and GENMENU, the screen and menu generating programs.

Textmerge combines low-level file handling with runtime evaluation of expressions and type conversion to provide an easy mechanism for generating complex text. In VFP 7, textmerge has been enhanced in several ways.

First, it's now possible to send textmerge output to a variable rather than a file. SET TEXTMERGE TO has a new MEMVAR clause that lets you specify a variable as the output destination, like this:

```
SET TEXTMERGE TO MEMVAR cContents
```

In addition, the TEXT … ENDTEXT command has been enhanced to increase its textmerge capabilities. The TEXT line now has several options, where in previous version it just said TEXT. Here's the new syntax:

```
TEXT [ TO VarName [ ADDITIVE ] [ TEXTMERGE ] [ NOSHOW ] ]
   Text lines
ENDTEXT
```

The new syntax lets you perform textmerge to a variable with just a TEXT … ENDTEXT sequence, without using SET TEXTMERGE. For example, this block of code sends a string plus the current date to the variable cDate, without echoing it to the screen:

```
TEXT TO cDate TEXTMERGE NOSHOW
Today is <<DATE()>>
ENDTEXT
```

The biggest change on the textmerge front is the addition of a new TextMerge() function that accepts an expression to be evaluated, a flag that indicates whether the evaluation is recursive and a set of textmerge delimiters, and returns the result of the evaluation. This line is equivalent to the example above:

```
cDate = TextMerge("Today is <<Date()>>")
```

The form TextMerge.SCX in the conference materials demonstrates all three ways of merging text.

Parsing strings

A new function, StrExtract(), has been added to aid in breaking strings into their component parts. Although this function was probably added to simplify handling of XML, it has uses in other processing as well.

The syntax for StrExtract() is:

```
cResult = StrExtract( cSource, cBeginningDelimiter [, cEndingDelimiter
                      [, nOccurrence [, nFlags ]]] )
```

The key aspect of this function is that the beginning and ending delimiters can be multiple characters. So you can apply the function to an XML string, passing beginning and ending tags. For example,

```
cXML = "<customer><custid>37</custid>" +
       "<name>Fred's Auto Parts</name></customer>"
?StrExtract( cXML, "<name>", "</name>")
```

displays:

```
Fred's Auto Parts
```

You can extract any occurrence between the specified delimiters by passing the optional nOccurrence parameter. The nFlags parameter is a sum of values. Add 1 to flag to make the search case-insensitive. Add 2 to the flag if the ending delimiter is optional – in that case, the function returns everything from the beginning delimiter to the end of the string. (That's also what happens when the ending delimiter is omitted.)

For example, using the cXML string defined above:

```
?StrExtract( cXML, "<NAME>", </NAME>")
```

returns the empty string, while:

```
?StrExtract( cXML, "<NAME>", </NAME>",1,1)
```

returns:

```
Fred's Auto Parts
```

Miscellaneous string changes

In addition to beefing up the textmerge facility, the VFP team migrated a couple of string functions from FoxTools, and enhanced several others. The result is that working with text in VFP 7 is even easier than it was in VFP 6.

The FoxTools' promotions are GetWordCount() and GetWordNum(), though they don't have those names in FoxTools. They do what their new names suggest: return the number of words and the specified word in their first parameter. Each accepts an optional parameter to indicate where one word ends and another begins – if it's omitted, a space, tab or return separates words. One way to use these two functions together is to output the words in a string one per line. Assuming cInputString contains the string in question, the code looks like this:

```
LOCAL nWordCount, nWordNum
nWordCount = GetWordCount(cInputString)
FOR nWordNum = 1 TO nWordCount
   ? GetWordNum(cInputString, nWordNum)
ENDFOR
```

I compared this code to equivalent code using STRTRAN() and ALINES() to pull out the words and to traditional Xbase code using AT() and SUBSTR() and found that, while the new functions were quite fast on small strings, as soon as cInputString got big, the STRTRAN()/ALINES() combination was a much better choice. Here's the alternative, faster, code:

```
LOCAL nWordCount, cModString
LOCAL ARRAY aWords[1]
cModString = STRTRAN(STRTRAN(cInputString," ",CHR(13)), ;
                     CHR(9),CHR(13))
nWordCount = ALINES(aWords,cModString)
FOR nWordNum = 1 TO nWordCount
   ?aWords[nWordNum]
ENDFOR
```

In fact, the fastest way to do this parsing task in VFP 7 uses just ALINES() because this function has a new parameter to specify additional characters that indicate the end of the line (other than CHR(13) and CHR(10)). Here's the enhanced syntax for ALINES():

```
nNumberOfLines = ALINES( aArrayOfLines, cExpression [, lTrim ]
                         [, cParseChars1, … , cParseCharsn ] )
```

So, the code to output each word in a string onto a separate line can be rewritten one more time as:

```
LOCAL nWordCount
LOCAL ARRAY aWords[1]
nWordCount = ALINES(aWords,cInputString, " ", CHR(9))
FOR nWordNum = 1 TO nWordCount
   ?aWords[nWordNum]
ENDFOR
```

The form WordsOut.SCX in the conference materials runs timing tests on all four approaches to parsing words from a string.

Note that each separator (cParseChars*n*) can contain multiple characters, so you can parse based on sequences of characters, not just single characters. It's also worth noting that the separators are processed in the order in which they occur in the function call.

STRTRAN() also has been changed in this version. It has a new flags parameter that indicates whether the search and replacement it performs should be case-sensitive or not. If the parameter is entirely omitted, the search is case-sensitive, as in older versions of FoxPro. When included, the parameter has three possible values:

0-case-sensitive search

1-case-insensitive search with no case change in the replacement string

3-case-insensitive search with case change in the replacement string

(Note that 2 is an acceptable value for this parameter, but gives the same results as 0.)

For example, without the new parameter, this call:

```
? STRTRAN("Now is the time", "now", "then")
```

returns the original string "Now is the time". With the new parameter, we can call the function like this:

```
? STRTRAN("Now is the time", "now", "then", -1, -1, 1)
```

and it returns:

```
"then is the time"
```

To maintain the original case, make this call:

```
? STRTRAN("Now is the time", "now", "then", -1, -1, 3)
```

and you see:

```
"Then is the time"
```

Note that you can't combine case-sensitive search with changing case in the replacement string.

Data-Related Changes

In addition to the new database events, a number of commands and functions related to data have been changed in VFP 7. Most of the changes are in response to requests from the developer community. All of them make it easier to write solid code.

First, the IN clause has been added to a number of the Xbase commands that didn't already have it. They include BLANK, SET FILTER, PACK, RECALL and CALCULATE. Using the IN clause with these and other Xbase commands means that you don't have to worry about setting the right work area before issuing the command. The IN clause specifies the work area for the individual command and doesn't affect any other commands.

Two changes affect the SQL SELECT command. First, there's a new READWRITE keyword for the INTO CURSOR clause. This keyword lets you create a cursor that can be modified, answering perhaps one of the most common wishes of FoxPro developers ever since SELECT was added in FoxPro 2.0.

When you issue SELECT … INTO CURSOR, the newly created cursor is read-only. There are some tricks you can do to make it read-write. But the new READWRITE clause makes those tricks obsolete. Cursors created with SELECT … INTO CURSOR name READWRITE can be modified. (Of course, the original data on which the cursor is based isn't affect by changes to the cursor.)

Here's an example. This query (which uses data from the example TasTrade database) creates a read-only cursor:

```
SELECT First_Name, Last_Name ;
   FROM (_SAMPLES + "TasTrade\Data\Employee") ;
   INTO CURSOR EmployeeNames
```

Add the READWRITE keyword, and the cursor can be changed:

```
SELECT First_Name, Last_Name ;
   FROM (_SAMPLES + "TasTrade\Data\Employee") ;
   INTO CURSOR EmployeeNames READWRITE
```

It's also worth noting that, in some situations, the first version of the query results in a filter of the original table unless you add the NOFILTER keyword. When you use the READWRITE keyword, you can omit NOFILTER since a read-write cursor cannot be a filtered version of the original table.

The SYS(3054) function that lets you check the optimization of queries has some new parameters in VFP 7. Pass 2 or 12 for the second parameter to include the query itself in the output. This is really handy when you're testing a number of queries. SYS(3054, 2) turns on filter optimization information with the query displayed, as well; in other words, it's the same as SYS(3054, 1) with the addition of the query display. SYS(3054, 12) is SYS(3054, 11) plus the query display – it shows filter and join optimization.

This function has a new parameter as well. You can pass it the name of a variable and the optimization information is stored in that variable rather than being displayed. For example:

```
SYS(3054, 2, "cOptim")
SELECT Last_Name, First_Name ;
   FROM _SAMPLES + "TasTrade\Data\Employee" ;
   WHERE Country = "UK" ;
   INTO CURSOR Junk
```

The variable ends up with these contents:

```
SELECT Last_Name, First_Name     FROM _SAMPLES + "TasTrade\Data\Employee"
WHERE Country = "UK"     INTO CURSOR Junk

Rushmore optimization level for table employee: none
```

Check out the form SYS3054.SCX in the conference materials for additional examples.

Two changes help you get a better picture of what's going on with your data. First, the IsReadOnly() function has been enhanced to work on databases as well as tables. You can test only the current database. To do so, pass 0 to the function like this:

```
?IsReadOnly(0)
```

If no database is open and current, you get an error.

The new ATagInfo() function is one that FoxPro developers have long asked for. It puts information about a table's indexes into an array with one row for each tag. There are columns for the tag name, tag type (primary, candidate, regular or unique), key, filter, ascending/descending status and collation sequence.

Like the other functions that put data into arrays, ATagInfo() creates the array you pass it or resizes it, if necessary. It returns the number of rows in the resulting array. This example runs the function on the Employee table from the TasTrade database and shows some of the results.

```
? ATAGINFO(aEmplTags)
LIST MEMORY like aEmplTags

AEMPLTAGS                 Pub             A
   (    1,    1)                          C     "GROUP_ID"
   (    1,    2)                          C     "REGULAR"
   (    1,    3)                          C     "GROUP_ID"
   (    1,    4)                          C     ""
   (    1,    5)                          C     "ASCENDING"
   (    1,    6)                          C     "MACHINE"
   (    2,    1)                          C     "LAST_NAME"
   (    2,    2)                          C     "REGULAR"
   (    2,    3)                          C     "UPPER(LAST_NAME)"
   (    2,    4)                          C     ""
   (    2,    5)                          C     "ASCENDING"
   (    2,    6)                          C     "MACHINE"
   (    3,    1)                          C     "EMPLOYEE_I"
   (    3,    2)                          C     "PRIMARY"
   (    3,    3)                          C     "EMPLOYEE_ID"
   (    3,    4)                          C     ""
   (    3,    5)                          C     "ASCENDING"
   (    3,    6)                          C     "MACHINE"
```

The form ShowATags.SCX in the conference materials lets you choose a tag and apply ATagInfo() to it.

A change to the GetNextModified() function makes it easier to write your conflict resolution code. By default, even though it doesn't actually write any data, GetNextModified() fires rules regarding the uniqueness of indexes. A new, optional parameter lets you turn off the firing of those rules, so that you can simply find all changed records and deal with them. As you deal with them, of course, you'll have to handle conflicts in primary and candidate indexes, but the appropriate time to do so is as you check the individual records for problems, not when you're identifying them. Here's the updated syntax for GetNextModified():

```
nRecordNumber = GetNextModified( nStartRecord [, cAlias | nWorkarea
                                 [, lIgnoreRules ] ]
```

Here's another VFP 7 enhancements that comes in response to developer requests. The VALIDATE DATABASE RECOVER command that lets you repair damaged databases is now permitted in code, not just at the Command Window. Be aware that VALIDATE DATABASE RECOVER isn't all that smart about fixing your database–usually, its solution is to simply remove whatever piece is causing the trouble and clean up the traces. A better, long-term solution for healthy databases is to use a third-party tool like the Stonefield Database Toolkit.

Resource Management

A number of the new and modified commands and functions make it easier to keep track of what's going on in your application. Some of the changes either fix existing language elements or make them more useful, while others add functionality.

One change that should make a lot of people happy is that the DISKSPACE() function finally works consistently with drives greater than 2GB. The function has also been improved – it now takes an optional parameter that lets you find out the total space on the drive. Omit the parameter or pass 2 for the free space. Pass 1 for total space. For example:

```
? DISKSPACE( "C" )    && 2053126144
? DISKSPACE( "C", 1)  && 3249307648
? DISKSPACE( "C", 2)  && 2053126144
```

Be aware that, for a sufficiently large disk, DISKSPACE()'s return value is in scientific notation.

The OS() function, which tells you what operating system you're running, now provides a lot of other information as well. Table 1 shows the parameters you can pass to OS() and the information it returns.

Table 1 Parameters for OS() – This function has been significantly enhanced in VFP 7. All return values are character, even those identified as numbers.

Parameter	Return Value
Omitted or 1	Operating system version, that is, Windows version.
2	Is double-byte character support available? If so, returns "DBCS"; if not, returns the empty string.
3	Operating system major version number.
4	Operating system minor version number.
5	Operating system build number.
6	Operating system platform (distinguishes Win95/98/ME from WinNT/2000/XP).
7	Operating system service pack.
8	Operating system service pack number.
9	Operating system service pack minor version number.
10	Bit flags to identify product suites available on the operating system, such as Small Business Server, Windows 2000 Advanced Server and so forth.
11	Additional information about the operating system. Currently, this value is only informative for Windows 2000, where it indicates which version is installed.

Table 2 shows the returns values for OS() on two machines: one running Windows 2000 Pro Service Pack 1 with US English, OS(), and the other running Windows XP Pro with US English and Terminal Services installed.

Table 2 OS() return values – The return values for OS() depend on the version of Windows, the Service Pack, the language support installed, and other factors.

Call	Return value - Windows 2000 Pro SP1	Return value - Windows XP Pro
? OS()	"Windows 5.00"	"Windows 5.01"
? OS(1)	"Windows 5.00"	"Windows 5.01"
? OS(2)	""	""
? OS(3)	"5"	"5"
? OS(4)	"0"	"1"
? OS(5)	"2195"	"2600"
? OS(6)	"2"	"2"
? OS(7)	"Service Pack 1"	""
? OS(8)	"1"	"0"
? OS(9)	"0"	"0"
? OS(10)	"0"	"256"
? OS(11)	"1"	"1"

Two changes make it easier to work with dynamic link libraries (DLLs). First, in previous versions, the CLEAR DLLS command was an all-or-nothing affair. While you could DECLARE functions individually, you could only clear them all from memory at the same time. VFP 7 adds the capability to clear individual functions by specifying the name of the function or the alias you assigned it when you declared it. Here's an example:

```
DECLARE INTEGER GetSystemDirectory ;
        IN Win32api AS GetSysDir;
        STRING @cWinDir, ;
        INTEGER nWinDirLength
DECLARE DOUBLE GetSysColor ;
        IN Win32API;
        INTEGER nIndex
* Now do something with these
* Now clean these up without releasing other DLL functions
CLEAR DLLS GetSysDir, GetSysColor
```

The new ADLLS() function lets you check what DLL functions have been declared. In VFP 6 and older versions, the only way to get this information was by parsing the output of LIST STATUS.

Like the other "A" functions, ADLLS() stores its results in an array. This one has three columns. The first column contains the actual name of the function, the second its alias and the third has the name of the library containing the function. Note that, even if the function was declared using the IN WIN32API syntax, ADLLS() lists the actual library that contains the function. Here's an

example that assumes we've issued the same declarations as in the previous example (the output has been slightly reformatted to fit the page):

```
ADLLS( aDeclaredDLLS )
LIST MEMORY LIKE aDeclaredDLLS

ADECLAREDDLLS                   Pub           A
   (    1,    1)                C     "GetSysColor"
   (    1,    2)                C     "GetSysColor"
   (    1,    3)                C     "C:\WINNT\system32\USER32.DLL"
   (    2,    1)                C     "GetSystemDirectory"
   (    2,    2)                C     "GetSysDir"
   (    2,    3)                C     "C:\WINNT\system32\KERNEL32.DLL"
```

ShowDLLs.SCX in the conference materials demonstrates both new capabilities.

Another new "A" function in VFP 7 is ASessions(), which fills an array with a list of the existing data sessions. Each array element contains the number of the data session. Executing ASessions() is simple. Here's an example:

```
nActiveSessions = ASessions( aActiveSessions )
```

Although, at first glance, the function would appear to give you an array where element 1 contains 1, element 2 contains 2, and so forth, that's not actually the case. Since data session numbers are assigned at the time at which forms and reports are instantiated and do not change once a form or report exists, it's possible to have holes in the sequence. Also, since available data sessions are reused (after a form or report is closed), it's possible for the list to be out of order. Run the form ShowSessions.SCX in the conference materials for a demonstration of this issue and the new function.

A more interesting question is how you would use the array created by ASessions(). The function makes it easier to write generic shut down code that handles outstanding changes. Code along these lines is useful both for forcing a shutdown and for handling those situations where you have abandoned data sessions, a problem that can occur due to errors or to dangling pointers.

```
* Table handling portion of generic shutdown code
* This version reverts all changes.
#DEFINE DB_BUFOFF               1

LOCAL nCurrentSession, nActiveSessions, nSession, nCursors, nCursor
nCurrentSession = SET("DATASESSION")

nActiveSessions = ASESSIONS( aActiveSessions )
FOR nSession = 1 TO nActiveSessions
   * Switch to this session
   SET DATASESSION TO aActiveSessions[ nSession ]
   nCursors = AUSED(aCursors)
   FOR nCursor = 1 TO nCursors
      * Process each open cursor
      SELECT (aCursors[ nCursor, 1])
      IF CURSORGETPROP("Buffering") <> DB_BUFOFF
         * If it's buffered, revert all changes
         TABLEREVERT(.T.)
      ENDIF
   ENDFOR
ENDFOR
```

```
SET DATASESSION TO nCurrentSession
```

There are times when you want to show the user a filename with its path, but the path is so long that it doesn't fit into the available space. The new DisplayPath() function solves this problem. Pass it a filename and length and it returns a shortened version of the filename, no longer than the specified length. Entire directories are removed from the path and an ellipsis (…) is substituted for them. The example here is a perfect case for the function, in fact. The path for the filename to be shortened is so long that it wraps onto a second line.

```
? DisplayPath( "C:\WINNT\PROFILES\TAMAR\APPLICATION
DATA\MICROSOFT\TEMPLATES\NORMAL.DOT", 40)
c:\...\microsoft\templates\normal.dot
```

Note that the result may be shorter than the specified length, since it always cuts off at a folder. In the example, the result string is 37 characters.

The ADIR() function, which fills an array with a list of files and/or directories, has been in FoxPro for a long time, but since the advent of 32-bit operating systems, it's had some weaknesses. The addition of a new parameter in VFP 7 addresses those shortcomings. The optional fourth parameter, nFlags, lets you specify whether file and directory names should be in capital letters (the default and the old behavior) or should reflect the actual case used. It also lets you request the DOS 8.3 names for files and folders. You can combine these two behaviors, if you wish, since the parameter, like other flags is additive. Add 1 for original case and add two for DOS 8.3 names.

One of the trickier things developers have had to do with VFP is figure out the actual size of the usable screen area, that is, the whole VFP area less the space taken up by the menu, the status bar and any docked toolbars. VFP 7 makes this much easier. The Height and Width properties of _SCREEN now reflect that interior space, while the properties of _VFP continue to show the entire area taken up by VFP.

On one of my systems, with the standard toolbar docked under the menu and the status bar on, these values are returned:

```
?_VFP.Height  && 733
?_VFP.Width   && 893
?_SCREEN.Height  && 624
?_SCREEN.Width   && 885
```

If I undock the toolbar, the values for _VFP don't change, but _SCREEN.Height returns 653. Docking the toolbar at the side changes _SCREEN.Width to 850, but none of the other values change.

Array Handling

In addition to the new functions that return arrays (ATagInfo(), ASessions(), ADLLs()), there are several changes to array-handling functions in Visual FoxPro 7, plus one big change regarding arrays that affects functions in general.

First, the big change. It's now possible to return an array from a function. In older versions of VFP, the return value from a function had to be a scalar (non-array). In VFP 7, a function can return an array, provided the array is still in scope when the function is done. What this means in

practice is that this ability is really limited to methods returning arrays that are properties. In this example (found in Colors.PRG in the conference materials), the class has an array property called aRGB. The method RGBComp accepts a color as a parameter and returns an array containing the red, green and blue components of the color. (It mimics the behavior of the RGBComp function in FoxTools, except for the way it returns the values.)

```
DEFINE CLASS Colors AS Custom
* Color handling code
DIMENSION aRGB[3]

FUNCTION RGBComp(nColor) AS Array
* RGBComp
* Returns the Red, Green and Blue Components
* of a color in an array

This.aRGB[1] = -1
This.aRGB[2] = -1
This.aRGB[3] = -1

IF VARTYPE(nColor)="N"
   This.aRGB[3] = INT(nColor/(256^2))
   nColor = MOD(nColor,(256^2))
   This.aRGB[2] = INT(nColor/256)
   This.aRGB[1] = MOD(nColor,256)
ENDIF

RETURN @This.aRGB

ENDDEFINE
```

Note that you have to precede the returned array with "@", the same way that you pass a parameter by reference.

To use the method, first we need to instantiate the class, of course:

```
oColors = NewObject("Colors", "Colors.PRG")
```

Then, we can call the method and store the return value in an array. If the array doesn't exist, it's created (with private scope). If it exists, but has the wrong dimensions, it's redimensioned:

```
aResult = oColors.rgbcomp(255)
```

Two of FoxPro's array-handling functions have been modified in VFP 7 to make them more useful. Both ASCAN() and ASORT() now have the ability to ignore case. ASCAN() has several other new features: you can now choose whether it does an exact search (in the SET EXACT sense); you can limit the search to a specific column; and you can tell it to return the row number rather than the element number in a two-dimensional array. These changes have all been requested by the VFP developer community over the years.

Let's look at the syntax and some examples. We'll take ASORT() first because it has changed less. The function has a new (fifth) parameter, nFlags, used to specify whether or not the sort is case-sensitive. If this parameter is omitted or 0, the sort is case-sensitive, as in older versions of FoxPro. If the parameter is 1, the sort is case-insensitive. (Of course, since this parameter is called nFlags, it's possible that additional flags will be added to it in future. See the discussion of ASCAN() below to see how flag values are added to combine choices.)

This query creates an array that we can work with. It pulls the first names out of the TasTrade Employee table three times, taking each one in its regular form, in lower-case and in upper-case.

```
SELECT First_Name FROM _SAMPLES+"TasTrade\Data\Employee";
UNION ;
SELECT LOWER(First_Name) FROM _SAMPLES+"TasTrade\Data\Employee" ;
UNION ;
SELECT UPPER(First_name) FROM _SAMPLES+"TasTrade\Data\Employee" ;
INTO ARRAY aNames
```

Here's part of the results from calling ASORT() without the new flag:

```
ANAMES                  Pub             A
   (    1,    1)                        C     "ALBERT    "
   (    2,    1)                        C     "ANDREW    "
   (    3,    1)                        C     "ANNE      "
   (    4,    1)                        C     "Albert    "
   (    5,    1)                        C     "Andrew    "
   (    6,    1)                        C     "Anne      "
   (    7,    1)                        C     "CAROLINE  "
   (    8,    1)                        C     "Caroline  "
   (    9,    1)                        C     "JANET     "
   (   10,    1)                        C     "JUSTIN    "
   (   11,    1)                        C     "Janet     "
   (   12,    1)                        C     "Justin    "
   (   13,    1)                        C     "LAURA     "
   (   14,    1)                        C     "LAURENT   "
   (   15,    1)                        C     "Laura     "
   (   16,    1)                        C     "Laurent   "
```

Note that none of the lower-case names are shown here. They're all clustered at the end of the listing.

To use the new flag, but not specify a start position or number of elements, pass –1 for the second and third parameters. Here's the call needed to get aNames properly sorted:

```
ASORT(aNames, -1, -1, 0, 1)
```

Here's a partial display of the results. Note that within each group, there's no predicting which version of the name comes first. The function sees "ANDREW", "Andrew" and "andrew" as identical.

```
ANAMES                  Pub             A
   (    1,    1)                        C     "ALBERT    "
   (    2,    1)                        C     "albert    "
   (    3,    1)                        C     "Albert    "
   (    4,    1)                        C     "ANDREW    "
   (    5,    1)                        C     "Andrew    "
   (    6,    1)                        C     "andrew    "
   (    7,    1)                        C     "anne      "
   (    8,    1)                        C     "ANNE      "
   (    9,    1)                        C     "Anne      "
   (   10,    1)                        C     "CAROLINE  "
   (   11,    1)                        C     "caroline  "
   (   12,    1)                        C     "Caroline  "
   (   13,    1)                        C     "Janet     "
   (   14,    1)                        C     "janet     "
   (   15,    1)                        C     "JANET     "
   (   16,    1)                        C     "Justin    "
   (   17,    1)                        C     "JUSTIN    "
```

```
(   18,    1)                    C     "justin    "
(   19,    1)                    C     "laura     "
(   20,    1)                    C     "Laura     "
(   21,    1)                    C     "LAURA     "
(   22,    1)                    C     "Laurent   "
(   23,    1)                    C     "LAURENT   "
(   24,    1)                    C     "laurent   "
```

ShowASort.SCX in the conference materials lets you experiment with ASORT().

ASCAN() has two new parameters. The added fifth parameter is nSearchColumn – it indicates which column to search. You can combine it with the parameters for start position and number of elements to search only particular elements within a specific column. Alternatively, you can specify –1 for both the start position and number of elements to search the entire specified column.

To set up our example, this line creates an array containing information about the persistent relations in the TasTrade database:

```
ADBOBJECTS(aRelns,"RELATION")
```

The array has five columns: the first is the name of the child table in the relation and the second is the name of the parent table. The third and fourth columns contain the names of the tags used to maintain the relation in the child and parent tables, respectively. The final column indicates the type of relational integrity for that relation.

If we want to find relations involving a particular table as child, we need to make sure we search only the first column. For example, this call finds the first relation involving the Products table as a child:

```
? ASCAN( aRelns, "PRODUCTS", -1, -1, 1) && displays 16
```

The new sixth parameter, nFlags, lets us improve that search in several ways. In VFP 7, nFlags has four, additive, bit values, as shown in Table 3.

Table 3 ASCAN() flags – Add the values shown together to create the nFlags parameter.

Bit	Value	Meaning
0	0	Search is case-sensitive. Default.
0	1	Search is case-insensitive.
1	0	Exact is off. Effective only when Bit 2 is set (4).
1	2	Exact is on. Effective only when Bit 2 is set (4).
2	0	Current SET EXACT setting applies.
2	4	Use the exactness setting from Bit 1.
3	0	Return the element number of the matching item.
3	8	Return the row number of the matching item, if this is a two-dimensional array.

Returning to the previous example, we can try several changes. The most useful is getting the row number rather than the element number:

```
? ASCAN( aRelns, "PRODUCTS", -1, -1, 1, 8) && displays 4
```

We also might not want to worry about the case of the data in the array, so we can add the flag for case-insensitivity to the flag for returning the row number:

```
? ASCAN( aRelns, "PRODUCTS", -1, -1, 1, 9) && displays 4
```

The flags for dealing with exact matches are a little confusing. It doesn't matter which value you pass for bit 1, unless you also add 4 for bit 2. This gives you the option of either following the current SET EXACT setting or overriding it for the search. Add 4 to nFlags to make sure you search with EXACT off; add 6 to force ASCAN() to search with EXACT on. No matter which value you pass, it doesn't affect anything outside the single ASCAN().

Date Handling

A couple of changes provide improvements related to dealing with dates. The new QUARTER() function takes a date and tells you which quarter of the year it falls in. You can specify a starting month for the year to handle fiscal years rather than calendar years. For example:

```
? QUARTER( {^ 2000-9-23} )
```

returns 3, since September 23 is in the third quarter. But, if you specify a year beginning July 1, like this:

```
? QUARTER( {^ 2000-9-23}, 7 )
```

the function returns 1, instead.

SET("CENTURY") has a new option. When you pass 3 as the second parameter, it returns the maximum rollover date as set in the Regional Options Applet in the Control Panel. This option is available only in Windows 98 and later. In earlier versions, SET("CENTURY", 3) returns -1.

Improved User Interface Tools

There are a number of changes related to building user interfaces. They can be broken into two categories, those related to controls and those related to other input mechanisms.

Control Improvements

Several of the changes to controls provide the flat look popularized by Windows 2000 and Office 2000. Many controls have a new "Hot Tracking" setting (2) for SpecialEffect that makes them flat except when the mouse passes over them. At that point, depending on the particular control, they become either raised or depressed. For some controls, the Hot Tracking setting works only when Style is set to Graphical.

All controls have new MouseEnter and MouseLeave events that give you the opportunity to take action as the mouse passes through the control. These events take the same parameters as

MouseMove: the mouse button or buttons that are currently pressed; which, if any, of the Ctrl, Shift and Alt keys is pressed; and the current mouse position.

CommandButtons have a new VisualEffect property, available only at runtime, that lets you raise or depress that button. By manipulating this property in the MouseEnter and MouseLeave methods, you can build your own Hot Tracking effect.

The form in Figure 1 demonstrates the new SpecialEffect setting, the MouseEnter and MouseLeave events, and the VisualEffect property, as well as a number of other interface changes, described in the next section. The form is included as UIChanges.SCX in the conference materials.

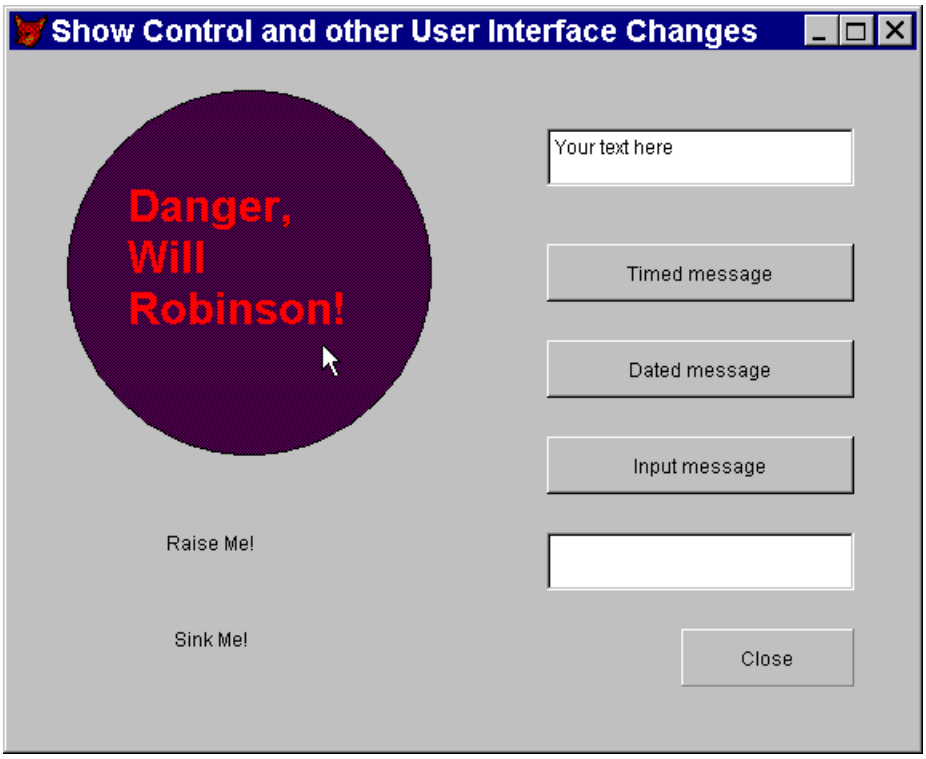

Figure 1 What's New in Formland? – This form demonstrates a few of the changes to controls and other input techniques, including the new MouseEnter and MouseLeave events, the VisualEffect property for buttons, and the new Hot Tracking setting for the SpecialEffect property.

A couple of changes to grids address long-time enhancement requests. A WordWrap property has been added to the Header object, so that headers can occupy more than one line. In the form in Figure 2 (included in the conference materials as GridSample.SCX), WordWrap is .T. for the header in the first column and .F. for the other headers.

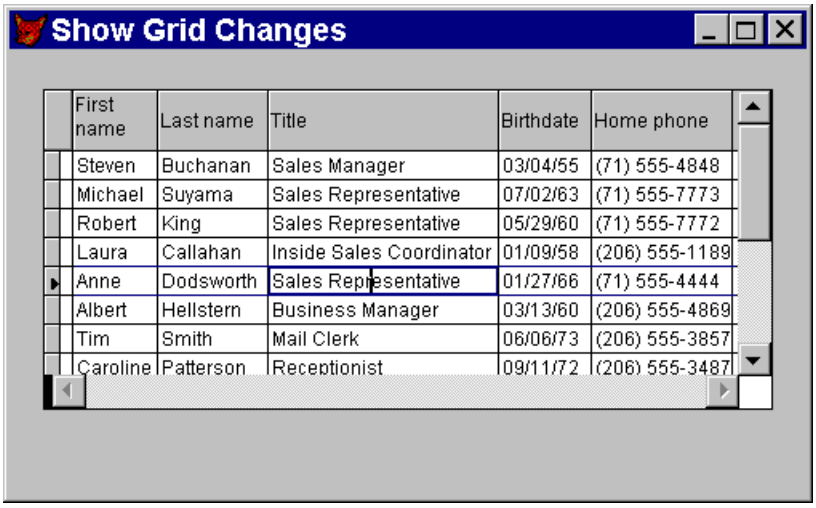

Figure 2 New grid features – Column headers now have the ability to wrap as needed. In addition, you can now find out whether it was a row change or a column change that caused BeforeRowColChange or AfterRowColChange to fire.

Ever since Visual FoxPro 3.0, developers have wondered why the BeforeRowColChange and AfterRowColChange events weren't divided into separate BeforeRowChange, BeforeColChange, AfterRowChange and AfterColChange events. While VFP 7 doesn't go quite that far, the new RowColChange property does make it easy to know why the two events fired. It contains a value that indicates what changed: the row (1), the column (2), both (3) or neither (4). In the form shown in Figure 2, AfterRowColChange displays a message box showing what fired it. Here's the code:

```
LPARAMETERS nColIndex

#DEFINE MB_ICONINFORMATION      64      && Information message

LOCAL cWhatChanged

DO CASE
CASE This.RowColChange = 0
   cWhatChanged = "nothing has"
CASE This.RowColChange = 1
   cWhatChanged = "row only has"
CASE This.RowColChange = 2
   cWhatChanged = "column only has"
CASE This.RowColChange = 3
   cWhatChanged = "row and column have"
ENDCASE

MESSAGEBOX("In AfterRowColChange - " + cWhatChanged + " changed", ;
           "Grid Information",MB_ICONINFORMATION)

RETURN
```

Another new property that cuts down on the code you need to write is hWnd, which has been added to forms and toolbars. This property always contains the window handle of the form or toolbar. It can be used in Windows API functions. In older versions, you need to make a couple of function calls to get this information. Figure 3 shows a form that passes its hWnd to an API

function and changes itself to a circle. The code for the form (as exported by the Class Browser) is shown below. (Thanks to David Frankenbach for showing me this technique.) This form is available in the conference materials as MkCircle.SCX.

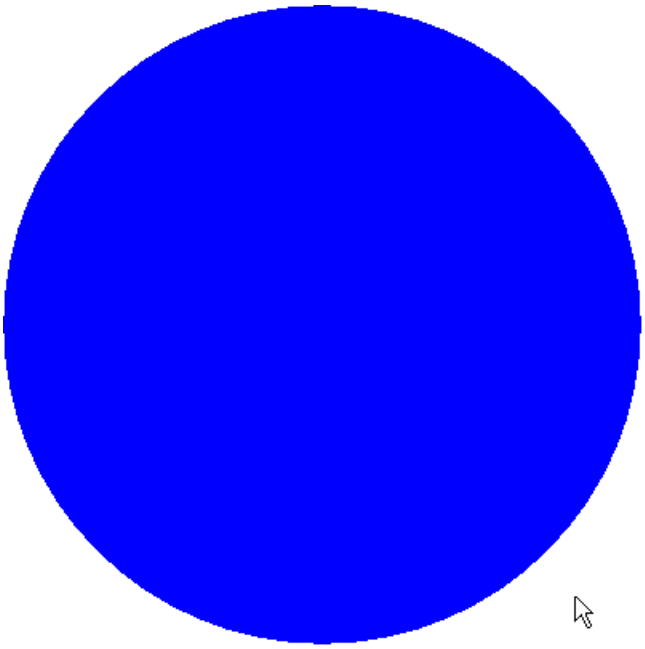

Figure 3 Getting a handle on a form – This form passes its Windows' handle to the API function SetWindowRgn in order to change the form into a circle.

```
PUBLIC ofrmcircle

SET CLASSLIB TO d:\fox\testcode\vfp6test\controls\forms.vcx ADDITIVE

ofrmcircle=NEWOBJECT("frmcircle")
ofrmcircle.Show
RETURN

DEFINE CLASS frmcircle AS frmform

   Height = 400
   Width = 400
   DoCreate = .T.
   AutoCenter = .T.
   BorderStyle = 0
   Movable = .F.
   TitleBar = 0
   BackColor = RGB(0,0,255)
   Name = "frmCircle"


   PROCEDURE Init
      LOCAL nhWnd, nWidth, nHeight, nRegion

      DECLARE INTEGER CreateEllipticRgn IN gdi32 ;
              INTEGER X1 , INTEGER Y1 , INTEGER X2 , INTEGER Y2
      DECLARE INTEGER SetWindowRgn IN user32 ;
              INTEGER HWND, INTEGER hRgn , INTEGER bRedraw
```

```
      nhWnd = This.HWnd
      nWidth = This.WIDTH / 1 && change ratio
      nHeight = This.HEIGHT / 1 && change ratio
      nRegion = CreateEllipticRgn(0, 0, nWidth, nHeight)
      SetWindowRgn(nhWnd, nRegion, 1)
   ENDPROC


   PROCEDURE DblClick
      ThisForm.Release()
   ENDPROC

ENDDEFINE
```

In line with the other changes to _SCREEN, note that _VFP and _SCREEN both have an hWnd property, but they contain different values.

Other UI Enhancements

Not all interaction occurs through forms and controls. There are a variety of other ways to communicate with users and some of the changes in VFP 7 impact those approaches.

FoxPro's menus have been enhanced to keep up with the latest interaction techniques. The DEFINE BAR command has several new clauses. PICTURE and PICTRES let you add pictures to menu items. With PICTURE, you specify a file, including path. With PICTRES, you specify the bar number for an item from the FoxPro system menu and the graphic associated with that item is used. In order to use graphics, the menu popup containing the bar must be defined with the MARGIN clause.

INVERT allows you to make a bar appear with a lighter background and as though it were depressed. This is the way the lesser-used items in the Office applications appear (when they appear at all).

The final new clause is the most complex because you need to write code to make it useful. MRU stands for "most recently used." When you add this clause to a bar, the bar appears as a chevron character pointing downwards and implies that the menu can be expanded. When the user hovers the mouse over that item or clicks on it, you can respond by adding menu items (or any other code you want to run). Presumably the items you add will be inverted; otherwise, they'd have been displayed initially.

This program creates a new menu pad on the system menu with three items. The first two have pictures associated with them. The third is an MRU item. When the user chooses the MRU item, a separate program is called that changes the menu popup. First, here's the program that creates the menu pad and popup initially:

```
* MRUMenu.PRG
* Create a menu with an MRU item.

DEFINE PAD MRUSample of _MSYSMENU prompt "MRU Demo"

DEFINE POPUP MRUpop MARGIN RELATIVE

ON PAD MRUSample of _MSYSMENU activate POPUP MRUPop
```

```
DEFINE BAR 1 of MRUPop Prompt "End MRU Demo" picture HOME() + "Fox.BMP"
DEFINE BAR 2 of MRUPop prompt "Second" pictres "_mfi_save"
DEFINE BAR 3 of MRUPop mru

ON SELECTION BAR 1 OF MRUPop DO ReleaseMRU
ON SELECTION BAR 2 OF MRUPop WAIT WINDOW "Not really saving anything"
ON SELECTION BAR 3 of MRUPop do ChgMRUPop
```

Now here's ChgMRUPop, called when the user chooses the MRU item. It modifies the popup, reactivates it, then after the user makes a choice, restores the menu to its original state:

```
* ChgMRUPop.Prg
* Change the contents of MRUPop menu popup
* in response to the MRU choice

* Remove the MRU bar
RELEASE BAR 3 of MRUPop

* Add the new bars
DEFINE BAR 3 of MRUPop prompt "New Third Bar" INVERT AFTER 1
DEFINE BAR 4 of MRUPop prompt "Fourth" INVERT

ON SELECTION BAR 3 OF MRUPop WAIT WINDOW "Hey! This works."
ON SELECTION BAR 4 OF MRUPop WAIT WINDOW "Even the new one at the bottom
works"

* Reactivate the popup
ACTIVATE POPUP MRUPop

* After the user makes a choice, clean up
* Remove the added bars
RELEASE BAR 3 of MRUPop
RELEASE BAR 4 of MRUPop

* Need to redefine the MRU bar
DEFINE BAR 3 of MRUPop mru
ON SELECTION BAR 3 of MRUPop do ChgMRUPop
```

Finally, here's the ReleaseMRU program that cleans up the menu when the user choose the End MRU Demo choice.

```
* ReleaseMRU.PRG
* Clean up MRUMenu

RELEASE POPUP MRUpop
RELEASE PAD MRUSample of _MsysMenu
```

Figure 4 shows the menu as it first appears, while Figure 5 shows the expanded menu.

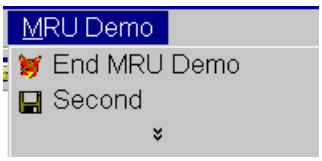

Figure 4 Menu enhancements – This menu popup shows both pictures for menu bars and the MRU menu item.

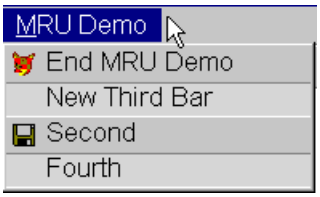

Figure 5 Menu expansion – You can write code to respond to the MRU item to add bars to the menu. The new INVERT clause provides the ability to have bars that are set back and dimmed.

Both kinds of pictures can be specified in the Menu Designer. However, the MRU and INVERT options can't be handled directly through the Designer. (It is possible to trick the Menu Designer into adding these options by including them in the SKIP FOR clause.)

VFP has long provided a Windows standard way for communicating messages to users with the MessageBox() function. In this version, the function gains a timeout parameter plus the ability to automatically convert the message to character. The latter means that you can now pass dates or numbers without needing to run them through TRANSFORM() or another conversion function. Of course, when creating a complex message from data of multiple types, it's still necessary to convert all the data to character before concatenating it.

The new timeout parameter means that you can display a message box until either the user dismisses it or time runs out. The timeout period is measured in milliseconds. If the box is cleared due to timeout, the function returns –1. Here's an example:

```
nResult = MessageBox("Don't you hate monolog boxes?",32+0,;
                     "Demonstrate timeout",2000)
```

This message is displayed for two seconds unless the user chooses the OK button first. Figure 6 shows the dialog.

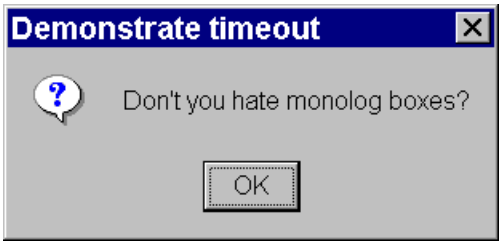

Figure 6 Message box with timeout – The MessageBox() function has a new timeout parameter that lets you clear the dialog after a specified time.

A new function that's sort of a cousin to MessageBox() is for those situations where you need just a single input from a user and implementing a whole form to get it seems like overkill. The function is called InputBox() and what it does is give you access to the same dialog that VFP uses to get view parameters. InputBox() accepts five parameters as follows:

```
cInput = InputBox( cPrompt [, cCaption [, cDefault [, nTimeOut
                   [, cTimeoutValue ] ] ] ] )
```

For example:

```
cUserName = InputBox( "Enter UserName", "Tamar's Application","",10000,;
                      "***No entry***")
```

Figure 7 shows the resulting dialog. As the syntax shows, the function always returns a character value.

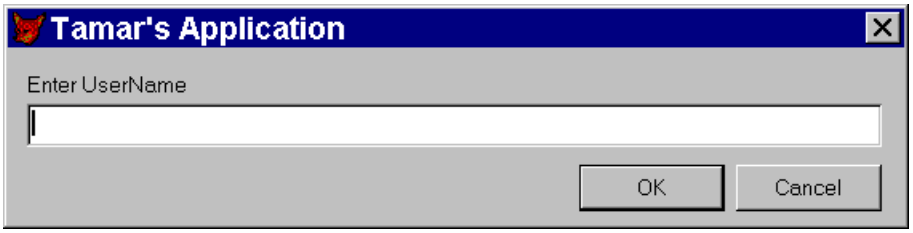

Figure 7 Easy input – The new InputBox() function makes it simple to collect single input values.

The form in Figure 1 includes buttons that demonstrate the changes to MessageBox() and a button that demonstrates InputBox().

Two functions that display system dialogs have been updated. GetFont() now has the ability to indicate the character set for the chosen font. In order to avoid breaking existing code, the function includes the character set number in the return value only if a value is passed for the optional nCharacterSet parameter. When that parameter is passed, the Script dropdown in the Font dialog is enabled. Figure 8 shows the dialog produced by this call:

```
? GETFONT("Tahoma",24,"",1)
```

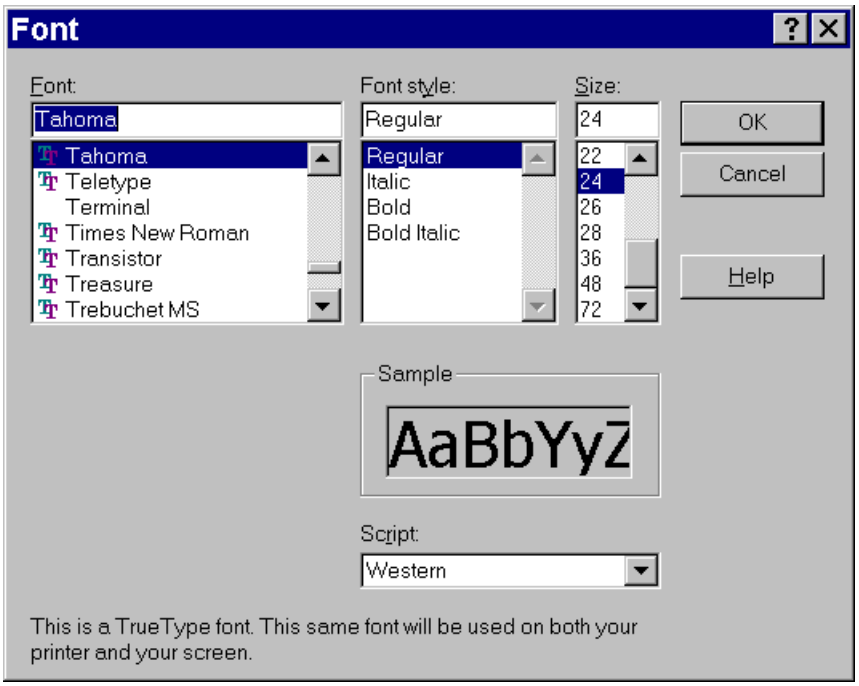

Figure 8 Choosing fonts – The new nCharacterSet parameter for GetFont() lets you enable the Script dropdown, so users can choose the appropriate character set.

If the user just clicks OK in Figure 8, the return value is:

```
"Tahoma,24,N,1"
```

The changes to the GETDIR() function are much more significant. First, in some circumstances, the function uses a treeview to display the drive and directory hierarchy, rather than the nested folders used in VFP 6 and earlier versions.

Second, three new parameters have been added. If any of these parameters are passed, the treeview version of the dialog is used. The new syntax is as follows:

```
cDirectory = GETDIR( [ cStartDir [, cPrompt [, cDialogCaption
                     [, nFlags [, lRootOnly ] ] ] ] ] )
```

As its name suggestions, the new cDialogCaption parameter lets you specify text that appears on the dialog's title bar. The nFlags parameter lets you specify 0 or more additive flags. They work like the flags for ASCAN() – choose the ones you want and add them together to get the single nFlags parameter. Table 4 shows the flag values – some of them are only meaningful in newer versions of Windows. The lRootOnly parameter indicates whether the treeview should stop at the root of the specified drive (when it's .T.) or continue upward (the default, .F., setting).

Table 4 GETDIR() flags – Choosing a directory is more configurable than ever. Add these flag values together for the fourth parameter to GETDIR(). Some of these values make it possible to use GETDIR() to choose a printer or another computer rather than a directory.

Flag value	Meaning
1	Only return directories that are part of the file system. If the user selects folders that are not part of the file system, the OK button is disabled.
2	Do not include network folders below the domain level in the tree.
8	Only return file system ancestors. If the user selects anything other than a file system ancestor, the OK button is disabled.
16	Include an edit control for the user to type the name of an item. (Only available in some versions of Windows.)
64	Use the new user interface. (Only supported in Windows 2000 and above.)
4096	Only return computers. If the user selects anything other than a computer, the OK button is disabled.
8192	Only return printers. If the user selects anything other than a printer, the OK button is disabled.
16384	Display files as well as directories. (Only available in some versions of Windows.)

CD ? uses the same new dialog as GETDIR(), though it doesn't offer all the new options.

The final user interface related change is one that FoxPro developers have been requesting for many, many versions. For a very long time, the _DBLCLICK system variable has controlled two different things. As its name suggests, it determines how long can elapse between clicks and still have the sequence considered a double-click. However, the same variable has also been used to determine the speed at which a user must type to have incremental search in combo boxes. In VFP 7, these two items have finally been separated and incremental search speed is now controlled by the new _INCSEEK system variable.

OOP Language

In addition to the various changes to controls described earlier, there are a couple of other changes to VFP's object-oriented language. First, the DEFINE CLASS command has been extended to allow you to include the class library of the class being subclassed. The new syntax for the DEFINE CLASS line is:

```
DEFINE CLASS ClassName AS ParentClass [ OF ClassLibrary ]
```

The advantage of this format is that you don't need to issue SET CLASSLIB or SET PROCEDURE before the class is instantiated. The class library can (and usually should) include a path and can be surrounded by quotes (for example, when the path includes spaces).

The AMEMBERS() function provides additional information, both for native VFP objects and for COM objects. The new value 3 for the third parameter indicates that the function should return a four-column array – the contents of the columns are shown in Table 5.

Table 5 What's in an object? – Calling AMEMBERS() with 3 as the third parameter returns a four-column array, with these contents.

Column	Contents
1	Name of the property, event or method
2	Type of item. For VFP objects, the possible values are "Property", "Event" or "Method". For COM objects, the possible values are "PropertyPut", "PropertyGet", "PropertyPutRef" and "Method".
3	Empty for properties of VFP objects. For methods of VFP objects, the parameter list. For properties and methods of COM object, the member's signature, consisting of the parameter list, plus the return value.
4	The help string for the item.

AMEMBERS() has also acquired a cFlags parameter that lets you specify which members to return, for native objects. Table 6 lists the flag characters. The flags in each filter group are mutually exclusive. However, by default, when you concatenate multiple flag characters into the cFlags parameter, they're combined with OR, so a cFlags parameter of "HP" includes all hidden and protected PEMs. Passing "GU" includes all members that are either public or user-defined in the result.

Table 6 Choosing members – AMEMBERS() new, fourth, parameter lets you filter the list of members returned.

Flag character	Filter group	Meaning
P	Visibility	Protected
H	Visibility	Hidden
G	Visibility	Public
N	Origin	Native PEMs
U	Origin	User-defined PEMs
I	Inheritance	Inherited PEMs
B	Inheritance	Base PEMs
C	Changed	Changed
R	Read-only	Read-only

There are two special flags. Including the "+" anywhere in the cFlags parameter indicates that the filters should be combined with AND rather than OR. So, for example, passing "GU+" includes only members that are both public and user-defined.

The second special flag is "#", which adds a column to the resulting array. The new column shows the flags that apply to each member. The "#" flag can't be passed alone-you need to include at least one other flag with it; if you want just to add the column, pass "+#".

Here's an example of both of the new features. This code creates an instance of the _MoverLists class from the FoxPro Foundation Classes, then lists the Protected members of the class, including their flags:

```
oObject = NewObject( "_MoverLists",HOME()+"FFC\_CONTROLS" )
AMEMBERS( aMemberList, oObject, 3, "P#" )
LIST MEMORY LIKE aMemberList
AMEMBERLIST          Pub            A
   (    1,    1)    C     "ADDTOPROJECT"
   (    1,    2)    C     "Method"
   (    1,    3)    C     ""
   (    1,    4)    C     "Dummy code for adding files to project."
   (    1,    5)    C     "CPUI"
   (    2,    1)    C     "NINSTANCES_ACCESS"
   (    2,    2)    C     "Method"
   (    2,    3)    C     ""
   (    2,    4)    C     "Access method for nInstances property."
   (    2,    5)    C     "CPUI"
   (    3,    1)    C     "NINSTANCES_ASSIGN"
   (    3,    2)    C     "Method"
   (    3,    3)    C     "vNewVal"
   (    3,    4)    C     "Assign method for nInstances property."
   (    3,    5)    C     "CPUI"
   (    4,    1)    C     "NOBJECTREFCOUNT_ACCESS"
```

```
   (    4,    2)    C     "Method"
   (    4,    3)    C     ""
   (    4,    4)    C     "Access method for nObjectRefCount property."
   (    4,    5)    C     "CPUI"
   (    5,    1)    C     "NOBJECTREFCOUNT_ASSIGN"
   (    5,    2)    C     "Method"
   (    5,    3)    C     "m.vNewVal"
   (    5,    4)    C     "Assign method for nObjectRefCount property."
   (    5,    5)    C     "CPUI"
```

This example includes all members for the class (because all three visibilities are listed), but also includes the flags. Only a portion of the output is shown.

```
AMEMBERS(aMemberList, oObject, 3, "GPH#")
LIST MEMORY LIKE aMemberList

AMEMBERLIST         Pub            A
   (    1,    1)    C     "ACTIVECONTROL"
   (    1,    2)    C     "Property"
   (    1,    3)    C     ""
   (    1,    4)    C     "References the active control on an object."
   (    1,    5)    C     "GRNI"
   (    2,    1)    C     "ADDOBJECT"
   (    2,    2)    C     "Method"
   (    2,    3)    C     "cName, cClass"
   (    2,    4)    C     "Adds an object to a container object at run time."
   (    2,    5)    C     "GNI"
   (    3,    1)    C     "ADDPROPERTY"
   (    3,    2)    C     "Method"
   (    3,    3)    C     "cPropertyName,eNewValue"
   (    3,    4)    C     "Adds a new property to an object."
   (    3,    5)    C     "GNI"
   (    4,    1)    C     "ADDTOPROJECT"
   (    4,    2)    C     "Method"
   (    4,    3)    C     ""
   (    4,    4)    C     "Dummy code for adding files to project."
   (    4,    5)    C     "CPUI"
   (    5,    1)    C     "AOBJECTREFS"
   (    5,    2)    C     "Property"
   (    5,    3)    C     ""
   (    5,    4)    C     "Array of object references properties."
   (    5,    5)    C     "GUI"
   (    6,    1)    C     "BACKCOLOR"
   (    6,    2)    C     "Property"
   (    6,    3)    C     ""
   (    6,    4)    C     "Specifies the background color used to display text
and graphics in an object."
   (    6,    5)    C     "GNI"
```

The form ShowAMembers.SCX in the conference materials lets you experiment with AMembers().

IDE Enhancements

Many of the most significant changes in VFP 7 are in the interactive development environment (IDE). As in earlier versions, the programming language has been enhanced to support the IDE changes. In addition to the new material, there are some language changes that support other, previously existing, areas of the IDE.

Support for new features

Several new system variables point to the support programs and tables for new IDE features; in addition, new functions provide programmatic support for new editor features. Table 7 shows the new system variables that reference new features; you can change these to point to programs or data you substitute for the ones provided with VFP.

Table 7 Finding new features – These new system variables point to new VFP applets and the data for them. You can substitute your own, if you're so inclined.

Variable	Default Value	Meaning
_CodeSense	HOME() + "FoxCode.App"	The IntelliSense Manager application
_FoxCode	<WinDir> + "\Profiles\" + <User> + "\<User Settings Dir>\Microsoft\Visual FoxPro\FoxCode.DBF"	The IntelliSense data table
_FoxTask	HOME() + "FoxTask.DBF"	The Task List data table
_ObjectBrowser	HOME() + "ObjectBrowser.APP"	The Object Browser application
_TaskList	HOME() + "TaskList.APP"	The Task List application

Along the same lines, the _VFP application object has two new properties, EditorOptions and LanguageOptions, that let you control which of the new editing features are automatically available. EditorOptions lets you manage the various Intellisense features, as well as drag and drop between words and having hyperlinks created automatically. Even when these features are turned off via EditorOptions, they can still be provided through the menu. LanguageOptions lets you determine whether strict typing is enforced at runtime.

The new EDITSOURCE() and APROCINFO() functions let you take advantage of some other new editing features. EDITSOURCE() opens the appropriate editor for a file, optionally positioning at a specific point in the file. If you pass a file that belongs to one of the visual designers, you must specify a line number. For example:

```
EditSource( HOME()+"FFC\_CONTROLS.VCX",3,"_MoverLists","SelectAll")
```

EDITSOURCE() also ties into the new Task List tool. You can pass a shortcut id from the task list and have the appropriate editor opened right at that shortcut.

APROCINFO() is a programmatic version of the new Document View. It fills an array with the same kind of information that's provided by choosing Document View interactively. The function lets you decide how much of the information you want to see. Here's the syntax:

```
nCount = AProcInfo( aArray, cFileName [, nWhichInfo ] )
```

The nWhichInfo parameter can take any of four values:

0 or omitted – include all document view information
1 – include only class definitions
2 – include only method information
3 – include only preprocessor directives, including #DEFINE

For example:

```
APROCINFO( aDefines, HOME()+"GENHTML.PRG", 3)
```

produces this result (shown only in part):

```
API                 Pub             A
   (     1,     1)  C      "VFP_DEFAULT_ID"
   (     1,     2)  N      32                          (          32.00000000)
   (     1,     3)  C      "Define"
   (     2,     1)  C      "M_CLASS_LOC"
   (     2,     2)  N      35                          (          35.00000000)
   (     2,     3)  C      "Define"
   (     3,     1)  C      "M_COULD_NOT_BE_INST_LOC"
   (     3,     2)  N      36                          (          36.00000000)
   (     3,     3)  C      "Define"
   (     4,     1)  C      "M_COULD_NOT_OPENED_EXCL_LOC"
   (     4,     2)  N      37                          (          37.00000000)
   (     4,     3)  C      "Define"
   (     5,     1)  C      "M_FILE_LOC"
   (     5,     2)  N      38                          (          38.00000000)
   (     5,     3)  C      "Define"
   (     6,     1)  C      "M_FILE_ALREADY_EXISTS_LOC"
   (     6,     2)  N      39                          (          39.00000000)
   (     6,     3)  C      "Define"
   (     7,     1)  C      "M_FILE_TYPE_LOC"
   (     7,     2)  N      40                          (          40.00000000)
   (     7,     3)  C      "Define"
```

APROCINFO() works only on textual program files, not visual class libraries (VCX) or forms.

In VFP 7, many of the system windows can be docked. The WDOCKABLE() function lets you find out, for any system window that can be docked, whether it's currently dockable. More than that, it lets you change the window's dockability status. Pass just the window name to check its status. Pass .T. or .F. for the second parameter to change the window's dockability. For example, to make the Command Window dockable:

```
WDOCKABLE("Command", .T.)
```

The ALANGUAGE() function was added to aid in customizing IntelliSense. It fills an array with various language components, based on the second parameter passed to it. Table 8 shows the choices:

Table 8 Language components – ALANGUAGE() fills an array with different portions of the VFP language, depending which of these values you pass it.

Second parameter	Result
1	Create a one-dimensional array of commands

2	Create a two-dimensional array of functions, including the number of parameters accepted. The second column may also contain the letter "M", indicating that the whole name of the function must be used rather than a shortened version.
3	Create a one-dimensional array of base classes.
4	Create a one-dimensional array of DBC events.

Here's an example, with only partial results shown:

```
ALANGUAGE( aFunctions, 2)
LIST MEMORY LIKE aFunctions

AFUNCTIONS                Pub             A
   (    1,    1)                         C     "ABS"
   (    1,    2)                         C     "1"
   (    2,    1)                         C     "ACLASS"
   (    2,    2)                         C     "2"
   (    3,    1)                         C     "ACOPY"
   (    3,    2)                         C     "2-5"
   (    4,    1)                         C     "ACOS"
   (    4,    2)                         C     "1"
   (    5,    1)                         C     "ADATABASES"
   (    5,    2)                         C     "1"
   (    6,    1)                         C     "ADBOBJECTS"
   (    6,    2)                         C     "2"
   (    7,    1)                         C     "ADDBS"
   (    7,    2)                         C     "1"
   (    8,    1)                         C     "ADEL"
   (    8,    2)                         C     "2-3"
   (    9,    1)                         C     "ADIR"
   (    9,    2)                         C     "1-4"
```

Updates to existing language

To aid tool developers and for consistency with other commands, MODIFY VIEW and MODIFY PROCEDURE now support the NOWAIT clause that allows them to be opened programmatically and remain open while execution continues.

The WriteMethod method has a new parameter that allows it to create methods on the fly. When the method specified by the first parameter does not exist and the lCreateMethod parameter is .T., the method is created. This approach works only at design-time and the form or class must be saved after the method is added. These limitations are reasonable since WriteMethod is intended for use in builders.

When project hooks were added in VFP 6, developers immediately found uses for them. But a few features were missing. VFP 7 plugs those holes with three new events: QueryNewFile, Activate and Deactivate. QueryNewFile fires when you begin the process of adding a file to a project. Interactively, that's when you click the New button in the Project Manager. Previously, no event fired when a new file was added.

The Activate and Deactivate events for the ProjectHook object are like those of other classes – they fire when the object becomes active and when it loses focus. In the case of project hooks,

however, that happens when the project associated with the project hook is activated or deactivated. This means we now have the ability to modify the VFP environment as we switch between projects, offering the chance to change things like the VFP PATH, field mappings, and other settings that are project-specific. However, beware-when the project is docked, Activate and Deactivate don't fire.

When the ability to specify captions for fields was added in VFP 3, one simple thing got harder. When you BROWSE a table, the captions are used instead of the field names. For a developer who just wants a quick look at a table, this can be an annoyance. VFP 7 finally provides a quick and easy solution with a NOCAPTIONS clause for BROWSE.

Using defined constants makes code far more readable and collecting them into include files (also known as header or .h files), using the #INCLUDE directive or the Include File item on the menu, is a powerful technique. But this technique has been difficult to use because the algorithm used to search for include files specified with a relative path wasn't always intuitive. When the file was recompiled, the header files couldn't always be found, causing compilation errors.

In VFP 7, the search path for include files has been expanded. For classes and forms, the search order for include files is now:

Containing File + #INCLUDE filename (including relative path)
CURDIR() + #INCLUDE filename (including relative path)
SET PATH + #INCLUDE filename (including relative path)

The last two items are new. For programs, the new search order for include files is:

CURDIR + #INCLUDE filename (including relative path)
SET PATH + #INCLUDE filename (including relative path)
Containing File + #INCLUDE filename (including relative path)
Containing File + #INCLUDE filename only (no path)
VFP Root + #INCLUDE filename only (no path)

The third item is new.

Odds and Ends

One new function is an aid to debugging. ASTACKINFO() provides a complete listing of the program stack at the time it's executed. It's similar to both the Call Stack window in the Debugger and to SYS(16), but provides more information than either of those. The array it creates has six columns, as shown in Table 9.

Table 9 Who's running? – ASTACKINFO() fills an array with information about all the programs running at the time it's called.

Column	Contents
1	Stack level, with 1 for the main program.
2	Name of the file containing the routine that's executing.
3	Name of the routine that's executing. Can be a procedure, function, or method. For a

	method, the entry includes the object name.
4	Name of the source file containing the routine.
5	Line number within the file (not within the routine).
6	Source code

While ASTACKINFO() seems most appropriate for debugging, it's not restricted to that environment, but can be used at any point to get a snapshot of the current execution situation.

The USE command has a new CONNSTRING clause that lets you pass a connection string for use with a remote view. This provides a way to specify a userid and password at runtime rather than at design-time. Be aware that the complete connection string must be specified, however, as in SQLStringConnect().

Final Thoughts

The development team made changes in a tremendous number of areas in VFP 7. As in earlier versions, many of them have the potential to truly improve the way we write code. Even better, with this version, a lot of the changes come in response to developer requests. The challenge for us is to assimilate this new material into our thinking, so that we can remember to use these features when they're appropriate.

Thanks to Randy Brown of Microsoft for his help in preparing these notes.