VFP: Ideal for Tools, Part 1

VFP has an extensive set of language elements that make it easy to build developer tools. This article explores data-related language that helps in tool-building.

Tamar E. Granor, Ph.D.

In my last few articles, I focused on Thor, a VFPX tool for managing developer tools. Thor and VFPX generally are part of a long history of developer tools for FoxPro written in FoxPro. In this series of articles, I'll look at the language elements that enable writing developer tools.

When FoxPro 2.0 shipped in 1991, it included two new tools written in FoxPro's programming language: GenScrn and GenMenu, which translated screen designs and menu designs, respectively, into FoxPro code. While these may not have been the first developer tools written in FoxPro, they mark a turning point because they were an essential part of developing FoxPro applications and because the programming language was extended to make it possible to write them. After that, every version of FoxPro and Visual FoxPro included multiple tools written in FoxPro, and included commands, functions or other capabilities added specifically to make tool-writing easier. Since then, also, the FoxPro community has created hundreds (perhaps, thousands) of tools written in FoxPro to manipulate FoxPro.

Starting in VFP 6, the source code for all the tools written in the FoxPro language that come with VFP has been included with the product (in Tools\XSource\XSource.ZIP). VFP 9 includes more than 20 so-called "Xbase tools," from the Class Browser to the three reporting applications that help extend the Report Designer. All this source code both enables the VFP community to modify and extend the existing tools, and to see how particular things were done. (In fact, the Xbase tools are now part of VFPX, so that extensions can be managed and distributed.)

The set of language capabilities that enable tool creation is extensive. In this series of articles, I'll look at those capabilities, grouped by what kind of thing they manipulate. To demonstrate them, I'll look at code from a number of different VFP tools. Most of the code shown in these articles was written by people other than me. (Note also that much of the code has been reformatted to fit this magazine.)

The Array functions

There are a large number of functions in VFP that collect information and put the results into an array. Before actually exploring any specific tool-building language features, let's take a quick look at what all these array functions have in common.

First, all of them have names beginning with the letter "A," such as AFIELDS() or AVCXClasses().

Second, each expects an array as the first parameter. The information collected by the function is put into the array, which is created, if necessary, and redimensioned as needed.

Finally, these functions almost all return the number of rows in the resulting array.

For more information on the array functions or on array-handling in general in VFP, see the white paper "You Need Arrays," available on my website at
http://www.tomorrowssolutionsllc.com/Conference%20Sessions/You%20Need%20Arrays.pdf.

Digging into Data

The big thing that sets VFP apart from conventional programming languages is its native database engine. Not surprisingly, there are a whole slew of functions that let you explore databases and tables and find out which are currently in use.

Even if you're working with a SQL database, VFP includes functions to help you understand the available data structures.

Determining VFP table structures

FoxPro has had the ability to determine the fields in a table from early days. The AFields() function first appeared in FoxPro 2.0; it puts the structure of a table into an array with one row for each field. The content of the array has expanded over the years, as tables gained new capabilities. In VFP 9, the resulting array has 18 columns, though only the first six are relevant for free tables. The function returns the number of fields in the table.

The syntax for AFields() is shown in Listing 1. Listing 2 shows most of the code from the Thor tool, Create SQL from cursor. It uses AFields() to get a list of the fields in the specified cursor and then loops through the resulting array, building a CREATE CURSOR statement. The result of this code, applied to the Northwind Products table, is shown in Listing 3.

Listing 1. AFields() fills an array with information about the fields in a table. It can work on the current workarea or a specified alias or workarea.

```
nFields = AFIELDS( ArrayName
                   [, cAlias | nWorkarea] )
```

Listing 2. This code, drawn from the Thor tool Create SQL from cursor, shows a fairly typical use of AFields().

```
lnFields = afields(laFields)
lcSQL    = ''
for lnI = 1 to lnFields
  lcType = laFields[lnI, 2]
  lcSQL  = lcSQL + ;
    iif(empty(lcSQL), 'create cursor TEMP ;' ;
    + ccCR + '(', ', ;' + ccCR) + ;
    laFields[lnI, 1] + ' ' + lcType
  do case
    case lcType $ 'CVQ'
      lcSQL = lcSQL + '(' + ;
        transform(laFields[lnI, 3]) + ')'
    case lcType $ 'NF'
      lcSQL = lcSQL + '(' + ;
        transform(laFields[lnI, 3]) + ',' + ;
        transform(laFields[lnI, 4]) + ')'
    case lcType = 'B'
      lcSQL = lcSQL + '(' + ;
        transform(laFields[lnI, 4]) + ')'
  endcase
next lnI
```

Listing 3. The result produced by the Thor tool Create SQL from cursor, applied to the Northwind Products table.

```
create cursor TEMP ;
  (PRODUCTID I, ;
  PRODUCTNAME C(40), ;
  SUPPLIERID I, ;
  CATEGORYID I, ;
  QUANTITYPERUNIT C(20), ;
  UNITPRICE Y, ;
  UNITSINSTOCK I, ;
  UNITSONORDER I, ;
  REORDERLEVEL I, ;
  DISCONTINUED L)
```

The other key thing we may want to know about a table is what indexes it has. Although this information has been available since early FoxPro days, collecting it got a lot easier in VFP 7 with the addition of the ATagInfo() function. As its name implies, this function puts information about index tags into an array. The resulting array has one row for each tag and six columns indicating the name, type, key, filter, order and collate sequence for the tag. The function returns the number of tags; its syntax is shown in Listing 4.

Listing 4. ATagInfo() fills an array with information about index tags.

```
nTags = ATAGINFO( ArrayName [, cIndexName
                  [, cAlias | nWorkarea] )
```

Note that the second parameter to ATagInfo() is the name of an index file. This allows you to collect information on a subset of a table's indexes, in the odd case where they're not all in the table's structural index (CDX). While you're unlikely to use that parameter, you need to be aware of it when you want to specify the alias or workarea of the table. To skip over the cIndexName parameter, pass the empty string, as in Listing 5.

Listing 5. To specify the alias or workarea for ATagInfo(), you must pass something for the cIndexName parameter. Pass the empty string to include all tags in the result.

```
nTags = ATAGINFO(aTags, '', 'MyAlias')
```

Listing 6 shows a block of code from the Thor Schema tool. This section shown creates the list of indexes for the schema. That portion of the output, for the Northwind Products table, is shown in Figure 1.

Listing 6. This block of code, from Thor's Schema tool, uses ATagInfo() to collect the list of tags for the table. It then loops through and produces output describing each.

```
If Not This.lView
  Dimension aTags[1]
  iTags = Ataginfo(aTags)
  If iTags = 0
    \ No Structural Index Tags
  Else
    \ <h4>Indexes:</h4><table>

    \ <table id='tblIndices'><tr><th>Tag
Name</th> <th>Type</th> <th>Expression</th>
<th>Filter</th> <th>Order</th> <th>
Collation</th></tr>
    For X = 1 To iTags
      \<tr>
      For Y = 1 To 6
        \ <td> <<aTags[X,Y]>>  </td>
      Next
      \</tr>
    Next
  Endif
  \</table>
Endif
```

Tag Name	Type	Expression	Filter	Order	Collation
CATEGORYID	REGULAR	CATEGORYID		ASCENDING	MACHINE
SUPPLIERID	REGULAR	SUPPLIERID		ASCENDING	MACHINE
PRODUCTNAM	REGULAR	UPPER(PRODUCTNAME)		ASCENDING	MACHINE
PRODUCTID	PRIMARY	PRODUCTID		ASCENDING	MACHINE

Figure 1. The tag section of the output from Thor's Schema tool, applied to the Northwind Products table.

Exploring VFP database structures

VFP also includes commands that let you discover the structure of a database, including what tables, views and connections it contains, and the details of those items. The ADBObjects() function is a one-stop shop for finding out what's in a database, while the DBGetProp() function lets you look up the details of database contents.

ADBObjects() fills an array with a list of one kind of thing in a database. You pass a parameter indicating whether you're interested in tables, views, connections or relations. Listing 7 shows the syntax; cInfoType is one of this list: "TABLE", "VIEW", "CONNECTION", "RELATION". For everything other than relations, the array created has a single column with the names of the specified objects. For relations, the function creates a five-column array providing the names of the tables involved, the tags used to create the relation, and a string indicating whether there are any relational integrity constraints based on this relation.

Listing 7. Use ADBObjects() to explore database contents.

```
nItemCount = ADBObjects( ArrayName, cInfoType)
```

ADBObjects() is also used in the Thor Schema tool, to collect information about the relations in the database. Listing 8 shows that part of the code, while Figure 2 shows that section in the output for the Northwind Products table.

Listing 8. This block of code from Thor's Schema tool reports on relationships in the database.

```
iDbObjects = Adbobjects(aDb,"RELATION")
If Ascan(aDb,Upper(This.cAlias))>0
\ <h4>Relations:</h4><table>
\ <tr><th>Parent Table</th><th>Parent
Tag</th><th>Child Table</th><th>Child
Tag</th></tr>
  For X = 1 To iDbObjects
    If aDb[X,1]=Upper(This.cAlias) Or
aDb[X,2]=Upper(This.cAlias)
\<tr> <td> <<aDb[X,2]>> </td><td><<aDb[X,4]>>
</td><td><<aDb[X,1]>> </td><td><<aDb[X,3]>>
</td></tr>
    Endif
  Next
Endif
```

DBGetProp() lets you find the value of a specific characteristic of an object in a database. You pass the name of the object, the type of object, and the name of the property you're interested in, and the function returns the value of that property. You can ask about anything from the default value of a field to the primary key of a table to the SQL that defines a view. Listing 9 shows the syntax for the function.

Relations:

Parent Table	Parent Tag	Child Table	Child Tag
CATEGORIES	CATEGORYID	PRODUCTS	CATEGORYID
PRODUCTS	PRODUCTID	ORDERDETAILS	PRODUCTID
SUPPLIERS	SUPPLIERID	PRODUCTS	SUPPLIERID

Figure 2. Thor's Schema tool uses ADBObjects() to collect information about relations involving the specified table.

Listing 9. DBGetProp() lets you explore the properties of a database and its contents.

```
uPropertyValue = DBGetProp( cItem, cItemType,
                            cProperty )
```

The possible values for cItemType are: "CONNECTION", "DATABASE", "FIELD", "TABLE", and "VIEW". The list of values you can pass for cProperty varies with the item type. There's a complete list of properties in the Help topic "DBGETPROP( ) Function."

Listing 10 shows a small block of code found in the DoSearch method of the RefSearchDatabase class of Code References. It uses ADBObjects() and DBGetProp() to add the tables in a database to the list of places to search.

Listing 10. This method inside Code References adds the list of tables in a database to the list of places to be searched.

```
* Add tables in DBC to search list
m.nCnt = ADBOBJECTS(aDBList, "TABLE")
FOR m.i = 1 TO m.nCnt
  TRY
    m.cTableName = DBGETPROP(aDBList[m.i], ;
      "TABLE", "PATH")
    THIS.AddFileToSearch( ;
      FULLPATH(m.cTableName, ;
      ADDBS(THIS.Folder)))
  CATCH
    * ignore error (we might possibly get
    * one on DBGETPROP)
  ENDTRY
ENDFOR
```

DBGetProp() has a sibling, DBSetProp(), that lets you set properties of the items in a database. It's handy for tools that manage databases. The syntax, shown in Listing 11, is similar to DBGetProp()'s, but there's a fourth parameter to provide the new value of the specified property.

Listing 11. DBSetProp() lets you change the properties of an object in a database.

```
lSuccess = DBSETPROP( cItem, cItemType,
                      cProperty, uNewValue )
```

Be aware that some properties of database objects can't be changed by DBSetProp(); the list of properties in Help indicates, for each, whether it's read-only or read-write. Those that can't be changed by DBSetProp() generally have another command to set them. For example, the SQL property of a view is set by the CREATE SQL VIEW command.

Listing 12 is drawn from the DoReplace method of the RefSearchDatabase class of Code References. It uses DBSetProp() to change the Comment for a database.

Listing 12. Use DBSetProp() to set those properties of database objects that aren't handled by other, more specific, commands.

```
DBSETPROP(JUSTSTEM(THIS.Filename), ;
  "DATABASE", "Comment", cNewText)
```

Exploring SQL database structures

VFP also includes some functions that let you explore the structure of a SQL database. What's particularly nice about these functions is that they work for pretty much any SQL database, and the results are structured the same way no matter which back-end you're talking to. Both of these functions require that you've already connected to the database, as they expect the handle as the first parameter.

SQLTables() returns a list of tables in the database. You can optionally limit the result to just tables, just views or just system tables. The syntax for SQLTables() is shown in Listing 13. The function returns 1 if it's done, 0 if it's still executing (only possible if you're executing it asynchronously), or a negative value if an error occurred.

Listing 13. SQLTables() fills a cursor with the list of tables in a SQL database.

```
nSuccess = SQLTables( nHandle [, cTableType
                      [, cResultCursor ] ] )
```

The CursorAdapter Builder uses SQLTables() to populate a combobox with the list of tables when you specify an ODBC data source. Figure 3 shows the relevant form and Listing 14 shows the code for that case.

Listing 14. This code, from the GetTables method of the SelectCommandBuilderForm, populates a combo on the form with the list of tables in the remote database.

```
case vartype(.uConnection) = 'N'
  sqltables(.uConnection, 'TABLE', '_Tables')
  scan for upper(TABLE_NAME) <> 'DTPROPERTIES'
    lcTable = trim(TABLE_NAME)
    .AddTableToList(lcTable, lcTable)
  endscan
  use in _Tables
```

SQLColumns() retrieves a list of the columns in a table. You can choose whether to format the list in the customary VFP way or in the format native to the server. Listing 15 shows the syntax for the function. Like SQLTables(), the return value can be 1, 0 or a negative value.

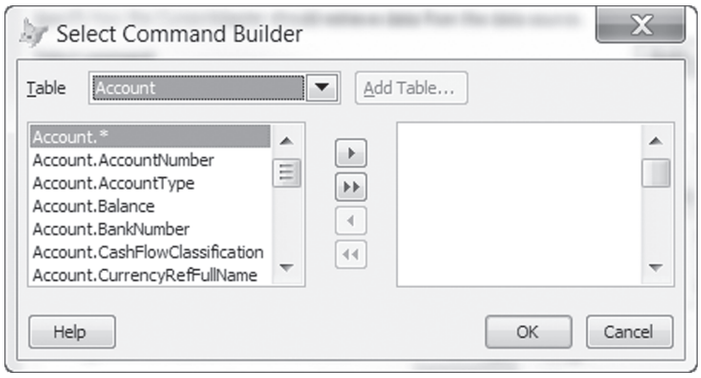

Figure 3. This form from the CursorAdapter Builder uses SQLTables() and SQLColumns() to retrieve data about an ODBC data source.

Listing 15. SQLColumns() can retrieve field information in the standard VFP format or the server's format.

```
nSuccess = SQLColumns(nHandle, cTable
                      [, cFormat
                      [, cResultCursor ] ] )
```

The CursorAdapter Builder uses SQLColumns() to collect the data for the mover in Figure 3. The code is shown in Listing 16.

Listing 16. This code, from the GetFieldsForTable method of the CursorAdapter Builder's SelectCommandBuilderForm, collects the list of fields used in the two-column mover in Figure 3.

```
case vartype(.uConnection) = 'N'
  * If we're using an ODBC connection handle,
  * use SQLCOLUMNS to get the fields.
  * If it fails the first time, try again
  * because sometimes it fails immediately
  * after using SQLCOLUMNS().
  sqlcolumns(.uConnection, lcTable, ;
    'NATIVE', '_Fields')
  if not used('_Fields')
    sqlcolumns(.uConnection, lcTable, ;
     'NATIVE', '_Fields')
  endif not used('_Fields')
  if used('_Fields')
    scan
      lcField  = trim(COLUMN_NAME)
      lcField  = ;
        GetObjectName(trim(COLUMN_NAME))
      lnFields = lnFields + 1
      dimension laFields[lnFields]
      laFields[lnFields] = lcTable + '.' + ;
        lcField
    endscan
  use in _Fields
endif used('_Fields')
```

Determining what's in use

Several functions let you find out what databases and tables are open and what data sessions are in use. These are useful in several ways:

- Allowing you to save and restore the data set-up, if you need to change it in a tool;
- Letting you figure out what to operate on;
- Reporting the data set-up in an error handler.

ASessions() fills an array with a list of data sessions in use. It takes an array as its only parameter; the array comes back with a single column, listing the data session numbers in use. You might think this function is unnecessary, but it is possible for the data sessions in use to have non-sequential numbers. Suppose you open three forms, each with a private data session. At that point, you'd have data sessions 1 through 4 in use (1 is the default, shared, data session). If you then close the first form you opened, data session 2 is no longer is use; you have data sessions 1, 3 and 4.

ADatabases() fills an array with the list of open databases. It, too, takes a single parameter, the array name. The resulting array has two columns; the first tells you the name (just the filestem) of the database, while the second contains the path to the DBF.

AUsed() fills an array with a list of tables in use in the current or a specified data session; its syntax is shown in Listing 17. As you'd expect, omitting the second parameter applies the function to the current data session.

Listing 17. Call AUsed() to find out what tables are open in a particular data session.

```
nTables = AUSED( ArrayName [, nDataSession] )
```

The code in **Listing 18** demonstrates both ADatabases() and AUsed(). It's part of the code to populate the combo and listboxes in the _tablemover class of the FFC (FoxPro Foundation Classes).

Listing 18. This code, from the InitData method of the _TableMover class, adds open databases to a combo box, and adds open tables to the list of tables.

```
m.nDBCCount=ADATABASES(aDBC)
FOR i = 1 TO m.nDBCCount
  * Add bar for popup
  IF m.i = 1
    THIS.cboData.AddItem("\-")
  ENDIF
  THIS.cboData.AddItem(aDBC[m.i,1])
ENDFOR

* Go thru workareas and see which tables open
m.nTotWorkAreas = AUSED(aWorkAreas)
FOR m.nCount = 1 TO m.nTotWorkAreas
  m.nWorkArea = aWorkAreas[m.nCount,2]

  * Avoid specific tables used by wizard and
  * not in a DBC
  DO CASE
  CASE ASCAN(aSkipTables,DBF(m.nWorkArea))#0
    LOOP
  CASE ISREADONLY(m.nWorkArea) AND ;
       !THIS.AllowReadOnly
    * skip for read-only tables and queries
    LOOP
  CASE EMPTY(THIS.GetDBCName(m.nWorkArea))
    * Add to free tables list
    IF ATC(".TMP",DBF(m.nWorkArea))#0 AND ;
       !THIS.AllowQuery
      LOOP
    ENDIF
    IF !EMPTY(aDBFList[1])
      DIMENSION aDBFList[ALEN(aDBFList,1)+1,2]
    ENDIF
    aDBFList[ALEN(aDBFList,1),1] = ;
      DBF(m.nWorkArea)
    aDBFList[ALEN(aDBFList,1),2] = ;
      ALIAS(m.nWorkArea)
  OTHERWISE
    * Need to determine if its a Table, Local
    * View or Remote View
    * Add to DBC tables list
    IF !THIS.AllowViews AND ;
     CURSORGETPROP("sourcetype",m.nWorkArea)#3
      LOOP
    ENDIF
    IF !EMPTY(aDBCList[1])
      DIMENSION aDBCList[ALEN(aDBCList,1)+1,2]
    ENDIF
    IF CURSORGETPROP("sourcetype", ;
      m.nWorkArea)#3  &&handle view here
      aDBCList[ALEN(aDBCList,1),1] = ;
        UPPER(CURSORGETPROP("sourcename", ;
        m.nWorkArea))
    ELSE
      aDBCList[ALEN(aDBCList,1),1] = ;
        DBF(m.nWorkArea)
    ENDIF
    aDBCList[ALEN(aDBCList,1),2] = ;
      ALIAS(m.nWorkArea)
  ENDCASE
ENDFOR
```

More to come

In my next article, I'll look at commands and functions that let you explore and work with classes and forms. The final article in this series will look at language elements that operate on code and on projects.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. Tamar is author or co-author of nearly a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with Visual FoxPro and Taming Visual FoxPro's SQL. Her latest collaboration is VFPX: Open Source Treasure for the VFP Developer. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar was a Microsoft Support Most Valuable Professional from the program's inception in 1993 until 2011. She is one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.

DOWNLOAD

Subscribers can download FR201307_code.zip in the SourceCode sub directory of the document portal. It contains the following files:

rickschummer201307_code.zip
Source code for the article "Application Updater" from Rick Schummer