One of the most anticipated features of version 5.0 is the ability to create OLE Servers. Now you can create components in Visual FoxPro and use them in other applications such as Visual Basic, Excel, Access, or Visual FoxPro itself. The major benefit to building and using OLE Automation Servers is reusability. This article will try to equip you with the basics you need to understand OLE from the perspective of objects, automation, and servers. In addition, John will cover the steps required to build an OLE Automation Server and implement it in different applications.

Object Linking and Embedding (OLE) is the foundation technology that enables different applications to interoperate. The technology used to be limited to just linking and embedding documents between applications. For example, a Word document could contain an embedded Excel worksheet or a PaintBrush picture. Today, OLE provides much more power through OLE Automation.

OLE Automation is simply the mechanism whereby one application controls another application, made possible when an application exposes a standard set of properties and methods that can then be accessed by other applications. Code in one application can manipulate those properties and invoke methods. Every OLE Automation scenario involves a client and server. The client is the controlling application; the server provides the services. For example, consider this block of code in Visual FoxPro:

```vfp
oExcelApplication = CreateObject("Excel.Application")
oExcelApplication.WorkBooks.Add
oExcelApplication.ActiveSheet.Range("A1").Value = 1
oExcelApplication.Visible = .T.
```

In this scenario, Visual FoxPro creates an instance of the Excel application, adds a workbook, assigns the first cell in the active worksheet the value of 1, and finally makes the Excel application visible. Excel has exposed properties and methods that make it possible for applications such as Visual FoxPro to control it. Excel is the OLE Automation Server because it has provided the interface of properties and methods. Visual FoxPro is the client because it has initiated the process and done the controlling.

**VFP as an OLE Automation Server**

Visual FoxPro 3.0 is only capable of being an OLE Automation client, because it can control other applications but other applications can't control it. Version 5.0 has the capability to be both an OLE Automation client and server.
The following example illustrates how Visual FoxPro can be used as an OLE Automation Server. To effectively use a VFP Application as an OLE Server, you'll need to understand three methods associated with Visual FoxPro when it's utilized as an OLE Object (assume that an instance of VFP was created and the variable "o" is used to store the object reference):

- **DoCmd()**: Executes a Visual FoxPro command for an instance of the Visual FoxPro application automation server. Syntax: ApplicationObject.DoCmd(cCommand)
  
  ```vba
  o.DoCmd("USE Customer SHARED")
  ```

- **Eval()**: Evaluates an expression and returns the result for an instance of the Visual FoxPro application automation server. Syntax: ApplicationObject.Eval(cExpression)
  
  ```vba
  o.Eval("_TALLY")
  ```

- **SetVar()**: Creates a variable and stores a value to the variable for an instance of the Visual FoxPro application automation server. Syntax: ApplicationObject.SetVar(cVariableName, eValue)
  
  ```vba
  o.SetVar("X",1)
  ```

With these basics out of the way, you can now create and examine the code in your Excel/VBA module. (If you don't want to type the code manually, it's available in download file 03PETERS.EXE: )
Sub GetVFPData()
    Dim o As Object
    Dim iCols, iRows, ifCount, iRecs As Integer
    Dim cArgument, cCountry As String
    Set o = CreateObject("VisualFoxPro.Application")
    ' Depending on your installation, you may have
    ' to change the directory in the following line.
    o.docmd ("CD \vfp50\samples\tastrade\data")
    cCountry = Range("A1").Value
    ' There is no line continuation character in VBA.
    ' While the following line appears to
    ' be on two lines, it's only wrapped
    ' for publication purposes.
    cArgument = "SELECT * FROM Customer WHERE Country = "+ Trim(cCountry) & "+ INTO CURSOR cTemp"
    o.docmd (cArgument)
    Range("C1").Value = o.Eval("_tally")
    Range("D1").Value = cArgument
    ifCount = o.Eval("FCOUNT()"
    iRecs = o.Eval("RECC()"
    For iCols = 1 To ifCount
        cArgument = "FIELD(" & Str$(iCols) & ")"
        ActiveSheet.Cells(2, iCols).Value = o.Eval(cArgument)
    Next iCols
    For iRows = 3 To iRecs + 2
        For iCols = 1 To ifCount
            cArgument = "EVAL(FIELD(" & Str$(iCols) & "))"
            ' The next line is again split in two for
            ' formatting purposes.
            ActiveSheet.Cells(iRows, iCols).Value = o.Eval(cArgument)
        Next iCols
        o.docmd ("SKIP 1")
    Next iRows
End Sub

The GetVFPData() VBA function is contained in a module within an Excel WorkBook. The user
types the name of a country in the first cell and then presses a CommandButton embedded on the
worksheet. The CommandButton has the GetVFPData() function assigned as a macro. When the
user clicks the CommandButton, the code inside GetVFPData() executes. Through OLE
Automation, a SQL-Select statement runs against the Customer Table of the Tastrade! Database.
The number of records found is entered in cell C1 and the passed SQL-Select string is entered in

cell D1. The second row of the worksheet contains the name of the columns in the SQL cursor.
The remaining rows and columns contain the data (see Figure 1).

Creating your own OLE Automation Servers

Up to this point, I've discussed OLE Servers in the context of an entire application. For example,
you can create an Excel Object in Visual FoxPro and a Visual FoxPro Object in Excel. While
VFP has rich functionality, it doesn't encompass all of the needs a developer might have. What if
you could create your own OLE servers, complete with properties and methods that you define yourself? With version 5.0, you can! Think of an OLE Custom Server as a UDF on steroids. Just as you define the characteristics and behavior of a UDF, you define the behavior and characteristics of a custom OLE Server. The two key differences between a UDF and an OLE Automation Server are: first, the OLE Automation Server is an application-independent object; and second, it can be used by any OLE-enabled client. This means OLE Servers created in Visual FoxPro can be used by applications created in Access, Visual Basic, and Excel, to name a few.

Creating an OLE Server is very easy. If you know how to create a class and build an application, you're already 80 percent of the way there. What might you use an OLE Server for? Consider the following scenario. Your organization uses a variety of development tools: Visual FoxPro, Access, Visual Basic, and Excel. You've received a request to create a program that will accept a five-digit ZIP code and return the state, city, and county related to the ZIP code. The added twist is that the program you write must be usable by all tools with which your company works. Because of the quantity of data involved, you've decided that Visual FoxPro is the perfect tool for the job.

Now that you have a specification, it's time to get to work. In the words of Julia Child, "I already have one prepared for you." All of the following sample code is available in download file 03PETERS.EXE. Your first step is to design a table that will hold the ZIP code data. Here's the structure of the ZIPLKUP Table:

<table>
<thead>
<tr>
<th>Field</th>
<th>Field Name</th>
<th>Type</th>
<th>Width</th>
<th>Dec</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ZIP</td>
<td>Character</td>
<td>5</td>
<td>Asc</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>STATE</td>
<td>Character</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CITY</td>
<td>Character</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>COUNTY</td>
<td>Character</td>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table in our sample file contains 5,000 ZIP codes and one index on the ZIP Field. Because ZIP code data changes only once a quarter (at most) and the user always needs access to the data, the table and index files are included in the project.

Next, you need to create a class that will open the table, accept a ZIP code parameter, seek the value, and return the needed information. The following is the class definition:
**ZIPLKUP.PRG
**Class code for zip code lookup OLE Server
**John V. Petersen
**9/2/96

DEFINE CLASS ziplkup AS CUSTOM OLEPUBLIC
    DOCREATE = .T.
    zipcode = SPACE(0)
    state = SPACE(0)
    city = SPACE(0)
    county = SPACE(0)
    NAME = "ziplkup"

PROCEDURE INIT
    * No need to trap for not being able to open
    * the table because the DBF/CDX will be
    * bound into the DLL.
    USE ziplkup ORDER zip
    RETURN
ENDPROC

PROCEDURE GetData
    LPARAMETERS tcZip
    IF TYPE("tcZip") = "C"
        tcZip = ALLTRIM(tcZip)
    IF SEEK(tcZip,"ziplkup")
        WITH THIS
            .zipcode = ziplkup.zip
            .state = ziplkup.state
            .city = ziplkup.city
            .county = ziplkup.county
        ENDWITH
    ELSE
        WITH THIS
            .state = SPACE(0)
            .city = SPACE(0)
            .county = SPACE(0)
        ENDWITH
    ENDIF
    ENDIF
    RETURN
ENDPROC
ENDDEFINE

This class is almost identical to one that could have been defined in version 3.0. The only difference is the OLEPUBLIC keyword. Including this keyword in your class specifies that the class in an custom OLE server can be accessed by an OLE Automation client. In this example, the OLE Server created will contain only one class definition, ZIPLKUP. Classes created in the Class Designer can also be specified as OLEPUBLIC in the Class Info dialog box (see Figure 2).

With the class definition in hand, it's time to create a project using the following steps (see
1. Create a new project called ZIP.

2. Add the ZIPLKUP Program to the ZIP Project.

3. Add the ZIPLKUP.DBF/.CDX to the ZIP Project (make sure to include these files).

4. Press the Build Button.

5. Choose Build OLE DLL.

By now you've already discovered several new features of the Project Manager. It should be noted that OLE Servers can either be .EXE or .DLL files. The following outlines the difference between the two types of OLE Server files:

• **.EXE files**: These files are known as out-of-process OLE Servers because they run in their own processes. The communication that occurs between an OLE client application and an out-of-process server is known as cross-process communication. Pro: .EXE files can be deployed remotely. Con: Performance hits can occur due to cross-communication overhead.

• **.DLL files**: These files are known as in-process OLE Servers because they run in the same process space as the OLE client applications that call them. Pro: .DLLs are faster than out-of-process servers due to lack of cross-communication overhead. Con: They can't be deployed remotely.

Two additional files are created when a .DLL or .EXE file is created: one has a .VBR extension and the other a .TLB extension. The .TLB file is called a type library; the .VBR file is a registration file. Two files have been created in this example: ZIP.VBR and ZIP.TLB. Here are the specifics on each file:

• **.TLB files**: These are type library files, containing information regarding the properties and methods that a class has exposed. The Class Browser in version 5.0 is capable of reading .TLB files (see Figure 5). This gives a developer the ability to browse the member methods and properties of an OLE Server. When a .DLL contains more than one type library, its associated type library file will have an .OLB extension.

• **.VBR files**: These files allow you to register the class definitions in the system registry when the executable file is moved to a different computer.

Building an OLE Server is basically the same as building a Visual FoxPro .APP/.EXE file. The big difference occurs at the end of the build process. When the type library file is built, an entry in the Windows registry is created to store information about the OLE Server just created. The following is a description of the Windows registry from the VFP online help file:

*Windows registry*: "A database maintained by Windows that stores configuration information about the operating system, all Windows applications, ActiveX, OLE, and optional components such as ODBC. For example, the registry is where Windows stores the associations between file
name extensions and applications, and where Visual FoxPro stores its application-specific configuration information."

In order to use your OLE Server, it must be registered in the Windows registry. If you have a problem with "class definition class.method not found" errors, it's likely that your OLE Server isn't registered properly. Figure 6 shows the entry for our OLE Server.

In addition to the new build options, there's also a new Servers Page in the Project Information dialog box. The following is from the online help file that accompanies VFP.

**Server classes**

Displays a list of all the available classes, both .VCX and in code, which are marked as OLEPublic. Each class in the list has a corresponding .PJX file.

**Class library**

Displays the directory path of the class library in which the selected class is stored.

**Class name**

Displays the name of the class.

**Instancing**

- **Single Use**: Specifies that you can create an instance of the class both inside and outside Visual FoxPro using OLE Automation. Each request for an instance of the class by an OLE client outside the project causes a separate copy of the OLE server to start.

- **Not Creatable**: Specifies that you can create instances of the class only inside Visual FoxPro.

- **Multi-Use**: Specifies that you can create an instance of the class both inside and outside Visual FoxPro using OLE Automation. Each request for an instance of the class by an OLE client outside the project causes an already running copy of the OLE server to be provided as the source for the new instance.

**Description**

Displays a small description of the class.

**Help file**

Displays the Help file associated with the class. Select the dialog button to display the Open dialog box if no file is listed.

**Help context ID**

Displays the context ID of the Help file associated with the application, if a Help file is
distributed with the application.

**Project name**

Displays the name of the project associated with the server classes.

**Typelib description**

Displays a description of the class library on which the selected class is based.

**Putting the OLE Server to work**

Now let’s get to the fun part: putting your new OLE Server to work. The OLE Server I just created will be implemented in Visual FoxPro, Visual Basic, Access, and Excel. The essence of using an OLE Server centers upon instantiating the server and invoking its methods or manipulating its properties. The code in each of the following examples -- each in a different application environment is almost identical.

Figure 7 illustrates the OLE Server in action with the different tools. In each case, the user enters a ZIP code in a text box (or in the case of Excel, cell A1). Then the user presses a CommandButton which instantiates the OLE Server, makes the call to the GetData() Method, and then updates the interface with state, city, and county information.

**Visual FoxPro -- Command1.Click()**

While the "o" variable is declared Local, you don't have to declare the variable or its data type prior to using it:

```vbnet
LOCAL o
o = CREATEOBJECT("zip.ziplookup")
o.GetData (THIS.PARENT.txtZip.Text)
THIS.PARENT.txtState.Value = o.state
THIS.PARENT.txtCity.Value = o.city
THIS.PARENT.txtCounty.Value = o.county
```

**Visual Basic -- Command1.Click()**

You must declare and specify the "o" variable as an Object Type prior to storing a reference to the OLE server because Visual Basic is a strongly typed language. The only other notable difference between VB and VFP syntax is the Set keyword when you assign a value to a variable. Also, the city, state, and county information can be assigned directly to the text box. It's assumed that the Value Property will hold information directly assigned to a text box control:
Private Sub Command1_Click()
    Dim o As Object
    Set o = CreateObject("zip.ziplkup")
    o.GetData (Screen.ActiveForm.txtZip)
    Screen.ActiveForm.txtState = o.state
    Screen.ActiveForm.txtCity = o.city
    Screen.ActiveForm.txtCounty = o.county
End Sub

Access -- Command0_Click()

Access Basic is almost identical to Visual Basic. The only difference is that you must first
invoke the SetFocus() method of a control before referencing any of its properties. Another
interesting point is the way the argument is passed to the GetData() Method. In Access, you don't
have to use the Text Property as in Visual Basic:

Private Sub Command0_Click()
    Dim o As Object
    Dim loForm As Object
    Set o = CreateObject("zip.ziplkup")
    Set loForm = Screen.ActiveForm
    loForm.txtZip.SetFocus
    o.GetData (Form_Form1.txtZip)
    loForm.txtState.SetFocus
    loForm.txtState = o.State
    loForm.txtCity.SetFocus
    loForm.txtCity = o.City
    loForm.txtCounty.SetFocus
    loForm.txtCounty = o.County
End Sub

Excel

The only difference in the implementation here is the substitution of cells for text boxes. In
addition, if you enter a ZIP code such as 19301 in a cell, it's assumed to be a numeric value
unless you place quotes around the expression. To keep data entry consistent, the Str$() function
converts the data in the method code:

Sub ZipLkUp()
    Dim o As Object
    Set o = CreateObject("zip.ziplkup")
    o.GetData (Str$(ActiveSheet.Range("B1").Value))
    ActiveSheet.Range("B2").Value = o.State
    ActiveSheet.Range("B3").Value = o.City
    ActiveSheet.Range("B4").Value = o.County
End Sub
Conclusion

The ability to build OLE Servers is a powerful skill to add to your toolkit. The future of development will center on putting different components together to form a single solution. The promise of different applications working together is here, and it works! The client/server arena has the most to gain from OLE Servers. With an OLE Server, true three-tier client/server applications can be written in Visual FoxPro using servers written in Visual FoxPro. Most importantly, however, is that these servers can be used by any OLE-enabled client.

With this technology firmly in place, Visual FoxPro is finally a full-fledged member of the Visual Development Tools family of Microsoft products and has once again proven itself to be the ideal environment for database application development.

John V. Petersen, MBA, specializes in the development of Visual FoxPro-based solutions for business. John has written for U.S. publications and has been a speaker at user group meetings and developer conferences. John is a Microsoft Most Valuable Professional (MVP) as well as coauthor of Developing Visual FoxPro 5.0 Enterprise Applications from Prima Publishing. 610-651-0879, johnpetersen@mail.com.

The Developers are Revolting

Whil Hentzen

Okay, that's what you secretly think your customers are saying about you behind your back, right? Well, I meant something a bit different <g>.

I'm writing this on New Year's Eve (all together now: "Whil, you need to get out more!") while watching a fight (and occasionally catching a bit of hockey). I've spent the last month poking around the various nooks and crannies of a suite of development tools from a major software manufacturer. Actually, I've spent a month just installing this suite, and it wasn't because I had difficulties. It just took that long to install and configure these tools800M worth. That isn't a typo, and it isn't an inefficient use of disk space -- it's just a lot of software.

Given what I've been through, let me illustrate a scenario in which you may soon find yourself.

A tool like VFP occupies between 50 and 100M, and you could spend a year or more just getting reasonably proficient with it. So it looks like somewhere between a five and 10-year investment to get reasonably proficient with this suite. Okay, yes, that's unreasonable. No one would need to get really good at all of the pieces -- just some of them, right? And not all development tools are as complex as VFP, right? So let's say that you could get reasonably proficient with the necessary pieces in a year -- ambitious, optimistic, maybe even naïve. But let's just assume, okay? (C'mon, work with me on this. . .)

So, here we are. We've spent a year mastering a very large, complex suite of tools. Guess what? The whole suite will be updated with a new rev somewhere between 12 and 18 months from
now. So about the time you've gotten good, a new stack of two dozen CDs will land on your desk. So now it's time to learn all over again. A new set of tools. New features, functionality, abilities, capabilities . . . it's exciting and invigorating to accomplish what you were trying to do a year ago with tools that didn't quite cut it.

What's wrong with this picture? When do you get a chance to ship some apps? When do you make money? When do you receive some payback on the financial, temporal, and emotional -- yes, emotional -- investment you've made over the last year?

How many years do you think you'll go through this routine, making a huge time investment only to have it vaporize about the time you're ready to capitalize on it? Yeah, me too.

At this point, you're probably thinking two things. First, you may be wondering when you're going to face this situation. It's possible that you're already tired from the transitions from FoxPro 1.0 to FoxPro 2.0 (Power Tools, SQL, and so on), and then FoxPro to Visual FoxPro. Do you really want to face that with several or a half-dozen tools?

And the second thing you're possibly thinking is, "Well, what's the magic solution? Whil seems to have some pretty good ideas for the issues he brings up -- how's he going to wrap this one up by the end of the show?"

Unfortunately, I'm stuck. I don't have a solution for you. Or for me, for that matter. So instead I'm making a prediction. I believe that the developer community will splinter into four factions. The first will ignore the issue and continue to develop in Clipper Summer '87 for another five years. The second will decide to skip generations -- trying to wangle some life out of an investment and then making two jumps instead of one. The third will try to keep up, grabbing each new rev with gusto and living on Jolt Cola and adrenaline. And the last will simply revolt.

What form is this revolution going to take? I think it depends on how large the contingent is.

Stay tuned . . . the complete story at eleven!

Custom Classes

Doug Hennig

This month's column uses great ideas from other applications to create some custom classes you can include in any application.

Last month we looked at properties, events, and methods (PEMs) of subclasses of the Visual FoxPro base classes I've created for my use. In this article, we'll examine some special subclasses of these classes: an EditBox that automatically expands keywords to complete text (similar to the AutoCorrect function in Microsoft Word), a ComboBox that supports a feature similar to Quicken's "quick fill" function, and a TextBox that makes it obvious when it has focus.

The great part about using custom classes like these is that they can become "black box" objects: just drop them on a form, set a few properties, and don't even worry about how they do what they
AutoCorrect EditBox

Several years ago, the company I worked for used a time and billing program called TBR to track consulting time and bill clients. Although it was a clunky program to use, one thing I really liked about it was the idea of dictionary codes: two-letter codes that would expand to a complete phrase. Dictionary codes made data entry much faster because I could predefine common phrases, such as "met with," "to discuss," "project status," or even "Prepared system documentation for project," and then enter those phrases by just entering the two-letter code. I could mix and match regular text with the dictionary codes; TBR used "@" to indicate that the next two letters were a code. For example, "@mw Bob Jones @td @ps" would expand to "Met with Bob Jones to discuss project status." The bothersome thing about it, though, was that it wouldn't expand the text on the screen -- only in printed copy (such as on timesheets and invoices).

I liked this feature so much that I implemented a version of it in FoxPro 2.x. I called it a "memo dictionary" because I generally used it only in memo fields. Unlike TBR, I made the Valid clause of the memo field process the text, looking for "@" characters and expanding the codes by looking them up in a table called MEMODICT. This simple table had just two fields: CODE (the code entered by the user) and DESCRIP (the text to put in place of the code). This feature allowed users to see their complete text as soon as they left the field. While this was an improvement, it still wasn't as handy as similar features in Microsoft Word (the AutoCorrect feature) and the Cob Editor Extensions (a FoxPro editor add-on written by Randy Wallin and Ryan Katri). What I really wanted was the ability to expand the code as soon as the user pressed the space bar. Unfortunately, I just couldn't do it in FoxPro 2.x using native FoxPro code.

Then came VFP. The KeyPress event in the EditBox control provided exactly what I needed: the ability to trap every keystroke, check whether the user entered a code or not, and if so, expand it immediately. The SFMemoDictEditBox class is the result.

This class, which is contained in download file 03HENNIG.EXE (CONTROLS.VCX), is based on the SFEditBox class I discussed last month (also in CONTROLS.VCX). I added several public properties to this class, which are listed in Table 1. Among other things, these properties allow you to customize the name, alias, and tag of the memo dictionary table, and the name of the field that contains the expanded text. I also added one protected property: lOpened, which is .T. if this control opened the memo dictionary table.

Table 1. SFMemoDictEditBox Public Properties.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cDictAlias</td>
<td>The alias of the memo dictionary table (default = MEMODICT)</td>
</tr>
<tr>
<td>cDictCodeChar</td>
<td>The character used to indicate the start of a dictionary code (default = @)</td>
</tr>
<tr>
<td>cDictField</td>
<td>The field containing the expansion text for the code (default = DESCRIP)</td>
</tr>
<tr>
<td>cDictFile</td>
<td>The name of the memo dictionary table (default = MEMODICT.DBF)</td>
</tr>
<tr>
<td>cDictTag</td>
<td>The tag to use for the SEEK in the memo dictionary table (default = CODE)</td>
</tr>
</tbody>
</table>
cExpandKeyCode A comma-delimited list of the keypress codes used to terminate the entry of a dictionary code (default = 9,13,32,44,46,59 which is Tab, Enter, Space, comma, period, and semicolon)

The Init() method opens the memo dictionary table if necessary and sets lOpened if we did so:

```vba
with This
    if not empty(.cDictAlias) and ;
        not used(.cDictAlias) and ;
            ((empty(dbc()) and file(.cDictFile)) or ;
                indbc(.cDictAlias, 'Table'))
        .lOpened = .T.
    use (.cDictFile) alias (.cDictAlias) ;
        again shared in 0
    endif not empty(.cDictAlias) ...
endwith
```

The Destroy() method closes the memo dictionary table if necessary:

```vba
with This
    if .lOpened and used(.cDictAlias)
        use in (.cDictAlias)
    endif .lOpened ...
endwith
```

The code in the KeyPress() event checks each keypress to see if the user typed a character that could terminate a code, such as the space bar, Enter, or punctuation; if so, it calls FindCode() to see if a code was entered:

```vba
LPARAMETERS nKeyCode, nShiftAltCt
clocal lcKey
    lcKey = ltrim(str(nKeyCode))
    if lcKey $ This.cExpandKeyCodes
        This.FindCode()
    endif lcKey $ This.cExpandKeyCodes
```

Valid() also calls FindCode() because the user could have entered a code as the last characters before leaving the field:

```vba
This.FindCode()
dodefault()
```

The real work is done in two custom public methods. FindCode() scans the content of the EditBox, starting from the current cursor position and going backwards to the start, to see if a
"start code" character (the default is "@") was entered. If so, it calls ExpandCode() to look up the code in the memo dictionary table and substitute the expanded text for the code. Here's the code for FindCode():

```vba
local lnI
with This
    for lnI = .SelStart to 1 step -1
        if substr(.Value, lnI, 1) = .cDictCodeChar
            .ExpandCode(lnI, .SelStart)
            exit
        endif
        substr(.Value, lnI, 1) = .cDictCodeChar
    next lnI
endwith
```

Here's ExpandCode():

```vba
lparameters tnStart, ;
    tnEnd
local lcCode, ;
    lcExact, ;
    lcExpand
with This
    lcCode  = substr(.Value, tnStart + 1, ;
        tnEnd - tnStart)
    lcExact = set('EXACT')
    set exact on
    do case
        case empty(lcCode) or empty(.cDictAlias) or ;
            not used(.cDictAlias)
        case seek(upper(lcCode), .cDictAlias, .cDictTag)
            lcExpand  = trim(evaluate(.cDictAlias + '.' + ;
                .cDictField))
            .Value    = stuff(.Value, tnStart, ;
                tnEnd - tnStart + 1, lcExpand)
            .SelStart = .SelStart + len(lcExpand) - ;
                (tnEnd - tnStart + 1)
        endcase
    if lcExact = 'OFF'
        set exact off
    endif
endwith
```

To see how this class works, run the SAMPLE1 form included in download file 03HENNIG.EXE. It uses MEMODICT.DBF as the memo dictionary table. Several codes have already been defined in this table (TD, MW, TD, and DS), but of course you can add new ones. The codes can be up to five characters long (although simply changing the size of MEMODICT.CODE will allow you to use shorter or longer ones). Try entering something like "@mw Bob Jones @td proposal" and notice that the codes are expanded as you type.
AutoFill ComboBox

More than a decade ago, I used a Macintosh database program called OverVUE. This program was revolutionary in many ways, but one feature called "precognition" stood out. As I typed in a field that had this feature turned on, OverVUE would automatically look to see if another record had these same characters entered, and if so, fill in the rest of the field with the complete entry. This is similar to an incremental search feature, but with a twist: The characters filled in by the program are highlighted so if the user keeps typing, the new characters replace the selected text and the incremental search tries again. This feature, which has been implemented in other programs such as Quicken, is great for entering data with frequently used values, because the user can enter just enough characters so the desired value is filled in, and then move to the next field. An obvious use of this is a lookup field into another table.

As with the memo dictionary feature, I decided I liked this so much that I created a class to provide it. Rather than using a TextBox, though, SFAutoFillComboBox is based on the SFComboBox class I discussed last month (both classes are contained in CONTROLS.VCX in download file 03HENNIG.EXE). I decided to use a ComboBox because this would give the user three ways to enter a value: typing the entire string, typing just a few characters and letting the field "autofill" with the nearest value, or selecting it from the drop-down portion of the ComboBox. However, this functionality could easily be added to a TextBox class as well; this might even be a better choice if the lookup table contains a lot of records.

The public properties I added to this class are listed in Table 2. There's also a protected property: lFoundMatch, which is .T. if an autofill match was previously found (you'll need this so you can determine what to do if the user presses the Backspace key).

Table 2. SFAutoFillComboBox Public Properties.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cAutoFillAlias</td>
<td>The alias to search for an autofill entry (required only if RowSourceType is 3-SQL Select or 4-Query)</td>
</tr>
<tr>
<td>cAutoFillTag</td>
<td>The tag to search on for the autofill entry (required only if RowSourceType is 2-Alias, 3-SQL Select, or 4-Query)</td>
</tr>
<tr>
<td>cAutoFillValue</td>
<td>The field to return for the autofill entry (required only if RowSourceType is 2-Alias, 3-SQL Select, or 4-Query)</td>
</tr>
<tr>
<td>IAutoFill</td>
<td>.T. to enable the autofill feature (default = .T.)</td>
</tr>
<tr>
<td>IUpper</td>
<td>.T. if the entered text should be uppercased before searching for an autofill value (required only if RowSourceType is 2-Alias, 3-SQL Select, or 4-Query)</td>
</tr>
</tbody>
</table>

The majority of the work in this class is done in the InteractiveChange() event. It looks for the value the user entered in one of a number of places, depending on the RowSourceType property. For example, if RowSourceType is 5-Array, it searches the array specified in RowSource. If it's 2-Alias, 3-SQL Select, or 4-Query and the specified cursor is indexed, it does a SEEK in the cursor specified in RowSource. Here's the code for this method:
#define cnBACKSPACE 127
local llFound, lnCursor, lcText, laArray[1], lnRow, lcAlias, lnPos, lcValues, lnStart, lnI, lcValue

* If you're doing autofill, save the current cursor position and get the current text entry.

llFound = .F.
with This
   if .lAutoFill and .Style = 0
      lnCursor = .SelStart
      lcText   = alltrim(.Text)
   endif

* If you previously found a matching item and the user presses the backspace key, you need to manually remove the last character.

if lastkey() = cnBACKSPACE and .lFoundMatch and not empty(lcText)
   lnCursor = lnCursor - 1
   if lnCursor < 1
      store '' to lcText, .DisplayValue
   else
      lcText = left(lcText, lnCursor)
   endif
endif

* If the user entered something, try to find a match.

if not empty(lcText)
do case
   case .RowSourceType = 5
      = evaluate('acopy(' + .RowSource + ', ;
      lnRow   = .ArrayScan(@laArray, lcText, 1)
      llFound = lnRow > 0
      if llFound
         .DisplayValue = laArray[lnRow, 1]
      endif
   endif
   case .RowSourceType = 3
      lnRow   = .ArrayScan(@lcValues, lcText, 1)
      llFound = lnRow > 0
      if llFound
         .DisplayValue = lcValues[lnRow, 1]
      endif
   endif
   case .RowSourceType = 1
      lnRow   = .ArrayScan(@lcAlias, lcText, 1)
      llFound = lnRow > 0
      if llFound
         .DisplayValue = lcAlias[lnRow, 1]
      endif
   endif
endcase
endif

* If the combobox is based on an alias or an indexed SQL select, seek for the value based on the
* text in the array.

case .RowSourceType = 3
   lnRow   = .ArrayScan(@lcValues, lcText, 1)
   llFound = lnRow > 0
   if llFound
      .DisplayValue = lcValues[lnRow, 1]
   endif
endif

case .RowSourceType = 1
   lnRow   = .ArrayScan(@lcAlias, lcText, 1)
   llFound = lnRow > 0
   if llFound
      .DisplayValue = lcAlias[lnRow, 1]
   endif
endif

ArrayScan() is a protected method that searches a particular column in an array for a specified value. This is the same code that I presented in my January 1997 column on arrays, but I added it again here so this control is completely self-contained:

```vbp
lparameters taArray, ;
    tuValue, ;
    tnColumn, ;
    tnOccur
external array taArray
local lnColumn, ;
    lnOccur, ;
    lnRow, ;
    lnStartElement, ;
    lnFound, ;
    lnColumns, ;
    lnElement, ;
    lnCol
lnColumn        = iif(type('tnColumn') = 'N', tnColumn, 1)
lnOccur         = iif(type('tnOccur')  = 'N', tnOccur, 1)
lnRow            = 0
lnStartElement = 1
lnFound          = 0
lnColumns        = alen(taArray, 2)

    * Use ASCAN to find the value in the array, then
    * determine if it's in the correct column. If not,
    * change the starting element number and try again.

    do while .T.
    lnElement = ascan(taArray, tuValue, lnStartElement)
    if lnElement <> 0
        lnCol = iif(lnColumns > 1, ;
                    asubscript(taArray, lnElement, 2), 1)
        if lnCol = lnColumn and (type('tuValue') <> 'C' or ;
                                   taArray[lnElement] = tuValue)
            lnFound = lnFound + 1
        endif lnCol = lnColumn ...
        if lnFound = lnOccur
            lnRow = iif(lnColumns > 1, ;
                        asubscript(taArray, lnElement, 1), lnElement)
            exit
        endif lnCol = lnColumn ...;
        lnStartElement = lnElement + 1
    else
        exit
    endif lnElement <> 0
endo while .T.
return lnRow
```

The SAMPLE1 form, included in download file 03HENNIG.EXE, also has a demo of this control. This form has the CUSTOMER table from the VFP 5.0 sample data in its
DataEnvironment; you'll need to modify the form if the sample data isn't installed or if you named the directory something other than VFP5. The SFAutoFillComboBox is bound to the COMPANY field in the CUSTOMER table. Try entering "A" (the case is important because this table isn't indexed on UPPER(COMPANY)) and notice that the rest of the field fills in with the name of the first company starting with "A" (Alfreds Futterkiste), and all but the first character is selected. Next, enter "r" (again, the case is important), and notice that the first company starting with "Ar" (Around the Horn) is now displayed. Press Backspace, and Alfreds Futterkiste is once again displayed.

Visible Edit TextBox

The contact manager we use at Stonefield Systems Group is GoldMine for Windows. One thing I like about its user interface is that you can easily tell which field you're currently editing: all fields appear plain with a grey background, while the field with focus appears 3-D with a white background. No more searching around, wondering where the cursor is, especially on a laptop. I liked this feature, so I created a simple SFVisibleEditTextBox class (based on SFTextBox, both of which are in CONTROLS.VCX) that provides similar functionality. (By the way, in case you're noticing a trend here, I do like to "borrow" the best features from programs I like.)

To create this class, I simply set the BackStyle property to 0-Transparent so the background color of the container shows through and SpecialEffect to 1-Plain. In the GotFocus() method of the control, I set BackStyle to 1-Opaque and SpecialEffect to 0-3D, and in LostFocus(), I set them back to the original values.

To see this class in use, run the SAMPLE2 form included in download file 03HENNIG.EXE. As with SAMPLE1, this form includes the VFP sample CUSTOMER table in its DataEnvironment, so you may need to modify the form. Only the field with focus has a white background and appears in 3-D. It's very obvious where the cursor is located now.

Conclusion

I love the object-oriented nature of Visual FoxPro. It allows us to subclass common controls like TextBoxes and ComboBoxes and add new functionality to them, then forget about how the new features were implemented. I hope you'll find the classes described in this article as useful as I have.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. 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 user groups and regional conferences all over North America. 75156.2326@compuserve.com. ▲
The FoxPro Report Writer is quite suitable for most reports; however, in this article Paul describes a report that couldn't be handled by any standard report writer. With the help of an HP LaserJet 4 Plus with the duplex option, some new drivers, and some ingenuity, he developed a rather unique report.

One of my clients has used a manual system for ages, and I was called in to help them computerize their operations. We've all had to explain the limitations of a new system to a client -- you know the ones: they have to fill in all the required fields, they have to spell things correctly, they have to fill in invoice numbers, and so on. This report presented a challenge to provide all the flexibility the client was used to, but with a new automated system.

The client, a printing shop, uses a single piece of paper called a Work Ticket that travels around the plant with each job and records the 'story' of that job. The front side of the page lists the type of paper that is required, the ink, the presses that will be used, how the final product will be cut, and so forth. The back side of the page lists all of the job cost information, the invoice information, and so on.

In the old system, the information on both sides of the page was printed in portrait mode and filled in by hand. Because the new system was being designed from the ground up, the client had the opportunity to ensure that they got exactly what they required. One of the changes they wanted to make was to print the front page in landscape mode and the back in portrait mode.

Another challenge I faced with this project is that the printing shop was accustomed to generating and modifying their own forms as they saw fit. They wanted to retain the ability to modify the layout of both the front and back page; thus they wanted to be able to issue a MODIFY REPORT command on both the front and back reports.

Another factor was that the budget was quite tight, so there was no allowance for additional software.

Because the client wanted all this flexibility at the push of a button, they also required a printer that could print in full duplex. They selected an HP LaserJet 4 Plus.

**Step 1: Define the requirements**

Based on the preceding information, I came up with the following requirements:

- Print the processing information on the front of the page in landscape mode.
- Print the financial information, including job costs, on the back of the page in portrait mode.
- Generate the report on an HP LaserJet 4 Plus.
- Allow for multiple pages (there could be multiple front pages, but there would always be
Step 2: Develop a conceptual solution

I tried a number of ideas, including running a portrait-oriented report (the back page) inside a UDF in a landscape-oriented report (the front page). I finally settled on the idea that two reports would have to be generated -- one for the front pages and one for the back page -- and then merged.

To generate the report I used the following steps: First, I copied the required information from the .DBF files into three cursors -- one for the header and financial information (one record), one for the processing information on the front of the page (multiple records), and one for the job cost information on the back of the page (multiple records). Next, I generated two report forms -- one for the front of the page in landscape mode, and one for the back of the page in portrait mode. Then I ran the report forms to create separate files, and merged them into a third file. I sent this third file to the printer.

Step 3: Implement the solution

Generating the report forms (.FRXs) and the paper forms was the easiest part. (The client did this <g>). All I had to do was finish the job.

Step 3a: Become intimate with the available tools

To make matters a whole lot easier, I decided that the client should have a single printer driver, and that I should write the code to work with that driver. Additionally, because I already had a good understanding of HP's Page Control Language (PCL), I decided that the driver should support that language instead of using PostScript.

The next step was to become intimately familiar with how the LaserJet driver worked, first by printing a number of reports to disk files and examining the output, and then examining the WIN.INI file (yes, this was Windows 3.11) when changes were made in the Control Panel. It became obvious very quickly that the driver that came with the LaserJet was inconsistent in its behavior, and that a new or updated driver was required. Apparently, numerous bugs have been reported with both the hardware and the drivers for the LaserJet 4 line, so a driver problem wasn't a huge surprise. I retrieved the latest driver from HP's Web site, and I was relieved to find that it was both stable and consistent.

It turns out that making changes to the .INI file is extremely easy. The Orient= line in the HP LaserJet 4 Plus/4M Plus is "1" for a portrait page and "2" for a landscape page, and it's a relatively simple matter to make the modifications via FOXTOOLS.FLL functions RegFn() and CallFn().

It also turned out that the .PCL files were very straightforward with the new driver. There was a
standard chr(12) -- a standard eject character -- at the end of each page, and nowhere else in the report (this wasn't true for other drivers I tried), and all of the page formatting information was positioned at the beginning of each page.

**Step 3b: Generate separate reports**

The pseudocode for this step is fairly straightforward:

- **Generate the cursors**
- **Set the orientation to Landscape**
- **Generate the front page(s) to a file (FILE1.TXT)**
- **Set the orientation to Portrait**
- **Generate the back page(s) to a file (FILE2.TXT)**

**Step 3c: Merge the reports**

Merging the reports was the hard part. The pseudocode for this was simple; it just took some time to implement it:

1. Read FILE1.TXT into a character string (m.lcReport1).
2. Read FILE2.TXT into a character string (m.lcReport2).
3. Remove the Start and End of the second report.
4. Remove Simplex/Duplex and paper tray commands, and remove everything past the final eject.
5. Add the Duplex command to the orientation.
6. Convert all Ejects to Double-Ejects.
7. Write out the information.

The sidebar accompanying this article, "Be Intimate with Your Driver!," outlines some of the characteristics of the PCL code that was generated. Without the consistent behavior of the updated driver, this exercise would have been significantly more difficult.

The first thing I had to do was set up the PCL codes that would be used later. I used memory variables for this, rather than #DEFINEs, out of habit. I also ignored the variable naming convention here, so that the variable names were easier to read and remember (they'd all be prefixed with "m.lc" anyway, so why bother?):
m.esc = chr(27)  && Escape
m.amp = chr(38)  && Ampersand
m.eject = chr(12)  && Page Eject
m.reset = m.esc+"E"  && Issue a Soft-Reset to the LaserJet
m.portrait = m.esc+m.amp+"l0O"
m.landscape = m.esc+m.amp+"l10"
m.simplex = m.esc+"10S"
m.duplex = m.esc+"11S"
* Notice how the landscape
* and duplex commands are combined
m.landduplex = m.esc+m.amp+"1101S"
m.upper = m.esc+m.amp+"11H"  && These are the paper trays
m.optional = m.esc+m.amp+"14H"
m.lower = m.esc+m.amp+"15H"
* This sets the resolution to 300 dots per inch
m.SetRes = m.esc+"*t300R"

The item numbers in the following sections correspond to the pseudocode listing in Step 3c:
Merge the reports. Items 1 and 2 were fairly simple.

**Items 1 and 2**

create cursor temp (text m)
append blank
append memo text from file1.txt overwrite
m.lcReport1 = temp.text
append memo text from file2.txt overwrite
m.lcReport2 = temp.text

**Item 3**
The first report, the front page, contains all the header and footer information that's required for the entire report; this information doesn't have to be duplicated for the back page. The only thing to be concerned with on the back page is that it's in portrait mode:
* Remove the beginning of the back page.
* Remove everything before the resolution is set.
m.lcReport2 = substr(m.lcReport2,at(m.SetRes,m.lcReport2))
* Remove everything up to the next <Escape> character.
m.lcReport2 = substr(m.lcReport2,at(m.esc,m.lcReport2,2))

* Remove the end of the back page, and prefix it
* with a page orientation.
if right(m.lcReport2,3) = m.eject+m.reset
  m.lcReport2 = left(m.lcReport2,len(m.lcReport2)-3)
endif
m.lcReport2 = m.portrait+m.lcReport2

Item 4

The Simplex and Duplex commands indicate whether the printing happens on one or both sides of the page. This command was always isolated with the driver I was using, and thus removing it was easy:

m.lcReport1 = strtran(m.lcReport1,m.simplex,"")
m.lcReport1 = strtran(m.lcReport1,m.duplex,"")

The paper tray might not have been set correctly for the application, so I had to change all references to the paper tray (in this case, the upper tray):

m.lcReport1 = strtran(m.lcReport1,m.lower,m.upper)
m.lcReport1 = strtran(m.lcReport1,m.optional,m.upper)

I truncated the text past the final eject so I could be sure when the last page had been processed:

m.lcReport1 = left(m.lcReport1,rat(m.eject,m.lcReport1)-1)
m.lcReport2 = left(m.lcReport2,rat(m.eject,m.lcReport2)-1)

Item 5

I had to add the Duplex operation to the orientation on the first page. From examining the driver results, I knew what the orientation looked like, so adding the Duplex command was easy:

m.lcReport1 = strtran(m.lcReport1,m.landscape,m.landduplex)

Item 6
Now that the entire report would print in Duplex mode, page 2 would print on the back of page 1; page 4 would print on the back of page 3; and so on. To prevent this behavior, I changed all single page breaks into double page breaks (the page break carried the "set resolution" command with it, so I doubled that as well):

\[
m.lcReport1 = \text{strtran}(m.lcReport1, ;
m.eject+m.SetRes, ;
m.eject+ m.SetRes m.eject+ m.SetRes")
\]

**Item 7**

After all of the preliminary work was complete, the report was written to a file (for the purposes of this article, I've assumed that the file was previously created with =FCREATE() and will be closed with =FCLOSE() ). I wanted to allow for more than a single front page. If there was only a single front page, then there would be no page break (I would remove it); however, if there was more than one front page, I would add a double page break between each of the pages. If there was only a single front page, then I'd add the back page to the end of it. If there were multiple front pages, then the back page would only be added after the first front page (between the first set of double page breaks):

\[
\text{if } m.eject+ m.SetRes \neq m.lcReport1
\]

* There are multiple front pages:
  * Print the first one, ;
  * Then print the back page, ;
  * Then print the rest of the front pages.
  
  \[
  =\text{fwrite}(m.lpFHandle, \text{left}(m.lcReport1, \text{at}(m.eject+ ;
  m.SetRes,m.lcReport1)-1))
  \]

  \[
  =\text{fwrite}(m.lpFHandle, m.lcReport2+m.landscape)
  \]

  \[
  =\text{fwrite}(m.lpFHandle, ;
  \text{substr}(m.lcReport1, ;
  \text{at}(m.eject+m.SetRes,m.lcReport1,2)+;
  \text{len}(m.eject+m.SetRes)) ;
  \]

else
  * There is only a single front page:
  * Print the Front page, ;
  * Then Print the Back Page
  
  \[
  =\text{fwrite}(m.lpFHandle, m.lcReport1)
  \]

  \[
  =\text{fwrite}(m.lpFHandle, m.lcReport2)
  \]

endif

\[
=\text{fwrite}(m.lpFHandle,m.eject+m.reset)
\]

release m.lcReport1, m.lcReport2

**Step 3d: Produce the final report**

This step was also very simple. I sent the previously created file to the printer using the
following DOS batch file:

```
copy /b %1 LPT2
erase %1
```

I used the "/b" parameter because the file may contain an EOF marker, chr(26). If so, then the file up to the chr(26) would be copied to the printer. The "/b" parameter indicates that the file should be copied in binary mode, and that all of the characters should be copied (this parameter seems to be relevant only when copying files to devices).

The FoxPro program calls this batch file like so:

```
RUN /0 PRINTFIL.BAT m.lcFileOut
```

The only persistent problem with this solution is that the memory variables take up a considerable amount of conventional memory, and this is where the .BAT file wants to run. Unfortunately, this is a limitation of this solution, and it can't be overcome in standard ways. One option to prevent it is to generate the result files, then have another process on the same machine (or on a different machine) monitor the directory to see if these result files exist and, if they do, copy them to the printer. But that's another article . . .

**Conclusion**

In a conversation with another consultant, I formulated a standard phrase when a customer asked for something a bit unusual: "Anything is possible with computers, given enough time and money." In the case of this report, I spent the bulk of my time figuring out the concept. Once I had that, the mechanics were fairly straightforward. This is because I already knew how to manipulate the contents of an .INI file, I already understood PCL (from more than 10 years ago), and I already had experience working with large and complex character strings in FoxPro.

Our customers will get the most bang for their buck if we're familiar with the tools that we're working with, because we'll be able to answer most "what-if" questions right away, instead of having to do some research first. With the new version of FoxPro on the streets, I encourage you to learn everything you can about it. Some developers are concerned that we spent a lot of time learning 2.x, and we now have to throw out all that knowledge and experience. This may be true, but I don't think so. A friend once asked me what was the easiest way to learn the newest version of FoxPro. My response was: "Know the old version, then just learn what's different."

*Paul Russell is a software specialist who works for Software Kinetics Ltd., a firm specializing in Internet applications, with offices in Dartmouth, Nova Scotia, and Ottawa, Ontario, Canada. He is also the president of the FoxPro user group in Nova Scotia. russell@atl.sofkin.ca, prussell@fox.nstn.ca, or 102440.2125@compuserve.com.*
Two types of information comprise a PCL file: commands and data. Generally, PCL commands begin with an ESC (chr(27)) character and one or two other characters that identify the classification of the command, followed by one or more numbers that represent the details of the command, and end with a single character that indicates the specific command within the classification. The data, on the other hand, is a simple text string that gets printed, usually a single word at a time.

So far so good.

This gets more complicated because a classification of a command can actually have multiple distinct commands concatenated, with only a single prefix. The letters at the end of each command that denote the specific command (before the last one) are in lowercase. The letter at the very end of this classification (that denotes the last command) is in uppercase. This means that, if you're looking for a command to set the page resolution, you can't simply perform a search for chr(27)+"*t300R" and be satisfied that you'll find something. Instead, you have to search for chr(27)+"*t", and then before the next chr(27), you have to find a number that's followed by either "r" or "R". That number could be 75, 150, 300, or 600. This leaves eight different possibilities. In this article, for example, the Landscape ((ESC)+(Ampersand)+"l1O") and Duplex ((ESC)+(Ampersand)+"l1S") commands are concatenated to create (ESC)+(Ampersand)+"l1o1S".

Because the commands can appear in different forms, you have to examine the results of your PCL driver very carefully. With some drivers, you can't be sure that they'll generate the same codes in different situations. This was the case with the original driver that came with the printer. The original driver didn't know about substituting fonts, so it sent a bitmap of each character that had to be printed. Frequently there was a chr(12) character within the bitmap code, and I was using this as a page delimiter. This made it much more difficult to split pages, so I obtained an updated driver that didn't have this problem.

With the driver that I was using and the configuration I specified, the page resolution command always looked like chr(27)+"*t300R". This made the programmatic interpretation of the PCL code very simple.

This does mean, however, that the customer has to be extremely cautious with the configuration of the machines. If the configuration of the machines, the printer ports, and the drivers is consistent, then the PCL driver will produce the same results every time. If any of these elements changes, the application will `break.'

Using OptionGroups in Visual FoxPro
The OptionGroup control in Visual FoxPro is sort of like that oil filter wrench hanging on the wall in the garage -- it's not something that you use on a regular basis, but when you really need it, there's just no alternative! Jeff points out some of the useful tricks you can perform with OptionGroups and discusses a few approaches to making sure your OptionGroup classes use your own sub-classed OptionButtons.

One of the most important aspects of developing software for a graphical user interface (GUI) is staying within the generally accepted guidelines for that interface. This means knowing exactly which control to use for a given interface need. As developers, we're also end users of the development products and tools we use, so think back on a piece of software you used recently that had a truly horrible interface. Chances are that the interface was horrible because the developers of that product chose the wrong controls for the context of the information you were working with. The result is an experience that feels awkward or unwieldy, likely leading to a dissatisfied customer.

OptionGroups are normally used in a portion of an interface where the user may choose from a fixed, relatively small, set of choices. If the set of choices is large (more than a half-dozen or so) or is driven dynamically by the data, then a Listbox control is usually a better choice. When there are only two options to choose from, and those options are the opposite of one another (such as "on" and "off"), then a Checkbox control should be used. I like to think of OptionGroups as meeting the need for all the cases that fall into the "gray area" between Checkbox and Listbox.

The anatomy of an OptionGroup

The OptionGroup control itself is merely a container for two or more OptionButton controls. In versions of FoxPro prior to 3.0, these were called "radio buttons" because their behavior is like that of a car radio. This description still fits quite well because, regardless of how many options you have in the group, you can select only one at a time, and the act of selecting any one will automatically de-select all the others (see Figure 1).

Like other containers in Visual FoxPro, the OptionGroup has a collection property that provides access to its member OptionButton controls. In this case, the Buttons member array contains one element for each OptionButton object found in the OptionGroup. The OptionGroup also has a ButtonCount property, which contains the current number of member OptionButton objects in the OptionGroup container. Using these two properties and the new FOR EACH construct in Visual FoxPro 5.0, you can write code in a specialized OptionGroup subclass that performs some action on each OptionButton member object:
LOCAL loButton
FOR EACH loButton IN THIS.Buttons
   *-- loButton is an object reference to the
   *-- current OptionButton in the collection.
ENDFOR

One nice enhancement that came when FoxPro 2.x "radio buttons" became Visual FoxPro OptionGroups is the ability to space the buttons using any sort of alignment. The buttons are aligned in a vertical column by default, but you can arrange them individually into horizontal rows, multiple columns, or anywhere else within the confines of the OptionGroup container.

The individual OptionButtons also contain a Style property that indicates the type of button to be displayed. As with check box controls, the choices are Standard (the default) and Graphical. The behavior of the OptionButtons and their parent OptionGroup container isn't affected by the Style property. It simply indicates the type of visual representation the OptionButton should use. If this property is set to 1 (Graphical), the OptionButton will look like a command button; if it's the selected OptionButton within the OptionGroup, it looks like a command button in a "pressed down" state.

**Using the OptionGroup with data**

To bind most controls to a data source, you'd set the ControlSource property for the control to the data item it refers to. If you populate the ControlSource of an OptionGroup, however, its Value property will contain the Caption for the selected member OptionButton. This probably isn't the behavior you'll normally want, because a change to any button's caption will introduce an inconsistency into the data. If you leave empty the ControlSource property for the OptionGroup, the Value property for that object will contain the index for the member OptionButton that's currently selected. For example, the OptionGroup in would have a value of 1 because the first object in its Buttons collection is currently selected.

Oddly enough, the class definition for individual OptionButtons contains a ControlSource property as well. You probably won't ever bother with it, simply because it doesn't make a great deal of sense to have a group of OptionButtons in the same OptionGroup bound to different data sources.

In my opinion, the cleanest approach to using an OptionGroup with data would be to not use a ControlSource on the OptionGroup but instead to store the numeric value into the data. That way, if you change the captions of the individual OptionButtons you won't affect the data. For reporting purposes, you'd refer to a lookup table to indicate which of the possible choices each numeric value corresponds to. As an alternative, you might use the numeric value from the OptionGroup to search the lookup table containing the actual values and simply store that value into the main table. Either way, a slightly more complex design makes maintenance significantly easier when the client later decides to change the interface or add new options to a group.

And where should you put the code that saves this numeric value or performs the lookup? That depends on when your data is being saved. If your form has a method that handles the saving of
new or changed records, then that would be the logical place to poll the OptionGroup for its Value. If the OptionGroup is part of a business object that's dropped onto a form, then the business object should be responsible for gathering the OptionGroup's settings.

The relationship between the values of the individual buttons and the group's value is fairly straightforward. Each OptionButton within the group has its own Value property. If a given OptionButton is the one currently selected within the group, it will have a value of either 1 or .T. (depending on whether the value was set to a numeric or logical value at design time). With that one selected, all the other buttons in the group will contain either 0 or .F.

When a user attempts to add a new record, you might set the OptionGroup's Value property to the numeric value that corresponds to the default choice within the group. This way, if the user doesn't change that selection, you've still got a valid default value in the data. There may be cases, however, when you don't want to have any of the buttons in the group initially selected. The user will then know to select something from the list of possible options. To do this, set the OptionGroup's Value to 0 initially and none of the member buttons in the group will be selected. If a choice is required, though, make sure to check that the value is no longer 0 before saving the user's new record.

**What's new in VFP 5.0**

Before you get too excited, there weren't a lot of changes to the OptionGroup or OptionButton base classes in Visual FoxPro 5.0. But, given all the other enhancements to the product, I think we can probably look the other way this time. As Table 1 illustrates, the new properties, events, and methods (PEMs) added to both these classes are primarily the same ones added to most other Visual FoxPro base classes.

**Table 1. New PEMs added to the OptionGroup and OptionButton base classes.**

<table>
<thead>
<tr>
<th>OptionGroup</th>
<th>OptionButton</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColorSource property</td>
<td>MiddleClick event</td>
</tr>
<tr>
<td>Destroy event</td>
<td>MouseIcon property</td>
</tr>
<tr>
<td>MiddleClick event</td>
<td>MouseWheel event</td>
</tr>
<tr>
<td>MouseIcon property</td>
<td>Parent property</td>
</tr>
<tr>
<td>MouseWheel event</td>
<td>ReadExpression method</td>
</tr>
<tr>
<td>Parent property</td>
<td>ReadMethod method</td>
</tr>
<tr>
<td>ReadExpression method</td>
<td>RightToLeft property</td>
</tr>
<tr>
<td>ReadMethod method</td>
<td>ShowWhatsThis method</td>
</tr>
<tr>
<td>ShowWhatsThis method</td>
<td>UIEnable event</td>
</tr>
<tr>
<td>WhatsThisHelpID property</td>
<td>WhatsThisHelpID property</td>
</tr>
<tr>
<td>WriteExpression method</td>
<td>WriteExpression method</td>
</tr>
</tbody>
</table>

Two changes with regard to the OptionButton and OptionGroup classes are worthy of note. First, do you notice anything odd about the OptionGroup list in Table 1? Yes, the good folks at
Microsoft fixed that bug found in VFP 3.0 and 3.0b wherein the OptionGroup class definition never had a Destroy( ) event! Now, if it's appropriate to the interface, you can create a generic OptionGroup class that saves the user's selected option into the registry and then restores it in the Init( ) event.

The second change is a minor enhancement in your ability to create OptionButton class definitions. Prior to VFP 5.0, you could create OptionButtons visually using the Class Designer. Unfortunately, this ability wasn't obvious because the dialog resulting from the CREATE CLASS command didn't contain the OptionButton as a base class. Instead, you had to issue something like the following:

```
CREATE CLASS opbMyOptionButton OF
  ClassLib AS OptionButton
```

Thankfully, the CREATE CLASS dialog has been modified in VFP 5.0 to include the OptionButton in the combo box that lists the VFP base classes.

**And speaking of subclasses . . .**

I wish I could report that VFP 5.0 has made it easier to use specialized OptionButton classes as members of your OptionGroup class definitions. Sadly, that's still not the case in version 5.0. If you create an OptionButton class and add three instances of it to an OptionGroup container in the Class Designer, VFP won't save those OptionButtons in the class definition when you save the container. VFP doesn't save a record into the Visual Class Library (VCX) table for each member OptionButton, as with most container-member class definitions. Instead, it writes "ButtonCount = 3" into the Properties memo field for the OptionGroup container class' record. The next time you open the OptionGroup class in the Class Designer, VFP evaluates the class definition and adds three instances of the VFP OptionButton base class to your subclassed OptionGroup container.

Obviously, this can wreak all sorts of havoc for developers who strive for the highest level of reusability (and who among us doesn't?). The idea of creating a custom OptionGroup builder that lets you use specialized OptionButton classes is appealing, but nobody wants to write a builder that VFP un-builds!

There's also the problem of the ButtonCount property in subclassed OptionGroups. When a specialized OptionGroup class is used in a form or other container, the number of buttons in the group can't be decreased because those members reside in the original class definition. You can always add member buttons to each instance of the OptionGroup, but then those buttons won't have any of the unique functionality possessed by the original buttons from the class definition.

There are, however, a few different ways that you can solve both these problems at once. You can use your specialized OptionButton classes in an OptionGroup class and make sure that each button in the group has the necessary functionality. The trick is to place the buttons you want into the container at runtime instead of design time so that VFP doesn't undo your changes. The following list describes three possible approaches that you might consider. (You'll find an
example of each in download file 03DONICI.EXE.)

- You can add a custom property, cButtonClass, to your OptionGroup class definition and populate it with the name of the OptionButton class you want the group to use. Then add a custom method named ReplaceButtons( ) to the class and call it from the OptionGroup's Init( ). Add code to this new method that goes through the group's Buttons collection and, for each member button found, performs the following tasks: Saves the caption and position-related properties for that button, removes it from the OptionGroup container, replaces it with a button of the class specified in cButtonClass, and resets the saved properties from the original button. Then you can simply set up each OptionGroup instance using the default OptionButton base class. At runtime, the ReplaceButtons( ) method will replace the default buttons with your specialized button class.

- Another approach is to create an OptionGroup class with no option buttons in it and add an AddButtons( ) method to the class definition. Place code in this method to receive two parameters: the caption for the added button and the class name for the button. This method will also need to find the last OptionButton in the Buttons collection and add the new button below it. Next, add code to the Init( ) for each OptionGroup instance that calls the AddButtons( ) method to add the necessary member buttons to the container.

- The last approach is the one that I think developers will find most appealing. It's similar to the previous approach but uses an array to maintain the list of buttons to create during instantiation. This class definition contains no option buttons. Instead, an array property, aButtons, is added to the class along with two methods -- FillButtonsArray( ) and AddButtons( ). To use the class, simply write code in the FillButtonsArray( ) method that populates the aButtons array property with one row for each button to be added. The array would contain two columns: the first for the button caption and the second for the OptionButton class definition to be used. The AddButtons( ) method will loop through this array and add each button in turn. In the Init( ) of the class definition, the FillButtonsArray( ) method would be called just before the AddButtons( ) method, allowing the OptionGroup to build itself.

Hopefully you'll be able to use one of these approaches, or some combination of them, in your own development to get the most flexible OptionGroup classes possible. Remember, there are times when no other control will do!

DOWNLOAD

Jefferey A. Donnici is an applications developer specializing in common code and developers' tools, at Resource Data International Inc. in Boulder, Colorado. Jeff is a Microsoft Certified Professional in Visual FoxPro and a Microsoft Developer Most Valuable Professional. 303-444-7788, fax 303-444-1286, 75460.437@compuserve.com or jdonnici@resdata.com.

▲

Ease Your VFP 5.0 Development with New
Properties

John V. Petersen

Object Referencing Techniques

What is the best way to reference objects from within other objects in method code?

There are two basic techniques to reference objects. I'll refer to them as the "top-to-bottom" and "bottom-to-top" approaches, respectively. In the top-to-bottom approach, the ThisForm object token is usually invoked. For example, consider the following code in a CommandButton's Click() Method:

```vba
thisform.text1.value = "Just pressed 
commandbutton " + this.name
```

What is the assumption? First, it assumes the TextBox is contained directly on the form. This may or may not always be the case. What happens two months down the road when a design change occurs and controls formerly on the form are now placed on a page in a PageFrame? When attempting to run the form, you'll be greeted with all kinds of error messages such as "Unknown Member ". Why? Because the text box's parent is no longer the form. Rather, it's a page on the newly added PageFrame. You'll have to endure the task of modifying every instance of code that refers to the TextBox.

While groups of controls are subject to moving around in a form during design, I've found that their relative positioning to each other usually remains constant. This brings us to the second approach, bottom-to-top. In this approach, the Parent object token is used. Referring back to the Click() Method example, using the Parent object token yields the following:

```vba
this.parent.text1.value = "Just pressed 
commandbutton " + this.name
```

While this produces exactly the same results, it provides a big boost in flexibility. Using this technique, you can move the CommandButton and TextBox anywhere; as long as their relative positions to each other in the containership hierarchy remain unchanged, the code will always work. With this technique, you won't have to make any code modifications. Obviously, if the relative positioning does change, code modifications will be inevitable.

DynamicInputMask property -- applies to Column
This is a useful property that should make it easier to create flexible grids. Just like the other dynamic properties (DynamicCurrentControl, DynamicForeColor, DynamicBackColor, and so on), the property value can change based on the value of an expression. Without these dynamic properties, developers would have to resort to writing lengthy code in the AfterRowColChange() event to change properties manually.

To illustrate how to use this new property, let's consider a table of customers. A complicating twist to the data is that customers can be both in the United States and Canada. One notable difference between the addresses of both countries lies in the difference between U.S. ZIP codes and Canadian postal codes. Typically, a U.S. ZIP code has the following input mask:

99999-9999

A Canadian postal code has the following input mask:

A9A 9A9

Assuming a field exists called "country," the DynamicInputMask Property of the Column Class would contain the following expression:

IIF(country = "USA", "@R 99999-9999", "@R! A9A 9A9")

Notice that the formatting codes are also included to enforce details such as uppercase characters. Figure 1 shows how the grid looks with the new DynamicInputMask Property in place.

**IntegralHeight property -- applies to ListBox, EditBox, and TextBox**

Fonts have always been a challenge in FoxPro, especially when working with controls such as EditBoxes and ListBoxes. It's a big chore to size the control so that all visible items or lines can be viewed in their entirety. Of course, once you size the control and need to change a font, you have to go through your sizing routine again. Unless you create a builder to do the sizing, you have to do a lot of manual work to get the job done. Enter the new IntegralHeight property.

The sole purpose of this new property is to ensure that the last item in a ListBox or EditBox is completely displayed. For a TextBox, one line of text will be completely displayed (see Figure 2). The form on the left has an EditBox and ListBox with their last rows only partially visible. This doesn't make for a very appealing interface because the user has to go through an extra step in order to view the last item. The form on the right has controls with the IntegralHeight Property set to .T. The interface is much cleaner.

No matter how you alter the font properties (name, size, and so forth), the controls will
automatically resize to accommodate the last item. Interestingly enough, the default value in the VFP BaseClasses for the IntegralHeight Property is .F. I can't think of a reason why it wouldn't be set to .T. This scenario represents another reason why it's a good idea to subclass each of the VFP baseclasses.

**AutoVerbMenu property -- applies to OLE Container and OLE Bound**

If you provide the ability to open and edit OLE Objects inside VFP forms, the AutoVerbMenu property was made for you! To illustrate, consider an Excel Chart Object in an OLE Container Control. In version 3.0, you had to manually invoke the DoVerb() Method to perform some action on the OLE Object. Two of the most common verbs associated with OLE Objects are Edit and Open. For OLE Objects such as movie clips (.AVI files) and sound clips (.WAV files), Play is a common verb as well.

Depending on the setting of the AutoActivate Property, it was easy to access the default verb. By default, double-clicking on the OLE Object would invoke the default verb. In most cases, the Edit verb is the default. For movie and sound clips, Play is the default verb. What happens when you want to give your users access to all verbs? In version 3.0, you had to remember the names of the different verbs and create a CommandButton with the following code in the Click() Method:

```vfp
This.Parent.OleControl1.DoVerb("Open")
```

Perhaps there isn't enough room to house the needed controls. Or, you may not want to give this capability to all users. The new AutoVerbMenu Property is a big help.

By setting the AutoVerbProperty to .T., when the user clicks the right mouse button over the OLE Object, a pop-up menu appears with an item for each verb (see Figure 3).

---

**Tip: Year 2000 Follow-Up**

*I've just come across David Diehl's article entitled "A Turn of the Century Solution" in the May 1996 issue of FoxTalk. His EPOCH() function offers a simple and elegant solution to the challenge of handling six-digit date entries for the year 2000 and beyond.*
However, there's a problem with using "<date> + 36,525" to generate the date 100 years hence. The problem stems from the fact that centuries aren't leap years, although millennia are leap years. Therefore, if the "epoch" spans a February that falls in a century that isn't a millennium, such as 1900, there are only 36,524 days in that 100-year period. By always using 36,525, the EPOCH() function goes one day too far for dates entered as 01/01/00 through 02/28/00. FoxPro initially interprets these dates as 01/01/1900 through 02/28/1900, and EPOCH() then returns 01/02/2000 through 02/29/2000, respectively.

Fortunately, there's an easy fix for this, which is to use FoxPro's GOMONTH() function inside of EPOCH(). Replacing "RETURN dUserEntry + 36,525" with "RETURN GOMONTH(dUserEntry, 1200)" tells FoxPro to add 1,200 months to the date and FoxPro handles the "centuries-that-are-not-leap-years" condition for you.

The only remaining problem is that the year 2000, as a millennium and therefore a leap year, has a Feb. 29, while the year 1900, as a century but not a millennium, does not. Try entering "02/29/00" in a date field: FoxPro initially interprets this as 02/29/1900 and traps it as an invalid date without giving EPOCH() -- or anything else -- a chance to change it to the intended date of 02/29/2000.

David's EPOCH() is still the best solution to the Y2K problem that I've seen so far. But I guess we'll still have to train our users to enter a four-digit year for that one lone exception, Feb. 29, 2000.

Rick Borup is with Information Technology Associates (ITA) in Champaign, Illinois. 217-359-0918, rborup@prairienet.org, 72632.1274@compuserve.com.

Views

Compiled by Barbara Peisch

Using 2.x Files in VFP Views

If you've been toying with the idea of including a 2.x file in a view, but didn't want to mess up any of your existing 2.x files to try it out, go ahead. But before you do, heed this sage advice from John Petersen.

If you elect to utilize a remote view in Visual FoxPro to access and update data in a 2.x table, take care to ensure that the key you select in the view designer is unique. For example, assume the 2.x table has a column called ID, C(5) and you select this as the key column in the view designer. Further, assume that two records have an ID value of '00001'. If you open the view via a USE command and delete one of these records and subsequently call TABLEUPDATE(), you may be surprised by the results. Inspecting the original 2.x table will reveal that both records were deleted. Why? First, because this is what you told SQL to do. In effect, you said, "Delete every record in the back-end table with '00001' in the key column." Because two records have this value, two records were deleted. At first you may say to yourself, "But they're two distinct
records with different record numbers. FoxPro should have known better!" Remember, though, that when you use an updateable view you aren't using FoxPro to update the back end -- you're using SQL, and SQL knows nothing about a record number. The SQL translation most likely looks like a FoxPro command such as DELETE FOR ID = '00001'.

Obviously, this whole issue would be eliminated if the FoxPro 2.x tables were accessed directly because VFP will do that quite nicely. However, if you need to have 2.x access as part of a database, a view is your only alternative because the 2.x table can't be a member of the database. If you were to include it, it would be inaccessible by FoxPro 2.x because the header would be converted to the 3.0 format. In my opinion, this scenario does a nice job of illustrating the differences in accessing and updating data via SQL vs. the methods employed in traditional FoxPro/xBase.

[I'd also like to add that John's caution holds true regardless of what kind of table the back end is. But unlike VFP, 2.x didn't have the ability to guarantee uniqueness in a tag so it's more likely to be a problem with a 2.x back end. -- B.P.]

Using a Compound Index on a View

Say you're doing a client/server app using views. You could really use a compound index on one of those views but you're not sure if it will work. Well, Benjamin P. Burnham has done the research for you.

If you need a compound structural index on a view, go ahead and build one! Even though the table is temporary and its actual TMP disk name changes with every REQUERY(), the index will remain attached, stay in sync, and delete itself when the view is closed. Just open the view, and INDEX ON <MyViewField> TAG <MyViewTag>.

What more could one want? ▲

Expanding Horizons

Les Pinter

Last November, when I was speaking at the Spanish FoxPro DevCon in Madrid, a young Portuguese friend of mine asked my advice about the future. My answer was so smart that my wife said I should follow it myself. So here it is.

We're all wondering about our future. Heaven knows that things are changing fast. I now publish a FoxPro newsletter () on the Internet, something I had scarcely heard of a year ago. I code using a notation that took me months to master, and which bears little resemblance to the language that I used during the previous 10 years. The paradigm shift that VFP represented has most of us scrambling to learn as quickly as possible what's important and what's not. ActiveX technology has mushroomed and is certain to be at the heart of all future development platforms. And the Internet has everyone hypnotized.

We've been promised two more generations of FoxPro before the name changes to Developer's
Studio, but the first merging of the two may be only months away. Will we be programming in FoxPro in two years? I can't honestly say that I program in FoxPro now. But I can tell you that now I can write applications in a day that used to take me a week.

I'm learning other languages, but not for lack of affection for FoxPro. If you read the Sunday classifieds, you'll see 50 ads for C++, 20 for VB, 10 each for Oracle, Delphi, Powerbuilder, and Access, and three for Visual FoxPro. I'm not in the FoxPro business; I'm in the database business, and each of these products (except maybe Access) has something to offer.

But there's a bigger picture. Each of these database environments is sandwiched between two other very important technologies, and is independent of both. I'm referring to structuring your database before you start to code, and extracting data after it's been entered. Each of the database development products takes a stab at these two critical areas, but each falls short to one degree or another. And good design ideas, which can be applied to any product, are in short supply. Third-party products take up the slack.

Take a look at Xcase from Resolution Technologies (). This is a powerful database design tool that's completely visual, exports not only to FoxPro but to various platforms, and allows you to diagram multiple pictures of the database as you model it. I often get hired to salvage database projects that have stumbled. Most of the problems stem from incorrect database design. Using a tool like Xcase or ERWin, you can't screw it up. Microsoft bought the predecessor to Word from me 16 years ago for practically nothing; Xcase would fit like a glove with the next version of FoxPro, so maybe they'll snap up this bargain, too. Download the free Xcase demo and test-drive it, and you won't want to start a project without it.

Report writers, can we talk? Crystal Reports runs circles around anything else on the market. And the lack of an HTML format option is getting to be hard to explain. Also, the lack of a really strong report generator has made tired and hungry men out of David Lewis (QueryMaker, 74660.1422@compuserve.com) and Alan Schwartz (Foxfire! from MicroMega, 73354.1745@compuserve.com), while demonstrating that there is a need for such tools. In fact, if you simply use Xcase to design local views and then use views to design your FoxPro reports, you'll find the report design process is much, much easier!

Finally, Neon Software (FoxExpress) and Lawson-Smith (ProMatrix) both sell application development environments that produce real-world examples of how to write commercial-quality applications. The examples that come with FoxPro do a good job of demonstrating how controls and other features work, but you can't ask users of a parent-child form whether they want to enter a new child key or not, because they have no clue as to what a child key is.

So we have several directions in which to expand: We can finally get good at building and normalizing databases, regardless of the language to be used; we can master SQL, which is the way to simplify reports; and we can master report design and become familiar with third-party reporting tools. Then, if customers want to use another language, we can say "that'll work" -- unless, of course, they want to use Access. But I still recommend FoxPro.

Les Pinter publishes the Pinter FoxPro Letter in the United States and Russia and on the Internet at . His
first novel, The Valley, will be out later this year.