Extending Visual FoxPro with Visual FoxPro

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Web site: http://www.stonefield.com
Web site: http://www.stonefieldquery.com
Blog: http://doughennig.blogspot.com

Overview

One of the coolest things about the Visual FoxPro interactive development environment (IDE) is that it's highly extensible. Even better, you can use VFP code to extend VFP itself. In this document, you'll learn how to extend the VFP IDE to boost your productivity and create simple tools that make your life easier.

Topics I'll discuss include:

- How to create your own property editors.
- Creating powerful IntelliSense scripts to boost your productivity.
- How the new New Property/Method dialog speeds form and class development.
- Automatically updating a LOCAL statement.
- An overview of how the My namespace in Sedna (the code name for the next version of VFP) works.

Introduction

FoxPro has long provided ways to hook into various aspects of the interactive development environment (IDE), and it gets better with each version. FoxPro 2.x allowed us to replace the functionality of certain Xbase components by changing the name of the applications pointed to by system variables, such as _GENSCRN and _GENMENU, and customize the system menus. VFP 3 gave us builders, which allow us to automate or simplify working on classes and forms. VFP 6 added project hooks; they allow us to receive events when the user does things in the Project Manager, such as adding or removing a file. IntelliSense was one of the major new features in VFP 7, and the data-driven approach the VFP team used means we can customize how it works.

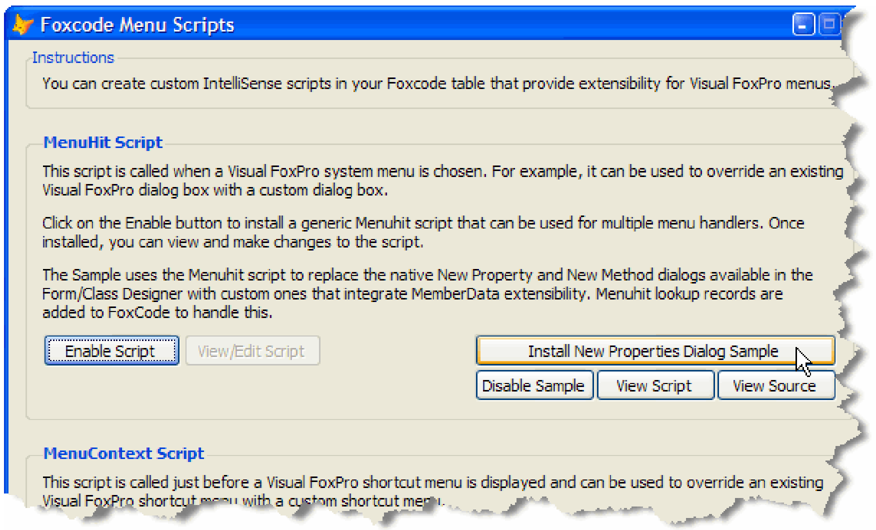
In VFP 9, Microsoft has blown the lid off the IDE. We can completely change the appearance and behavior of the dialogs used by the Report Designer because the Report Designer raises report events that we can react to. The new ReportListener base class allows us to receive events as a report is run, providing features such as label rotation, dynamic formatting, and custom rendered objects such as graphs. We can react to Windows events, giving us nearly complete control over both the VFP IDE and run-time environment.

In this document, I'll focus on several things I've added to my IDE to boost my development productivity. We'll look at a replacement for the New Property and New Method dialogs, a framework for creating property editors, fully utilizing Member Data as meta data for objects, and creating time-saving IntelliSense scripts.

Replace the New Property and New Method dialogs

One thing I've always wanted to do was replace the New Property and New Method dialogs. Since thanks to Member Data, which I'll discuss in more detail later, we now have the ability to display custom properties and methods in any case we desire, I'd rather have VFP use the case I enter into the New Property or New Method dialog than have to bring up the MemberData Editor and change the case for a new member. Also, it always annoys me that I have to click on the Add button to add a new property or method and then click on the Close button to close the dialog. Since I often just add one property or method at a time, I'd like to see a button that both adds the member and closes the dialog.

Such a replacement actually comes with VFP 9, although it's somewhat buried. To get at it, select the Solution Samples pane of the Task Pane Manager, expand the New in Visual FoxPro 9.0 heading, and choose the FoxCode Menu Scripts sample. In the dialog that appears (see **Figure 1**), click the Enable Script and Install New Properties Dialog Sample buttons to enable the replacement dialog.



You can create custom IntelliSense scripts in your Foxcode table that provide extensibility for Visual FoxPro menus.

This script is called when a Visual FoxPro system menu is chosen. For example, it can be used to override an existing Visual FoxPro dialog box with a custom dialog box.

Click on the Enable button to install a generic Menuhit script that can be used for multiple menu handlers. Once installed, you can view and make changes to the script.

The Sample uses the Menuhit script to replace the native New Property and New Method dialogs available in the Form/Class Designer with custom ones that integrate MemberData extensibility. Menuhit lookup records are added to FoxCode to handle this.

This script is called just before a Visual FoxPro shortcut menu is displayed and can be used to override an existing Visual FoxPro shortcut menu with a custom shortcut menu.

Figure 1. Click the Enable Script and Install New Properties Dialog Sample buttons to access the replacement New Property/Method dialog.

Don't think this is some lame sample; it's so useful, I haven't used the New Property or New Method dialogs than come with VFP in more than a year. Selecting either New Property or New Method from the Form or Class menus brings up the same dialog, shown in **Figure 2**. This dialog has the following features:

- It automatically updates the _MemberData property (adding that property if necessary) so the case entered for the new member is used (even for access and assign methods if they're created as well) and the member is displayed on the Favorites tab if that option is turned on in the dialog.
- It's non-modal. That means you can keep it open, add some properties or methods, switch to other things, come back, and add some more members.
- How many times have you accidentally selected the wrong dialog, entered the name and description, only to realize your mistake and have to close the dialog and start all over? With the replacement dialog, you can switch between Method and Property without closing the dialog.
- It's dockable: try tab-docking it with Properties window. This is very cool!
- It's resizable and persists its size and position to your resource (FOXUSER) file.
- It has an Add & Close button to perform both tasks with one click.
- The default value for a property is automatically set to a value based on the data type of the property if you use Hungarian names. For example, lTest would be logical, so the default value is .F. For nTest, the default is 0.
- It hides rather than disables non-applicable controls. Since this one dialog is used for both New Property and New Method for both the Class and Form Designers, some options may not be applicable for a given instance.
- It disallows invalid names when you enter them rather than when you click on Add or Add & Close.
- The Add buttons are only enabled if a name is entered.

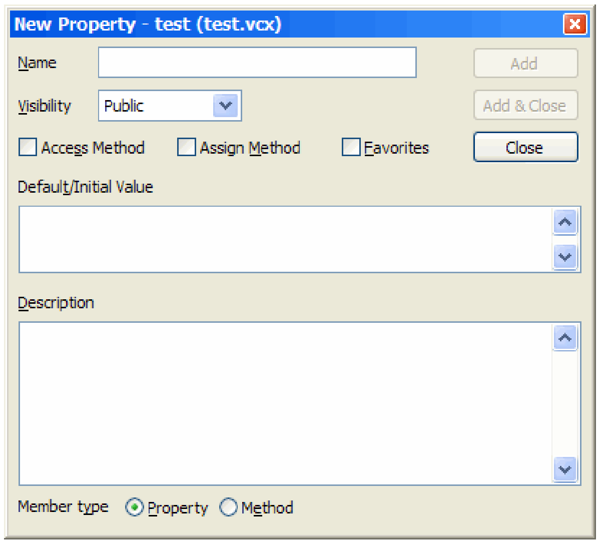


Figure 2. This replacement for the New Property and New Method dialogs has a lot more capabilities than the native dialogs.

I've found this replacement dialog to be a huge time-saver; I typically open it and leave it open for extended periods. I frequently copy and paste between this dialog and code windows, switch between Property and Method, and enjoy the fact that it automatically creates Member Data for the new property or method automatically.

How does VFP know to use this dialog rather than the native one? It's based on a new feature in VFP 9 that allows you to hook into system menu hits (that is, when the user selects an item from the VFP system menu bar).

MENUHIT

To trap a system menu hit, create a script record in the IntelliSense table, with TYPE = "S" (for "script"), ABBREV = "MENUHIT," and the code to execute in the DATA memo field. (The IntelliSense table is pointed to by the

_FOXCODE system variable; by default, it's FOXCODE.DBF in the HOME(7) directory.) The code should accept an object as a parameter. This object has several properties, but the important ones as far as MENUHIT is concerned are shown in **Table 1**.

Table 1. The properties of the parameter object passed to the MENUHIT script provide information about the menu hit.

Property	**Description**
UserTyped	The prompt of the pad the menu hit is from.
MenuItem	The prompt of the bar the user selected.
ValueType	A return value to VFP: blank means continue with the default behavior and "V" or "L" means prevent the default behavior (similar to using NODEFAULT in class code).

Here's a simple example (taken from SimpleMenuHit.PRG included in the source code with this document):

```
text to lcCode noshow
lparameters toParameter
wait window 'Pad: ' + toParameter.UserTyped + chr(13) + ;
  'Bar: ' + toParameter.MenuItem
endtext
delete from (_foxcode) where TYPE = 'S' and ;
  ABBREV = 'MENUHIT'
insert into (_foxcode) ;
    (TYPE, ABBREV, DATA) ;
  values ;
    ('S', 'MENUHIT', lcCode)
```

This simply displays the pad and bar prompts for the selected menu item and then carries on with the default behavior for that item. This is fine for testing code, but in "real" code, you may want to disable the default behavior and replace it with your own. In that case, set the ValueType property of the parameter object to "V" or "L" and have the code return .T. (In case you're wondering why there are two values, it's because these values are used for other purposes by the IntelliSense manager, and the VFP team decided to use the same mechanism for MENUHIT.)

The MENUHITs just keep coming

One slight complication: what if you have a script that does something cool and so does someone else, and you want to use both of them? The problem is that there can only be one MENUHIT record (if more than one is present, VFP will use the first one in the IntelliSense table). For this reason, the sample dialog that installs the replacement dialog uses some generic code for MENUHIT. This code delegates to something else rather than perform the desired IDE customization directly. It does this in two ways. First, it looks for another record in the IntelliSense table with TYPE = "M" (which stands for "menu") and ABBREV containing the name of the selected menu item. If it finds such a record, it executes the code in the DATA memo field. If a record is not found, it then looks for a program whose name is stored in the TIP field (which is empty by default but you could put the name of a program in there if desired) and executes that program if it's found. Here's the code:

```
LPARAMETERS toParameter
LOCAL lnSelect, lcCode, llReturn, lScriptHandled
TRY

  * First try FoxCode lookup for Type="M" records
  lnSelect = SELECT()
  SELECT 0
  USE (_FOXCODE) AGAIN SHARE ORDER 1
  IF SEEK('M' + PADR(UPPER(toParameter.MenuItem), ;
    LEN(ABBREV)))
    lcCode = DATA
  ENDIF
  USE
  SELECT (lnSelect)
  IF NOT EMPTY(lcCode)
```

```
    llReturn = EXECSCRIPT(lcCode, toParameter)
    lScriptHandled=.T.
  ENDIF

  * Handle by passing to external routine as
  * specified in Tip field
  IF !lScriptHandled
    lcProgram = ALLTRIM(toParameter.Tip)
    IF FILE(lcProgram)
      DO (lcProgram) WITH toParameter,llReturn
    ENDIF
  ENDIF

  * Custom script successful so let's disable native
  * behavior
  IF llReturn
    toParameter.ValueType = 'V'
  ENDIF
CATCH
ENDTRY
RETURN llReturn
```

There are a few interesting things about this code. First, normally you'd use ORDER <tag name> rather than ORDER <n> to set the order for a table. However, the IntelliSense table is unusual: if you open the table and use ATAGINFO() to retrieve information about the indexes for this table, you'll see that there are two tags, both marked as primary and both without tag names. So, you have to use ORDER 1 or ORDER 2 to set the order for this table. The second thing is that the code is wrapped in a TRY structure to prevent any errors, such as problems opening the table or errors that may exist in the code in the other record. The third issue is that this code doesn't check for the prompt of the pad the user's menu selection is from, only the bar prompt. That's because Microsoft decided to do a SEEK for performance reasons and the tag used is UPPER(TYPE + ABBREV). Since this is a small table, they could probably get away with putting the pad prompt into the EXPANDED column and using LOCATE FOR TYPE = "M" AND ABBREV = toParameter.MenuItem AND EXPANDED = toParameter.UserTyped to ensure the exact record is found if you wish.

In addition to this record, the sample dialog installs two other records with TYPE = "M", one with ABBREV set to "NEW METHOD..." and the other with "NEW PROPERTY..." in ABBREV. These records contain nearly the same code in DATA, which brings up the replacement dialog when the appropriate menu function is selected. Here's the code for the New Method record (in place of <path> will be the path to your VFP home directory):

```
LPARAMETERS toParameter
LOCAL llReturn, ;
  llMethod, ;
  llClass
TRY
  llMethod = toParameter.MenuItem  = 'New Method...'
  llClass  = toParameter.UserTyped = 'Class'
  RELEASE _oNewProperty
  PUBLIC _oNewProperty
  _oNewProperty = NEWOBJECT('NewPropertyDialog', ;
    'NewProperty.vcx', ;
    '<path>\samples\solution\europa\NewProp\' + ;
    'NewPropertyDialog.APP', llMethod, llClass)
  _oNewProperty.SHOW()
  llReturn = .T.
CATCH
ENDTRY
RETURN llReturn
```

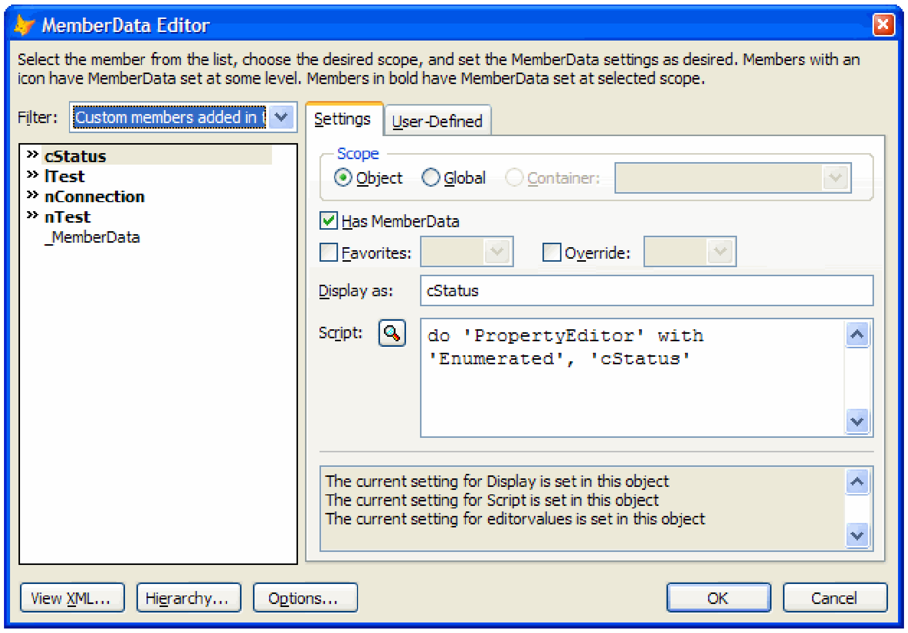
We won't look at the code that makes up NewPropertyDialog.APP. It's not complicated code; at its heart, it calls the AddProperty and WriteMethod methods of the object being edited in the Class or Form Designer to add a new property or method. Both of these methods accept two new parameters in VFP 9: the visibility (1 for public, 2 for protected, or 3 for hidden) and description for the new property or method.

Property editors

Since its release, VFP has provided editors that make it easier to specify the value of a property in the Properties window. Some properties, such as BorderStyle, have a combobox of acceptable values you can choose from. Others have a button with an ellipsis that launches the appropriate dialog, such as the Color dialog for BackColor and ForeColor. For years, VFP developers have lobbied Microsoft for the ability to specify editors for our own custom properties. In VFP 9, they deliver.

You specify an editor for a property in the script attribute of the Member Data for the property. Member Data is something new in VFP; it's an XML string that defines various attributes about the members of an object, including what case a member should be displayed in and whether it appears on the Favorites tab of the Properties window. Member Data is stored in one of two places: a custom _MemberData property of the object, which can contain Member Data for multiple members of the object, or in records in the IntelliSense table (FoxCode.DBF), each of which contains the Member Data for a single member.

Rather than typing XML, which is tedious, VFP 9 includes the MemberData Editor, accessible from the Form and Class menus. To specify the editor for a property, select that property in the MemberData Editor, turn on the Has MemberData setting, and type the code to execute when the editor for that property is invoked in the Script editbox. **Figure 3** shows an example.



Select the member from the list, choose the desired scope, and set the MemberData settings as desired. Members with an icon have MemberData set at some level. Members in bold have MemberData set at selected scope.

» cStatus
» lTest
» nConnection
» nTest
_MemberData

Scope: (●) Object ( ) Global ( ) Container:

[✓] Has MemberData

[ ] Favorites: [ ] Override:

Display as: cStatus

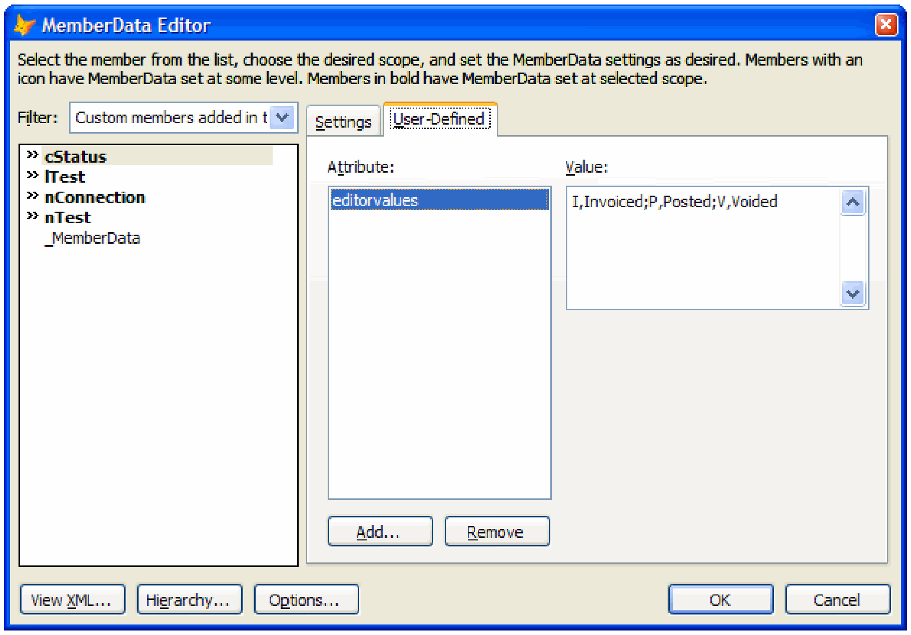
Script:

```
do 'PropertyEditor' with
'Enumerated', 'cStatus'
```

The current setting for Display is set in this object
The current setting for Script is set in this object
The current setting for editorvalues is set in this object

Figure 3. You can define the code to execute when the editor for a property is invoked in the MemberData Editor.

One of the interesting things about Member Data is that it's extensible; you can add custom attributes to the XML. One of the uses we'll make of this is storing meta data for our property editors. The MemberData Editor makes it easy to add custom attributes to a member—on the User-Defined page (see **Figure 4**), click Add to add a new custom attribute and then enter the value for that attribute in the Value editbox.



Select the member from the list, choose the desired scope, and set the MemberData settings as desired. Members with an icon have MemberData set at some level. Members in bold have MemberData set at selected scope.

» cStatus
» lTest
» nConnection
» nTest
_MemberData

Attribute: editorvalues

Value: I,Invoiced;P,Posted;V,Voided

Figure 4. You can define custom Member Data attributes in the User-Defined page of the MemberData Editor.

Here's the Member Data for the properties in the Test class in Test.VCX. The lTest property, which contains a logical value, specifies the Toggle editor. (Note: you'll likely need to specify the path to PropertyEditor.PRG but that's been excluded in these samples for simplicity.)

```
<memberdata
name="ltest"
type="property"
display="lTest"
script="do PropertyEditor with 'Toggle', 'lTest'"/>
```

nTest contains numeric values between 1 and 3, so it also specifies Toggle but uses custom lowvalue and highvalue attributes to specify the range of values.

```
<memberdata
name="ntest"
type="property"
display="nTest"
script="do PropertyEditor with 'Toggle', 'nTest'"
lowvalue="1"
highvalue="3"/>
```

nConnection specifies what type of connection to use for remote data; 1 means use a DSN, 2 means use a connection defined in a DBC, and 3 means use a DSNless connection. Since these are enumerated values, nConnection uses the Enumerated editor and its custom editorvalues attribute provides a list of the values in the format "value, description to display; value, description to display; etc."

```
<memberdata
name="nconnection"
type="property"
display="nConnection"
script="do PropertyEditor with 'Enumerated',
'nConnection'"
editorvalues="1,DSN;2,DBC Connection;
3,DSNless Connection"/>
```

cStatus is also enumerated, with I meaning invoiced, P meaning posted, and V meaning voided.

```
<memberdata
name="cstatus"
type="property"
display="cStatus"
script="do PropertyEditor with 'Enumerated',
'cStatus'"
editorvalues="I,Invoiced;P,Posted;V,Voided"/>
```

Before we look at how to create property editors, here are a few things about them you should know:

- You can invoke the editor for a property in one of two ways: click on the button with the ellipsis that appears when an editor is specified for a property or double-click the property row in the Properties window.
- Property editors are hard to debug: if you put SET STEP ON into the code, execution is suspended at that point. However, even if you choose Step Into, execution continues without stopping. This means you need to add SET STEP ON after each line of code you want execution to be suspended at. To prevent this, use SYS(2030, 1), which enables debugging in system components, just before the SET STEP ON.
- If you want to use classes in earlier versions of VFP, you can't use _MemberData. Well, you can, but _MemberData must have less than 255 characters or you won't be able to open the class in earlier versions, and you'll find that it's very easy to surpass that limit given the verbosity of XML. In that case, store Member Data in the IntelliSense table instead.

PropertyEditor.PRG

Because much of the work in editing a property is the same from one editor to the next, I created a framework for property editors. The starting point is PropertyEditor.PRG, the program specified as the property editor in the examples above. PropertyEditor.PRG doesn't do very much; it simply instantiates and calls the appropriate class for the specified editor name. Editor classes are registered in PropertyEditors.DBF, a simple table in the same directory as the PRG with ID, NAME, CLASS, and LIBRARY columns.

PropertyEditor.PRG accepts as parameters the name of the editor and optionally the property the editor will edit. It opens PropertyEditors.DBF if necessary, looks for the specified name in the NAME column, and if found, instantiates the class specified in the CLASS and LIBRARY columns. Note that it uses TEXTMERGE() on LIBRARY, so you can specify a library in the VFP home directory using "HOME() + 'LibraryName.VCX'." Also, if a path isn't specified, PropertyEditor.PRG looks for the library in the same directory it's in. After instantiating the class, PropertyEditor.PRG calls the editor's EditProperty method to do the work.

```
lparameters tuEditorName, ;
  tcProperty
local lcDirectory, ;
  lcLibrary, ;
  loEditor

* Determine the directory we're running in.

lcDirectory = addbs(justpath(sys(16)))

* Create the specified editor, handling the cases
* where it isn't specified or isn't registered.

do case
  case not vartype(tuEditorName) $ 'CN' or ;
    empty(tuEditorName)
    messagebox('Must specify editor name.', 48, ;
      'Property Editor')
    return
  case not OpenEditorsTable(lcDirectory)
    return
  case (vartype(tuEditorName) = 'C' and ;
    seek(upper(tuEditorName), 'PropertyEditors', ;
    'Name')) or (vartype(tuEditorName) = 'N' and ;
    seek(tuEditorName, 'PropertyEditors', 'ID'))
    lcLibrary = textmerge(PropertyEditors.Library)
    if not file(lcLibrary)
      lcLibrary = forcepath(lcLibrary, lcDirectory)
    endif not file(lcLibrary)
    if file(lcLibrary)
      loEditor = newobject(PropertyEditors.Class, ;
        lcLibrary)
    else
      messagebox('The library specified for ' + ;
        transform(tuEditorName) + ;
        ' cannot be located.', 48, 'Property Editor')
      return
    endif file(lcLibrary)
  otherwise
    messagebox(transform(tuEditorName) + ;
      ' is not a registered editor.', 48, ;
      'Property Editor')
    return
endcase
use in PropertyEditors

* If a property name was specified (for example, for
* an editor that can work with several properties),
* set the cProperty property of the editor to it.

if vartype(tcProperty) = 'C' and not empty(tcProperty)
  loEditor.cProperty = tcProperty
endif vartype(tcProperty) = 'C' ...

* Tell the editor to do its thing.

loEditor.EditProperty()

* Try to open the PropertyEditors table in the same
* directory as this PRG is.

function OpenEditorsTable(tcDirectory)
do case
  case used('PropertyEditors')
  case not file(tcDirectory + 'PropertyEditors.dbf')
    messagebox('PropertyEditors.dbf could not ' + ;
      'be located.', 48, 'Property Editor')
  otherwise
    try
      use (tcDirectory + 'PropertyEditors') in 0 ;
        again shared
    catch to loException
      messagebox('The following occurred while ' + ;
        'trying to open PropertyEditors.DBF:' + ;
        chr(13) + chr(13) + loException.Message, ;
        48, 'Property Editor')
    endtry
endcase
return used('PropertyEditors')
```

SFPropertyEditor

SFPropertyEditor, defined in a VCX of the same name, is the base class for property editors. SFPropertyEditor shouldn’t be used directly; subclass it to create the desired editor. Since PropertyEditor.PRG calls the EditProperty method of the editor, let’s start there.

The first task is to get a reference to the object or objects selected in the Form or Class Designer. The GetObjectReference method fills and returns a collection with the references, so EditProperty puts the collection into the oObjects property. Next, EditProperty calls ValidateProperty to ensure that a property name was specified in the cProperty property and that the specified property is a member of the selected object. (PropertyEditor.PRG puts the second parameter it’s passed into cProperty; for single property editors, fill in a value in the Properties window for the editor subclass.) GetCurrentValue returns the current value of the property (so the editor can display it if necessary) for the first selected object into the uCurrentValue property and cDataType contains the data type of that value. EditProperty then calls GetAttributes to read the Member Data attribute names and their values into the oAttributes collection so the editor can use them if necessary. If your editor needs to test that things are set up properly (for example, that cDataType contains the proper data type the editor is expecting), put that code into the TestCustomAsserts method that EditProperty calls; that method is abstract in this class. Finally, the PropertyEditor method, which is also abstract in this class, is called to perform the actual editing tasks.

```
local loException as Exception
with This
  try

* Get a reference to the selected objects.

    .oObjects = .GetObjectReference()

* Ensure we have a valid property name and get its current value and any
* attributes in _MemberData.

    .ValidateProperty()
    .uCurrentValue = .GetCurrentValue()
    .cDataType     = vartype(.uCurrentValue)
    .oAttributes   = .GetAttributes()

* Test any other asserts, then call the "real" editor method.

    .TestCustomAsserts()
    .PropertyEditor()

* Handle a failed assert.

  catch to loException when not empty(loException.UserValue)
    .Warning(loException.UserValue)

* Handle any other type of error.

  catch to loException
    .Warning(loException.Message)
  endtry
endwith
```

We won't look at all of the other methods in SFPropertyEditor, just those that contain important or interesting code.

GetObjectReference returns a collection of selected objects. It uses ASELOBJ() to fill an array of these objects and copies the references into a collection that it returns. Note the call to the Assert method; this method simply tests the specified condition and if it's false, THROWs the specified error message. Assert is similar to the ASSERT command but allowed me to consolidate all assertion failure into one place: the first CATCH statement in EditProperty.

```
local laObjects[1], ;
  loObjects, ;
  lnI
This.Assert(aselobj(laObjects) > 0 or ;
  aselobj(laObjects, 1) > 0, ;
  'No object is selected.')
loObjects = createobject('Collection')
for lnI = 1 to alen(laObjects)
  loObjects.Add(laObjects[lnI])
next lnI
return loObjects
```

GetAttributes fills a collection with the names and values of all Member Data attributes defined for the property the editor is editing. It uses an XML DOMDocument object to make short work of the searching and parsing. Note that if the object doesn't have a _MemberData property or it's empty, it obviously must have "global" Member Data, which is stored in the IntelliSense table, or else the property editor wouldn't have been launched. So, the Member Data is read from the TIP memo field of the appropriate record in the IntelliSense table in that case.

```
local loAttributes, ;
  lnSelect, ;
  loDOM as MSXML2.DOMDocument, ;
  loObject, ;
  loNode as MSXML2.IXMLDOMElement, ;
  loAttribute as MSXML2.IXMLDOMAttribute, ;
  lcName, ;
  lcValue
loAttributes = createobject('Collection')
try

* Create an XML DOM object and load the XML in the
* _MemberData property of the first selected object.
* If that property doesn't exist or is empty, see if
* there's any global Member Data for it in the
* IntelliSense table.

  loDOM = createobject('MSXML2.DOMDocument.4.0')
  loDOM.async = .F.
  loObject = This.oObjects.Item(1)
  if pemstatus(loObject, '_MemberData', 5) and ;
    not empty(loObject._MemberData)
    loDOM.loadXML(loObject._MemberData)
  else
    select 0
    use (_foxcode) again shared alias __FOXCODE
    locate for TYPE = 'E' and ;
      upper(ABBREV) = upper(This.cProperty)
    if found()
      loDOM.loadXML(TIP)
    endif found()
    use
  endif pemstatus(loObject, '_MemberData', 5) ...

* Find the node for the property we're editing. If
* there is one, load all of the attributes into the
* collection.

  loNode = loDOM.selectSingleNode('//memberdata' + ;
    '[@name="' + lower(This.cProperty) + '"]')
  if vartype(loNode) = 'O'
    for each loAttribute in loNode.attributes
      lcName  = loAttribute.name
      lcValue = loAttribute.value
      loAttributes.Add(lcValue, lcName)
    next loAttribute
  endif vartype(loNode) = 'O'
catch
  use in select('__FOXCODE')
endtry
select (lnSelect)
return loAttributes
```

UpdateProperty isn't called from EditProperty but can be called from PropertyEditor in your subclass to write the new value to the property in each of the selected objects.

```
lparameters tuValue
local loObject, ;
  lcProperty
for each loObject in This.oObjects
  lcProperty = 'loObject.' + This.cProperty
  store tuValue to (lcProperty)
next loObject
```

SFPropertyEditorToggle

Let's look at a practical example. While VFP 9 was still in beta, I watched VFP guru Rick Schummer double-click a property in the Properties window to change it from .F. to .T. No big deal, except I did a double-take when I realized it was a custom rather than native property he'd double-clicked. I asked him whether that was some new feature that'd escaped my attention, and he gave a sly grin and said, "Nope, it's a custom property editor." After thinking about it a bit, I realized that such an editor would also be useful for numeric properties so they'd work just like double-clicking native properties with a pre-defined range of values such as BorderStyle. So I created SFPropertyEditorToggle.

The PropertyEditor method checks to see if custom lowvalue and highvalue Member Data attributes exist, and if so, that both of them exist and they have reasonable values (feel free to change this range check if you wish). It then toggles the property by calling UpdateProperty with the appropriate value: NOT the current value for a logical property, the current value + 1 for a numeric property that hasn't reached the maximum value yet, or the minimum value if the property is at the maximum.

```
local lnValue, ;
  lnLowValue, ;
  lcLowValueType, ;
  lnHighValue, ;
  lcHighValueType
with This

* Ensure that if either lowvalue or highvalue
* attributes exist that they both exist and that
* they're reasonable values.

  if .oAttributes.GetKey('lowvalue') > 0
    lnValue    = .oAttributes.Item('lowvalue')
    lnLowValue = int(val(lnValue))
  endif .oAttributes.GetKey('lowvalue') > 0
  lcLowValueType = vartype(lnLowValue)
  if .oAttributes.GetKey('highvalue') > 0
    lnValue    = .oAttributes.Item('highvalue')
    lnHighValue = int(val(lnValue))
  endif .oAttributes.GetKey('highvalue') > 0
  lcHighValueType = vartype(lnHighValue)
  .Assert(lcLowValueType = lcHighValueType, ;
    '_MemberData must specify both lowvalue and ' + ;
    'highvalue attributes.')
  .Assert(lcLowValueType = 'L' or ;
    (lcLowValueType = 'N' and ;
    between(lnLowValue, 0, 10)), ;
    'The _MemberData lowvalue attribute must be ' + ;
    '0 - 10.')
  .Assert(lcHighValueType = 'L' or ;
    (lcHighValueType = 'N' and ;
    between(lnHighValue, 0, 10)), ;
    'The _MemberData highvalue attribute must ' + ;
    'be 0 - 10.')

* Toggle the property value.

  do case
    case .cDataType = 'L'
      .UpdateProperty(not .uCurrentValue)
    case lcHighValueType = 'N' and ;
      .uCurrentValue >= lnHighValue
      .UpdateProperty(lnLowValue)
    otherwise
      .UpdateProperty(.uCurrentValue + 1)
  endcase
endwith
```

To see this editor in action, open the Test class in Test.VCX and double-click the lTest or nTest properties. As we saw earlier, the Member Data for both properties specifies Toggle as the editor name passed to PropertyEditor.PRG, and Toggle is registered in PropertyEditors.DBF as the SFPropertyEditorToggle class. Also, nTest has lowvalue and highvalue attributes that specify the range of values for nTest as 1 to 3. Double-click nTest several times to see the complete range of values.

SFPropertyEditorEnumerated

Some properties contain “code” values, such as BorderStyle, in which 0 means no border, 1 means a fixed single border, 2 means a fixed dialog border, and 3 means a resizable border. Since there’s a pre-defined list of values and what they mean, these are enumerated properties. You may have your own enumerated properties, such as an invoice status property that contains I for invoiced, P for posted, or V for voided. Rather than forcing the developer to remember the possible values, use SFPropertyEditorEnumerated as the editor for these properties. As you can see in **Figure 5**, it displays a list of the values and their descriptions, similar to the combobox VFP displays in the Properties window for native enumerated properties.

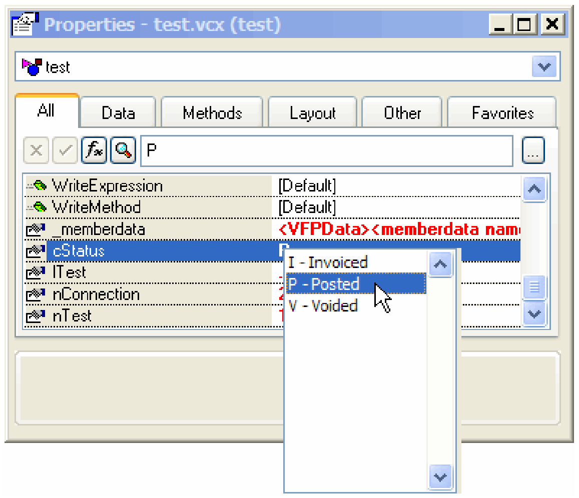


Figure 5. SFPropertyEditorEnumerated displays a listbox of possible values and their descriptions for enumerated properties.

SFPropertyEditorEnumerated needs a list of values to display, so the PropertyEditor method gets the value of the custom editorvalues attribute from the Member Data and parses it into separate value-description sets. PropertyEditor then instantiates SFEnumeratedValueForm, a simple form consisting of a listbox and not much code, and calls its AddValue method to add each value and description to the listbox and its SelectValue method to ensure the current value of the property is selected by default. The trickiest code involves determining where the form should go. Normally, I’d like it to be placed right at the mouse position (MROW() and MCOL() will tell us that) but if the Properties window is dockable, that won’t work because the form can’t be placed on top of the Properties window. So, the code uses some Windows API functions to figure out where the Properties window is located (we won’t look at those methods here), decides whether to put the form to the left or right of the Properties window, and sets the form’s Top and Left properties appropriately. Finally, it calls the Show method of the form, and upon return, retrieves the value the user selected from the form and updates the property depending on the data type required.

```
local lcValues, ;
  lcDirectory, ;
  loForm, ;
  laLines[1], ;
  lnLines, ;
  laValues[1], ;
  lnI, ;
  lcValue, ;
  lcDescription, ;
  lnTop, ;
  llDockable, ;
  lnhWnd, ;
  lcBuffer, ;
  lnWLeft, ;
  lnWRight, ;
  lnLeft
with This

* Get the editorvalues attribute.
```

```
  .Assert(.oAttributes.GetKey('editorvalues') > 0, ;
    'There is no editorvalues attribute in ' + ;
    '_MemberData.')
  lcValues = .oAttributes.Item('editorvalues')

* Create the form to display values in and split out
* into the individual values.

  lcDirectory = sys(16)
  lcDirectory = addbs(justpath(substr(lcDirectory, ;
    at(' ', lcDirectory, 2) + 1)))
  loForm      = newobject('SFEnumeratedValueForm', ;
    lcDirectory + 'SFPropertyEditor.vcx')
  lnLines     = alines(laLines, lcValues, 1, ';')
  .Assert(lnLines > 0, 'The editorvalues ' + ;
    'attribute in _MemberData does not contain a ' + ;
    ' valid set of values.')
  dimension laValues[lnLines]
  for lnI = 1 to lnLines
    lcValue       = laLines[lnI]
    lcDescription = strextract(lcValue, ',')
    lcValue       = strextract(lcValue, '', ',')
    laValues[lnI] = lcValue
    loForm.AddValue(lcValue, lcValue + ' - ' + ;
      lcDescription)
  next lnI

* Select the current value.

  loForm.SelectValue(.uCurrentValue)

* Figure out where to put the form. If the Properties
* window is dockable (whether it's currently docked
* or not), we have to put the form beside it since we
* can't go on top of it. In that case, we have to use
* some Windows API functions to locate the Properties
* window and determine its location. If we have
* enough room to put the form to the right of the
* Properties window, do so; otherwise, put it to the
* left.

  lnTop      = min(mrow('', 3), _screen.Height - ;
    loForm.Height)
  llDockable = wdockable('Properties')
  if llDockable
    lnhWnd = .FindWindow(0, 'Properties')
    if lnhWnd <> 0
      lcBuffer = replicate(chr(0), 16)
      if GetWindowRect(lnhWnd, @lcBuffer) <> 0
        lnWLeft  = ctobin(left(lcBuffer, 4), 'rs')
        lnWRight = ctobin(substr(lcBuffer, 9, 4), 'rs')
      endif GetWindowRect(lnhWnd, @lcBuffer) <> 0
      if _screen.Width - loForm.Width > lnWRight
        lnLeft = lnWRight
      else
        lnLeft = lnWLeft - loForm.Width - 5
      endif _screen.Width - loForm.Width > lnWRight
    else
      lnLeft = 0
    endif lnHWnd <> 0
  else
    lnLeft = min(mcol('', 3), _screen.Width - ;
      loForm.Width)
  endif llDockable
  loForm.Top  = lnTop
  loForm.Left = lnLeft

* Display the form and get the selected value.

  loForm.Show()
  lcValue = laValues[loForm.nSelectedValue]
  do case
    case vartype(loForm) <> 'O' or ;
      loForm.nSelectedValue = 0
    case .cDataType = 'C'
      .UpdateProperty(lcValue)
    otherwise
      .UpdateProperty(int(val(lcValue)))
  endcase
endwith
```

To see this editor in action, open the Test class in Test.VCX and double-click the nConnection or cStatus properties. As we saw earlier, the Member Data for both properties specifies Enumerated as the editor name passed to PropertyEditor.PRG, and Enumerated is registered in PropertyEditors.DBF as the SFPropertyEditorEnumerated class. Both of these properties have an editorvalues attribute that specifies the value and description pairs.

Other property editor thoughts

Another generic property editor, SFPropertyEditorGetFile, is used for properties that contain file names. Create custom attributes called fileext and extdescrip that contain the extension for the file and the description for the extension (for example, "DBF" and "Table") and specify "GetFile" as the editor name passed to PropertyEditor.PRG and the name of the property as the second parameter. We won't look at the code for this editor since it's relatively simple. See the Member Data for the cFileName property in the Test class for an example of how to use it.

Not all property editors you create have to be generic like SFPropertyEditorToggle, SFPropertyEditorEnumerated, and SFPropertyEditorGetFile; your property editors can be specific for a single property. In that case, set the value of cProperty to the name of the property and don't pass a second parameter to PropertyEditor.PRG. Also, a property editor doesn't just have to affect a single property; if the value of one property has an impact on another property, feel free to change both of them at the same time.

Because properties in the Properties window have a limit of 8K characters, and _MemberData can grow quite large quickly when you have a lot of members due to the verbosity of XML, you might want to conserve space in the XML by renaming PropertyEditor.PRG to PE.PRG, putting it in a directory in your VFP path, and referencing a specified editor by ID rather than name. For example, you could use script="PE(1,'PropertyName')" to use SFPropertyEditorToggle, which has an ID of 1, as the editor for the property.

Special thanks to Rick Schummer for testing this, providing valuable suggestions, and, of course, the original idea spark.

Making the most of IntelliSense

IntelliSense, added in version 7, was likely the biggest productivity improvement ever added to VFP. It almost entirely eliminates the need to bring up the VFP help, even for little used commands and functions, because of how it provides tips on clauses and parameters in a context-sensitive manner.

One of the coolest things about the way IntelliSense is implemented in VFP is that, because some of the pieces are written in VFP and it's data-driven, it's extensible. For example, you can easy add your own shortcuts using the IntelliSense Manager in the Tools menu. One of the first I added is "mp" for "MODIFY PROJECT," a command I type a dozen times a day. This shortcut saves me more than just the 2-second difference in time it takes to type "mp" versus "MODI PROJ", because I often mistyped "MODI PROJ" as "MOID PROJ" or "MODI PRJO,", resulting in an error message that I had to dispatch and then try again.

However, you may not be aware that you can create shortcuts that do more than just expand to a simple command. For example, one of the shortcuts that come with VFP is "dc." This shortcut expands to a multi-line block of code (a programmatic class definition) and highlights the text "classname" automatically so you can type the desired class name. To see how that was done, examine the "dc" record in the IntelliSense table:

```
use (_foxcode) again shared
locate for Abbrev = 'DC'
browse
```

Table 2 shows the contents of this record. See the "Customizing IntelliSense Using FoxCode" topic in the VFP Help file for a detailed description of the structure of the FoxCode table.

***Table 2**. The contents of the “dc” shortcut record in the FoxCode table (only fields of interest are shown).*

Field	Value
TYPE	U (user-defined shortcut)
ABBREV	DC
CMD	{}

DATA	LPARAMETERS oFoxcode
	
	IF oFoxcode.Location #1
	   RETURN "DC"
	ENDIF
	
	oFoxcode.valuetype = "V"
	
	TEXT TO myvar TEXTMERGE NOSHOW
	DEFINE CLASS ~classname~ AS Session OLEPUBLIC
	
	PROCEDURE Init
	
	ENDPROC
	
	PROCEDURE Destroy
	
	ENDPROC
	
	PROCEDURE Error(nError, cMethod, nLine)
	
	ENDPROC
	
	ENDDEFINE
	
	ENDTEXT
	
	RETURN myvar

The fields of interest in this record are ABBREV, CMD, and DATA. ABBREV specifies the shortcut that triggers the current script or command (“dc” in this case). CMD specifies the script: A blank value means there’s no script code, a value between curly braces (such as “{color}”) specifies the ABBREV value of a script (type “S”) record, and empty curly braces (as in the case of the “dc” shortcut) means the script is contained in this record. DATA contains the script code.

Now let’s analyze the script itself. The first line in the code shown in the DATA field in Table 2 is an LPARAMETERS statement. All FoxCode script is passed a FoxCodeScript object. IntelliSense creates this object, filling its properties with information about the FoxCode record, what you typed, and the current environment (see the “FoxCode Object Reference” topic in the VFP Help file for a description of each property of this object). The “dc” script code checks the value of the Location property and simply returns “DC” if it isn’t 1. Location indicates where you were typing when the shortcut was invoked; a value of 1 means the PRG editor. So, if you type this shortcut anywhere but in a PRG, nothing appears to happen, which makes sense, because that’s the only logical place for a class definition. The return value, which must be a string, is actually ignored, so a blank string could have been returned with the same results.

Next, the code sets the ValueType property to “V.” As I discussed earlier, this property specifies what happens after the script code is done; “V” means IntelliSense replaces the shortcut with the return value of the code (since it wasn’t set earlier, IntelliSense ignored the previous “DC” return value). The code then uses the TEXT command to place several lines of text (in this case, the class definition code) into a variable and returns that variable. Note the “~” characters surrounding “classname” in the text. A single “~” tells IntelliSense where to place the cursor after replacing the shortcut with the text (without a “~”, it places the cursor at the end of the text), while two of them specify text to highlight. This makes it easy for you to complete the text by entering more information (such as the class name in this case). If you’d rather use something other than “~”, set the CursorLocChar property of the passed object to the desired character.

Creating your own custom scripts isn’t difficult to do, and can boost your productivity boost every time you use them.

Custom scripts

Here’s an example of a useful shortcut implemented with scripting. One block of code you’ve probably written many times either selects a table if it’s already open or opens it if not:

```
if used(some alias)
  select some alias
else
  select 0
  use some alias again shared
endif used(some alias)
```

Since the only thing varying in this code is the alias, it’s possible to create a shortcut that asks you for the alias and then expands to this entire block of code. Here’s the script code that does this:

```
lparameters toFoxCode
local lcReturn, ;
  lcTable
if toFoxCode.Location <> 0
  lcTable = inputbox('Table/view name:')
  if not empty(lcTable)
    toFoxCode.ValueType = 'V'
    lcReturn = GetText(lcTable)
  endif not empty(lcTable)
endif toFoxCode.Location <> 0
return lcReturn

function GetText(tcTable)
local lcDelimiter, ;
  lcTable, ;
  lcText
lcDelimiter = iif(left(tcTable, 1) = '(', '', "'")
lcTable = iif(empty(lcDelimiter), ;
  strtran(strtran(tcTable, '('), ')'), ;
  lcDelimiter + tcTable + lcDelimiter)
text to lcText textmerge noshow
if used(<<lcTable>>)
  select <<tcTable>>
else
  select 0
  use <<tcTable>> again shared
endif used(<<lcTable>>)
endtext
return lcText
```

Like the “dc” script code, this script accepts a FoxCodeScript object parameter and checks where you typed the shortcut by examining the Location property. It doesn’t make sense to expand this shortcut in the Command Window, so if Location contains 0, the rest of the code is skipped. Otherwise, the code prompts you for the name of a table or view. If you enter one, the code sets the ValueType property of the FoxCode to “V” and calls the GetText routine to place the desired block of code, with the name you entered inserted at the appropriate places, into the return value of the script. Note that you can either enter an alias, in which case the script places it in quotes in the “IF USED” statement, or the name of a variable surrounded with parentheses (such as “(lcAlias)”); the latter will generate the following:

```
if used(lcAlias)
  select (lcAlias)
else
  select 0
  use (lcAlias) again shared
endif used(lcAlias)
```

To create this shortcut, bring up the IntelliSense Manager, select the Custom page, enter the shortcut code you want to use (such as “us” for “Use or Select”) but leave the With text box blank, and click on the Add button. Click on the Script button and enter the code shown earlier. You can, of course, also manually add a record in the IntelliSense table, or append the records in the NewFoxCode table that accompanies this document into your IntelliSense table. To use it, simply type “us” and press Enter in a code window, enter the desired table name when prompted, and carry on.

Here’s another example. This shortcut inserts program header comments. Like the previous example, it only works from a code editor, not the Command Window, it uses text merge to create the text to replace the abbreviation in the command line, and, with the “~” character, it tells IntelliSense to put the cursor in the “Purpose” comment line after the expansion is complete. It has a couple of interesting wrinkles, though. First, it reads your name, company name, and e-mail address from the Registry using the FoxPro Foundation Classes (FFC) Registry class so it can insert them into the header. Second, it uses WONTOP() to insert the name of the file being edited. As you can see, script code can be considerably more complex than simply outputting some text.

```
lparameters toFoxCode
local lcReturn, ;
  lcTable
if toFoxCode.Location <> 0
  toFoxCode.ValueType = 'V'
  lcReturn = GetText()
endif toFoxCode.Location <> 0
return lcReturn

function GetText
local loRegistry, ;
  lcKey, ;
  lcCompany, ;
  lnResult, ;
  lcContact, ;
  lcAccount, ;
  lcEmail, ;
  lcText
loRegistry = newobject('Registry', ;
  home() + 'FFC\Registry.vcx')
lcKey      = iif('NT' $ os() or '5.0' $ os(), ;
  'Software\Microsoft\Windows NT\CurrentVersion', ;
  'Software\Microsoft\Windows\CurrentVersion')
lnResult   = ;
  loRegistry.GetRegKey('RegisteredOrganization', ;
  @lcCompany, lcKey, -2147483646)
if lnResult <> 0
  lcCompany = ''
endif lnResult <> 0
lnResult = ;
  loRegistry.GetRegKey('RegisteredOwner', ;
  @lcContact, lcKey, -2147483646)
if lnResult <> 0
  lcContact = ''
endif lnResult <> 0
lcKey    = 'Software\Microsoft\Internet Account ' + ;
  'Manager'
lnResult = ;
  loRegistry.GetRegKey('Default Mail Account', ;
  @lcAccount, lcKey, -2147483647)
if not empty(lcAccount)
  lcKey    = lcKey + '\Accounts\' + lcAccount
  lnResult = ;
    loRegistry.GetRegKey('SMTP Email Address', ;
    @lcEmail, lcKey, -2147483647)
endif not empty(lcAccount)
if lnResult <> 0 or empty(lcEmail)
  lcEmail = ''
else
  lcEmail = ', mailto:' + lcEmail
endif lnResult <> 0 ...
text to lcText textmerge noshow
*=====================================================
* Program:         <<wontop()>>
* Purpose:         ~
* Author:           <<lcContact>>
* Contact:          <<lcEmail>>
* Copyright:        (c) <<year(date())>> <<lcCompany>>
* Last revision:    <<date()>>
* Parameters:
* Returns:
* Environment in:
* Environment out:
*=====================================================

endtext
return lcText
```

Here's an example of the output of this shortcut:

```
*=====================================================
* Program:          TESTPROGRAM.PRG
* Purpose:          the cursor is positioned here
* Author:           Doug Hennig
* Contact:          mailto:dhennig@stonefield.com
* Copyright:        (c) 2005 Stonefield Software Inc.
* Last revision:    03/19/2005
* Parameters:
* Returns:
* Environment in:
* Environment out:
*=====================================================
```

The NewFoxCode table accompanying this document specifies "header" as the shortcut for this script, but you can of course use any shortcut you wish. To use this shortcut, start a new program, type "header" and press Enter. You can then update any of the information in the header as you see fit.

Automating LOCAL statements

One other IDE enhancement I use every day is a PRG called DeclareLocals.PRG. This program was written by Andy Kramek and Marcia Akins and published in *MegaFox: 1002 Things You Wanted to Know About Extending Visual FoxPro*, a book from Hentzenwerke Publishing (www.hentzenwerke.com) that I highly recommend. It automates the construction of the LOCAL statement at the start of a routine by allowing you to put the insertion point somewhere in a variable name and hit a shortcut key to add that variable to the LOCAL statement if it isn't there already. This saves having to scroll up to add the variable and scroll back down to where you were working, or worse, forgetting to declare a variable as LOCAL.

I won't show the code for DeclareLocals.PRG here as it's fairly long. I've customized the version published by Andy and Marcia to work slightly differently—I like to have each variable specified on its own line.

Adding DeclareLocals to your VFP IDE is easy. Specify the name of a startup program (mine is called VFPStart.PRG) by adding _STARTUP = *some program name* to your CONFIG.FPW file and put code similar to the following into that program:

```
lnBar    = 0
lcPrompt = 'Add \<Variable to LOCAL'
lcBar    = strtran(lcPrompt, '\<')
for lnI = cntbar('_MEDIT') to 1 step -1
  if prmbar('_MSM_EDIT', ;
    getbar('_MSM_EDIT', lnI)) = lcBar
    lnBar = lnI
    exit
  endif prmbar('_MSM_EDIT' ...
next lnI
if lnBar = 0
  lnBar = cntbar('_MSM_EDIT') + 1
  define bar lnBar     of _MSM_EDIT ;
    after _mlast prompt '\-'
  define bar lnBar + 1 of _MSM_EDIT ;
    key Alt+6, 'Alt+6' after _mlast ;
    prompt lcPrompt
else
```

```
  lnBar = lnBar - 1
endif lnBar = 0
on selection bar lnBar + 1 of _MSM_EDIT ;
  do home() + 'DeclareLocals'
```

This code adds an "Add Variable to LOCAL" bar to the Edit menu with Alt-6 as its hotkey.

How My Works

*** THIS CODE AND TEXT NEEDS TO BE UPDATED: SEE Code Focus ARTICLE!!!
New to Visual Basic.NET 2005 is the My namespace. My makes .NET Framework classes more discoverable and allows you to write less code. Here's an example taken from Duncan Mackenzie's MDSN article "Navigate the .NET Framework and Your Projects with My" (http://msdn.microsoft.com/msdnmag/issues/04/05/VisualBasic2005/default.aspx). Instead of writing the following to read the contents of a text file:

```
Dim sr As New IO.StreamReader("c:\mytextfile.txt")
contents = sr.ReadToEnd
sr.Close()
```

you could write this:

```
contents = My.Computer.FileSystem.ReadAllText("c:\mytextfile.txt")
```

Thanks to IntelliSense on the My namespace, not only is it easier to figure out how to do this task, it's also less code to write and debug.

Sedna, the next version of VFP, includes a My namespace as well, for the same reasons that VB.NET 2005 does. In this section, we'll look at how My is implemented in Sedna.

Many of the classes that come with My are wrappers for SYS() functions, Windows API functions, Windows Script Host properties and methods, and so on. For example, the Play method of Audio, which plays an audio file, is a wrapper for the sndPlaySound Windows API function. So, without having to DECLARE this function or even know that it exists, your VFP application can play a sound file.

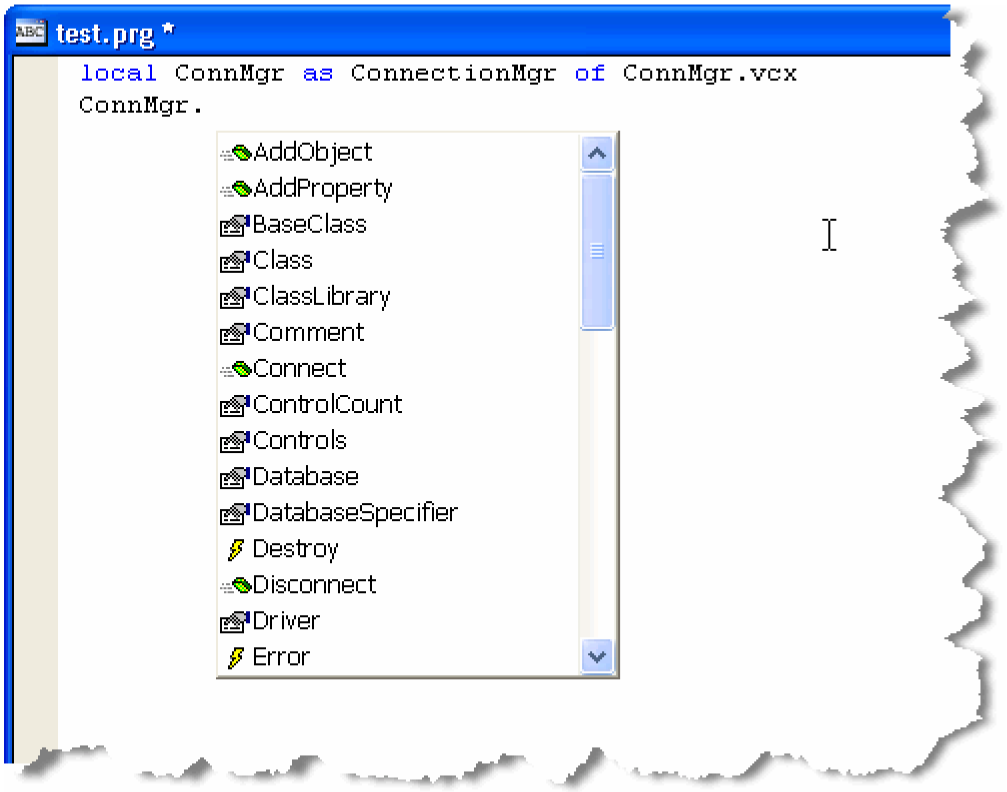
```
lparameters SoundFile as String
#define SND_SYNC       0
#define SND_ASYNC      1
#define SND_NODEFAULT  2
#define SND_LOOP       8
#define SND_NOSTOP    16
declare integer sndPlaySound in WinMM.dll ;
  string lpszSoundName, integer uFlags
sndPlaySound(SoundFile, SND_ASYNC)
```

There are two aspects that make My useful: IntelliSense at design-time and the class hierarchy at run-time.

IntelliSense for My

IntelliSense is easily the best feature ever added to Visual FoxPro. It provides a greater productivity boost to VFP developers than anything added before or since. However, one thing that bugs me about IntelliSense is that when used with a class, it displays all members of that class rather than the ones I really want to see.

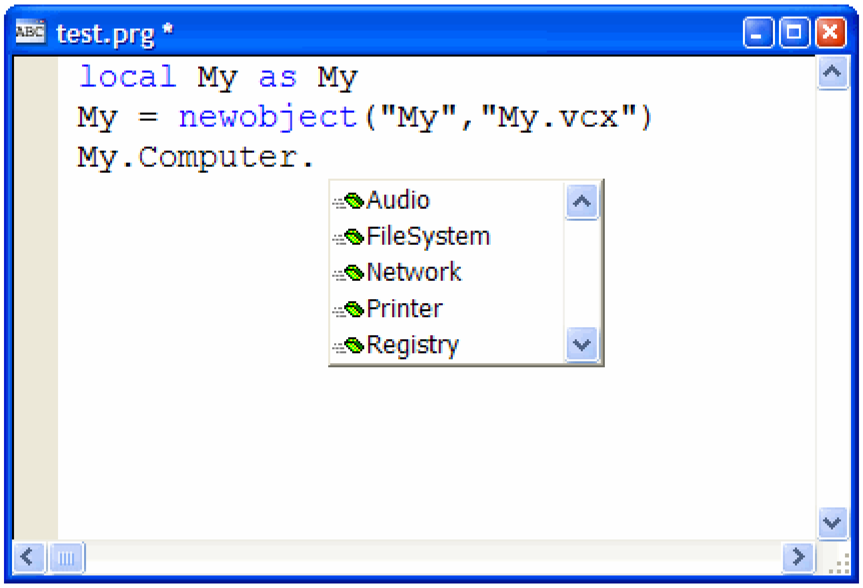
For example, **Figure 6** shows the IntelliSense display for the ConnectionMgr class. Note that although there are only a few custom properties and methods we're interested in, IntelliSense displays everything. This requires more effort to select the exact member you want, especially if you're not very familiar with the class.



```
local ConnMgr as ConnectionMgr of ConnMgr.vcx
ConnMgr.
```

Figure 6. Although IntelliSense allows you to choose a member name from a list, it displays more items than we usually need.

However, as you can see in **Figure 7**, IntelliSense on members of the My namespace show only those properties and methods we're actually interested in.



```
local My as My
My = newobject("My","My.vcx")
My.Computer.
```

Figure 7. IntelliSense on My members shows only the members of interest.

The secret behind My's IntelliSense lies in two things: how IntelliSense deals with things defined as "types" in the IntelliSense table and IntelliSense scripts. Types are normally use for data types, such as Integer or Character, and base classes, such as Checkbox and Form. However, other things can be defined as types as well, either by manually adding records with TYPE set to "T" or using the IntelliSense Manager in the Tools menu. Other kinds of type records are usually custom classes or COM objects so you get IntelliSense on them. That's what My uses type records for as well, but it customizes how IntelliSense works using a script and a custom IntelliSense-handling class.

If you look in your IntelliSense table (USE (_FOXCODE) AGAIN and BROWSE) after registering My, you'll see two new records at the end of the table. One is the type record for the namespace; it doesn't contain much information other than "My" in the ABBREV and DATA fields and "{myscript}" as the name of the script to use for IntelliSense purposes in the CMD field. The other is a script record, with TYPE set to "S" and ABBREV containing "myscript."

The script record has the following code in its DATA memo (in this code, Path is replaced with the path for MY.VCX):

```
lparameters toFoxcode
local loFoxCodeLoader, ;
  luReturn
if file(_codesense)
  set procedure to (_codesense) additive
  loFoxCodeLoader = createobject('FoxCodeLoader')
  luReturn        = loFoxCodeLoader.Start(toFoxcode)
  loFoxCodeLoader = .NULL.
  if atc(_codesense, set('PROCEDURE')) > 0
    release procedure (_codesense)
  endif atc(_codesense, set('PROCEDURE')) > 0
else
  luReturn = ''
endif file(_codesense)
return luReturn

define class FoxCodeLoader as FoxCodeScript
  cProxyClass    = 'MyFoxCode'
  cProxyClasslib = 'Path\my.vcx'

  procedure Main
    local loFoxCode, ;
      luReturn
    loFoxCode = newobject(This.cProxyClass, ;
      This.cProxyClasslib)
    if vartype(loFoxCode) = 'O'
      luReturn = loFoxCode.Main(This.oFoxCode)
    else
      luReturn = ''
    endif vartype(loFoxCode) = 'O'
    return luReturn
  endproc
enddefine
```

This code defines a subclass of the FoxCodeScript class defined in the IntelliSense application specified by the _CODESENSE system variable. This subclass overrides the Main method, which is called by IntelliSense, to instantiate the MyFoxCode class in MY.VCX and call its Main method, passing it a reference to the IntelliSense data object, which contains information about what the user typed and other IntelliSense settings. As a result of this script, MyFoxCode.Main is called for all IntelliSense tasks for My, such as when you select "My" from the IntelliSense list displayed when you type LOCAL My AS or when you type one of the "trigger" characters—such as a period, an opening parenthesis, or an equals sign—in a statement containing My.

MyFoxCode

The MyFoxCode class does all of the custom IntelliSense work for My, so let's examine this class in detail.

The Init method just does two things: turns on debugging in system components (without this, you can't easily debug problems in the code) and opens the My table, which contains information about the My namespace members (we'll discuss this table in more detail later), by calling OpenMyTable. If the table can't be opened, Init displays an error message and returns .F. so the class isn't instantiated. Since My uses a table of members, it's data-driven, which, as we'll see later on, will give us a number of benefits.

```
* Turn debugging on.
```

```
sys(2030, 1)

* Open the My table.

local llReturn
llReturn = This.OpenMyTable()
if not llReturn
  messagebox('Could not open My.DBF.', 64, ;
    'My Framework IntelliSense Handler')
endif not llReturn
return llReturn
```

The Main method, called from the IntelliSense script, handles all of the IntelliSense tasks for My. As we saw earlier, the script passes a FoxCode object to Main. If “My” is found in the MenuItem property of the FoxCode object, we must be on the LOCAL My AS statement, so Main calls the HandleLOCAL method to deal with it. Otherwise, Main determines which character triggered IntelliSense and calls the GetMyMember method to determine which My member you typed (it could also be My itself) and return a SCATTER NAME object from the appropriate record in the My table. If the trigger character is a period, we need to display a list of the registered My members, so Main calls DisplayMembers to do the work. If the trigger character is an opening parenthesis and the LIST field in the My table is filled it, we’ll call DisplayEnumeratedValues to display a list of enumerated values available for a parameter for the method (similar to what IntelliSense displays when you type “DBGETPROP(“). Finally, if the trigger character is an opening parenthesis, an equals sign, or a comma and the TIP memo of the My record is filled it, Main uses it as the tooltip for IntelliSense. This is usually used to display the signature of a method (for example, “Login(UserName as String, Password as String) as Boolean”).

```
lparameters toFoxCode
local lcReturn, ;
  lcTrigger, ;
  loData
with toFoxCode
  .ValueType = 'V'

* If we're on the LOCAL statement, handle that by
* returning text we want inserted.

  lcReturn = ''
  if atc('My', .MenuItem) > 0
    lcReturn = This.HandleLOCAL()

* Get the character that triggered IntelliSense and
* figure out which member the user typed.

  else
    lcTrigger = right(.FullLine, 1)
    loData    = This.GetMyMember(.UserTyped)
    do case

* We can't figure out which member the user typed, so
* do nothing.

      case vartype(loData) <> 'O'

* If we were triggered by a ".", display a list of
* members.

      case lcTrigger = '.'
        This.DisplayMembers(toFoxCode)

* If we were triggered by a "(" (to start a method
* parameter list) and the method accepts enumerated
* values specified in the LIST memo, display them.

      case lcTrigger = '(' and not empty(loData.List)
        This.DisplayEnumeratedValues(toFoxCode, ;
          loData)
```

```
* If we were triggered by a "(" (to start a method
* parameter list) or "," (to enter a new parameter),
* display the parameters for the method.

      case inlist(lcTrigger, '(', ',') and ;
        not empty(loData.Tip)
        .ValueTip  = loData.Tip
        .ValueType = 'T'
    endcase
  endif atc('My', .MenuItem) > 0
endwith
return lcReturn
```

We won’t look at the code for HandleLOCAL here; feel free to examine this method yourself. Main calls it when you type LOCAL My AS and choose My from the type list. All it does is generate a NEWOBJECT() statement for the class so you don’t have to. The only complication for this method is determining the case to use for NEWOBJECT(). (I could have hard-coded this as “newobject” since I always use lower-case for FoxPro keywords, but decided to be nice to other developers who may have different ideas about casing.) This is solved by looking in the IntelliSense table for a record with TYPE = “F” (for “function”) and ABBREV = “NEWO” (the abbreviation for NEWOBJECT) and using the value in the CASE field.

GetMyMember, called from Main, looks for the member you typed in the My table. It uses the UserTyped property of the FoxCode object (passed as a parameter), which contains the text you typed pertaining to the namespace. For example, when you type “llStatus = My.Computer.Audio.Play(“, UserTyped contains “Computer.Audio.Play”. GetMyMember finds the record for the appropriate member in the My table and it returns a SCATTER NAME object from that record.

```
lparameters tcUserTyped
local lcUserTyped, ;
  lcAlias, ;
  loReturn, ;
  lnSelect
lcUserTyped = alltrim(tcUserTyped)
if right(lcUserTyped, 1) = '('
  lcUserTyped = substr(lcUserTyped, ;
    len(lcUserTyped) - 1)
endif right(lcUserTyped, 1) = '('
loReturn = .NULL.
if seek(upper(lcUserTyped), '__MY')
  lnSelect = select()
  select __MY
  scatter memo name loReturn
  select (lnSelect)
endif seek(upper(lcUserTyped), '__MY')
return loReturn
```

DisplayMembers is called from Main to tell IntelliSense to display a list of registered My members when you type a period in the command line. DisplayMembers calls GetMembers to retrieve a collection of members for the specified member (we won’t look at that method here). It then fills the Items array of the FoxCode object with the names and descriptions of the members and sets the object’s ValueType property to “L,” which tells IntelliSense to display a listbox with the contents of the Items array. This code shows one slight design flaw in IntelliSense: the FoxCode object has a single Icon property which contains the name of the image file to display in the listbox. What is actually needed is an additional column in the Items array, since in this case, we want to display different images for properties and methods. Unfortunately, we get only a single image displayed for all items.

```
lparameters toFoxCode
local loMembers, ;
  lnI
with toFoxCode
  loMembers = This.GetMembers(.UserTyped)
  if loMembers.Count > 0
    dimension .Items[loMembers.Count, 2]
    for lnI = 1 to loMembers.Count
      loMember       = loMembers.Item(lnI)
      .Items[lnI, 1] = loMember.Name
      .Items[lnI, 2] = loMember.Description
      if loMember.Type = 'P'
        .Icon = home() + 'ffc\graphics\propty.bmp'
      else
        .Icon = home() + 'ffc\graphics\method.bmp'
      endif loMember.Type = 'P'
    next loMember
    .ValueType = 'L'
  endif loMembers.Count > 0
endwith
```

Run-time Class Hierarchy

IntelliSense is one thing; it's another to actually have the My namespace work when you run the code. Although it would be simple to have a class called My that has members named App, Computer, Data, and so forth, My is actually more extensible than that; like IntelliSense, it's data-driven (in fact, using the same My table).

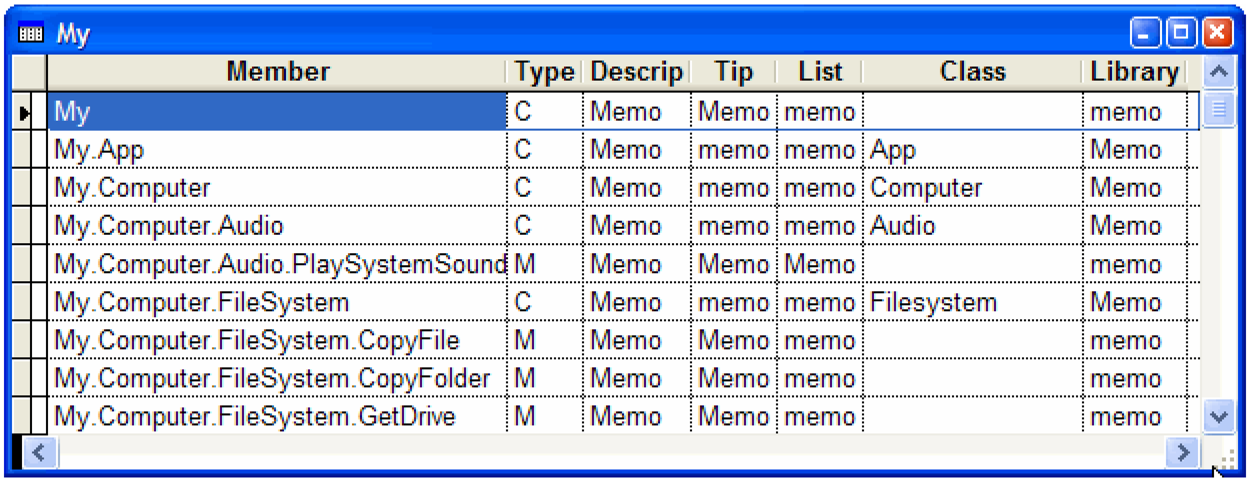
The My class is actually a subclass of MyBase, as is the Computer, User, and other classes. MyBase, a subclass of Custom, dynamically adds members to itself based on what it finds in the My table. This work is done in AddMembers, called from Init. Here's the code for AddMembers:

```
local lnSelect, ;
  lcNameSpace, ;
  lnLen, ;
  lcCursor, ;
  lcMember, ;
  lcLibrary

* Create a cursor of all objects in this namespace.

lnSelect    = select()
lcNameSpace = upper(This.cNameSpace) + '.'
lnLen       = len(lcNameSpace) + 1
lcCursor    = sys(2015)
select * from __MY ;
  where upper(MEMBER) = lcNameSpace and ;
  not empty(CLASS) and not deleted() ;
  into cursor (lcCursor)

* Go through the members, adding any that are directly
* within this namespace (for example, if this is "My",
* we'll add "My.Computers" but not "My.Computers.
* Audio").

scan
  lcMember  = alltrim(substr(MEMBER, lnLen))
  lcLibrary = fullpath(alltrim(LIBRARY), ;
    This.ClassLibrary)
  if at('.', lcMember) = 0 and file(lcLibrary)
    This.NewObject(lcMember, alltrim(CLASS), ;
      lcLibrary)
  endif at('.', lcMember) = 0 ...
endscan
use
select (lnSelect)
```

This code selects records from the My table matching the namespace specified in the custom cNameSpace property, which contains the namespace of this object (for example, "My" for the My class and "My.Computer" for the Computer class). It then instantiates the classes specified in those records and adds them as members. For example, for the My namespace, the My table has records for members named My.App, My.Computer, My.Data, and My.User. Thus, instantiating the My class, which is based on MyBase, causes all of the members registered in the My table to be created dynamically. My actually has no code; it simply has cNameSpace set to "My."

Computer, the class representing the My.Computer member, is also a subclass of MyBase. So, when it's instantiated from the AddMembers method of My, its AddMembers method goes through the My table, looking for members of the My.Computer namespace, such as My.Computer.Audio, My.Computer.FileSystem, and so on. Those classes in turn are based on MyBase, so a hierarchy as deep as necessary is automatically built simply by instantiating one class (My). For example, four levels of classes are automatically setup for the My.Computer.FileSystem.SpecialFolders namespace.

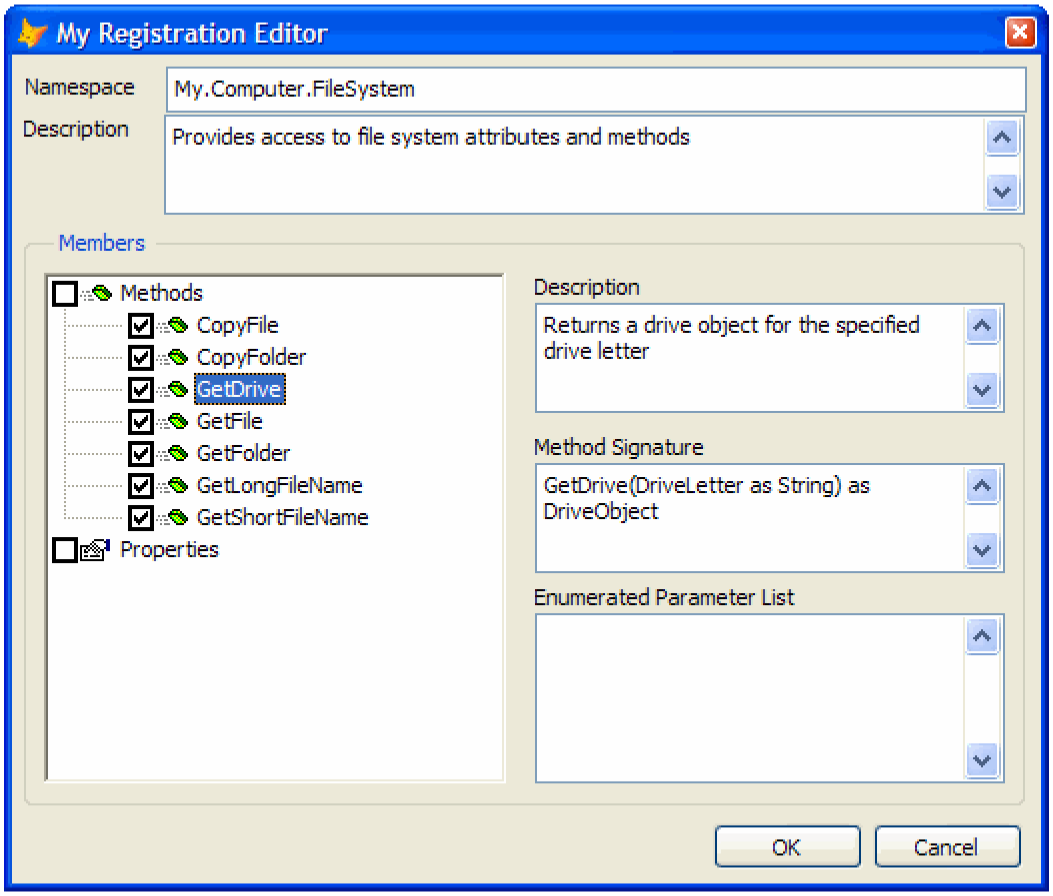
Figure 8 shows the structure of the My table. The MEMBER field contains the name of the member the record is for, with a fully qualified namespace. The TYPE column indicates what type of record this is: "C" for class, "M" for method, and "P" for property. DESCRIP contains a description for the member displayed as a tooltip in the IntelliSense member list. TIP contains the tooltip for a method displayed when you type the opening parenthesis; this is usually used to display the signature of the method. LIST contains a list of enumerated values displayed for the parameter of a method; this was discussed earlier. CLASS and LIBRARY contain the class and class library for the class to instantiate for "C" records.



Member	Type	Descrip	Tip	List	Class	Library
My	C	Memo	Memo	memo		memo
My.App	C	Memo	memo	memo	App	Memo
My.Computer	C	Memo	memo	memo	Computer	Memo
My.Computer.Audio	C	Memo	memo	memo	Audio	Memo
My.Computer.Audio.PlaySystemSound	M	Memo	Memo	Memo		memo
My.Computer.FileSystem	C	Memo	memo	memo	Filesystem	Memo
My.Computer.FileSystem.CopyFile	M	Memo	Memo	memo		memo
My.Computer.FileSystem.CopyFolder	M	Memo	Memo	memo		memo
My.Computer.FileSystem.GetDrive	M	Memo	Memo	memo		memo

Figure 8. The My table allows My to be data-driven.

Given that there's one record for every class, property, and method, it would be pretty tedious to fill this table out by hand. Fortunately, there's an easier way: with a builder. MyBase has a custom Builder property containing "My.vcx,MyBuilderForm". This tells VFP that MyBuilderForm in MY.VCX is the builder for this class and any class based on it. After creating a subclass of My, it and its members can be registered in the My table by right-clicking the class and choosing Builder. **Figure 9** shows what the builder form looks like.



Namespace: My.Computer.FileSystem

Description: Provides access to file system attributes and methods

- ☐ Methods
  - ☑ CopyFile
  - ☑ CopyFolder
  - ☑ GetDrive
  - ☑ GetFile
  - ☑ GetFolder
  - ☑ GetLongFileName
  - ☑ GetShortFileName
- ☐ Properties

Description: Returns a drive object for the specified drive letter

Method Signature: GetDrive(DriveLetter as String) as DriveObject

Figure 9. The My Registration Builder makes short work of registering a class in the My table.

The My Registration Builder allows you to specify the namespace for the class. It defaults to "My." plus the name of the class, but you can specify something else if you wish. For example, the FileSystem class is a member of My.Computer, so its namespace is My.Computer.FileSystem. The description is used as the tooltip for the class in the IntelliSense list. It defaults to the description for the class as specified in the Class Info function in the Class menu or by choosing the Edit Description function in the Project menu when the class is selected in the Project Manager.

The TreeView shows public custom properties and methods for the class; if you want native members displayed as well, change the AMEMBERS() statement in the LoadTree method of the MyBuilderForm class in MY.VCX. The checkbox before the name indicates whether the member is included in IntelliSense or not; by default, all custom members are included. The description is used as the tooltip for the member in the IntelliSense list; it defaults to the description you entered for the member when you created it. The method signature is displayed as a tooltip for a method when you type an open parenthesis or a comma in the parameter list for the method; this tooltip shows you what parameters can be passed to the method. The signature defaults to the method name and the contents of any LPARAMETERS statement in the method, but you can edit it to display anything you wish, including the data type of the return value. The Enumerated Parameters editbox allows you to the list of enumerated values for the method's parameter.

What if you want to add your own namespaces to My? You could do that by subclassing MyBase to create new classes with the desired functionality, but what if you already have a class you want to use that isn't based on MyBase? No problem: open the class and DO FORM MyBuilderForm. This form is an instance of the MyBuilderForm class and can register any class in the My table. Of course, since classes that aren't based on MyBase won't dynamically add members to themselves, these classes won't have a dynamic hierarchy below them nor will you get IntelliSense on objects members.

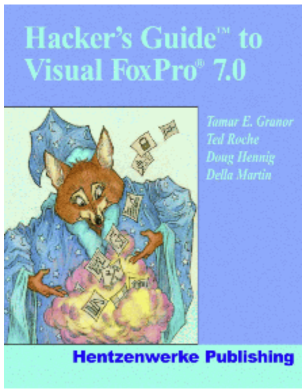
To see this in action, open the ConnectionMgr class in ConnMgr.VCX, then DO FORM MyBuilderForm, make the desired changes, and choose OK. Close the class. In a PRG window, type LOCAL My as My followed by My.ConnectionMgr. Notice you see IntelliSense on the members you specified.

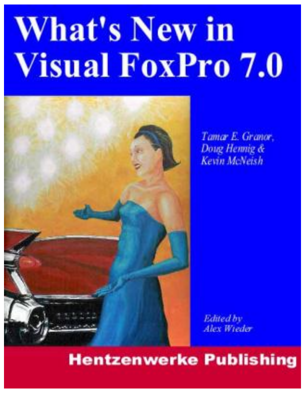
Summary

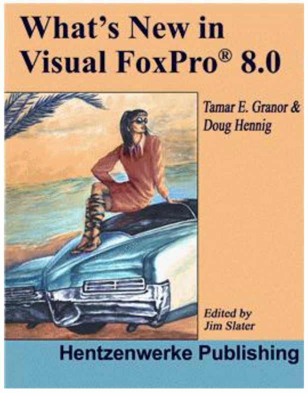
The fact that you can customize and extend the VFP IDE using VFP code makes it possible to improve your productivity immensely. The ideas I showed in this document should help you to reduce the time it takes to create the great applications VFP makes possible.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT), the award-winning Stonefield Query, and the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro, and the My namespace and updated Upsizing Wizard in Sedna. Doug is co-author of the "What's New in Visual FoxPro" series (the latest being "What's New in Nine") and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). Doug wrote over 100 articles in 10 years for FoxTalk. He has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over the world. He is one of the administrators for the VFPX VFP community extensions Web site (http://www.codeplex.com/Wiki/View.aspx?ProjectName=VFPX). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://fox.wikis.com/wc.dll?Wiki~FoxProCommunityLifetimeAchievementAward~VFP).




What's New in Visual FoxPro 7.0
Tamar E. Granor, Doug Hennig & Kevin McNeish
Edited by Alex Wieder
Hentzenwerke Publishing


What's New in Visual FoxPro® 8.0
Tamar E. Granor & Doug Hennig
Edited by Jim Slater
Hentzenwerke Publishing

stonefield Database TOOLKIT

stonefield Query

stonefield SYSTEMS GROUP INC

Copyright © 2005-2007 Doug Hennig. All Rights Reserved.