December, 2004

Build your own Property Editors

VFP 9 lets you attach custom code to the Property Sheet

By Tamar E. Granor, Technical Editor

One of the most exciting changes in VFP 9 is the ability to add custom editors to the Property Sheet. Some properties already let you specify a value using a specialized tool. For example, all the color-related properties give you access to the Color Picker, while Icon and Picture use the GETPICT() dialog. Other properties, like AlwaysOnTop and FontName, use a dropdown combo. Until VFP 9, though, you were stuck with the default textbox for any properties where such a tool wasn't specified, including your custom properties. That's changed; VFP 9 lets you specify an editor to use for any property that doesn't use one of the built-in mechanisms. If you want, you can specify a different editor for each property.

Why would you want to specify a property editor? To make it easier to set a property correctly. For example, the new Anchor property is complicated, so the VFP team included a custom property editor for it. (See the Advisor Answers column in the November, 2004 issue for more on Anchor and the Anchor Editor.) When the set of possible values for a custom property is limited, a property editor lets you present that list to the developer and encourage him to choose from that set. Be aware, though, that Property editors don't prevent a developer from typing an invalid value into the textbox in the Property Sheet.

Using a Property Editor

When a property has a property editor, an ellipsis (three dots) button appears next to the textbox in the Property Sheet, as in Figure 1. Click the button and the specified property editor appears.

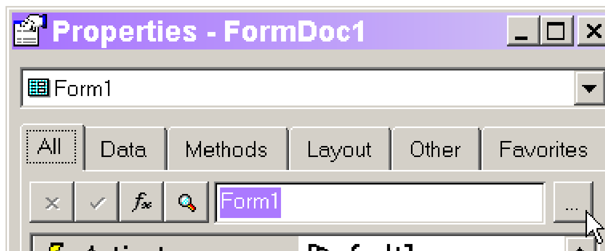

Figure 1. Invoking a property editor—When a property has a property editor, the ellipsis button appears next to the textbox. Click the button to run the property editor.

Property Editors and _MemberData

Propertyeditors work through the new _MemberData property, which you can add to any form or class to specify the behavior of that form's or class's PEMs in the development environment. (See the Advisor Answers column in the November issue for a discussion of other uses of _MemberData.)

_MemberData contains an XML string in a format easy to convert to a VFP table. The top-level element is <VFPData>. Within the <VFPData> element, there are <memberdata> elements for those properties and methods that have customization specified. Each memberdata element has attributes. Two attributes are required: name (the name of the PEM) and type ("property", "event" or "method"). You can add any other attributes you want, but support for several is included in the product.

For properties, one optional attribute is script, which specifies a property editor. A script is just VFP code. A typical script prompts the developer for the data and stores it appropriately.

Specifying a property editor

The easiest way to specify a property editor is by using the MemberData Editor. This tool (shown in Figure 2) is listed on the Form or Class menu when you're working in the Form Designer or Class Designer. It provides a convenient interface for entering _MemberData contents. (The version of the MemberData Editor shown here is significantly different from the one included with the public beta of VFP 9. However, the public beta does support property editors.) The list on the left shows the PEMs of the object. When you select a PEM in the list, its memberdata is shown on the right and can be edited.

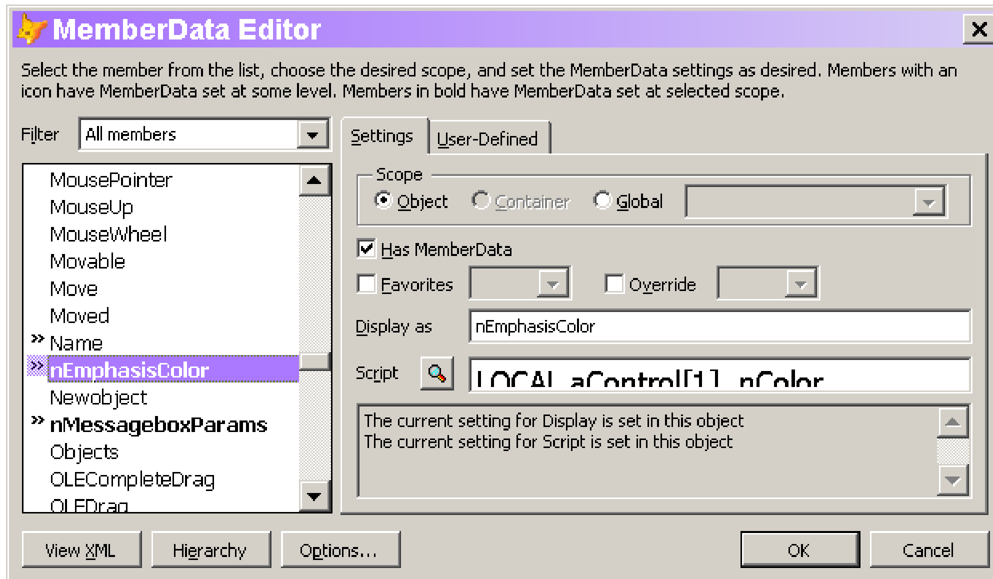

Figure 2. The MemberData Editor—Use this new tool to specify property editors, as well as capitalization of custom properties and methods, and other Property Sheet customization.

To specify a property editor, select the property and click the Zoom button next to Script. A code window opens and you can type your script. Close the code window to save the script in the MemberData Editor. When you close the MemberData Editor, your new property editor will be available. The MemberData Editor adds the _memberdata property to the form or class, if it doesn't already exist, and populates it with the appropriate XML.

Building a property editor

Property editors have a lot in common with builders. Both let you set properties at design-time and both have to figure out what to modify. However, property editors are a lot easier to build and distribute than builders.

A property editor has three key tasks: identifying the object(s) involved, soliciting input from the user (throughout this article, the "user" is a developer creating or modifying a form or class), and saving the user's input to the relevant property or properties. Here's the code for a property editor that calls on the Color Picker for a custom property called nEmphasisColor:

```
LOCAL aControl[1], nColor

IF ASELOBJ(aControl) = 0
   IF ASELOBJ(aControl, 1) = 0
```

```
      RETURN
   ENDIF
ENDIF

* Grab default value
IF VARTYPE(aControl[1].nEmphasisColor) = "N"
   nColor = aControl[1].nEmphasisColor
ELSE
   nColor = 0
ENDIF

nColor = GETCOLOR(nColor)

aControl[1].nEmphasisColor = nColor

RETURN
```

The first section of code uses ASELOBJ() to get an object reference to the selected control. If no control is selected, ASELOBJ() is called again to get a reference to the form or class. The second section of code calls GETCOLOR(), passing in the current setting for nEmphasisColor. The final part of the code is the single line that sets the nEmphasisColor property of the selected control, form or class to the value specified.

Using a form in a property editor

While VFP offers a lot of functions, such as GetColor(), GetPict(), GetFile(), GetDir() and InputBox(), that make it easier to specify property values, some properties need more complicated input. The Anchor Editor demonstrates that a custom form may be the appropriate mechanism for specifying a given property. The Anchor property is specified by adding together a number of values based on individual bit settings within a single byte. Another value created by summing bits is the second argument to the MessageBox() function. A form that makes it easy to specify the parts of the value and adds them together is handy. Figure 3 shows such a form.

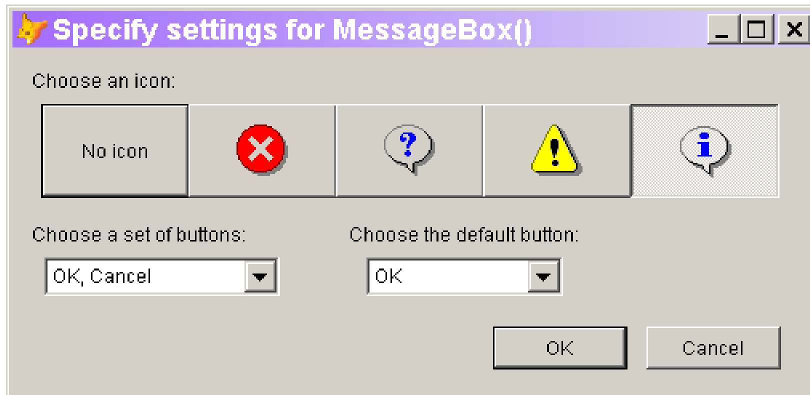

Figure 3. MessageBox() settings—Specifying the second parameter for MessageBox() is much easier with a form like this.

If you have a property to hold that parameter value, then it's convenient to call on the form as a property editor. Here's the code—of course, in a production setting, you need to make sure the form is either in the VFP path or that the reference to the form is fully pathed:

```
LOCAL aControl[1], nParam

IF ASELOBJ(aControl) = 0
   IF ASELOBJ(aControl, 1) = 0
      RETURN
   ENDIF
ENDIF

DO FORM MessageboxParams WITH aControl[1].nMessageBoxParams TO nParam

aControl[1].nMessageboxParams = nParam
```

This code has the same three parts as the previous example: identifying the control to modify, getting input from the user, and storing the result to the specified property. (The form MessageBoxParams.SCX is included on this month's Professional Resource CD, along with an example form, PropertyEditors.SCX, that includes the nMessageBoxParams property.)

Setting multiple properties

A property editor isn't restricted to setting a single property. You can use one property editor to set multiple, related properties. For example, using the property editor for FontCharSet lets you also set the FontName, FontSize, FontBold and FontItalic properties at the same time. (This property editor is actually built into the VFP engine, but you could build it using the GetFont() function, if it weren't.)

Another pair of properties that's related is Height and Width. Often, you can set these just by dragging, but sometimes you need to resize a control or form and want to be sure to keep the new size proportional to the current size. Figure 4 shows a property editor that lets you specify Height and Width with a spinner. If the checkbox is checked, changing one changes the other proportionally.

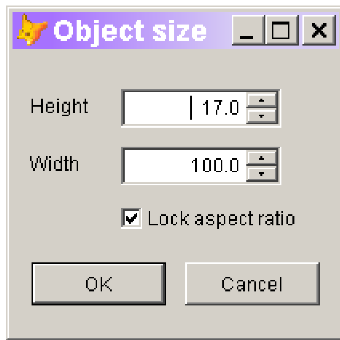

Figure 4. This property editor lets you change the height and width of a control proportionally.

This property editor (included on this month's PRD as PropEditSize.SCX) has more code than the previous examples, but the key code is quite similar. The Load method of the form grabs a reference to the selected control or form and saves it in a form property. The Click method of the OK button calls a custom method, UpdateSize, that saves the new Height and Width values to the control or form. One big difference between this example and the last one is that this form is designed to work only as a property editor. The MessageBox() parameters form could be called as a standalone form in other contexts.

This property editor should be called from both the Height and Width properties of controls that use it. All it takes is one line of code as the Script:

```
DO FORM PropEditSize && add path, if necessary
```

This month's PRD also contains PropertyEditors.VCX, a class library that holds imgEditSize. The class's _MemberData is configured to use the PropEditSize form as the property editor for both Height and Width.

Making property editors generally available

The previous example shows one way to make a property editor available for more than a single form. When you set up a property

editor for a class, it applies not only to that class and its instances, but to any subclasses, as well. That works well when you have a property that's specific to a single class hierarchy.

In many cases, though, you want to use the same property editor for every class or form in which the given property appears. For example, the property editor for Height and Width in Figure 4 is generally useful.

You can make a property editor available across the board by storing it in your IntelliSense table, which is referenced by the _FoxCode system variable. By default, the table is called FoxCode.DBF and is found in your application data directory (referenced by HOME(7)). The easiest way to get a look at the IntelliSense table is using the IntelliSense Manager (Tools | Intellisense Manager on the menu); click Browse on the General Page. Property Editor records in the IntelliSense table have a type of "E".

The MemberData Editor (shown in Figure 2) offers a simple way to add a property editor to the IntelliSense table. Choose the Global option for Scope, then check Has MemberData and specify the script. When you leave the MemberData Editor, the appropriate record is added to the IntelliSense table.

Once you specify a global property editor for a given property, it's available to any form or class that includes that property. However, you can override it in a particular form or class. With the MemberData Editor pointing at the property, choose the Object option for Scope and check Has MemberData. To use a different property editor for the property in this form or class, specify a different script. To simply turn off the property editor, check the Override checkbox; be aware that doing so turns off all global settings for this property, not just the property editor. That is, if the specified property has a display name or was set for the Favorites page, checking Override overrides those settings as well, so you can easily end up eliminating functionality you previously added. You can work around this problem by specifying an "empty" property editor (such as RETURN "").

Making property editors generic

Some property editors can apply to more than one property. For example, you might want to use the script that calls the Color Picker for a variety of custom color-related properties. (You don't need it for the native color properties because they already call on the Color Picker.) The problem is that the code sets the property directly, but

there's no ASelObj() analogue to tell us what property invoked the property editor.

Fortunately, the Fox team provided a way to do this, though it's a little clumsy. A new method, called RunPropertyEditor, was added to the IntelliSense engine. This method receives a parameter, to which you can pass the name of the relevant property. You can use RunPropertyEditor to run code in a type "S" (script) record in the IntelliSense table. The parameter received by RunPropertyEditor is passed along to the script code it runs.

VFP 9 includes a property editor for the Caption property that calls the InputBox() function and sets Caption to the user's entry. This property editor uses the RunPropertyEditor mechanism.

The IntelliSense table that comes with VFP 9 includes a type "E" record for Caption; the key fields are shown in Table 1. The Tip field includes the script attribute:

```
DO (_CODESENSE) WITH 'RunPropertyEditor','','caption'
```

_CODESENSE is a pointer to the IntelliSense application. When you call that application, passing a method name as the first parameter, it executes the specified method, passing on the third through sixth parameters to the method. (In the code above, there's only one parameter to pass to the RunPropertyEditor method.)

Table 1. Using a generic property editor—The IntelliSense table in VFP 9 contains this record to provide a property editor for the Caption property.

Field	**Value**
Type	"E"
Abbrev	"Caption"
Cmd	"{CaptionScript}"
Tip	"<VFPData><memberdata name="caption" type="property" favorites="True" script="DO (_CODESENSE) WITH 'RunPropertyEditor','','caption'"/></VFPData>"
Data	""

The RunPropertyEditor method finds the relevant record in the IntelliSense table—the one with Type="E" and Abbrev equal to the parameter it receives. If the Cmd field in that record contains the name of a script (as in Table 1), the method looks for a record with the specified script name and Type="S" and executes the contents of the Data field of that record. If the Cmd field of the Type "E" record doesn't contain a script name (no curly braces), RunPropertyEditor executes whatever code is found in the Data field of the Type "E" record. Whichever code it runs, RunPropertyMethod passes on the parameter it received.

You can see all this for yourself by examining the appropriate records near the end of the IntelliSense table that comes with VFP 9 and the code in FoxCode.PRG. To find the code, unzip the file XSource.ZIP found in Tools\XSource of your VFP 9 installation. After unzipping, FoxCode.PRG is in Tools\XSource\VFPSource\FoxCode.

The CaptionScript record in the IntelliSense table is shown in Table 2. The script actually isn't quite as generic as you might like, since it sets the caption of the InputBox() to "Caption Property Editor." If you want to use it more generally, you'll probably want change that. It's also worth noting that this script is set up to handle multiple selected objects.

Table 2. A generic property editor—This record in the IntelliSense table specifies a generic property editor for strings. It receives the name of the property to modify as a parameter.

<table>
<tr><th>Field</th><th>Value</th></tr>
<tr><td>Type</td><td>"S"</td></tr>
<tr><td>Abbrev</td><td>"CaptionScript"</td></tr>
<tr><td>Cmd</td><td>"{}"</td></tr>
<tr><td>Tip</td><td>""</td></tr>
<tr><td>Data</td><td><pre>#DEFINE   IBOX_CAPTION "Caption Property Editor"
#DEFINE   IBOX_TEXT   "Enter value for property: "
#DEFINE   USER_CANCEL   "__usercancelled__"

LPARAMETERS tcProp
LOCAL ARRAY laObjs[1]
LOCAL lcRetVal, lnCnt, loCtl,lcDefValue, lnSuccess
IF ASELOBJ(laObjs)=0
       IF ASELOBJ(laObjs,1)=0
              RETURN</pre></td></tr>
</table>

```
		ENDIF
	ENDIF
	lcDefValue=IIF(ALEN( laObjs,1)=1,laObjs[1].&tcProp,"")
	lcRetVal=INPUTBOX(IBOX_TEXT + tcProp, IBOX_CAPTION, lcDefValue, 0,
	"", USER_CANCEL)
	IF lcRetVal==USER_CANCEL
		RETURN
	ENDIF
	FOR lnCnt = 1 TO ALEN( laObjs,1)
		loCtl = laObjs[lnCnt]
		IF PEMSTATUS( loCtl, tcProp, 5 )
			loCtl.&tcProp = lcRetVal
		ENDIF
	ENDFOR
```

To use this property editor for another property (say Name), we need to add a type "E" record for the property. While you can add the record using the MemberData Editor by specifying global scope, that tool doesn't provide a way to set the Cmd field, so once you've added the record, you need to open the IntelliSense table and point the new record to the appropriate script record, CaptionScript.

Get it right

Custom property editors mean you can create classes and increase the chances of their being used correctly. While this might seem most important in a team environment, where one developer creates classes used by other developers, the truth is when you return to your own classes after some time has elapsed, it can be hard to remember the correct range of values or even data type for some properties. With a custom property editor, that information is carried right in the class.

FoxPro's open architecture has proved useful over and over. Property editors are just one more example of the Fox team handing us the ability to make using VFP easier.