Session VFP222

Customizing the Property Sheet

By Tamar E. Granor

Since its introduction in VFP 3, the Property Sheet has seen mostly minor changes. Each new version brought a few tweaks. VFP 9 brings significant changes to this tool, introducing a Favorites tab, and most importantly, a mechanism to let you customize behavior.

The ability to customize Property Sheet contents is tied to a new, optional, property, _MemberData, which contains an XML string. You can set _MemberData manually or use the new MemberData Editor.

What kind of things can you do with _MemberData? First, _MemberData controls the Favorites tab. You can indicate whether each property, event, or method (PEM) of a class or form should be on the Favorites tab. _MemberData also lets you specify capitalization for custom PEMs. In earlier versions of VFP, once you add a property or method, it appears in lower-case on the Property Sheet and in IntelliSense. With _MemberData, you can specify the way it appears. Finally, for properties, you can specify a “property editor” to use for entering the property’s initial value.

Unlike other properties, _MemberData is not included in new forms and classes automatically. You have to add it. (This is actually analogous to the Builder property you can add to a class to specify a custom builder.) However, the MemberData Editor can add it for you automatically.

The structure of _MemberData

_MemberData is an XML string containing VFP data (in the format created by CursorToXML()). Between the <VFPData> and </VFPData> tags, there’s one element, called memberdata, for each PEM with customization specified. The element has an attribute for each customization item for that PEM. For example, here’s the _MemberData string for a form with a custom property, lFlag, and a custom method, MyMethod, both capitalized as shown here, and with the AutoCenter property added to the Favorites page:

```
<VFPData><memberdata name="autocenter" type="property"
favorites="True"/><memberdata name="lflag" type="property"
display="lFlag"/><memberdata name="mymethod" type="method"
display="MyMethod"/></VFPData>
```

The definition for the memberdata element includes six attributes, as shown in **Table 1**. The schema is open, however, so you can add custom attributes and use them at design-time or runtime. Note that capitalization is significant in the _MemberData string, both for the attribute names and for True/False values. It appears not to matter for the value of the type attribute.

Table 1. The memberdata element for each property, event, or method includes a subset of these attributes.

Attribute	**Values**	**Purpose**
name		Contains the name of the PEM. Required.
type	"property" "event" "method"	Contains the type of PEM. Required.
display		Contains the name as it should display in the Property Sheet and with IntelliSense. Must be the same string as the name; only the capitalization can vary. Applies only to custom PEMs.
favorites	"True" "False"	Indicates whether the PEM is included on the Favorites page.
override	"True" "False"	Indicates whether settings unspecified at this level should be inherited from the parent class (False) or draw their values from the default settings for this PEM (True).
script		Contains VFP code to run as a Property Editor. (See "Creating property editors" later in this document.)

Be aware that the Property Sheet is sorted alphabetically using a case-sensitive sort. That's why custom properties, which normally display in lower-case, have always been listed at the bottom. If the display attribute you specify for a PEM begins with an upper-case letter, that PEM will be sorted in with the native PEMs.

The behavior of the override attribute is explained in "Inheriting _MemberData" later in this document.

Setting attributes globally

In addition to customizing a particular PEM in a particular class, you can specify that some customization applies to a PEM in every form or class with that PEM. For example, by default, the Caption and Anchor properties appear on the Favorites page.

Information about global customization is not stored in the _MemberData property of individual forms or classes, but in the IntelliSense table (referenced by the _FoxCode system variable and by default, FoxCode.DBF in the directory indicated by HOME(7)). PEM customization items have "E" in the Type field, the name of the PEM in the Abbrev field, and a MemberData string in the Tip field. For example, the default IntelliSense table contains a record for the Caption property, with the values shown in **Table 2**.

Table 2. You can specify custom behavior globally by adding records to the IntelliSense table. Here, the Caption property is added to the Favorites page and a custom property editor is specified.

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

You're not stuck with the global customization for a PEM; to override it in a particular form or class, just change it in the _MemberData string for that PEM in the form or class.

Inheriting _MemberData

_MemberData can be inherited either through the inheritance hierarchy or the containership hierarchy. The rules make sense, but they're different from those for other properties. Here's the order VFP uses to search for a memberdata attribute. Once it finds a value for a particular attribute, VFP stops searching for that attribute.

1. The _MemberData property of the object itself.
2. The _MemberData property of classes in the inheritance hierarchy for the object, working upwards in the normal way.
3. The _MemberData property of any containers, working upwards through the containership hierarchy.
4. Any global memberdata settings, stored in the IntelliSense table.

For a given PEM, each attribute may be found in memberdata at a different point in the list.

Although memberdata can be "inherited" through the containership hierarchy, you can't add the _MemberData property to an object once you put it on a form or class. (That's because you can't ever add a property to a contained object.) However, if a contained object already has a _MemberData property, you can edit it for the instance on the form or class.

The override attribute lets you change the search behavior. For attributes specified at the object level, it's irrelevant; they have whatever value you assign. Override applies only to attributes you don't specify at the object level, and determines where those attributes get their values. When override is set to True, VFP doesn't search the two hierarchies for the attribute; it just uses the default setting for this PEM.

For example, consider a command button class (cmdTopLevel) with two custom properties: cMyFirstProp and cMySecondProp. In the button class, both have display and favorites attributes. The MemberData XML for the class is:

```
<VFPData>
<memberdata name="cmyfirstprop" type="property" display="cMyFirstProp"
favorites="True"/>
<memberdata name="cmysecondprop" type="property"
display="cMySecondProp" favorites="True"/>
</VFPData>
```

With these settings, both properties appear on the Favorites page and they're capitalized as you'd want them.

Now consider a subclass (cmdMiddleLevel). Suppose you set the override attribute for cMySecondProp to True, but also specify the display attribute. The MemberData XML for this class is:

```
<VFPData>
<memberdata name="cmysecondprop" type="property"
display="cMySecondProp" override="True"/>
</VFPData>
```

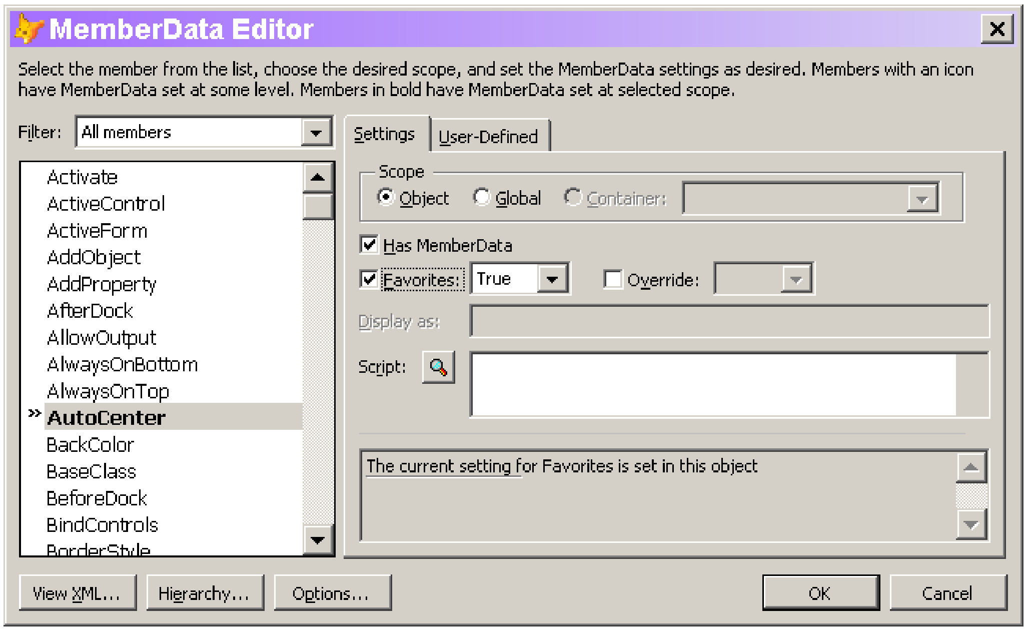
There's nothing specified for cMyFirstProp, so it draws its behavior from the parent class. That means cMyFirstProp appears on the Favorites page (and the property is capitalized as cMyFirstProp). However, with override set to True, cMySecondProp uses only the settings provided at this level. So, it doesn't appear on the Favorites page, though it's still properly capitalized, because that attribute is specified for this class.

Now consider a subclass of cmdMiddleLevel, called cmdBottomLevel. If you change none of the attributes at this level (that is, specify nothing for _MemberData), cMyFirstProp inherits its behavior from cmdTopLevel, so it appears on the Favorites page. cMySecondProp inherits from cmdMiddleLevel and does not appear in Favorites. The session materials include Chapter2.VCX, a class library that contains the cmdTopLevel, cmdMiddleLevel, and cmdBottomLevel classes described here.

The MemberData Editor

Once you understand the structure of MemberData, it's not hard to specify it, but it's tedious enough that you're not likely to do so. That's especially true for global member data, where you have to modify the IntelliSense table.

Fortunately, there's an alternative. VFP 9 includes a new tool called the MemberData Editor, available from the Form or Class menu. The MemberData Editor (**Figure 1**) handles all the heavy lifting involved in adding the _MemberData property and populating it.



MemberData Editor

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.

Filter: All members

Activate
ActiveControl
ActiveForm
AddObject
AddProperty
AfterDock
AllowOutput
AlwaysOnBottom
AlwaysOnTop
» **AutoCenter**
BackColor
BaseClass
BeforeDock
BindControls
BorderStyle

Scope: [✓ Object] Global Container:

[✓] Has MemberData

[✓] Favorites: True   [ ] Override:

Display as:

Script:

The current setting for Favorites is set in this object

Figure 1. The MemberData Editor makes it easy to customize the Property Sheet.

The MemberData Editor lists all PEMs of the form or class in alphabetical order. PEMs with customization are shown in bold (like AutoCenter in Figure 1). You can limit the display in the list using the Filter dropdown. Filters include custom members only, custom members added in this class, native members only, and favorites only.

To customize a PEM, choose it in the list and check the Has MemberData checkbox. Next, specify the attributes you want. **Table 3** shows the relationship between the controls in the MemberData Editor and the _MemberData attributes.

Table 3. The items in the MemberData Editor map to the attributes of the _MemberData string.

Control(s)	**Attribute**	**Notes**
Object/Global/Container	None	Determines whether the settings for this PEM are stored in the local _MemberData property (Object), the IntelliSense table (Global), or in the container's _MemberData property (Container).
Has member data	None	Indicates the PEM has member data. Provides a one-click way to remove all customization for a PEM.
Favorites checkbox and dropdown	favorites	Determines whether this PEM appears on the Favorites page. The checkbox indicates whether the memberdata element for this PEM has the favorites attribute. The dropdown specifies the setting for that attribute.

Control(s)	Attribute	Notes
Override checkbox and dropdown	override	Determines whether unspecified settings at this level use the inherited settings or the defaults. The checkbox indicates whether the memberdata element for this PEM has the override attribute. The dropdown specifies the setting for the attribute.
Display as	display	Specifies the capitalization for this PEM.
Script	script	Specifies the code for a property editor for this property. (See “Creating property editors” later in this chapter.)

If a control on a form or class is selected when you open the MemberData Editor, the tool attempts to edit _MemberData for that control. If the control’s class doesn’t have a _MemberData property, you get a warning, after which the MemberData Editor opens showing the PEMs for the object, but saving any changes you make to the _MemberData property of the containing form or class.

Be aware that “global” here really means global. If you change any settings when scope is set to global, the corresponding record in the IntelliSense table is added, modified or deleted and your customizations change or disappear for every form and class with the specified PEM.

The Description Pane (the disabled editbox near the bottom) shows you the current settings for the selected PEM. It indicates, for each attribute with memberdata at some level, which setting is in control. Figure 1 shows the simplest case, with the attribute set at the local level. The listing can also indicate a global setting, a setting drawn from a container (in which case, the container is named), and a setting inherited from a parent class (in which case, the parent class is named).

Two buttons on the MemberData Editor also help you see exactly what settings apply. Click View XML to see the string that will be stored to the object’s _MemberData property. Click Hierarchy… for a detailed look at each level in the inheritance and containership hierarchy that affects the currently selected PEM, as well as the result, the settings that will be used for that PEM. **Listing 1** shows the result for the cMySecondProp property of the cmdMiddleLevel class in Chapter2.VCX.

Listing 1. When you click Hierarchy… in the MemberData Editor, you see each setting that has an effect on the selected PEM’s memberdata.

```
Class - cmdmiddlelevel
<memberdata name="cmysecondprop" type="property"
display="cMySecondProp" override="True"/>
  Class - cmdtoplevel of chapter2.vcx
<memberdata name="cmysecondprop" type="property"
display="cMySecondProp" override="True"/>
Effective settings:
```

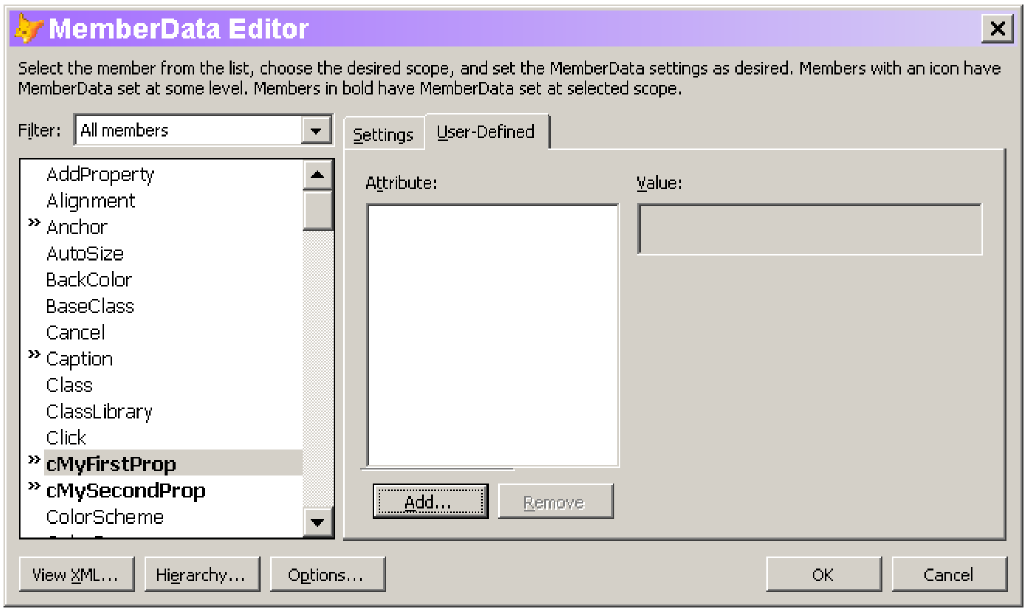
```
<memberdata name="cmysecondprop" type="property"
display="cMySecondProp" override="True"/>
```

When you click the MemberData Editor's OK button and at least one PEM has local customization specified, the MemberData Editor checks whether the current form or class has a _MemberData property, adds it if necessary, and generates the appropriate string. At the same time, any global customization specified is handled by adding or modifying records in the table specified by _FoxCode.

Adding custom attributes

In addition to specifying the standard memberdata attributes, the MemberData Editor lets you define and specify your own attributes. While VFP itself won't use your custom attributes, they give you a place to store information (even code) for a class and have that information always present whatever you do with the class.

Use the user-defined page of the MemberData Editor (**Figure 2**) to add custom attributes. The page is available for a PEM only if Has MemberData is checked. Click the Add button to add an attribute—an input box appears for you to specify the name. Once you do that, use the Value textbox to provide the attribute's value.


MemberData Editor

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.

Filter: All members

AddProperty
Alignment
» Anchor
AutoSize
BackColor
BaseClass
Cancel
» Caption
Class
ClassLibrary
Click
» **cMyFirstProp**
» **cMySecondProp**
ColorScheme

Figure 2. The User-Defined page of the MemberData Editor allows you to add custom attributes, which you can use to ensure information travels with a class.

What can you do with custom attributes? Here's one example. While you can specify a property editor for a custom property (see "Creating Property Editors" later in this chapter), you can't force people to use it. So imagine having an attribute called Valid that contains an expression determining the validity of the property's value. Of course, you need code to look for the attribute and evaluate the expression—you could put such code in a project hook. Here's a simple example of code to do so in the BeforeBuild method:

```
LPARAMETERS cOutputName, nBuildAction, lRebuildAll, ;
            lShowErrors, lBuildNewGuids
```

```
#DEFINE CRLF CHR(13) + CHR(10)

LOCAL oFile, aObj[1], cMembData, lFileResult, lResult
LOCAL cProblems

lResult = .T.

FOR EACH oFile IN This.oProject.Files
  DO CASE
  CASE INLIST(oFile.Type, "K", "V")
    * Open without running code
    IF oFile.Type="K"
      MODIFY FORM (oFile.Name) NOWAIT
    ELSE
      MODIFY CLASS (oFile.Name) NOWAIT
    ENDIF

    cProbs = "Validity checking " + oFile.Name + CRLF

    * Grab memberdata
    ASELOBJ(aObj, 1)
    IF PEMSTATUS(aObj[1],"_MemberData",5)
      cMembData = aObj[1]._MemberData
      * Convert to cursor
      XMLTOCURSOR(cMembData, "__MembData")
      * Look for Valid specs
      SELECT __MembData
      IF TYPE("__MembData.Valid") <> "U"
        * At least one item has a Valid attribute, so process it.
        lFileResult = .T.
        SCAN
          IF NOT EMPTY(__MembData.Valid)
            cValidExpr = __MembData.Valid
            * Substitute for "This"
            cValidExpr = STRTRAN(cValidExpr, "This", "aObj[1]")
            IF NOT EVALUATE(cValidExpr)
              cProbs = cProbs + "Failed test: " + ;
                       __MembData.Valid + CRLF
              lFileResult = .F.
            ENDIF

          ENDIF
        ENDSCAN
      ENDIF
    ENDIF

    * Close the form or class
    OTHERWISE
      * Do nothing
  ENDCASE

  cProbs = cProbs + CRLF
  lResult = lResult AND lFileResult
ENDFOR

IF NOT lResult
```

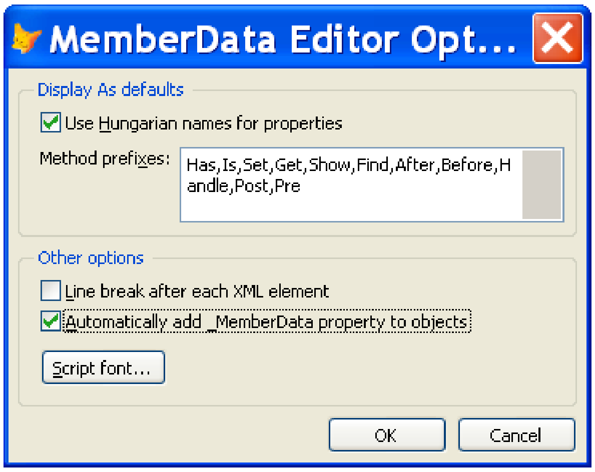
```
  MESSAGEBOX(cProbs)
ENDIF

RETURN lResult
```

This simple version doesn't drill down to check for a Valid attribute at another level of the inheritance or containership hierarchy, but should give you a sense of the possibilities offered by custom attributes. This project hook is included in Chapter2.VCX in the session materials. The materials also contain a project and form (UseCustomAttr.PJX and UseCustomAttr.SCX) to demonstrate its use.

Customizing the MemberData Editor

The Options button on the MemberData Editor lets you control several aspects of the tool's behavior. When you click it, the MemberData Editor Options dialog (**Figure 3)** opens. The settings there are remembered between invocations of the MemberData Editor, and between classes. They're stored in the Resource table (by default, FoxUser.DBF in the directory specified by HOME(7)); the relevant record's ID field contains "MEMBERDATAED" while the Name field is "MemberDataEditor".


MemberData Editor Opt...

Display As defaults

☑ Use Hungarian names for properties

Method prefixes: Has,Is,Set,Get,Show,Find,After,Before,Handle,Post,Pre

Other options

☐ Line break after each XML element

☑ Automatically add _MemberData property to objects

Figure 3. You can control the behavior of the MemberData Editor using the settings in the Options dialog.

The Display As defaults section simplifies the task of setting capitalization through the display attribute. When Use Hungarian names for properties is checked and you check Has Member Data for a property, its name is shown in the Display as textbox with the first letter in lowercase and the second letter in uppercase; this is useful for developers who use the first letter of a property name to specify the type and begin the actual name with the second character.

It's not unusual for many method names to begin with one of just a few words. Method prefixes lets you specify a list of strings that should be seen as words at the beginning of a method and capitalized appropriately. That is, when a method name begins with one of the specified strings, the suggested Display as value capitalizes the first letter of that string and the first letter following the string. For example, using the default settings (shown in Figure 3), the suggested capitalization for a method named isnumeric would be "IsNumeric".

The Other options section lets you control several types of appearance. If you check the Line break after each XML element checkbox, the XML created for _MemberData puts the memberdata element for each PEM on a separate line in the XML string. For example, with this setting checked, the _MemberData string shown in "The structure of _MemberData" section of this document would appear as:

```
<VFPData>
<memberdata name="lflag" type="property" display="lFlag"/>
<memberdata name="mymethod" type="method" display="MyMethod"/>
</VFPData>
```

Newly created forms and classes don't have a _MemberData property. The MemberData Editor adds it as needed. However, you may prefer for each form or class to have the property; if so, check Automatically add _MemberData property to objects. When that item is selected, a record is added to your IntelliSense table that automatically adds the _MemberData property to each form or class as you create or open it.

Finally, the Script font button lets you set the font used for the Script editbox on the Member Data page. Note that the font you choose isn't used for the window that opens when you click the Zoom button for the script; that's controlled by your setting for PRG files on the IDE tab of the VFP Options dialog.

Replacing the MemberData Editor

Like many other tools in the VFP development environment, the MemberData Editor is written in VFP. That means you can replace it with your own tool if you prefer.

However, the mechanism for replacing the MemberData Editor is different than for other tools written in VFP. Rather than providing a system variable (like _GenMenu or _CodeSense), the MemberData Editor is hooked into the Builder system. The table that drives the built-in Builder system (by default, Builder.DBF in the Wizards directory) contains a record for the MemberData Editor that points to MemberDataEditor.APP in the VFP home directory.

If you prefer to use another MemberData Editor, you have several choices. Once you create or acquire another application to serve this purpose, you can put it in the VFP home directory and name it MemberDataEditor.APP. (Of course, if you choose to name your replacement MemberDataEditor.APP, you should probably save a copy of the version that comes with VFP 9.) Alternatively, you can modify the record in the Builder table to point to the application you want to use. A third choice is to add another record to Builder.DBF for your editor; and then, when you invoke the MemberData Editor, you're prompted to choose between the editor provided and your custom version.

Finally, as with the other VFP tools written in VFP, the source code for the MemberData Editor comes with VFP. (Look in the XSource.ZIP file located in VFP's Tools\XSource directory.) If you just want to make minor changes, your best bet may be to modify the source code and build a custom version of the tool.

Playing favorites

While you can add PEMs to the Favorites tab using the MemberData Editor, there's actually an easier way. Right-click any PEM in the Property Sheet and choose Add to Favorites. Doing so automatically generates the appropriate MemberData string, adding the _MemberData property, if necessary.

Removing a PEM from the Favorites page isn't quite as easy; there's no Remove from Favorites item on the shortcut menu. You have to edit the _MemberData string directly or use the MemberData Editor.

Creating property editors

While a Favorites page and displaying the names of custom PEMs as you want them are both useful, the truly exciting feature enabled by _MemberData is the ability to create custom property editors.

A number of VFP's built-in properties let you choose a value rather than simply typing it in. For example, the various color properties (such as BackColor) use the Color Picker, while the Icon and Picture properties use a special version of the Open dialog (the same as the GetPict() function). You invoke these editors by double-clicking the property's current value or by clicking the ellipsis (...) button next to the textbox (called the "Property settings box" in Help) in the Property Sheet. In VFP 9, you can create your own dialogs or call on built-in dialogs for any property. You can even create property editors with no user interface.

VFP 9 includes two property editors, both defined globally in FoxCode.DBF for the relevant property. The first uses the InputBox() function to let you specify a Caption. It's designed so you can use it for other properties as well. (See "Using IntelliSense for property editors" later in this document.)

The second Property Editor is for the new Anchor property. Anchor requires a numeric value, computed by adding the values of the appropriate settings. The Property Editor (AnchorEditor.App in the VFP home directory) lets you choose the settings you want and test them.

A Property Editor is, essentially, a builder, though it's generally focused on one or a few properties, where a builder addresses many properties of a control. Like a builder, a Property Editor has to do the heavy lifting involved in setting properties at design-time in its code. It receives no parameters and must figure out what object it's addressing and what property it's intended to change. (For global property editors, IntelliSense offers a somewhat smarter alternative; see "Using IntelliSense for property editors" later in this document.)

Use the ASELOBJ() function to figure out which object you're working on. There's one complication; if the property belongs to a form or container class rather than a control, ASELOBJ() doesn't find a selected object. In that case, you need to call it again, passing 1 for the optional second parameter, so it can find the form or container. This code finds the selected control, if there is one, and the form or container, if no control is selected. If it can't find either, it gives up:

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

After executing this code (if it doesn't issue RETURN), aControl[1] contains an object reference to the selected control, form, or container. (In fact, if multiple objects are selected, the array has one row for each.)

Unfortunately, there's no generic way to figure out which property called the Property Editor, so you have to hard code the property name. **Listing 2** shows a Property Editor for a custom nEmphasisColor property; it brings up the Color Picker dialog.

Listing 2. This Property Editor lets you use the Color Picker to choose a value 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
```

Clearly, a Property Editor is most useful for items that can't be easily typed in, like RGB color values, or that are hard to figure out, like Anchor values. Another such item is the second parameter to the MessageBox() function, which specifies the icon and buttons to use. **Figure 4** shows a form that lets you make your choices and computes the value to pass. A Property Editor that calls on this form is shown in **Listing 3**; it's connected to a property called nMessageBoxParam. (Of course, if the form you're working on is in a different directory than the Message Box settings form, the DO FORM command needs to include the appropriate path.)

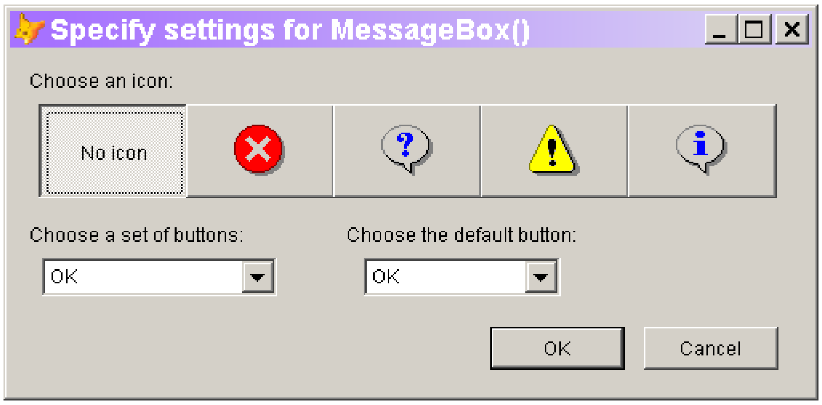


Figure 4. Calculating the second parameter for MessageBox() is complicated. A Property Editor like this one makes it much easier.

Listing 3. Using the form in Figure 4 as a Property Editor is simple.

```
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
```

You can also use a property editor to restrict users to a limited set of values. For example, you can double-click on built-in logical properties to switch them between True and False. Listing 4 shows a property editor that offers the same behavior for a custom property called lToggleMe.

Listing 4. This property editor toggles the value of a logical property.

```
LOCAL aControl[1], nParam

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

aControl[1].lToggleMe = NOT aControl[1].lToggleMe
```

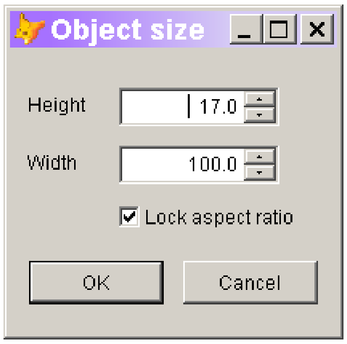
You could extend this idea to let a user double-click through a list of specified values, as they can for built-in properties like WindowState.

The session materials include MessageBoxParams.SCX (the form in Figure 4) and PropertyEditors.SCX, a form that demonstrates the Color Picker Property Editor, the Property Editor for MessageBox() parameters and the Toggle Property Editor.

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 5** 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.



Height 17.0
Width 100.0
☑ Lock aspect ratio

Figure 5.This property editor lets you change the height and width of a control proportionally.

This property editor (included in the session materials 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
```

The session materials include 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. The image on PropertyEditors.SCX is based on imgEditSize, so you can test this property editor.

Using IntelliSense for property editors

Defining a Property Editor every time you need it would get tedious enough that you'd do it only for the most complicated properties. Fortunately, the VFP team included an alternative approach that takes advantage of the existing IntelliSense script system.

In addition to the new “E” record in the FoxCode table to support global _MemberData, the IntelliSense engine was enhanced with a method called RunPropertyEditor; this method lets you execute the code in a script record (type “S”) as a Property Editor.

The Property Editor for the Caption property uses this mechanism, so we’ll examine it to see how it works. The “E” record for Caption has “{CaptionScript}” in the Cmd field. The Tip field contains:

```
<VFPData><memberdata name="caption" type="property" favorites="True"
script="DO (_CODESENSE) WITH 'RunPropertyEditor','','caption'"/>
</VFPData>
```

The key item there is the script attribute, with the value:

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

This line runs the IntelliSense engine, telling it to execute its RunPropertyEditor method and pass “caption” to that method. RunPropertyEditor finds the type “E” record in FoxCode corresponding to the parameter it receives. If the Cmd field of that record contains the name of a script record, the method locates that script record and executes the contents of the script record’s Data field. For the Caption record, therefore, RunPropertyEditor looks for a record in FoxCode with Type = “S” and Abbrev = “CaptionScript”. There is such a record; its Data field contains the code in **Listing 5**, which receives the property name as a parameter, finds all selected controls (or the form or container if no control is selected), prompts the user for a new caption, and then assigns the new value to the specified property of each selected object.

Listing 5. This script (which comes with VFP 9) uses the InputBox() to prompt for a new string value.

```
#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
  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
```

This two-record architecture makes it easy to use a single Property Editor (defined in a type “S” record) for many different properties. For example, to use the CaptionScript for the Name property, add a type “E” record for Name with “{CaptionScript}” in the Cmd field and specify this line as the script for Name (using the MemberData Editor):

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

Note that the line of code is identical to the one used for Caption, except for the final parameter.

You can apply the same mechanism to other property editors. For example, you might want to make the MessageBox() parameters Property Editor available as a script. To do so, first add a record to the FoxCode table, with the values in **Table 4** (remembering to provide a proper path to the form).

Table 4. To create a new script record for the MessageBox() parameters, add a record to FoxCode with these settings.

Field	**Value**
Type	“S”
Abbrev	“MessageScript”
Cmd	“{}”

Data	`LPARAMETERS tcProp`
	
	`LOCAL ARRAY laObjs[1]`
	`LOCAL lnRetVal, lnCnt, loCtl,lnDefValue, lnSuccess`
	`IF ASELOBJ( laObjs)=0`
	`  IF ASELOBJ( laObjs,1)=0`
	`    RETURN`
	`  ENDIF`
	`ENDIF`
	
	`lnDefValue=IIF(ALEN( laObjs,1)=1,laObjs[1].&tcProp,0)`
	`lnRetVal =0`
	`DO FORM "MessageBoxParams.SCX" ;`
	`   WITH lnDefValue to lnRetVal`
	
	`FOR lnCnt = 1 TO ALEN( laObjs,1)`
	`  loCtl = laObjs[lnCnt]`
	`  IF PEMSTATUS( loCtl, tcProp, 5 )`
	`    loCtl.&tcProp = lnRetVal`
	`  ENDIF`
	`ENDFOR`

Be aware that debugging IntelliSense scripts is difficult. If a script contains any compile-time errors, it will fail to run without any messages. If you’re having trouble getting a script to run, try copying it to a PRG and compiling to find your errors. In addition, tracing may or may not work with property editors; in some cases, issuing SYS(2030,1) prior to running your Property Editor may work.

Once you have the script record set up, add a record to FoxCode for each property name you want to use this script. **Table 5** shows an example, for a property called nMessageParam. You can set up type “E” records that use the same script for as many properties as you want. Unfortunately, there doesn’t appear to be a way to call on a script record without adding a type “E” record; it would be handy to be able to create MemberData at the class level (ideally, using the MemberData Editor) that calls RunPropertyEditor to use a property editor defined in the IntelliSense table.

Table 5. To use the MessageBox() parameters script, add a record like this to FoxCode.

Field	**Value**
Type	“E”
Abbrev	nMessageParam
Cmd	{MessageScript}

Tip	`<VFPData><memberdata name="nmessageparam" type="property" display="nMessageParam" script="DO (_CODESENSE) WITH 'RunPropertyEditor','','nmessageparam'`
	`"/>`
	`</VFPData>`

The session materials include AddMessageScript.PRG, a program that adds the records needed for the MessageBox() parameter Property Editor to the IntelliSense table.

The bottom line

FoxPro developers spend a lot of their time working in the Property Sheet. With VFP 9, working there is easier and more configurable. We expect the FoxPro community to create and make available a number of property editors that will smooth your way through form and class definition.

This document is excerpted from *What's New in Nine: Visual FoxPro's Latest Hits* by Tamar E. Granor, Doug Hennig, Rick Schummer, Toni M. Feltman and Jim Slater, Hentzenwerke Publishing, 2005. My thanks to my co-authors and to publisher Whil Hentzen.