Cool Controls for Your Applications

Doug Hennig
Stonefield Software Inc.
2323 Broad Street
Regina, SK Canada S4P 1Y9
Email: dhennig@stonefield.com
Web sites: www.stonefield.com
www.stonefieldquery.com
Blog: DougHennig.BlogSpot.com
Twitter: [DougHennig](http://twitter.com/DougHennig)

This document examines some controls you can add to your applications to provide a cleaner or fresher user interface. Controls discussed include a splitter, a combobox that drops down to a TreeView control, object-oriented Microsoft Office-like menus, a 100% VFP code date picker control, a control for displaying "balloon tips", and a modern-looking progress bar control.

Introduction

One of the reasons I hear that VFP developers are told to move to .Net is that .Net has controls that provide a newer, fresher look to applications. VFP apps look old, they say. Old fonts like Arial, old colors like the background grey, old icons for buttons, and old-style menus and toolbars. However, there's no reason for that. You can easily use newer, cleaner fonts like Segoe UI (the standard system font in Windows Vista and Windows 7), you can use modern, colorful 32-bit icons (there are hundreds or even thousands of web sites that provide free or paid icons), and you can use some of the projects on VFPX (http://vfpx.codeplex.com) to provide modern-style menus, toolbars, and other graphical user interface elements in your applications.

This document focuses on controls that have a more modern interface than the versions you've probably used in the past. Some of them are VFPX projects while others are classes I or others created. Specifically, we're going to look at a splitter, a combobox that drops down to a TreeView control, object-oriented Microsoft Office-like menus, a 100% VFP code date picker control, a control for displaying "balloon tips", and a modern-looking progress bar control.

Splitter control

Splitters are interesting controls: they may or may not have a visual appearance themselves, but they allow you to change the relative size between two or more other controls by adjusting the size of one at the expense of the other. Splitters appear in lots of places in Windows applications; for example, in Windows Explorer, you can adjust the relative sizes of the left and right panes using a splitter. Splitters can be horizontal (they adjust objects to the left and right) or vertical (they adjust objects above and below), and you can have both types of splitter on the same form.

Splitters are useful when you have multiple resizable controls. For example, **Figure 1** shows a sample form (TestSplitter.SCX) with three editboxes. If the Anchor property of these controls is set, VFP automatically resizes them when the user resizes the form. The problem, though, is that there are competing resizing interests: the two editboxes on the left side of the form should both become taller as the form gets taller, and all three should get wider as the form gets wider. Setting Anchor of all three controls to 15 (resize vertically and horizontally) causes the controls to overlap as the form is resized. Although the Anchor property supports values that resize the control relatively rather than absolutely, I've never been happy with the results. Instead, what we'll do is set Anchor so there's no competition for resizing (the upper left editbox gets wider and taller, the bottom left editbox gets only wider, and the right editbox only gets taller) and let the user decide the relative sizes of the control using splitters.

The sample form shown has two splitters: a vertical one between the two editboxes at the left and a horizontal one between those editboxes and the one at the right. Dragging the vertical splitter up or down changes the heights of the editboxes above and below it. Dragging the horizontal splitter left or right changes the widths of all three editboxes.

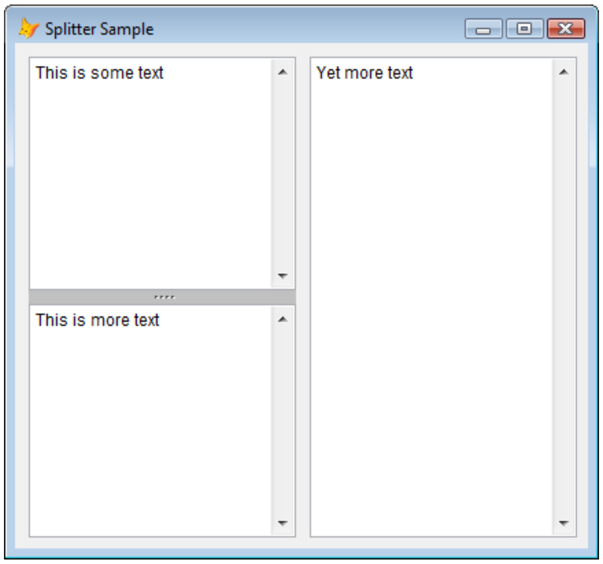


Figure 1. The vertical splitter is visible with a "gripper" but the horizontal splitter is invisible.

There are two styles in which the splitter can appear: visible with a "gripper" or invisible. The vertical splitter in the sample form appears as a grey bar with four dots in the middle (the gripper) while the horizontal one has no visual appearance other than the shape of the mouse pointer changes to an "east-west" arrow when it's over the control. Although the visible splitter is more "discoverable," I feel it also detracts from the appearance of the form. You can decide which style you want simply by setting a property of the splitter control to .T. or .F.

I first wrote about a splitter control in the July 1999 issue of FoxTalk ("Splitting Up is Hard to Do"). That control was fairly complex: it used a couple of collaborating objects and OLE Drag and Drop. The code in the splitter I'm presenting in this document is much simpler yet the control has more capabilities than the older one. Thanks to Matt Slay for creating the gripper control and adjusting the code to use this control in the splitter.

SFSplitter

The classes that make up the splitter control are defined in SFSplitter.VCX. The main class, SFSplitter, is actually an abstract class; you'll use either the SFSplitterH or SFSplitterV subclasses, depending on whether you want a horizontal or vertical splitter.

SFSplitter is based on Container. It contains an instance of the Gripper class, discussed later. SFSplitter has changes to the following properties:

- BackColor: 192,192,192 (grey). BackColor is only used if the lShowGripper property, discussed later, is .T.
- BackStyle: 0-Transparent. This makes the splitter invisible at run time. If you set lShowGripper to .T., the Init method sets this property to 1-Opaque so the BackColor shows.

- BorderColor: 255,0,0 (red) and BorderWidth: 2. These are only used so the splitter has a visual appearance at design time. The Init method changes BorderWidth to 0 so there's no border at run time.

SFSplitter has six custom properties:

- lShowGripper: set this to .T. to make the splitter visible and display the gripper.
- nDots: set this to the number of dots (up to 5) you want in the gripper.
- cObject1Name: the name of the object (it can be a comma-delimited list if there's more than one object) to the left of a horizontal splitter or above a vertical one.
- cObject2Name: the name of the object (again, use a comma-delimited list for multiple objects) to the right of a horizontal splitter or below a vertical one.
- nObject1MinSize: the minimum size for the object(s) named in cObject1Name.
- nObject2MinSize: the minimum size for the object(s) named in cObject2Name.

The Init method of SFSplitter sets up the control so it has the correct run time appearance.

```
with This

* Set BorderWidth to 0 so it doesn't appear at run time.

   .BorderWidth = 0

* If we're showing a gripper image (thanks to Matt Slay for the gripper controls),
* set BackStyle so we can see the color and set up the gripper.

   if .lShowGripper
      .BackStyle = 1
      .SetupGripper()
   endif .lShowGripper

* Call AdjustMinimum to adjust the form so it can't be sized too small.

   .AdjustMinimum()
endwith
dodefault()
```

SetupGripper sets up the gripper control if it's being used:

```
with This
   .Gripper.Visible = .T.
   .Gripper.SetupGripper()
endwith
```

Like many other methods, AdjustMinimum is abstract in this class because the behavior depends on whether it's a vertical or horizontal splitter.

The splitter action starts when the user drags the splitter; that is, when they move the mouse while holding down the left button. MouseMove takes care of this:

```
lparameters tnButton, ;
   tnShift, ;
   tnXCoord, ;
   tnYCoord
local lnPosition
with This
   if tnButton = 1 and .Enabled
      lnPosition = .GetPosition(tnXCoord, tnYCoord)
      .MoveSplitterToPosition(lnPosition)
   endif tnButton = 1 ...
endwith
```

GetPosition is abstract in this class. MoveSplitterToPosition is responsible for moving the splitter and the controls it's associated with. If you want to start the splitter at a certain spot (for example, restoring the former position when the user runs a form again), call MoveSplitterToPosition manually.

```
lparameters tnPosition
local lnPosition, ;
   laObjects1[1], ;
   lnObjects1, ;
   lnI, ;
   loObject, ;
   laObjects2[1], ;
   lnObjects2, ;
   lnAnchor
with This

* Move the splitter to the specified position. Ensure it doesn't go too far, based
* on the nObject1MinSize and nObject2MinSize settings.

   lnPosition = tnPosition
   lnObjects1 = alines(laObjects1, ;
      .cObject1Name, 4, ',')
   for lnI = 1 to lnObjects1
      loObject   = evaluate('.Parent.' + laObjects1[lnI])
      lnPosition = max(lnPosition, .GetObject1Size(loObject))
   next lnI
   lnObjects2 = alines(laObjects2, .cObject2Name, 4, ',')
   for lnI = 1 to lnObjects2
      loObject   = evaluate('.Parent.' + laObjects2[lnI])
      lnPosition = min(lnPosition, .GetObject2Size(loObject))
   next lnI
   lnAnchor = .Anchor
   .Anchor  = 0
   .SetPosition(lnPosition)
   .Anchor = lnAnchor

* Now move the objects as well.
```

```
   for lnI = 1 to lnObjects1
      loObject = evaluate('.Parent.' + laObjects1[lnI])
      with loObject
         lnAnchor = .Anchor
         .Anchor  = 0
         This.MoveObject1(loObject)
         .Anchor = lnAnchor
      endwith
   next lnI
   for lnI = 1 to lnObjects2
      loObject = evaluate('.Parent.' + laObjects2[lnI])
      with loObject
         lnAnchor = .Anchor
         .Anchor  = 0
         This.MoveObject2(loObject)
         .Anchor = lnAnchor
      endwith
   next lnI

* Since the object sizes have changed, we need to adjust the form as necessary.

   .AdjustMinimum()

* Call a hook method.

   .SplitterMoved()
endwith
```

Note what the code does with its own Anchor property and that of the associated objects. If you manually change the size or position of an object that has Anchor set to a non-zero value, the next time the form is resized, the objects moves and resizes based on the original values. To prevent this, the code saves the Anchor value, sets it to zero, and restores it again after moving and resizing objects.

All of the methods called from MoveSplitterToPosition are abstract in this class.

SFSplitterH and SFSplitterV

The two classes you'll actually use are SFSplitterH, a horizontal splitter, and SFSplitterV, a vertical one. The Height and Width of these subclasses are set such that the splitter has the appropriate shape. MousePointer contains 9-Size WE and 7-Size NS, respectively, and Anchor is set to 13 (resize vertically and bound to the right edge) and 14 (resize horizontally and bound to the bottom edge), respectively. Let's look at the code in SFSplitterH to see how it implements the desired behavior. The code in SFSplitterV is almost identical but generally uses Top instead of Left and Height instead of Width.

GetPosition, which is called from MouseMove, determines the new location of the splitter based on the location of the mouse:

```
lparameters tnXCoord, ;
   tnYCoord
```

```
return tnXCoord + This.Left - objtoclient(This, 2)
```

GetObject1Size, called from MoveSplitterToPosition, determines the size of the specified object on the left, taking into account its minimum width.

```
lparameters toObject
return toObject.Left + This.nObject1MinSize
```

GetObject2Size is similar but for objects on the right.

```
lparameters toObject
return toObject.Left + toObject.Width - This.nObject2MinSize - This.Width
```

SetPosition sets This.Left to the specified position.

MoveObject1, also called from MoveSplitterToPosition, moves the specified left object.

```
lparameters toObject
with toObject
   .Move(.Left, .Top, This.Left - .Left, .Height)
endwith
```

MoveObject2 is a little more complicated but still not a lot of code.

```
lparameters toObject
with toObject
   .Move(This.Left + This.Width, .Top, max(.Width + .Left - This.Left - ;
      This.Width, 0), .Height)
endwith
```

The last method called from MoveSplitterToPosition, AdjustMinimum, adjusts the MinWidth property of the form. After all, while the splitter respects the settings of nObject1MinSize and nObject2MinSize so the objects can’t be sized too small, it wouldn’t make sense to allow the user to size them too small by simply resizing the form.

```
local laObjects[1], ;
   lnObjects, ;
   lnWidth, ;
   lnI, ;
   loObject
with This
   lnObjects = alines(laObjects, .cObject1Name, 4, ',')
   lnWidth   = -1
   for lnI = 1 to lnObjects
      loObject = evaluate('.Parent.' + laObjects[lnI])
      lnWidth  = max(lnWidth, loObject.Width)
   next lnI
   Thisform.MinWidth = max(Thisform.MinWidth, Thisform.Width - lnWidth + ;
      .nObject1MinSize
endwith
```

Gripper and GripperDot

After looking at my splitter classes, Matt Slay decided he wanted one that had a visual appearance, similar to the splitter in Outlook between the Mail Folders section and the buttons below it. After a couple of attempts involving images, he created the Gripper and GripperDot classes that use VFP Shape objects for the dots.

Gripper is a Container-based class that acts as a container for the dots. It has five GripperDot objects, although how many are actually displayed depends on the nDots property.

SetupGripper, called from SFSplitter.Init, makes sure the container and the dots use the same MousePointer value as the splitter and sets up the container depending on whether it's a vertical or horizontal splitter.

```
local loDot
with This

* Get the number of dots to use.

   .nDots = .Parent.nDots

* Use the same MousePointer as the splitter, both for ourselves and each dot.

   .MousePointer = .Parent.MousePointer
   for each loDot in .Controls foxobject
      loDot.SetAll('MousePointer', .MousePointer)
   next loDot

* Adjust the gripper based on whether this is a vertical or horizontal splitter.

   if .Parent.Width > .Parent.Height
      .SetupForVerticalSplitter()
   else
      .SetupForHorizontalSplitter()
   endif .Parent.Width > .Parent.Height
endwith
```

SetupForVerticalSplitter and SetupForHorizontalSplitter are almost identical, the main difference being replacing Top with Left and Width with Height. They make sure the container displays the desired number of dots and that they're laid out in the appropriate orientation. Here's the vertical version:

```
local lnDotWidth, ;
   lnI, ;
   loDot
with This

* Position the dots in a horizontal orientation for a vertical splitter.

   lnDotWidth = .GripperDot1.Width
   for lnI = 1 to 5
      loDot      = evaluate('.GripperDot' + transform(lnI))
      loDot.Left = (lnDotWidth * lnI - 1) + 1
      loDot.Top  = 2
   next lnI

* Adjust the container so it shows the correct number of dots.

   .Height = .nDots * lnDotWidth
   .Height = lnDotWidth + 1

* Center the container and set Anchor so it stays centered.

   .Top    = (.Parent.Height - .Height)/2 - 1
   .Left   = (.Parent.Width  - .Width)/2
   .Anchor = 256
 endwith
```

MouseMove passes the drag operation up to the splitter by calling This.Parent.MouseMove in case the user started it on the container.

GripperDot is also based on Container. It has three shapes with different colors to give a shadowed appearance to each dot. As with Gripper, the MouseMove method of each shape and the container itself call the parent's MouseMove method to bubble the drag up to the splitter.

Checking it out

Run the form shown in **Figure 1**, TestSplitter.SCX, and try moving the splitters. Note that they only move so far, to prevent the objects they're associated with from being sized too small. Also note that as you adjust the relative sizes of the editboxes, the MinWidth and MinHeight properties of the form adjust, preventing you from making the editboxes too small by resizing the form.

Summary

A splitter control removes the need for you to decide between competing resizing behaviors between resizable controls and gives your users the ability to decide for themselves the relative sizes of the controls. Adding a splitter to a form is as easy as dragging it to the form, setting a few properties, and positioning it between resizable controls. Thanks again to Matt Slay for enhancing this control.

SFComboTree

SFComboTree is so named because it combines a VFP ComboBox control with a Microsoft TreeView ActiveX control. Although it can be used for a variety of needs, SFComboTree is most useful for two specific tasks: a hierarchical list of data and multiple checkboxes. In both cases, its compact size makes it ideal for forms lacking space for a large control like a list box or TreeView.

In its "closed" state, SFComboTree looks like a combo box, so it's only 24 pixels high and only as wide as you size it. When you "open" the control, it's as tall as you wish, temporarily overlapping any other controls as necessary. **Figure 2** shows a sample form (TestComboTree.SCX) with two closed SFComboTree controls. In **Figure 3**, the top control is open so it displays a hierarchical list of folders and temporarily covers the control below it. **Figure 4** shows an example of a list of multiple checkboxes.

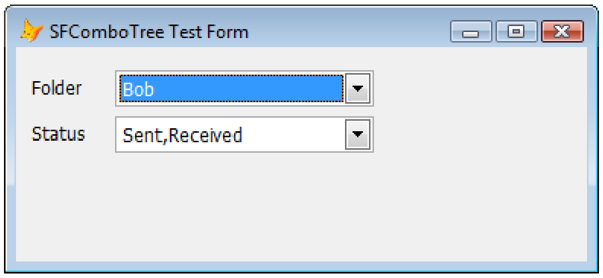


Figure 2. When it's closed, SFComboTree takes up very little space.

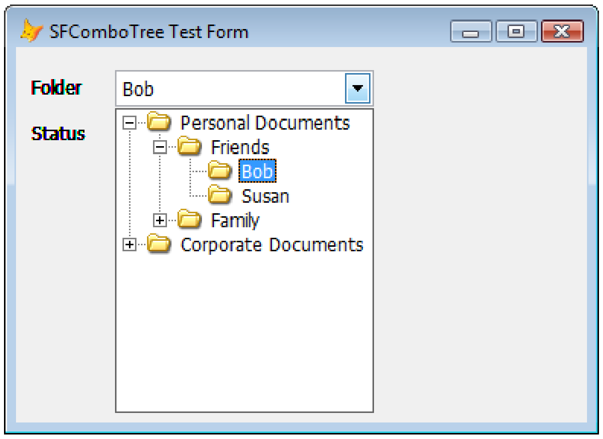


Figure 3. SFComboTree provides a combined combo box and TreeView control so the user can easily view a hierarchical list.

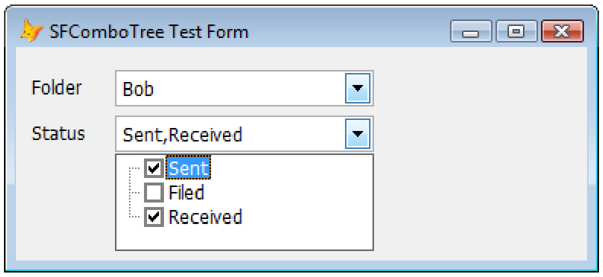


Figure 4. The second SFComboTree in the sample form demonstrates a list of items with checkboxes.

Here are some of the characteristics of SFComboTree:

- The combo box displays the text of the selected node in the TreeView. However, as you'll see later, you can override that to, for example, display a comma-delimited list of all checked items.
- Programmatically, you can read from or write to the Value property to find out which node is selected or to automatically select a node (such as displaying a value from an existing record).
- Nodes in the TreeView control can have images or not, checkboxes or not, and be arranged hierarchically or not.
- The control automatically resizes when its form or container does. Although the combo box doesn't get taller, it does get wider. The TreeView resizes both vertically and horizontally, regardless of whether the control is opened or closed when the form is resized.
- Clicking the combo box's down arrow opens the control. You can configure how the control closes: clicking the down arrow again, when the user clicks a node, when the user double-clicks a node, or when the control loses focus.

Using SFComboTree

To use SFComboTree, drop it on a form and set the properties shown in **Table 1** as desired. Fill in the LoadTree and LoadImages methods with code that loads the nodes in the TreeView and images used by the TreeView, respectively. Leave LoadImages empty if you don't want images in the TreeView (for example, if you're using it with checkboxes).

Table 1. Important properties of SFComboTree.

Property	Description
FontName	The font to use for the combo box and TreeView (default Tahoma).
FontSize	The font size (default 9).
ToolTipText	The tool tip text for the container and combo box.
lCloseOnClick	.T. to close the control when the user clicks an item in the TreeView (default .F.).
lCloseOnDblClick	.T. to close the control when the user double-clicks an item in the TreeView (default .F.).
lLoadImagesOnInit	.T. to load images when the control is initialized (default .F.).
lLoadTreeOnInit	.T. to load the TreeView when the control is initialized (default .F.).
lMoveToBack	.T. to set the ZOrder of the control to the back when it's closed (default .T.)
lNoClose	.T. to not have LostFocus close the control (default .F.).

If you need to do some setup tasks before loading images into the ImageList (for example, you have to wait until the form the control is on has initialized), set lLoadImagesOnInit to .F. and call LoadImages manually once the setup is done. Otherwise, set that property to .T. so LoadImages is called from Init.

Init calls LoadTree to load the TreeView if lLoadTreeOnInit is .T. That isn't strictly necessary because opening the control calls LoadTree if the TreeView doesn't have any nodes. However, if for some reason you want the TreeView loaded earlier, set lLoadTreeOnInit to .T.

TreeView controls are notoriously slow for loading if you have a lot of nodes. To improve performance when loading hierarchical nodes into the TreeView, you can just load the top-level nodes in LoadTree. SFComboTree can then load the child nodes for a top-level node when it's expanded for the first time. Of course, you'll need to load at least one child node for every top-level node or the "+" won't appear for the node. To make this work, create a child node under each top-level node with "Loading…" as the text for the node. When the user expands a node, the TreeView's Expand event fires, and code in that event calls SFComboTree's LoadExpandedNode method if it finds a "Loading…" child node. Fill in the code in LoadExpandedNode to load the child nodes for the specified parent node. The net result is that the child nodes are only loaded the first time a parent node is expanded, which is much less of a performance hit than loading all nodes at one time, regardless of whether the user will ever expand the parent nodes or not.

You can change any of the properties of oTree or oImageList as necessary. For example, if you want checkboxes, set oTree.CheckBoxes to .T.

If you have code you want to execute when the user selects an item, put the code into the ItemSelected method. Otherwise, you can check the lChanged (.T. if the user changed the selected node in the TreeView) and Value (the text of the selected node) properties as necessary, such as when the user saves a record or closes the form.

Try it out

The sample TestComboTree.SCX included with the downloads for this article shows a couple of typical uses for SFComboTree: a hierarchical list and multiple checkboxes. The top instance, named oFolder, displays a list of "folders" as stored in a table named FOLDERS.DBF, some of which are children of other folders. oFolder has lCloseOnClick set to .T. so the control closes when the user selects a node.

LoadTree has the following code:

```
local lcKey, ;
   lcName, ;
   lcParentKey

* Open the Folders table if necessary.

if used('FOLDERS')
   select FOLDERS
else
   select 0
   use FOLDERS again shared
endif used('FOLDERS')

* Go through each record and create a node in the TreeView under the appropriate
* parent node.

with This.oTree
   scan
      lcKey       = 'F' + transform(ID)
      lcName      = trim(NAME)
      lcParentKey = 'F' + transform(PARENT)
      if empty(PARENT)
         loNode = .Nodes.Add(, 1, lcKey, lcName, 'Folder')
      else
         loNode = .Nodes.Add(lcParentKey, 4, lcKey, lcName, 'Folder')
      endif empty(PARENT)
   endscan
endwith
```

LoadImages just loads a single image into the ImageList control. Because lLoadImagesOnInit is .T., oFolder loads the image from Init.

```
This.oImageList.ListImages.Add(1, 'Folder', loadpicture('Folder.bmp'))
```

ItemSelected simply displays the folder the user selected by passing This.Value to MESSAGEBOX().

Figure 2 shows the form when oFolder is open. Because the TreeView is loaded from the Folders table, it serves as an example for a dynamically loaded hierarchical list.

The second SFComboTree on the form, named oStatus, shows how to create a list of items with checkboxes. Its LoadTree method loads a hard-coded list of status values (which could easily be loaded from a table in a real application) but also turns on checkboxes for nodes. This allows you to select multiple status values, such as “Sent” and “Received”.

```
local loNode
with This.oTree
   .Checkboxes = .T.
   loNode = .Nodes.Add(, 1, 'S1', 'Sent')
   loNode = .Nodes.Add(, 1, 'S2', 'Filed')
   loNode = .Nodes.Add(, 1, 'S3', 'Received')
endwith
```

Because we want the combo box to show all selected items, not just the last one clicked, ItemSelected concatenates all checked items into a comma-delimited list and sets Value to that list.

```
local lcStatus, ;
   lnI, ;
   loNode
with This.oTree
   lcStatus = ''
   for lnI = 1 to .Nodes.Count
      loNode = .Nodes(lnI)
      if loNode.Checked
```

```
          lcStatus = lcStatus + iif(empty(lcStatus), '', ',') + loNode.Text
       endif loNode.Checked
   next lnI
endwith
This.Value = lcStatus
```

Figure 4 shows the form when you open oStatus.

SFComboTree details

SFComboTree, defined in SFComboTree.VCX, is subclass of Container with four controls: a ComboBox named cboCombo, a TreeView control named oTree, an ImageList control named oImageList, and a shape named shpTreeView that provides a border for the TreeView control. The container has BorderWidth set to 0 so it doesn't appear as a container and Height set to 24, the same height as cboCombo.

cboCombo has RowSourceType set to 1-Value and RowSource set to nothing because we don't need the combo box to contain a list of choices; it simply provides the visual appearance of a text box and a drop down arrow. Style is 2-Dropdown List so the user can't type a value.

The custom changes to oTree's properties control its appearance and behavior. Appearance is 0-Flat, HideSelection is .F., HotTracking is .T., Indentation is 10, LabelEdit is 1-Manual (so the user can't edit the nodes), and LineStyle is 1-RootLines.

The Init method of SFComboTree sets up the controls

```
with This

* Set up the combobox.

   .cboCombo.Anchor      = 0
   .cboCombo.Width       = .Width
   .cboCombo.Anchor      = .Anchor
   .cboCombo.ToolTipText = .ToolTipText

* Save the current height of the control and the form and our Anchor value.

   dimension .aParentHeights[1, 2]
   .aParentHeights[1, 1] = .Height
   .aParentHeights[1, 2] = .Height
   .nInitialFormHeight   = Thisform.Height
   .nSavedAnchor         = .Anchor

* Set our font name and size to their own values so Assign takes care of setting the
* other controls.

   .FontName = .FontName
   .FontSize = .FontSize

* Call CloseControl so everything is sized properly for a closed appearance.
```

```
   .CloseControl()

* If we're supposed to, load the images now.

   if .lLoadImagesOnInit
      .LoadImages()
   endif .lLoadImagesOnInit

* If we have any images, use the ImageList control with the TreeView.

   if .oImageList.ListImages.Count > 0
      .oTree.Object.ImageList = .oImageList
   endif .oImageList.ListImages.Count > 0

* If we're supposed to, load the TreeView now.

   if .lLoadTreeOnInit
      .LoadTree()
   endif .lLoadTreeOnInit
endwith
```

The DropDown event of cboCombo, fired when the user clicks the down arrow, opens or closes the control by calling SFComboTree's OpenControl or CloseControl methods. There's also some code in that event to handle what may be a bug in VFP; see the comments for details.

OpenControl is a fairly complex but well-documented method. It's responsible for adjusting the control so the TreeView is visible and sized appropriately, and ensuring the selected node in the TreeView matches the value displayed in the combo box.

```
local lnAnchor, ;
   loParent, ;
   lnParent, ;
   lnI, ;
   loNode
with This

* If we haven't already done so, load the TreeView the first time we're opened.

   if .oTree.Nodes.Count = 0
      .LoadTree()
   endif .oTree.Nodes.Count = 0

* If we loaded images later than from Init, use the ImageList control with the
* TreeView.

   if not .lLoadImagesOnInit and .oImageList.ListImages.Count > 0
      .oTree.Object.ImageList = .oImageList
   endif not .lLoadImagesOnInit ...

* Turn off anchoring since we'll be resizing and moving controls.
```

```
   lnAnchor = .Anchor
   store 0 to .Anchor, .cboCombo.Anchor, .oTree.Anchor, .shpTreeView.Anchor

* Save our height, then set it to the desired height, accounting for any resize of
* the form.

   .aParentHeights[1, 1] = .Height
   .Height               = min(.nOriginalHeight + Thisform.Height - ;
      .nInitialFormHeight, Thisform.Height - This.Top - 5)
   .aParentHeights[1, 2] = .Height

* Save the height of all parent containers and adjust them if necessary. Also, save
* the current anchor values and add 5 if necessary so they resize vertically.

   loParent = This.Parent
   lnParent = 1
   do while vartype(loParent) = 'O' and lower(loParent.BaseClass) = 'container'
      lnParent = lnParent + 1
      dimension .aParentHeights[lnParent, 3]
      if loParent.Height < .Top + .Height
         .aParentHeights[lnParent, 3] = loParent.Anchor
         loParent.Anchor              = 0
         .aParentHeights[lnParent, 1] = loParent.Height
         loParent.Height = .Top + .Height
         .aParentHeights[lnParent, 2] = loParent.Height
         loParent.Anchor = .aParentHeights[lnParent, 3]
         if not bittest(loParent.Anchor, 0) and not bittest(loParent.Anchor, 2)
            loParent.Anchor = loParent.Anchor + 5
         endif not bittest(loParent.Anchor ...
      endif loParent.Height < .Top + .Height
      loParent = loParent.Parent
   enddo while vartype(loParent) = 'O' ...

* Adjust the size of the TreeView and shape in case the container was resized while
* we were closed.

   .oTree.Width        = .Width - 2
   .oTree.Height       = .Height - .cboCombo.Height - 4
   .oTree.Left         = .shpTreeView.Left + 1
   .oTree.Top          = .shpTreeView.Top + 1
   .shpTreeView.Width  = .Width
   .shpTreeView.Height = .Height - .cboCombo.Height - 2

* If the current value doesn't match the selected item in the TreeView, find and
* select the appropriate item.

   if vartype(.oTree.SelectedItem) <> 'O' or ;
      (not empty(.cboCombo.DisplayValue) and ;
      not .cboCombo.DisplayValue == .oTree.SelectedItem.Text)
      for lnI = 1 to .oTree.Nodes.Count
         loNode = .oTree.Nodes.Item(lnI)
         if .cboCombo.DisplayValue == loNode.Text
            loNode.Selected = .T.
            exit
         endif .cboCombo.DisplayValue ...
      next lnI
   endif vartype(.oTree.SelectedItem) ...

* Enable the controls appropriately, then set focus to the TreeView.

   .oTree.Visible       = .T.
   .shpTreeView.Visible = .T.
   .ZOrder(0)
   .shpTreeView.ZOrder(0)
   .oTree.ZOrder(0)
   .lComboTreeOpen = .T.
   .oTree.SetFocus()

* Restore anchoring and add 5 to it so we resize vertically.

   .cboCombo.Anchor = lnAnchor
   store lnAnchor + 5 to .Anchor, .oTree.Anchor, .shpTreeView.Anchor
endwith
```

I won't show the code for CloseControl for space reasons; it too is well-documented and should be easy enough to understand. It has to do the opposite of OpenControl: reset the height of the control so the TreeView is no longer visible. CloseControl also calls the abstract ItemSelected method so you can add some code in a subclass or instance to do something when the user closes the control.

The custom FontName and FontSize controls have Assign methods so changing the font for the control changes it for the combo box and TreeView. Enabled also has an Assign method for similar reasons. The custom Value property has Access and Assign methods that read from and write to cboCombo.DisplayValue so you can simply reference Control.Value rather than Control.cboCombo.DisplayValue.

Summary

I use SFComboTree in lots of places in my applications. It's even used to display the hierarchy of controls for a form or class in PEM Editor, a very cool replacement for the VFP New Property, New Method, and Edit Property/Method dialogs, available from VFPX (http://vfpx.codeplex.com). All of those uses have one thing in common: the need to display a list of items (hierarchical or not) while taking up very little space in a form.

PopMenu

I haven't used VFP's native menus directly in many years. I always hated the fact that the Menu Designer is a clunky tool and that menus are hard-coded procedural code rather than object oriented. Because I wanted a more flexible menuing system, I created a set of OOP menu classes that are now part of VFPX (http://tinyurl.com/2a2jnak). Internally, they're just wrapper for the VFP DEFINE MENU, DEFINE PAD, DEFINE BAR, and other menu-related commands, but at least I can manipulate my menus as objects now.

However, the other issue is that, as you can see in **Figure 5**, VFP's native menus (the one on the left) look out of date compared to menus in more recent applications, such as Microsoft Outlook (the one on the right).

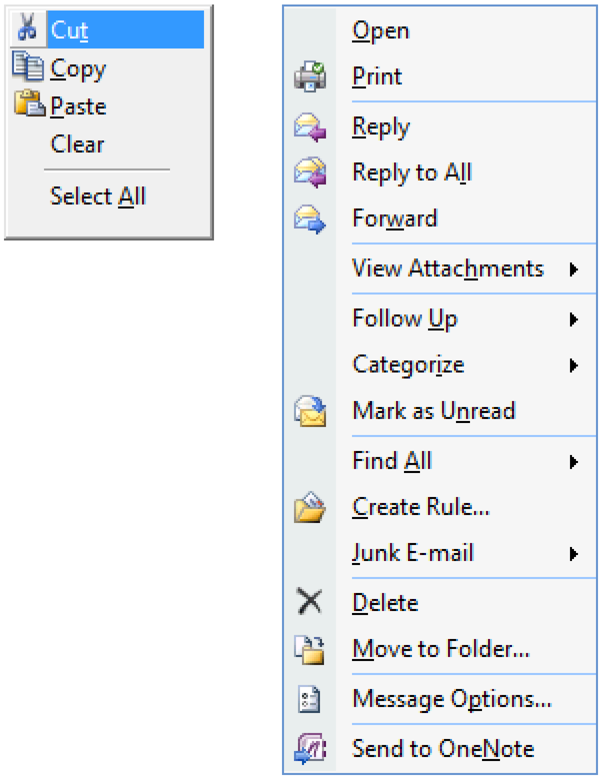


Figure 5. VFP's native menus look out of date compared to newer applications.

Fortunately, a VFPX project called PopMenu, written by LingFeng Shi, provides an entirely new way of doing menus. Not only are they object-oriented, they also use the native Windows menuing system rather than VFP's menuing system, meaning that your application's menus can look just like those in Microsoft Office applications.

One downside of PopMenu is that there's no documentation and all comments (in code and the Properties window) are in Chinese. So, I dug through the code, played with the samples, and found that I really like this class.

To use PopMenu, download it from VFPX (http://tinyurl.com/2c9ecr2), unzip it into any folder, and add the main class library, VCX_Tools.VCX, to your project.

Note that PopMenu is primarily for shortcut rather than system menus. I'll discuss this more in detail later.

Creating a menu

To create a shortcut menu, instantiate PopMenu or a subclass, then add items to it using either Add or AddItem.

AddItem is the simpler of the two methods: AddItem(cTitle [, cCommand [, vEnabled [, nFlag]]]). cTitle is the caption for the item, cCommand is the command to execute when the item is selected, vEnabled is an expression which indicates whether the item is enabled or not, and nFlag is a numeric value indicating how the item should appear. These parameters match up with properties of menu item object discussed below. Here's an example:

```
loMenu = newobject('PopMenu', 'VCX_Tool.vcx')
loMenu.AddItem('\<New')
loMenu.AddItem('\<Open')
loMenu.AddItem('\<Save')
```

Add provides more flexibility: Add(cParentKey, cKey, cTitle, cCommand, cPicture, vEnabled, nFlag). cParentKey is the key assigned to the parent item for this item and cKey is the key assigned to this item; these parameters are used when you want to create a submenu. cTitle, cCommand, vEnabled, and nFlag have the same meaning as for AddItem. cPicture is the name of an image file to use for the picture for the item. Here's an example:

```
loMenu = newobject('PopMenu', 'VCX_Tool.vcx')
loMenu.Add('', '', '\<New',  'messagebox("You chose New")', 'Images\New.bmp')
loMenu.Add('', '', '\<Open', '',                            'Images\Open.bmp')
loMenu.Add('', '', '\<Save', '',                            'Images\Save.bmp')
```

Both Add and AddItem create a _MenuInfo object, set it properties, add it to an internal collection, and return a reference to the object. This object has the following properties you can set as necessary:

- cCommand: the command to execute when the item is selected.
- cEnabled: an expression which indicates whether the item is enabled or not. This can either be a logical value or an expression which evaluates to a logical value. For example, "GetUserRights('Payroll')" calls the GetUserRights function to determine whether the user has rights to payroll and disables the item if not.
- cKey: the key for the item. If you don't assign a key, a SYS(2015) value is used.
- cMessage: the status bar text for the item.
- cParentKey: the key for the parent item for this item. If this contains a valid key, this item appears in a submenu of the parent item.
- cPicture: the name of an image file to use for the item.
- cTitle: the caption for the item. Use "\<" or "&" to indicate that the following character is a hotkey.
- nFlags: there are several constants defined in WIN32API.H you can use for nFlag:
    - MF_MENUBARBREAK (0x20): places the item in a new column without a vertical separator between columns.
    - MF_MENUBREAK (0x40): like MF_MENUBARBREAK but separates the columns with a vertical line.

- MF_CHECKED (0x8): displays a checkmark next to the item. This only works when "owner drawn" menus aren't used (discussed later).

- nHeight: the height of the bar in pixels. The default is the same as the nItemHeight property of PopMenu, which I'll discuss later.
- nIndex: the order for the item (1 for the first item, 2 for the second, etc.).
- nWidth: the width of the bar in pixels. The default is the same as the nItemWidth property of PopMenu, which I'll discuss later.
- oRefMenu: if this contains a reference to another PopMenu object, that object is used as a submenu for this item. If you set this, be sure to call CreateContext(oItem, oItem.oRefMenu) to properly initialize the submenu.

PopMenu contains a few helper methods:

- SetMessage(cMessage [, nOrder]): sets the message for the specified menu item; if nOrder isn't specified, the last added item is used.
- SetPicture(cImageFile [, nOrder]) : sets the picture for the specified menu item; if nOrder isn't specified, the last added item is used.
- Clear(): removes all items from the menu.

To display the menu, call Show. Here's an example that displays a typical "edit" menu. Each item is enabled if the appropriate function in the Edit menu is enabled and performs the same action when selected.

```
loMenu = newobject('PopMenu', 'VCX_Tool.vcx')
with loMenu
   .Add('', '', 'Cu\<t',       "sys(1500, '_MED_CUT',   '_MEDIT')", ;
      'Images\CutXPSmall.bmp', "not skpbar('_MEDIT', _MED_CUT)")
   .Add('', '', '\<Copy',      "sys(1500, '_MED_COPY',  '_MEDIT')", ;
      'Images\CopyXPSmall.bmp', "not skpbar('_MEDIT', _MED_COPY)")
   .Add('', '', '\<Paste',     "sys(1500, '_MED_PASTE', '_MEDIT')", ;
      'Images\PasteXPSmall.bmp', "not skpbar('_MEDIT', _MED_PASTE)")
   .Add('', '', 'Clear',       "sys(1500, '_MED_CLEAR', '_MEDIT')", ;
      , "not skpbar('_MEDIT', _MED_CLEAR)")
   .Add('', '', '\-')
   .Add('', '', 'Select \<All', "sys(1500, '_MED_SLCTA', '_MEDIT')", ;
      , "not skpbar('_MEDIT', _MED_SLCTA)")
endwith
loMenu.Show()
```

As another example, the code in the RightClick method of TestPopMenu.SCX creates the relatively plain looking menu shown in **Figure 6**.

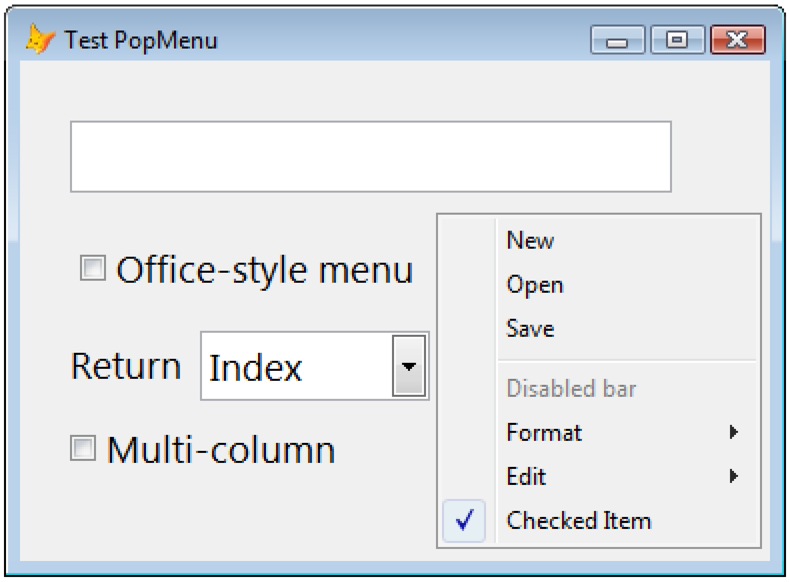


Figure 6. PopMenu can create object-oriented menus.

Getting fancier

So far, other than having an object-oriented menu, PopMenu hasn't done much for us yet. However, let's look at some properties that make menus really pop (no pun intended).

The key to having more modern looking menus is setting lOwnerDraw to .T. This provides much better control over the appearance. The menu shown in **Figure 7** has this property set to .T. and a few other properties discussed later set so the menu has an Office-like appearance.

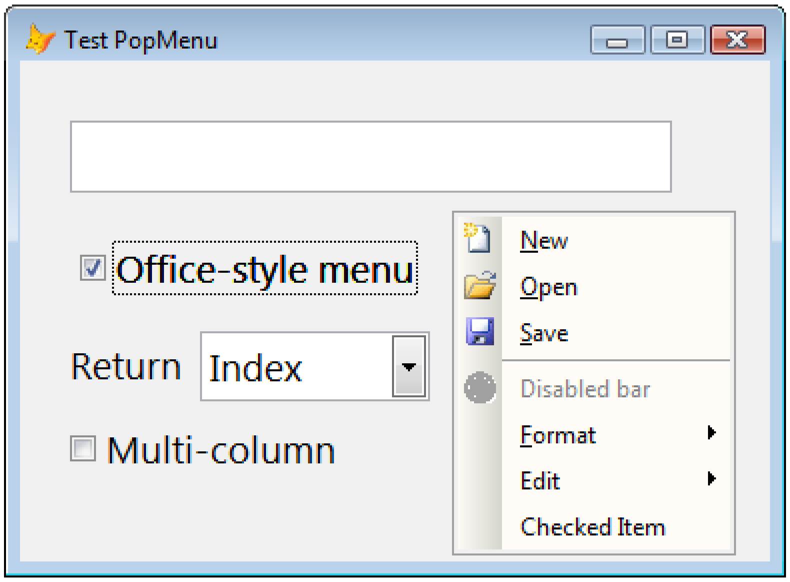


Figure 7. Setting lOwnerDraw to .T. is the key to more modern looking menus.

PopMenu has a few properties that control the dimensions of the menu items, as shown in **Figure 8**.

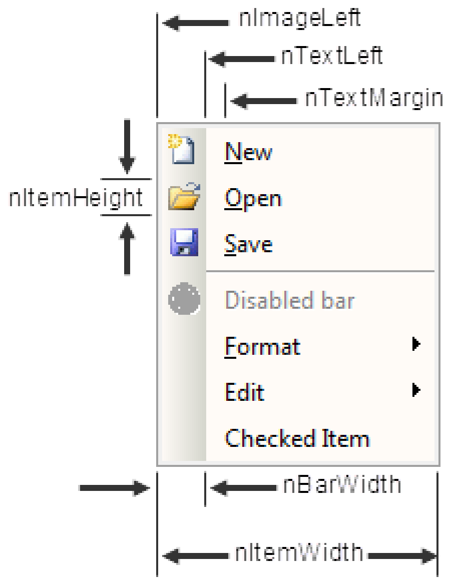


Figure 8. PopMenu has properties controlling the dimensions of menu items.

I created a subclass of PopMenu called SFOfficeMenu (in SFPopMenu.VCX) that has several properties set to create the menu shown in **Figure 7**. It has lOwnerDraw set to .T., nBarWidth set to 22, nItemHeight and nTextLeft set to 23, and nTextMargin set to 8. Other properties are set as described below. I created a subclass of SFOfficeMenu called SFEditMenu that provides a pre-defined menu with Cut, Copy, Paste, Clear, and Select All menu items.

nMenuBackColor is the color for the background of the menu items. It defaults to -1, which means use the system color for menus. SFOfficeMenu has this property set to RGB(255, 251, 247).

nBarStyle controls the appearance of the bar to the left of the menu items. The values are:

- 0: the bar doesn't have a separate appearance from the menu items.
- 1: fills the bar with the color specified in nBarFillColor1.
- 2: fills the bar with a left-to-right horizontal gradient from nBarFillColor1 (RGB(255, 251, 247) in SFOfficeMenu) to nBarFillColor2 (RGB(214, 215, 206) in SFOfficeMenu). SFOfficeMenu has nBarStyle set to 2.
- 3: fills the bar with a vertical gradient from nBarFillColor2 (top) to nBarFillColor1 (bottom).

If lSelectedEnabled is .F. (the default), a selection bar appears when you move over disabled items. Set this to .T. to display no bar.

If lShareIcons is .T. (the default), images for menu items are cached in a collection that's shared between all menus.

nSelectedForeColor determines the color of the text for the selected menu item; the default of -1 means use the system highlight text color. SFOfficeMenu has this set to RGB(0, 0, 0).

nSelectedStyle affect how the selected menu item appears. A related property is nSelectedImageStyle, which affects how the image for the selected menu item appears. These properties have the following values:

- 0: a solid bar of the color specified in nSelectedBackColor (the default of -1 means use the system highlight color) appears over the selected menu item. If nSelectedImageStyle is 0, the image is included in the same bar or rectangle as the text of the menu item; otherwise, it appears in a separate rectangle.
- 1: a rectangle appears over the selected menu item. The border of the rectangle uses nSelectedBorderColor as its color (the default of -1 means use the system highlight color). The fill color is specified in nSelectedBackColor if it isn't the default of -1; if it is the default, the fill color is a blend of nSelectedBorderColor (or the system highlight color if it's -1) and nMenuBackColor (or the system menu color if it's -1). SFOfficeMenu has nSelectedStyle set to 1.
- 2: same as 1 but uses a rounded rectangle with nSelectedRoundX and nSelectedRoundY specifying the roundedness of the rectangle (they both default to 12).
- 3: displays a raised rectangle over the selected menu item. The fill color is specified in nSelectedBackColor if it isn't the default of -1; if it is the default, the system highlight color is used.
- 4: like 3, but displays a sunken rectangle.

If the user closes the menu without selecting an item, Show returns .NULL. nReturn determines what Show returns when the user selects an item:

- 0 (the default): return the index of the selected menu item.
- 1: return the value of the cKey property for the selected item.
- 2: return the title of the selected item.
- 3: returns an object reference to the selected item.

Show can accept four optional parameters. The first two are X and Y coordinates for the menu; if they aren't specified, the menu is positioned at the mouse location. Pass .F. for the third parameter if the X and Y coordinates are relative to the VFP window or .T. if they're absolute values (that is, relative to the screen). The last parameter is a numeric additive flag indicating how the menu should appear. Use one of the following values to specify how to position the menu horizontally:

- 0x4 (constant TPM_CENTERALIGN in Win32API.H): centers the menu horizontally.
- 0x0 (TPM_LEFTALIGN): positions the menu so its left side is at the X coordinate.
- 0x8 (TPM_RIGHTALIGN): positions the menu so its right side is at the X coordinate.

Use one of the following values to specify how to position the menu vertically:

- 0x20 (TPM_BOTTOMALIGN): positions the menu so its bottom is at the Y coordinate.
- 0x0 (TPM_TOPALIGN): positions the menu so its top is at the Y coordinate.
- 0x10 (TPM_VCENTERALIGN): centers the menu vertically.

Use one of the following values to specify which mouse button can be used:

- 0x0 (TPM_LEFTBUTTON): only the left mouse button can select items.
- 0x2 (TPM_RIGHTBUTTON): the user can use both the left and right buttons.

There are several values (and associated constants) that provide animation of the menu, but I couldn't get these to work.

You can also display the menu by calling ShowBy(oObject [, tnAddX [, tnAddY]]), where oObject is a reference to an object whose upper-left corner is used as the location of the menu and tnAddX and tnAddY are offset values to add to the X and Y values.

I fixed a few issues in PopMenu; see the comments in Init, CreateContext, CreateMenus, and Show for details.

To try out PopMenu, run the TestPopMenu form. Right-click in the text box to see a VFP shortcut menu, then turn on "Office-style menu" to use PopMenu instead. Right-click the form; it displays either a plain menu ("Office-style menu" turned off) or an Office-like menu (that setting turned on). Try turning on "Multi-column" and "Select disabled items" to see the effect they have.

What about system menus?

We've seen that PopMenu can make your shortcut menus look more modern. However, what about system menus? Unfortunately, that's a little more difficult to do. PopMenu uses the Windows API TrackPopupMenu function, which displays a shortcut menu. Shortcut menus are almost identical to popup menus (the menu that appears under a pad in a menu bar) except for one thing: they're modal. You have to click off a shortcut menu to close it without making a selection. A popup menu, on the other hand, closes when you move the mouse off it.

There's one other issue: where to display the menu. With shortcut menus, you expect the menu to appear at the mouse location. However, popup menus should appear directly under the menu pad, regardless of where the mouse is on that pad.

I created a sample program, TestPopMenu.PRG, that shows how to use PopMenu with a system menu bar. This program defines a menu bar using DEFINE PAD functions and uses ON SELECTION PAD to call functions that use PopMenu to display a menu. The vertical location for the menu can be determined using some SYSMETRIC() functions:

```
lnY = _vfp.Top  + sysmetric(9) + sysmetric(4) + sysmetric(20)
```

The complication is that the horizontal location of the menu isn't the location of the mouse but the location of the pad, which can't really be determined easily. For example, the location for the Tools pad uses a empirically determined offset of 40 to determine where the menu should go:

```
lnX = _vfp.Left + sysmetric(3) + 40
```

An easier solution is to dispense with the VFP menuing system and implement your own. I created a couple of classes in SFPopMenu.VCX, SFMenubar and SFPadButton, that provide a menu bar and pads as a proof of concept. TestSystemMenu.SCX uses these classes for a simple menu in a form.

Regardless of which mechanism you use, you can't escape the fact that the menus are modal, so the user can't slide the mouse from one pad to the next once they've clicked a pad; they have to click on the next pad to close the current popup and open the next one.

The Windows API also contains functions for dealing with system menu bars, so hopefully LingFeng Shi or someone else will implement them to allow us to deal with menus more gracefully.

Summary

PopMenu provides an easy way to add object-oriented shortcut menus that look like menus in modern applications such as Microsoft Office. However, it needs more work if you want to do something similar for system menu bars.

RCSCalendar

VFP includes the Microsoft Date and Time Picker Control (DTPicker), an ActiveX control that provides a date/time control that appears like a combobox (**Figure 9**). You can type a date or click the down arrow to drop down a calendar control so the user can visually select the desired date. However, there are numerous shortcomings with this control:

- Since it's an ActiveX control, it's another file (MSComCtl2.OCX) you have to install and register.

- If you instantiate the control programmatically rather than visually (that is, you didn't drop it on a form or class), the user gets a license error when they run the form unless you specifically hack the Registry to install the license key for the control.
- It can't handle empty dates. The user has no way to blank the date and binding to a control source that has an empty value results in an error: "OLE IDispatch exception code 0 from DTPicker: A date was specified that does not fall within the MinDate and MaxDate properties... Unbinding object."
- The combobox looks old: it has a 3-D sunken appearance rather than the flat appearance of modern controls.

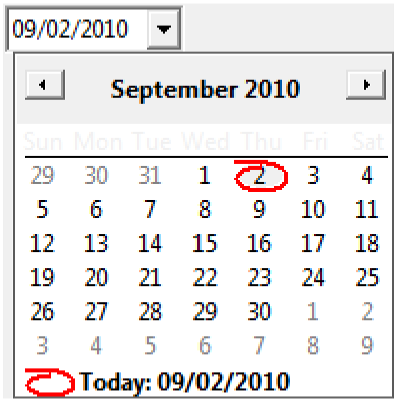


Figure 9. The Microsoft DTPicker.

Fortunately, Paul Mrozowski has created a replacement for DTPicker called RCSDateTimePicker. It's one of several calendar controls in RCSCalendar.VCX, which you can download from http://tinyurl.com/cus8b9. As you can see in **Figure 10**, it has a more modern appearance. Even better, it's written entirely in VFP so there are no ActiveX issues and Paul provides the source code so you can make any changes you wish.

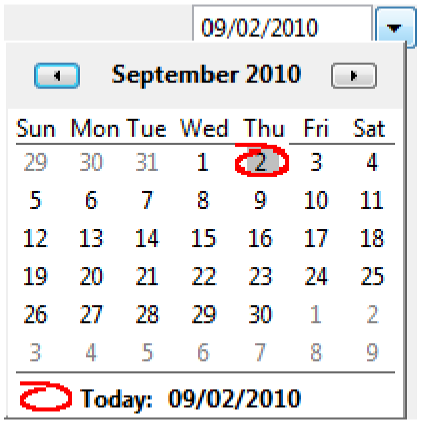


Figure 10. The RCSDateTimePicker has a more modern appearance.

I won't document the properties or methods of RCSDateTimePicker because Paul has done a great job of that in the help file included with the source code. Instead, I'll just focus on some of the uses for this and the other controls he provides.

Using RCSDateTimePicker

Using RCSDateTimePicker is easy: simply drop it on a form, set cControlSource if you want to data bind it, set lConvertToDateTime to .F. if you're binding to a Date value or .T. for a DateTime value, and you're done. The user can type a date, press "D" for today's date, "+" for the next day, "-" for the previous day, Delete to blank the date, or click the down arrow or press the down arrow key or Alt+down arrow to display the calendar. Clicking a date in the calendar closes it and displays that date in the control. If the calendar doesn't show the current month, clicking "Today" or the red circle jumps to the current month. Clicking the arrows move back or ahead a month.

If you don't want to data bind the control, you can either put some code into ValidateParentChild, fired whenever the user changes the date or selects a date from the calendar (don't forget to use DODEFAULT() to ensure the default behavior occurs) or reference the Value property when you want to retrieve the date.

The calendar is an instance of RCSCalendarForm, stored in the oCalendar property. RCSCalendarForm is just a simple borderless form containing an instance of RCSCalendar named ctrCalendar. Clicking the down arrow in the control calls DisplayCalendar to display the calendar. So, if you want to change something about the calendar, override DisplayCalendar, DODEFAULT(), and add whatever code you want. For example, to use Segoe UI as the font for the calendar, use this for DisplayCalendar:

```
dodefault()
This.oCalendar.SetAll('FontName', 'Segoe UI')
```

Some changes require calling RefreshCalendar to redisplay it. For example, if you want the calendar to only display days for the current month, omitting days from the previous and next months on the first and last lines, use this code:

```
dodefault()
This.oCalendar.ctrCalendar.lOnlyDisplayCurrentDays = .T.
This.oCalendar.ctrCalendar.RefreshCalendar()
```

In addition to RCSDateTimePicker's built-in hotkeys, it's easy to add support for others. In the KeyPress method of txtDate in the first RCSDateTimePicker in TestDateCtrls.SCX, I added the following code:

```
lparameters tnKeyCode, ;
   tnShiftAltCtrl
local luValue
luValue = evl(This.Parent.Value, date())
do case
```

```
   case inlist(tnKeyCode, asc('m'), asc('M'))
      This.Parent.Value = luValue - day(luValue) + 1
      nodefault
      This.SetFocus()
   case inlist(tnKeyCode, asc('h'), asc('H'))
      This.Parent.Value = gomonth(luValue - day(luValue) + 1, 1) - 1
      nodefault
      This.SetFocus()
   case inlist(tnKeyCode, asc('y'), asc('Y'))
      This.Parent.Value = date(year(luValue), 1, 1)
      nodefault
      This.SetFocus()
   case inlist(tnKeyCode, asc('r'), asc('R'))
      This.Parent.Value = date(year(luValue), 12, 31)
      nodefault
      This.SetFocus()
   otherwise
      dodefault(tnKeyCode, tnShiftAltCtrl)
endcase
```

You can now press “M” for the first day of the month, “H” for the last day of the month, “Y” for the first day of the year, and “R” for the last day of the year. These keys are easy to remember as they are the first and last letters of the appropriate word, “month” or “year.”

You can connect two (or more) RCSDateTimePicker controls so they synchronize values; this is useful in the case of start and end date controls to prevent the user from specifying an end date that’s earlier than the start date. Set the uChildPicker property of the start date control to either a reference to the end date control or an expression that can be evaluated to it (such as “Thisform.ctrEndDate”). Similarly, set the uParentPicker property of the end date control to tell it about the start date control. When the user specifies a date for either control, RCSDateTimePicker will automatically make the dates the same if the end date is less than the start date.

Let’s extend this idea: suppose you have starting and ending date controls but want to allow the user to select both dates in a single calendar. The sample form TestDateCtrl.SCX shows how to do this:

- Add two RCSDateTimePicker controls to the form and set the uChildPicker and uParentPicker properties appropriately.
- Override the DisplayCalendar method of both controls to the following code. The code in RCSDateTimePicker.DisplayCalendar uses BINDEVENT() to bind the calendar’s AddDate event to the control’s DateSelectedEvent so the latter method is fired when the user clicks a date. However, we don’t want that to happen just yet so this code unbinds it. It then calls the default code to do most of the setup and the form’s DisplayCalendar method to do the rest.

```
if vartype(This.oCalendar) = 'O'
   unbindevents(This.oCalendar.ctrCalendar, 'AddDate', This, 'DateSelectedEvent')
endif vartype(This.oCalendar) = 'O'
dodefault()
```

```
Thisform.DisplayCalendar(This)
```

- Add a DisplayCalendar method to the form with the following code:

```
lparameters toControl
local loCalendar
loCalendar = toControl.oCalendar.ctrCalendar

* We'll use Segoe UI for the calendar font.

loCalendar.SetAll('FontName', 'Segoe UI')

* Allow the user to select more than one date.

loCalendar.lMultiSelect = .T.

* Ensure we have both starting and ending dates in the calendar.

loOther = toControl.GetParentChildReference()
if not empty(loOther.Value)
   if isnull(toControl.uParentPicker)
      loCalendar.AddDate(loOther.Value)
   else
      loCalendar.aSelectedDates[1] = loOther.Value
      loCalendar.AddDate(toControl.Value)
   endif isnull(toControl.uParentPicker)
   loCalendar.RefreshCalendar()
endif not empty(loOther.Value)

* Change the event binding so AddDate fires before DateSelectedEvent.

bindevent(loCalendar, 'AddDate', toControl, 'DateSelectedEvent', 1)
```

- Override the DateSelectedEvent of both controls so the default code doesn't execute but instead calls the form's DateSelected method:

```
lparameters tdDate
Thisform.DateSelected(This, tdDate)
```

- Add a DateSelected method to the form with the following code. Because it's called from the DateSelectedEvent of both controls, it fires when the user clicks a date. The first time they click a date, the calendar's iSelectedDateCount property is 1, so the code puts that date into the starting date control. When they click a second date, the code puts that date into the ending date control and closes the calendar. If they open the calendar again (meaning two dates are already selected) and they click a new date, either to replace the starting or ending date, the code puts that date into the appropriate control and closes the calendar.

```
lparameters toControl, ;
   tdDate
local loOther, ;
   loStart, ;
   loEnd
```

```
loOther = toControl.GetParentChildReference()
if isnull(toControl.uParentPicker)
   loStart = toControl
   loEnd   = loOther
else
   loStart = loOther
   loEnd   = toControl
endif isnull(toControl.uParentPicker)
with toControl.oCalendar.ctrCalendar
   do case
      case .iSelectedDateCount = 1
         loStart.txtDate.Value = tdDate
      case .iSelectedDateCount >= 2
         lnIndex = .iSelectedDateCount
         if .iSelectedDateCount > 2 and tdDate < .aSelectedDates[1]
            loStart.txtDate.Value = tdDate
            lnIndex = .iSelectedDateCount - 1
         endif .iSelectedDateCount > 2 ...
         loEnd.Value = .aSelectedDates[lnIndex]
         toControl.oCalendar.Visible = .F.
   endcase
endwith
```

The effect of this is that the user can click the down arrow for either control and select two dates, the first being the starting date and the second being the ending date.

Other controls

RCSCalendar.VCX contains some other classes you can use:

- You can drop RCSCalendar, the class used as the calendar for RCSDateTimePicker, on a form and use it as a calendar control. Used this way, RCSCalendar has one behavior it doesn't have when used from RCSDateTimePicker: you can click the month to display a popup menu of other months to select from. You can add code to DateClick, DateDoubleClick, DateRightClick, DateMouseEnter, DateMouseLeave, or any of the other events. Set lMultiSelect to allow the user to select multiple dates; you can then look at aSelectedDates to see which dates the user selected. The sample TestCalendar.SCX that comes with RCSCalendar, shown in **Figure 11**, shows an example of this and the next three classes mentioned.
- RCSSmallCalendar and RCSTinyCalendar are smaller versions of RCSCalendar.
- RCSctrDateTimePicker contains both an RCSCalendar and an RCSctrTime control so the user can specify a date and time.

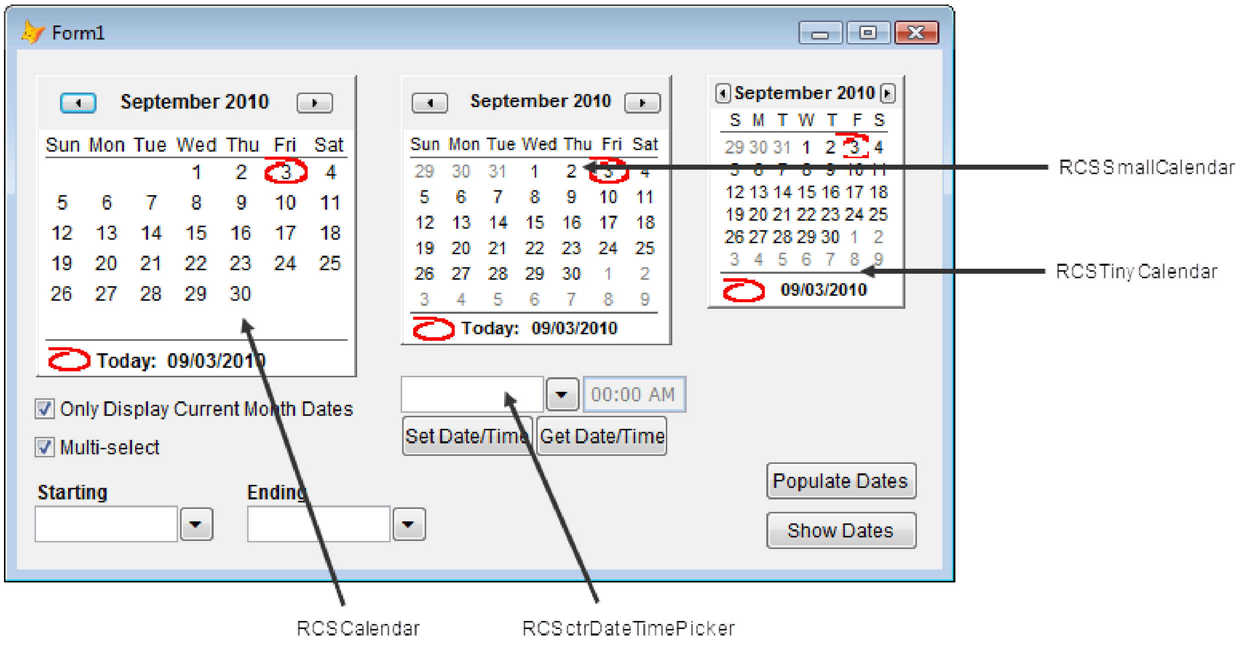


Figure 11. TestCalendar.SCX includes several of the controls in RCSCalendar.VCX

- RCSLargeCalendar is a larger version of RCSCalendar but has another behavior: it can display text and icons in each cell, such as to show events. Use code such as the following, taken from TestLargeCalendar.SCX (shown in **Figure 12**), to add events to the calendar:

```
loEvent = Thisform.ctrCalendar.GetEventObject()
loEvent.dDate = ldDate
loEvent.cClickEval = [MESSAGEBOX('You clicked on: ' + TRANSFORM(loEvent.dDate))]
loEvent.cImage = lcIcon
loEvent.cCaption = "Item " + TRANSFORM(lnCount)
loEvent.nColor = lnColor
loEvent.cType = "Demo"
loEvent.uKey = lnCount
ThisForm.ctrCalendar.AddEvent(loEvent)
```

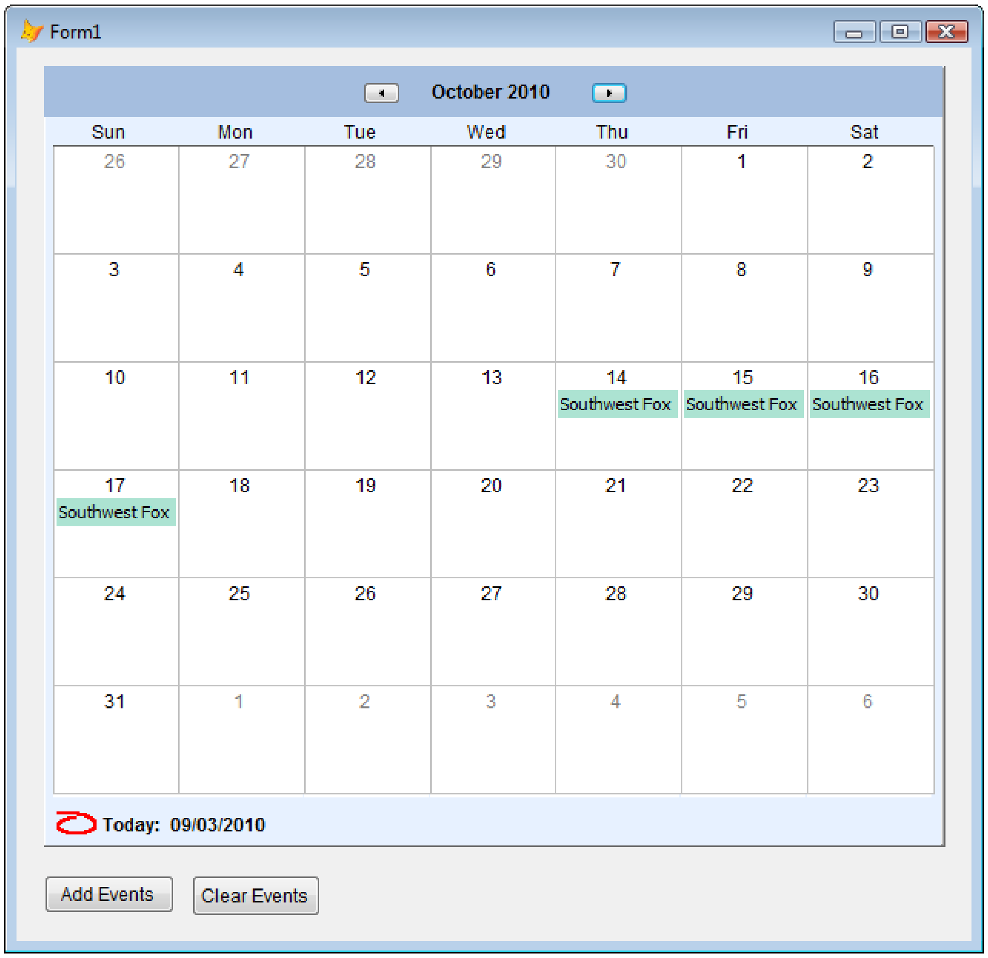


Figure 12. RCSLargeCalendar can display events.

Summary

Paul Mrozowski has done an excellent job of creating calendar controls you can easily add to any VFP application. Because he used 100% VFP code, there are no ActiveX issues. Also, he provides both code and documentation, making it easy to figure out how the controls work, how to use them, and how to extend their features.

Balloon Tips

The tooltips VFP displays (which are actually defined by Windows, not VFP) are kind of boring looking. As you can see in **Figure 13**, the tooltip window is tiny, with a yellow background, and provides no control over its appearance other than the text. For a more attractive appearance and much better control, try using ctl32_BalloonTip.

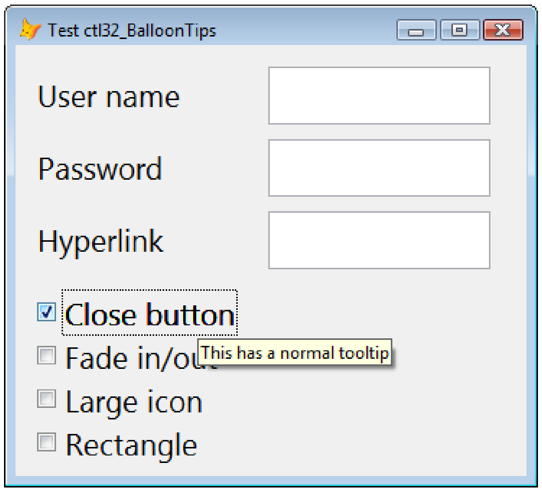


Figure 13. VFP's tooltips are kind of boring looking.

ctl32_BalloonTip, part of the ctl32 controls created by Carlos Alloatti of Argentina, displays tooltips such as that in **Figure 14**. Note the tooltip window has a “word balloon” appearance, a gradient background, a title, an icon, and of course, the tooltip text.

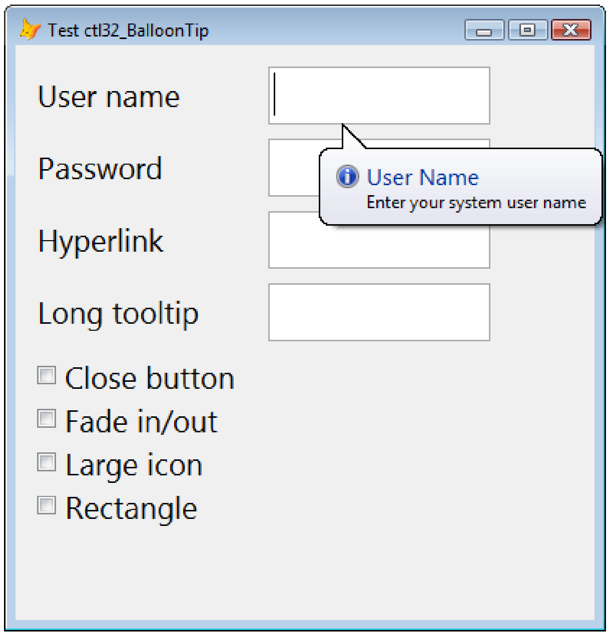


Figure 14. ctl32_BalloonTip displays a more attractive tooltip.

Download the ctl32 controls from http://www.ctl32.com.ar and unzip it in some folder. Add the following files added to your project:

- ctl32.prg
- ctl32.vct
- ctl32.vcx

- ctl32_api.prg
- ctl32_classes.prg
- ctl32_functions.prg
- ctl32_structures.prg
- ctl32_vfp2c32.prg
- vfpx.vct
- vfpx.vcx

These files add about 1.5MB to your executable.

In addition to ctl32_BalloonTip, the ctl32 controls include many other useful controls, such as a shortcut menu similar to PopMenu, a date picker similar to RCSDateTimePicker, a progress bar (discussed in the next section), a status bar, and so forth. You should spend some time going through the samples that come with ctl32 to see what's available and how they work.

Using ctl32_BalloonTip

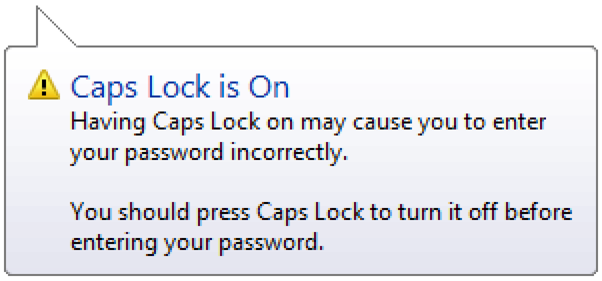
Start by dropping an instance of ctl32_BalloonTip on a form. Even though individual controls will have different tooltips, there can only be one instance of ctl32_BalloonTip per form. When a control that should display a balloon tip receives focus, call the ctl32_BalloonTip's ctlShow method with the appropriate parameters to display the balloon. Similarly, when the control loses focus, call ctlShow(0) to hide the balloon. If you want balloon tips displayed when the mouse moves over the control, regardless of whether it has focus or not, call ctlShow.

To make it easy to use balloon tips for textboxes, I created a subclass of Textbox called SFTextBox with the following changes:

- I added custom cBalloonTipTitle (the title of the balloon tip), lHaveBalloonTipControl (set to .T. in Init if the form has a balloon tip control), and nBalloonTipIcon (the icon to use for the balloon tip) properties.
- Init sets lHaveBalloonTipControl if the form has an object named oBalloonTip, which is assumed to be an instance of ctl32_BalloonTip.
- I added two custom methods: ShowBalloonTip and HideBalloonTip. HideBalloonTip simply calls Thisform.oBalloonTip.ctlShow(0) to hide the balloon tip. ShowBalloonTip has the following code to display the balloon tip:

```
lparameters tlGotFocus
#define CON_BTPOS_ACTIVECTRL  2
#define CON_BTPOS_MOUSE         6
do case
   case empty(This.cBalloonTipTitle) or not This.lHaveBalloonTipControl
   case not empty(This.PasswordChar)
      Thisform.oBalloonTip.ctlCapsLockStyle = .T.
   otherwise
      Thisform.oBalloonTip.ctlShow(iif(tlGotFocus, CON_BTPOS_ACTIVECTRL, ;
         CON_BTPOS_MOUSE), This.ToolTipText, This.cBalloonTipTitle, ;
         This.nBalloonTipIcon)
endcase
```

This code sets ctlCapsLockStyle to .T. if this is a password textbox, which causes a special balloon tip to appear if the Caps Lock key is turned on (**Figure 15**). Otherwise, it displays a balloon tip with the appropriate information.


Caps Lock is On
Having Caps Lock on may cause you to enter your password incorrectly.

You should press Caps Lock to turn it off before entering your password.

Figure 15. Setting ctlCapsLockStyle to .T. displays this balloon tip when the Caps Lock key is turned on.

- GotFocus and MouseEnter call ShowBalloonTip, although GotFocus passes .T. to indicate that we're receiving focus rather than the mouse moving over the control. The code in ShowBalloonTip puts the balloon tip over the control if passed .T. and near the mouse if passed .F.
- LostFocus and MouseLeave call HideBalloonTip.

To use SFTextBox, first drop a ctl32_BalloonTip control on a form and name it oBalloonTip. Then drop an SFTextBox on a form, set TooltipText to the text to display, cBalloonTipTitle to the title, and nBalloonTipIcon to the icon to use. If you want to change how a balloon tip appears, such as using a close button or large icons, set the appropriate properties of oBalloonTip. If you want to change something about this control's balloon tip, such as specifying which icon to use, override the code in ShowBalloonTip. For example, this code shows how to add a hyperlink to a balloon tip and how to specify a different icon:

```
#define TTI_ERROR      3

lparameters tlGotFocus
local lcText
with Thisform.oBalloonTip
   lcText = .ctlMakeLink('ctl32_BalloonTip Documentation', ;
      'http://www.ctl32.com.ar/ctl32_balloontip_members.asp')
   This.ToolTipText = 'This displays a hyperlink. Check ' + ;
      lcText + ' for details.'
   This.nBalloonTipIcon = TTI_ERROR
endwith
dodefault(tlGotFocus)
return
```

There are lots of other properties you can set to control the appearance of the balloon tip, including its background color, text font and size, position, margins, and so on. http://tinyurl.com/2dg4md8 has complete documentation on the properties and methods for ctl32_BalloonTip. Some you might find useful are:

- ctlCloseButton: set this to .T. to display a close button in the balloon tip window.
- ctlFadeIn and ctlFadeOut: set these to .T. to have the balloon tip window fade in and fade out.
- ctlIconLarge: set this to .T. to display a large version of the specified icon.
- ctlActive: set this to .F. to disable all balloon tips (you might do this for more experienced users, for example).
- ctlStyle: set this to 2 to display a rectangle rather than the default balloon shape (1).

Summary

ctl32_BalloonTip allows you to display more attractive and informative tooltips that your users will appreciate. It's pretty easy to add balloon tips to your forms, especially if you make changes similar to those in SFTextBox to your base class controls and add a ctl32_BalloonTip to your base class form. All you have to do then is set the ToolTipText and cBalloonTipTitle properties of the various controls and you're done.

Progress Bar

For several years, I used the Microsoft ProgressBar control to show the progress of some action. However, like the Microsoft DateTime control I discussed earlier, this is an ActiveX control, so it has the same issues as the DateTime control. And like the DateTime control, it looks dated (see the top progress bar in **Figure 16**).

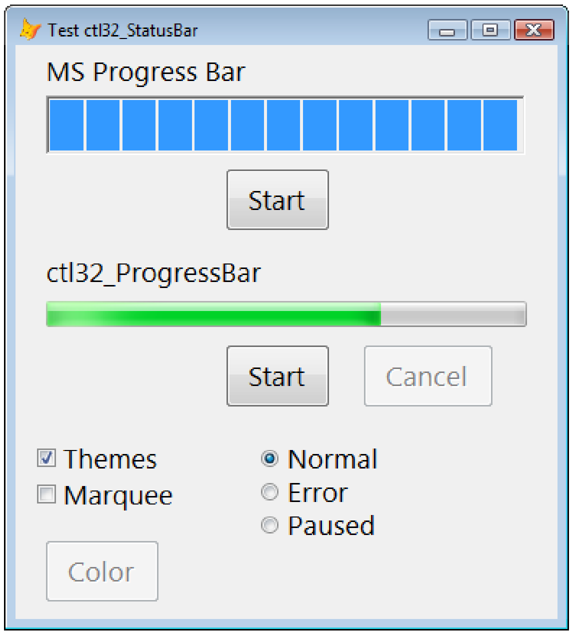


Figure 16. The ctl32_ProgressBar control is more attractive and flexibility than the Microsoft ProgressBar control.

Fortunately, the ctl32 library includes ctl32_ProgressBar, a very attractive, modern looking progress bar that also has more flexibility than the Microsoft control (see the second progress bar in **Figure 16**). Because it's part of ctl32, it comes along for the ride if you're using any of the other ctl32 controls.

http://tinyurl.com/2v9vty7 has documentation for the properties and methods of ctl32_ProgressBar. I'll only discuss the more commonly used properties here.

Using ctl32_ProgressBar

Using ctl32_ProgressBar is very straightforward:

- Drop an instance on a form.
- Set some properties as necessary. If the range of values isn't 0 to 100, set ctlMinimum and ctlMaximum to the desired range. If you don't wanted a themed progress bar (I'm not sure why you wouldn't), set ctlThemes to .F. and other properties for non-themed progress bars, such as ctlBorderColor, ctlForeColor, ctlBackColor, and so on. If you want the bar to be red or yellow, set ctlState to 2 or 3, respectively.
- To indicate that something is happening, set ctlValue to the desired value. The progress bar displays a bar of the appropriate length.
- If you want to show a "marquee" effect (a block continuously scrolls across the progress bar without you changing ctlValue), set ctlMarquee to .T.

- If you want to know the current percentage that ctlValue is of ctlMaximum, use ctlPercent (a numeric value) or ctlValuePercent (a character string formatted with a %).

If you run the sample TestProgressBar form, you'll notice something interesting: the progress bar length lags behind the percentage slightly. That's because the progress bar actually uses animation to draw the bar to the correct length. For example, if you click the "Set to 50" button, you'll notice that the bar doesn't jump to 50 but instead grows towards it.

Summary

There's very little effort in using ctl32_ProgressBar, and if you're using any of the other ctl32 controls, there's no additional overhead in your EXE.

Summary

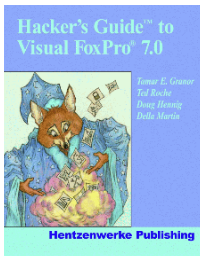
There's no excuse for creating a boring looking VFP application. Using the controls discussed in this document, and others available on VFPX or other sites, you can create a new, modern user interface for your forms that'll add years to the life of your applications. With a few days of effort, your apps can be as pretty as any .Net application. Get started today!

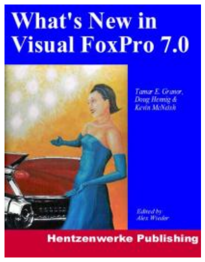
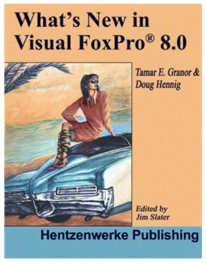
Biography

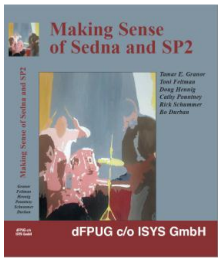
Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

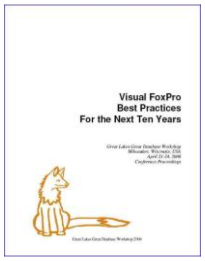
Doug is co-author of "Making Sense of Sedna and SP2," the "What's New in Visual FoxPro" series (the latest being "What's New in Nine"), "Visual FoxPro Best Practices For The Next Ten Years," and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).




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





Copyright, 2010 Doug Hennig.