Splitting Up is Hard to Do

Doug Hennig

While they aren't used everywhere, splitter controls can add a professional look to your applications when you have left/right or top/bottom panes in a window. This month's article implements a splitter using the new OLE drag and drop features added in VFP 6.

Splitters are interesting controls: they don't really 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 the 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 could have both types of splitter on the same form. A VFP example of a splitter is in the Class Browser; you can adjust the sizes of the four panes using both horizontal and vertical splitters.

Recently, I needed to add a splitter control to a form I was working on. I looked at the ctSplitter ActiveX control from DBI Technologies (who make wonderful ActiveX controls, most of which work just splendidly with VFP) but couldn't make it work because it expects to work with windows, and VFP forms are not true Windows windows (in other words, it isn't DBI's fault). So, I figured that since the Class Browser has a splitter control and we now have source code for the Class Browser (included in XSOURCE.ZIP in the TOOLS subdirectory of the VFP home directory), I'd just steal, er, borrow the splitter control used in that tool. However, after looking at the code, I ran into a couple of issues: I don't like the visual behavior of the control (the other objects on the form become invisible while you drag the splitter), and it's so specific to the Class Browser that making it work with my form (or even better, creating a reusable tool out of it) would be more work than it's worth.

So, I was stuck with building my own. After going down the same path as the Class Browser splitter for a while (trapping the MouseDown event of a shape object and then using DOEVENTS to allow the user to drag the shape around) and not enjoying where I was going, I decided to step back and think about things a bit. It occurred to me that the behavior I wanted was to drag a shape on a form and have it adjust the size and position of other controls as I did so. My first thought was to use VFP drag and drop, but I quickly discarded that idea when I remembered that the source object (the thing being dragged) doesn't get any events during the drag operation; instead, the target objects (the things being draggged over) get a DragOver event. I didn't want to put code into the DragOver event of every object on a form (which would be required so the shape could be dragged anywhere), so I discarded that idea. Then I remembered that with OLE drag and drop, new to VFP 6, the source object *does* receive events when it's dragged over a target object. Looks like we're on the right track so far.

Not having used OLE drag and drop before, I turned to my main resource for information on VFP, a book who's name I won't mention because I was the technical editor, FoxTalk editor Whil Hentzen is the publisher, and the authors are friends of mine (OK, it's the Hacker's Guide to VFP 6 <g>). The event that seemed most promising was OLEGiveFeedback, which is fired in the source object after the OLEDragOver event of the target object is fired. This seemed like the perfect place to adjust the sizes of the objects being managed by the splitter. That led to another complication: only objects that have the OLEDropMode property set to 1 (Enabled) would cause OLEGiveFeedback to fire. So, I was back to setting this property to .T. for each object on the form, right?

Well, a thought occurred to me. What if we place an invisible object on the form that covers everything else and have it be the target for the OLE drag and drop by setting its OLEDropMode to 1? That proved to be the winning strategy.

OK, enough of following my twisted thought processes. Let's get into some code!

SFSplitter

The first class I created is SFSplitter, contained in SFSPLITTER.VCX. This class, which is based on SFShape (our shape base class in SFCTRLS.VCX), won't be used directly but will serve as the parent class for horizontal and vertical splitter classes. Although the shape will be invisible at runtime, it's easier to work with if it has a visual appearance at design time, so I left the BorderStyle property at the default and

made the Init method set it to 0 (none) at runtime. If it's invisible, how does the user know a splitter is available? First, we'll drop it between two controls (such as listboxes, TreeViews, ListViews, etc.) that might logically be adjusted with a splitter, and second, we'll set the MousePointer property so when the mouse is over the control, it looks like a splitter. We won't set MousePointer in this class, because which pointer to use depends on whether we have a horizontal or vertical splitter, so we'll do that in a subclass.

The Init method sets BorderStyle to 0 as I mentioned, then it adds an SFSplitterCover object (which we'll talk about later) to the form. Notice it uses the new NewObject method (added in VFP 6) of the form and specifies its own ClassLibrary property as the location of the SFSplitterCover class. It also assigns a unique name to the object if one wasn't specified in the cSplitterCoverName property, and stores the assigned name in this same property (we need to know the name because we'll be talking to it later). Because SFSplitter adds this object (which collaborates with it during the drag operation) to the form, this class doesn't require that you do anything to the form or other objects to use it. That's my idea of a reusable tool! Here's the code for Init:

```
local lcName
with This

* Make the border invisible.

  .BorderStyle = 0

* Add an SFSplitterCover object to the form.

  lcName = iif(empty(.cSplitterCoverName), sys(2015), ;
    .cSplitterCoverName)
  .cSplitterCoverName = lcName
  Thisform.NewObject(lcName, 'SFSplitterCover', ;
    .ClassLibrary)
endwith
```

To enable OLE drag and drop, I set the OLEDragMode property to 1 (Automatic) so OLE drag and drop starts as soon as the user starts dragging the shape. The three OLE drag and drop events we're interested in are OLEStartDrag, which fires when the drag starts, OLEGiveFeedback, which fires when the object is dragged over a target, and OLECompleteDrag, which fires when the drag operation is done. Here's the code for OLEStartDrag:

```
lparameters toDataObject, ;
  tnEffect
This.StartMoving()
```

Here's the code for OLEGiveFeedback:

```
lparameters tnEffect, ;
  tuMouseCursor
tuMouseCursor = This.MousePointer
This.MoveSplitter()
```

Here's the code for OLECompleteDrag (unlike its name suggests, OLE drag and drop is actually kind of fun <g>):

```
lparameters tnEffect
This.DoneMoving()
```

With the slight exception of OLEGiveFeedback, each of these events simply calls a custom method of the form (I'm taking to heart Steven Black's advice that "events call methods"). OLEGiveFeedback also sets the mouse pointer for the drag operation (passed by reference to this event in the tuMouseCursor parameter) to the same value as the class itself uses, so we get a consistent look as the splitter is dragged.

StartMoving, called from OLEStartDrag, makes the SFSplitterCover object it added to the form visible, the same size as the form, and "above" everything else on the form so it will be the sole target object for the

drag operation. StartMoving also calls the SetStartingPositions method, which isn't implemented in this class because it's specific to the type (horizontal or vertical) of splitter we'll use.

```
local loCover

* Size the SFSplitterCover object to cover the entire
* form, make it visible, and move it to the top of the
* Z order.

loCover = evaluate('Thisform.' + This.cSplitterCoverName)
with loCover
  .Width   = Thisform.Width
  .Height  = Thisform.Height
  .Visible = .T.
  .ZOrder(0)
endwith

* Call a method to set the starting positions.

This.SetStartingPositions()
```

MoveSplitter, called from OLEGiveFeedback, is an abstract method because its behavior depends on whether the splitter is horizontal or vertical. I also added an abstract AfterMoveSplitter method that can be used in instances or subclasses of a splitter to perform additional operations.

DoneMoving, called from OLECompleteDrag, simply hides the SFSplitterCover object when the drag operation is done:

```
local loCover
loCover = evaluate('Thisform.' + This.cSplitterCoverName)
loCover.Visible = .F.
loCover.ZOrder(1)
```

Two other methods, GetFirstObject and GetObjectNames, aren't called by anything in SFSplitter, but will be called from subclasses. These methods accept a comma-delimited list of names and return the first name in the list and populate an array with the names, respectively. We'll see how they're used later.

SFSplitterCover

Like SFSplitter, SFSplitterCover is based on SFShape. Its purpose is very simple: act as a target for an OLE drag and drop operation of an SFSplitter object. Its OLEDropMode property is set to 1 (Enabled) so it can be a target, and its BorderStyle is set to 0 so it doesn't have any visual representation. Its OLEDragOver event, fired when the SFSplitter object is dragged over it, simply stores the X and Y coordinates passed to it to custom nXCoord and nYCoord properties:

```
lparameters toDataObject, ;
  tnEffect, ;
  tnButton, ;
  tnShift, ;
  tnXCoord, ;
  tnYCoord, ;
  tnState
This.nXCoord = tnXCoord
This.nYCoord = tnYCoord
```

SFSplitterH and SFSplitterV

SFSplitterH is a horizontal splitter subclass of SFSplitter. The way it works is that one or more objects to the left of the splitter are considered to be anchored at their left edges, so only their widths will be adjusted as the splitter is moved. One or more objects to the right of the splitter are anchored at their right edges, so both their Left and Width are adjusted accordingly. The net effect is that as the splitter is moved to the right, the left objects get wider at the expense of the right objects, and vice versa as the splitter is moved to the left. The relative distances of the left objects from the left edge of the form, the right objects from the right edge of the form, and the left and right objects from the splitter and each other stay fixed.

I set the Height property of SFSplitterH to 200, Width to 10 (you'll likely change the Height of an instance of this class to match the objects you're splitting but probably won't need to change the Width), and MousePointer to 9 (Size W E). Only two methods are overridden in this class. The first is SetStartingPositions:

```
local lcObject, ;
  loObject
with This

* Save the distance between the left object and the
* splitter.

  lcObject = .GetFirstObject(.cLeftObjectName)
  loObject = evaluate('.Parent.' + lcObject)
  .nLeftObjectSpace = .Left - loObject.Width

* Save the distance between the right object and the
* splitter and the object's width.

  lcObject = .GetFirstObject(.cRightObjectName)
  loObject = evaluate('.Parent.' + lcObject)
  .nRightObjectLeft  = loObject.Left - .Left
  .nRightObjectWidth = loObject.Left + loObject.Width
endwith
```

This code calls the GetFirstObject method, passing it the value of the custom cLeftObjectName property. The idea of cLeftObjectName is that you'll enter a comma-delimited list of the names of the objects to the left of the splitter control. SFSplitterH assumes that while there may be more than one "left" object being managed by the control, they're all have the same Width, so only the first one is used. nLeftObjectSpace contains the distance between the right edge of the left controls and the left edge of the splitter (not quite true because we aren't taking the Left property of the left objects into consideration, but since we won't be changing that property of these objects, we can factor that out of the equation). A similar calculation is done for the first right object listed in cRightObjectName, with the distances between it and the splitter and it and the right edge of the form stored in nRightObjectLeft and nRightObjectWidth, respectively. Why are these values calculated when a drag is started rather than in the Init of the class? You might programmatically assign the names in cLeftObjectName and cRightObjectName rather than doing it in the Property Sheet. Why are names listed in the properties rather than storing object references to them? It's easier to specify the objects by listing them in the Property Sheet than having to write code that places object references somewhere (likely an array), plus it avoids the issue of dangling object references caused by not destroying extra references to objects.

The other method with code in SFSplitterH is MoveSplitter, which is called from OLEGiveFeedback when the splitter is dragged over the target.

```
local loCover, ;
  lnXMovement, ;
  lcObject, ;
  loObject1, ;
  loObject2, ;
  lnWidth1, ;
  lnLeft, ;
  lnWidth2, ;
  laObjects[1], ;
  lnObjects, ;
  lnI, ;
  loObject
with This

* Get the current X coordinate for the drag from the
* SFSplitterCover object.

  loCover = evaluate('Thisform.' + .cSplitterCoverName)
  lnXMovement = loCover.nXCoord

* Get object references to the left and right objects
```

```
* and adjust their width and left/width properties,
* respectively, as long as we don't make one of them
* smaller than the minimum width. Then move ourselves
* to the X position.

  lcObject  = .GetFirstObject(.cLeftObjectName)
  loObject1 = evaluate('.Parent.' + lcObject)
  lcObject  = .GetFirstObject(.cRightObjectName)
  loObject2 = evaluate('.Parent.' + lcObject)
  lnWidth1  = lnXMovement - .nLeftObjectSpace
  lnLeft    = lnXMovement + .nRightObjectLeft
  lnWidth2  = .nRightObjectWidth - lnLeft
  if lnWidth1 >= .nLeftMinWidth and ;
    lnWidth2 >= .nRightMinWidth
    lnObjects = .GetObjectNames(.cLeftObjectName, ;
      @laObjects)
    for lnI = 1 to lnObjects
      loObject = evaluate('.Parent.' + laObjects[lnI])
      loObject.Width = lnWidth1
    next lnI
    lnObjects = .GetObjectNames(.cRightObjectName, ;
      @laObjects)
    for lnI = 1 to lnObjects
      loObject = evaluate('.Parent.' + laObjects[lnI])
      loObject.Width = lnWidth2
      loObject.Left  = lnLeft
    next lnI
    .Left = lnXMovement

* Call a hook method so a subclass could do something
* else (like moving other objects).

    .AfterMoveSplitter(lnXMovement)
  endif lnWidth1 >= .nLeftMinWidth ...
endwith
```

This code gets the value of the nXCoord property of the SFSplitterCover object; this property was set to the current X coordinate of the drag operation by the OLEDragOver method of that object. It then determines how much to adjust the width of the left objects and the left positions and widths of the right objects. If the width of either the left or right objects would be made too small (enter the minimum values into the nLeftMinWidth and nRightMinWidth properties), nothing is moved or adjusted. Otherwise, each of the left and right objects is adjusted and the Left property of the splitter itself changed to match the mouse position. Finally, the abstract AfterMoveSplitter is called with the mouse position as a parameter so an instance or subclass of the splitter could do some additional tasks.

SFSplitterV is a vertical splitter subclass of SFSplitter. I won't show its code here because it's almost identical to SFSplitterH, except instead of managing the Left and Width properties of left and right objects, it manages the Top and Height properties of objects above and below the splitter. Enter the names of the objects to manage in the cTopObjectName and cBottomObjectName properties and their minimum heights in the nTopMinHeight and nBottomMinHeight properties. The Height and Width of this class are reversed from SFSplitterH so it appears as a horizontal shape, and its MousePointer is 7 (Size N S).

As an extra goody, I've created BuilderD builders (see my March 1999 column for information on BuilderD) for both SFSplitterH and SFSplitterV. Simply open the BUILDERD table in the WIZARDS subdirectory of your VFP home directory and append from the BUILDERS table included with this month's source code.

Sequence of Events

Just so it's clear what happens when, here's the sequence of events that fire. When the SFSplitter subclass is instantiated, it adds an SFSplitterCover object to the form, but this object is invisible for now. When the user starts dragging the splitter object, its OLEStartDrag event fires, which makes the SFSplitterCover object visible (although it's transparent and has no border so it doesn't really appear to be visible) and the topmost object. As the splitter is dragged, the OLEDragOver event of the SFSplitterCover object fires, which stores the current mouse location. Then the OLEGiveFeedback event of the SFSplitter object fires,

which moves all the managed objects to their new locations. When the user lets go of the mouse, the OLECompleteDrag event of the SFSplitter object fires, which makes the SFSplitterCover object invisible again and moves it to the back of the Z order.

Sample Form

Let's check out these splitter controls. To make it interesting, let's put both vertical and horizontal splitters on a form *and* let's throw an SFResizable object (adjusts the sizes of controls as a form is resized as discussed in my December 1998 column) on the form just for grins.

TEST.SCX contains four ListBox objects, named List1 to List4 (see Figure 1). The Resize method of the form calls the AdjustControls method of an SFResizable object named oResizer; this object has its properties set so List4 is resized in both directions, List3 is only resized vertically, List2 is only resized horizontally, and List1 isn't resized at all. An SFSplitterH object sits between the left and right list pairs and has cLeftObjectName set to "List1, List3", cRightObjectName set to "List2, List4", and both nLeftMinWidth and nRightMinWidth set to 100. An SFSplitterV object sits between the top and bottom list pairs and has cTopObjectName set to "List1, List2", cBottomObjectName set to "List3, List4", and both nTopMinHeight and nBottomMinHeight set to 40. The DoneMoving method of both splitters calls the base behavior with DODEFAULT(), then calls the Reset method of the SFResizable object so it can adjust itself to new object sizes.

Figure 1. TEST.SCX.

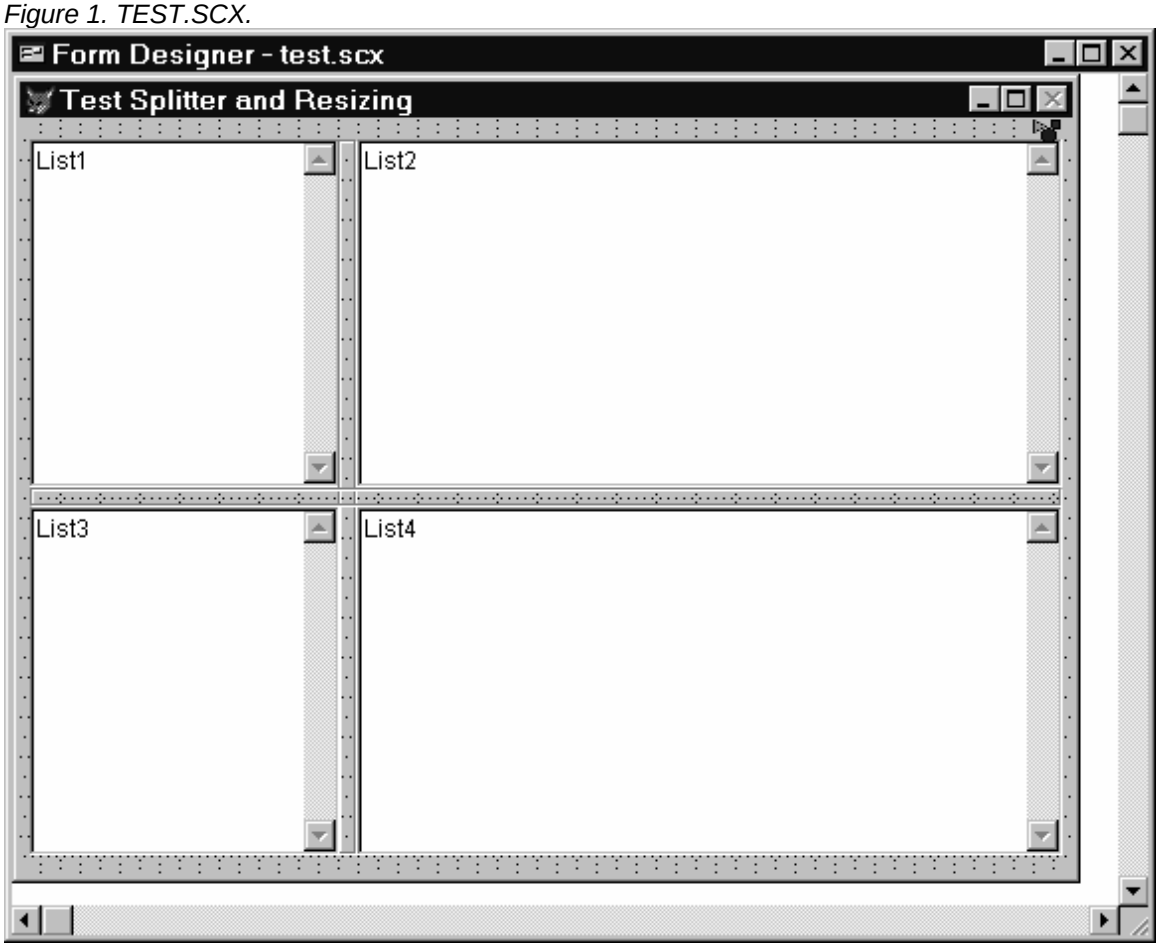
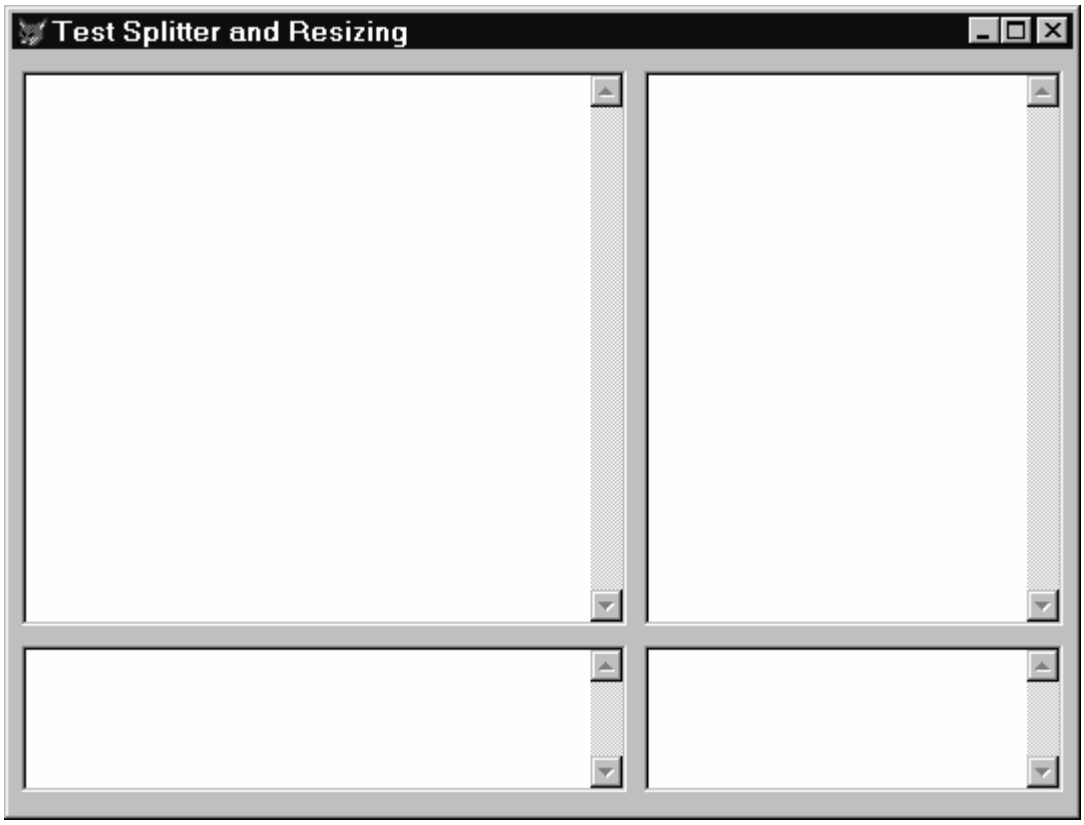


Figure 2 shows the results of running this form and using both the horizontal and vertical splitters to resize the listboxes. Try resizing the form and seeing how that affects the appearance of the objects.

Figure 2. TEST.SCX after using both splitters.



Conclusion

Splitter controls add a professional polish to an application because users are starting to expect that they can adjust the sizes of left/right or top/bottom panes in a window. The reusable classes presented in this article are self-contained: just drop them on a form, set some properties, and your users suddenly have new freedom to rearrange your forms as they see fit.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit and Stonefield Query. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP). He can be reached at dhennig@stonefield.com.