Custom UI Controls: Splitter

Doug Hennig

Adding a splitter control to your forms gives them a more professional behavior and allows your users to decide the relative sizes of resizable controls.

Over the next few issues, I'm going to discuss some custom UI controls I frequently use in my forms. We'll start with a splitter 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.

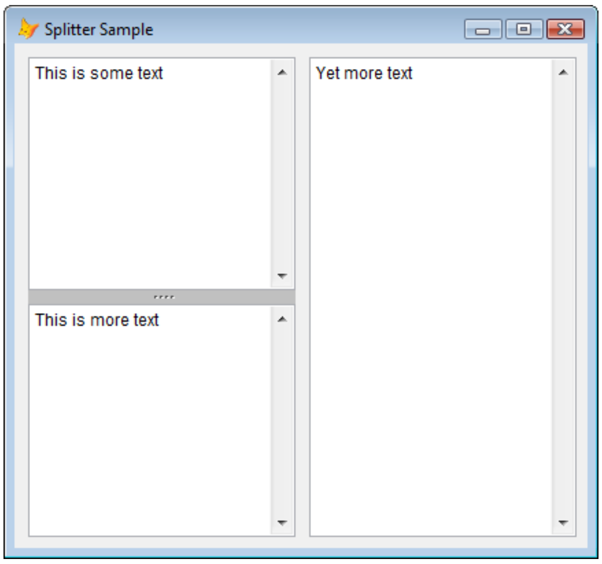

Figure 1. The vertical splitter is visible with a "gripper" but the horizontal splitter is invisible.

There are two styles in which the splitter can appear: visible with a "gripper" or invisible. The vertical splitter 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 article 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 MoveSplitter-ToPosition 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 MoveSplitter-ToPosition 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 Mouse-Move, 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 MoveSplitterTo-Position, 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 MoveSplitter-ToPosition, 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 MoveSplitter-ToPosition, AdjustMinimum, adjusts the Min-Width 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.

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://fox.wikis.com/wc.dll?Wiki~FoxProCommunity LifetimeAchievementAward~VFP).