# Taking Control

*Doug Hennig*

**This month, Doug Hennig discusses a simple way to make anchoring work the way you expect it to and how to control the appearance and behavior of a report preview window.**

There are a couple of issues I've wanted to discuss for some time, neither of which is big enough for a complete article. I kept putting them off, hoping they'd fit with the overall topic of another article. That hasn't happened yet, and since they're kind of important, I decided to combine them into one article even though they don't seem related. However, there is a common theme between them: controlling the appearance and behavior of something in ways other than VFP would normally provide. These topics are dealing with some problematic anchoring issues and controlling the appearance and behavior of a report preview window.

## The trouble with Anchors

The Anchor property added in VFP 9 is a wonderful addition; it allows you to eliminate reams of ugly code (if you handled resizing forms manually in the past) and allows you to easily specify exactly how each control in a form behaves when its container is resized. However, sometimes VFP seems to have a mind of its own when it comes to resizing or moving controls. The problem is that it remembers the original size and placement of the controls and the original size of the form and bases the decisions it makes on those values. If you change things programmatically, resizing no longer works as expected.

Here's one example. Sometimes I hide controls under certain conditions (security, configuration, and so forth), and move the controls below the hidden ones up so there isn't a gap in the spacing of the visible controls in the form. The problem is when the controls have Anchor set to something other than 0, VFP remembers the original placement of the moved controls and the form doesn't look right. **Figure 1** shows what the ResizeDemo1 form included in this month's Subscriber Downloads looks like in the Form Designer.

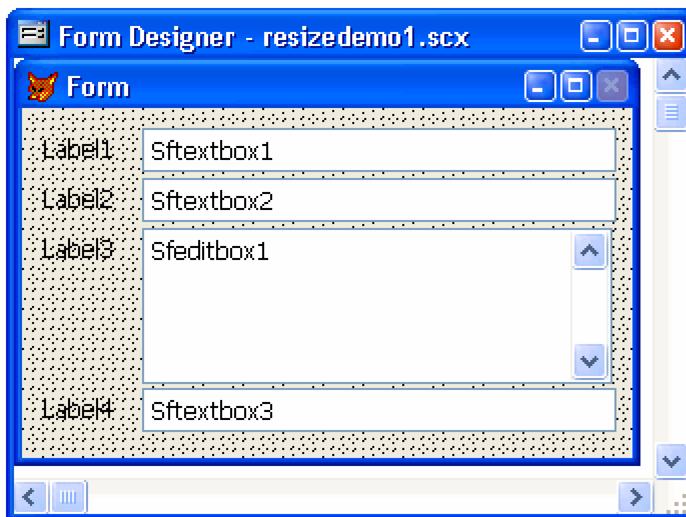


*Figure 1. ResizeDemo1.SCX as it appears in the Form Designer.*

The following code in the Init method hides the second label and text box and moves the controls below these up. However, as you can see in **Figure 2**, the edit box and third textbox don't move to the proper place when you run the form.

```
with This
  store .F. to .Sflabel2.Visible, .Sftextbox2.Visible
  .Sflabel3.Top    = .Sflabel2.Top
  .Sfeditbox1.Top = .Sftextbox2.Top
```

```
  .Sftextbox3.Top = .Sfeditbox1.Top + ;
    .Sfeditbox1.Height + 3
  .Sflabel4.Top    = .Sftextbox3.Top + 3
  .Height          = .Height - 25
endwith
```

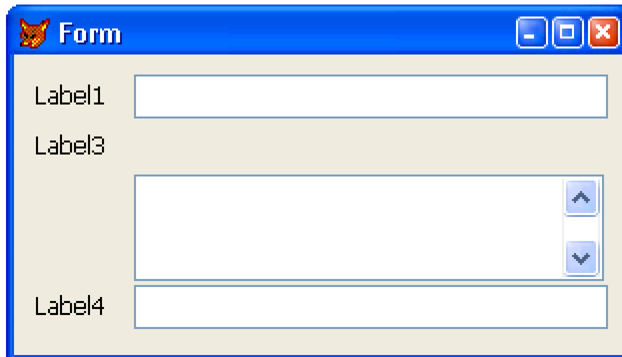


*Figure 2. The controls aren't placed properly because their Anchor properties cause them to move unexpectedly.*

The solution to the problem is simple: set the Anchor property of each control to 0 before moving them and reset Anchor back to the proper values afterward. Of course, that's easier to say than it is to do; the code would have to iterate through every control on the form, including drilling down into containers such as page frames, save the current Anchor value somewhere (such as an array), and then reverse the process after moving the controls.

To make that seemingly ugly chore easier, I added nSavedAnchor and lSaveAnchor properties to every one of my base classes (in SFCtrls.VCX) with an Anchor property. lSaveAnchor has an assign method with the following code:

```
lparameters tuNewValue
if tuNewValue
  This.nSavedAnchor = This.Anchor
  This.Anchor       = 0
else
  This.Anchor = This.nSavedAnchor
endif tuNewValue
```

The purpose of these two properties is to save and restore the anchor value on demand. Rather than having SaveAnchor and RestoreAnchor methods, having a single property (lSaveAnchor) with an assign method means you can use Container.SetAll('lSaveAnchor', .T.) to have every control in a container save its current Anchor value and set it to 0, and Container.SetAll('lSaveAnchor', .F.) to restore the saved values.

I use lSaveAnchor in lots of places to avoid resizing problems:

- To resolve the problem with ResizeDemo1.SCX, add This.SetAll('lSaveAnchor', .T.) before the moving code in Init and This.SetAll('lSaveAnchor', .F.) after. These two statements are actually there but commented out; uncomment them and run the form again, and this time it will appear as it should (**Figure 3**).

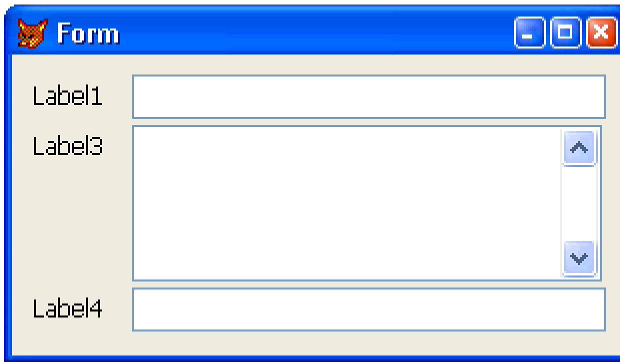

*Figure 3. Turning off anchoring before moving the controls results in the proper placement.*

- I have a class (SFWizardForm) used as the base class for wizard-based dialogs. SFWizardForm has a page frame sized to fill the form and with Tabs set to .F., giving the appearance of different steps in a wizard without showing the tabs for each page. At design time, the form is actually sized slightly wider than the page frame so there's some space I can double-click to bring up a code window for the form rather than the page frame. At run time, the Show method sets the Width of the form to the Width of the page frame. The problem, of course, is that this causes the controls in each page of the page frame to resize, which isn't needed. So, Show uses the following code to resize the form:

```
with This
  .SetAll('lSaveAnchor', .T.)
  .MinWidth = min(.MinWidth, .pgfWizard.Width)
  .Width    = .pgfWizard.Width
  .SetAll('lSaveAnchor', .F.)
endwith
```

- When controls are dynamically instantiated, as opposed to added to the form in the Form or Class Designers, they aren't placed properly if their Anchor values are non-zero. It's even worse if you resize the form prior to instantiating the controls. If this sounds like a contrived scenario, consider the following: in a wizard-based dialog, the choices the user makes in step 1 determine what controls they see in step 2. So, the controls aren't instantiated in step 2 until that page of the page frame is activated. To support this, my SFPage class (the base class for pages in page frames, defined in SFCtrls.VCX) includes the following code in its Activate method (other code omitted here):

```
with This
  if not empty(.cMemberClass) and ;
    type('.oMember.Name') <> 'C'
    if '\' + upper(.cMemberLibrary) $ set('CLASSLIB')
      .AddObject('oMember', .cMemberClass)
    else
      .NewObject('oMember', .cMemberClass, ;
        .cMemberLibrary)
    endif '\' ...
*   llSave = pemstatus(.oMember, 'lSaveAnchor', 5)
    with .oMember
     if llSave
       .lSaveAnchor = .T.
     endif llSave
      .Top      = 10
      .Left     = 10
      .Width    = .Width  + This.Parent.Width  - ;
        Thisform.nInitialWidth
      .Height   = .Height + This.Parent.Height - ;
        Thisform.nInitialHeight
      .Visible = .T.
```

```
      if llSave
        .lSaveAnchor = .F.
      endif llSave      .ZOrder(1)
    endwith
  endif not empty(.cMemberClass) ...
endwith
```

In a particular wizard form, I override this method to set cMemberClass and cMemberLibrary based on the choices in page 1, then DODEFAULT() to dynamically instantiate the desired controls. However, if you run ResizeDemo2.SCX, resize the form, and then select page 2, you'll notice that the controls aren't sized or positioned properly on the page. To fix this, uncomment the commented line in SFPage.Activate and run the form again. This time, when you resize the form and select page 2, the controls will be sized as if they were already members of the form when you resized it. Notice part of that is due to knowing the original size of the form: the nInitialWidth and nInitialHeight properties. My base class form, SFForm, initializes these two custom properties in its Load method:

```
with This
  .nInitialWidth  = .Width
  .nInitialHeight = .Height
endwith
```

- I like to save the size and position of a form when the user closes it and restore those settings the next time the user runs it. I discussed a helper class, SFPersist, and various subclasses in my January 2000 article, "Persistence without Perspiration." Normally, I just drop an SFPersist subclass on a form (the specific class I used depends on where I want the settings saved; there are classes that save to tables, INI files, and the Windows Registry), set some properties, and I'm done. The Init method of SFPersist takes care of restoring saved settings and Destroy saves the settings, so it works automatically. However, there's a problem with automatically restoring the size of a form when its controls have non-zero Anchor values; any controls instantiated after the SFPersist object (because they're lower in ZOrder) won't resize properly because the form was resized before they were instantiated. To resolve this, SFPersist has an lRestoreOnInit property; Init only calls the Restore method if that property is .T. So, after I drop an SFPersist object on a form, I set its lRestoreOnInit property to .F. and then call its Restore method from the form's Init method. That ensures all controls are completely instantiated prior to resizing the form to its former size.

### Controlling a report's preview window

In earlier versions of VFP, controlling the appearance of a report preview window isn't easy: you have to create a dummy window with the characteristics you want, then specify that window in the WINDOW clause of the REPORT command. Even with all this ugly code, you still have very little control over how the preview window looks.

```
loForm = createobject('SFPreviewForm')
with loForm
  .Caption = 'My Preview Caption'
  .Width   = _screen.Width
  .Height  = _screen.Height
endwith
keyboard '{CTRL+F10}'
if wexist('Print Preview') and ;
  not wvisible('Print Preview')
  show window 'Print Preview'
endif wexist('Print Preview') ...
report form MyReport to printer prompt preview ;
  nodialog noconsole window (loForm.Name)
if wvisible('Print Preview')
  hide window 'Print Preview'
endif wvisible('Print Preview')
```

```
define class SFPreviewForm as Form
  ScrollBars = 3
  DoCreate = .T.
  Caption = ""
  HalfHeightCaption = .T.
  MinButton = .F.
  BackColor = rgb(255,255,255)
  Name = "SFPreviewForm"
enddefine
```

Controlling the preview window in VFP 9 is much easier because it's a VFP form, so it has the usual properties like Top, Left, Width, Height, Caption, and so forth. In addition, it has some custom properties that control its appearance and behavior.

The key to controlling the preview window is to instantiate your own copy rather than letting VFP create a preview window. This gives you a chance to set some properties of the window before running the report. Note that you don't actually instantiate a preview form, but rather a proxy object that controls the form. The way you do that is by calling ReportPreview.APP, passing it a variable that will contain a reference to the proxy object. Once you have the proxy object, you can set whatever properties you want. To use the customized proxy object and form, you must instantiate a report listener object, either a base class ReportListener or whichever subclass you wish, and put a reference to the proxy object into its PreviewContainer property. When you run the report using that report listener, the report engine will see that PreviewContainer already contains a preview object and will use it rather than instantiating its own.

Here's an example, taken from TestPreview.PRG. Note that this code sets the Caption, size, and position of the preview window to desired values.

```
local loPreview, ;
  loListener

* Create the preview proxy object and set the desired
* properties.

do (_reportpreview) with loPreview
with loPreview
  .Caption = 'This is my report'
  .Top     = 10
  .Left    = 10
  .Height  = 500
  .Width   = 500
endwith

* Create a listener object and tell it to use our
* preview object.

loListener = createobject('ReportListener')
loListener.ListenerType     = 1
loListener.PreviewContainer = loPreview

* Run the report using the listener.

use _samples + 'Northwind\Orders'
report form TestPreview.FRX preview object loListener
```

### *Controlling other settings*

The "Leveraging the Default Preview Container" topic in the VFP help file discusses other properties of the preview proxy object you can control, including:

- ToolbarIsVisible: set this to .F. to hide the toolbar.
- CanvasCount: determines the number of pages visible in the preview window. For example, set it to 4 to display four pages at once.
- ZoomLevel: specifies the zoom level for the preview. These are enumerated values from 1 (for 10%) to 10 (for whole page); see the VFP help topic for the list of values.

- CurrentPage: the initial page to display in the window.

### *Disabling the Print button in the toolbar*

This is a frequently asked question on the Universal Thread (www.universalthread.com) and the solution is simple: set the undocumented AllowPrintFromPreview property to .F. See TestPreview.PRG for an example.

### *Docking the toolbar*

Rather than having it float, you may want the toolbar for the preview window to be docked automatically. This is a little trickier to do than changing other settings because the proxy object doesn't have a property for this, and since the preview form and its toolbar aren't instantiated until the last moment before the report is displayed (in the Show method of the proxy object), we can't talk directly to the form or toolbar when the other proxy properties are set.

Looking at the code in the Show method of the proxy object for ideas, I noticed that if the oForm property already contains a reference to a preview form, the proxy object doesn't instantiate another one. I also discovered that the toolbar is instantiated in the Init method of the preview form class. So, similar to how instantiating our own proxy object rather than letting VFP do it gives us more control, if we instantiate our own preview form, that should give us the ability to control the toolbar before the report is run.

I added the following code to TestPreview.PRG inside the WITH loPreview block:

```
  .oForm = newobject(.PreviewFormClass, .ClassLibrary)
  .oForm.oToolbar.Dock(0)
```

The PreviewFormClass property contains the name of the class to instantiate for the preview form, and since the same VCX contains both the proxy class and the preview form class, this ought to do what we need. Unfortunately, there are two problems with this:

- PreviewFormClass is protected, so the first statement gives a "Property PreviewFormClass is not found" error.
- Even if PreviewFormClass was public, its value is set dynamically in the proxy object's Show method since the exact class to use depends on various factors, such as whether a top-level form is needed. As I discussed earlier, Show is too late for us.

Fortunately, there's another way to do this: via the preview extension handler. Microsoft figured we'd want to do more with preview windows than comes in the box, so they opened up the architecture by providing for a hook object. If this hook object, called a preview extension handler, exists, the preview form calls the appropriate method of the object at various places: when the preview window is displayed or closed, when the user presses a key, when the shortcut menu is displayed, and when the current report page is drawn on the preview window.

The same "Leveraging the Default Preview Container" topic in the VFP help file I mentioned earlier discusses the interface of the extension handler object. Here's the definition of one that automatically docks the toolbar when the preview window is displayed. Notice that the empty methods must be defined because if the extension handler object is used, all of these methods are called at some point.

```
define class SFPreviewExtensionHandler as Custom
  PreviewForm = .NULL.

* Dock the toolbar if there is one.

  procedure Show(tnStyle)
    if vartype(This.PreviewForm) = 'O'
      This.PreviewForm.Toolbar.Dock(0)
    endif vartype(This.PreviewForm) = 'O'
  endproc

* Clean up when we're released.

  procedure Release
```

```
    if type('This.PreviewForm') = 'O'
      This.PreviewForm.ExtensionHandler = .NULL.
      This.PreviewForm = .NULL.
    endif type('This.PreviewForm') = 'O'
  endproc

  procedure AddBarsToMenu(tcShortcutMenuName, ;
    tnBarCount)
  endproc

  procedure HandledKeyPress(tnKeyCode, ;
    tnShiftAltCtrl)
  endproc

  procedure Paint
  endproc
enddefine
```

Tell the proxy object to use this preview extension handler with its SetExtensionHandler method. TestPreview.PRG uses the following code within the WITH loPreview block:

```
loHandler = createobject('SFPreviewExtensionHandler')
.SetExtensionHandler(loHandler)
```

## Summary

Anchoring is a great addition to VFP 9, but unless you're aware of some of its idiosyncrasies, you can pull your hair out fighting it to do what you want. Similarly, rather than fighting with the report preview window, take control of it by instantiating your own copy and making it look and act like you want it to.

*Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, and the MemberData Editor, Anchor Editor, New Property/Method Dialog, and CursorAdapter and DataEnvironment builders that come with VFP. He is co-author of the "What's New in Visual FoxPro" series and "The Hacker's Guide to Visual FoxPro 7.0," all from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a long-time Microsoft Most Valuable Professional (MVP), having first been honored with this award in 1996. Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com*