I Got Rendered Where?

Doug Hennig

Doug Hennig continues his discussion of report listeners by presenting a set of classes that output the contents of a report run to a cursor and use that cursor to provide a "live" report preview surface.

When you run a report in VFP 8 or earlier, the output is sort of a black box: you have little control over the preview window, don't have any information about what got rendered where, and can't provide a "live" preview surface (one in which click events can be trapped to perform some object-specific action) to the user.

Over the past two months, I've discussed the new ReportListener class in VFP 9 and how it can be used to control report output in ways that previously weren't possible. The ReportListener subclasses I've shown so far performed some type of visible output, such as HTML, reports with dynamic formatting, and so forth. This month, the output from a listener won't really go anywhere obvious to the user; instead, it's sent to a cursor so we can track what got rendered where. Having this information provides all kinds of interesting uses, such as a live preview surface, a dynamically generated table of contents, the ability to find text, conditionally highlighting certain report objects, and so on.

DBFListener

DBFListener, contained in DBFListener.PRG, is a subclass of _ReportListener, a report listener subclass defined in _ReportListener.VCX in the FFC subdirectory of the VFP home directory. I discussed _ReportListener in my February 2005 FoxTalk column, "Listening to a Report." Because it's a subclass of _ReportListener, DBFListener can either be used as the sole listener for a report or as one of a chain of listeners, each of which performs some function during the report run. The ListenerType property of DBFListener is set to 3 to suppress output to a preview window or printer.

Just before the report is run, the code in the BeforeReport event creates a cursor or table to hold the rendered report contents. To create a table, set the lUseCursor property to .F. and cOutputDBF to the name and path of the table to create (if you don't specify the name, a SYS(2015) name is used in the Windows temp directory). To create a cursor, set lUseCursor to .T. and cOutputAlias to the alias to use for the cursor (if you don't specify the alias, a SYS(2015) name is used). In either case, the table or cursor has columns for the record number in the FRX for the report object, the OBJTYPE and OBJCODE values from the FRX (which indicate what type of object it is), the left, top, width, and height of the rendered object, the "continuation type" parameter passed to the Render method (see the VFP help topic for Render for a discussion of this parameter), the contents of the object if it's a field or label, and the page it appears on.

```
function BeforeReport
  local lcTable
  with This

* If a table name and/or an alias wasn't specified,
* create default names.

    if empty(.cOutputDBF)
      .cOutputDBF = addbs(sys(2023)) + sys(2015) + ;
        '.dbf'
    endif empty(.cOutputDBF)
    if empty(.cOutputAlias)
      .cOutputAlias = ;
        strtran(juststem(.cOutputDBF), ' ', '_')
    endif empty(.cOutputAlias)

* If the cursor is already open it, close it. If the
* table already exists, nuke it.

    use in select (.cOutputAlias)
    if file(.cOutputDBF)
      erase (.cOutputDBF)
      erase forceext(.cOutputDBF, 'FPT')
```

```
      erase forceext(.cOutputDBF, 'CDX')
    endif file(.cOutputDBF)

* Create either a cursor or a table.

    lcTable = iif(.lUseCursor, 'cursor ' + ;
      .cOutputAlias, 'table ' + .cOutputDBF)
    create &lcTable (FRXRECNO I, OBJTYPE I, ;
      OBJCODE I, LEFT I, TOP I, WIDTH I, HEIGHT I, ;
      CONTTYPE I, CONTENTS M nocptrans, PAGE I)
    index on PAGE tag PAGE
  endwith

* Do the usual behavior.

  dodefault()
endfunc
```

As each object in the report is rendered, the Render event fires. The code in this event in DBFListener adds a record to the cursor. Since the text of a field or label is passed in Unicode, it has to be converted back to normal text using STRCONV() to make it useful. Render uses the helper methods SetFRXDataSession and ResetDataSession defined in its parent class to switch to the data session the FRX cursor is in and back again; this allows Render to get the OBJTYPE and OBJCODE values from the FRX for the current object. Note this code doesn't currently do anything special with images because I haven't decided what to do with them yet.

```
function Render(tnFRXRecNo, tnLeft, tnTop, tnWidth, ;
  tnHeight, tnObjectContinuationType, ;
  tcContentsToBeRendered, tiGDIPlusImage)
  local lcContents, ;
    liObjType, ;
    liObjCode
  with This
    if empty(tcContentsToBeRendered)
      lcContents = ''
    else
      lcContents = strconv(tcContentsToBeRendered, 6)
    endif empty(tcContentsToBeRendered)
    .SetFRXDataSession()
    go tnFRXRecno in FRX
    liObjType = FRX.OBJTYPE
    liObjCode = FRX.OBJCODE
    .ResetDataSession()
    insert into (.cOutputAlias) ;
      values (tnFRXRecNo, liObjType, liObjCode, ;
      tnLeft, tnTop, tnWidth, tnHeight, ;
      tnObjectContinuationType, lcContents, ;
      .PageNo)
  endwith
endfunc
```

The Destroy method (not shown here) closes the cursor or table and deletes the table if the lDeleteOnDestroy property is .T.

When DBFListener is used as the listener for a report, nothing appears to happen; the report isn't previewed, printed, output to HTML, or anything else. However, after the report run, a cursor or table containing information about what got rendered where is available.

SFPreview

Once you have the rendered content of a report in a cursor or table, you can use it for lots of things. I'll show you a couple of uses for it in this article.

SFPreviewForm is a form class providing a report preview dialog with different capabilities than the preview window that comes with VFP. It raises events when report objects are clicked and supports other capabilities such as finding text. It has to use some trickery to do this: since a preview page is a GDI+ image, nothing specific happens when you click on some text in the image. SFPreviewForm supports report

object events by creating a shape object on the preview surface for every rendered object. These shapes can, of course, capture events such as mouse movement or clicks, making it possible to have a live preview surface. The shapes aren't added to the form itself, but to a container that sits on the form. Since a different set of shapes must be created for each page, it's easier to delete the container, which deletes all of the shapes at once, and create a new one than to remove each individual shape prior to adding new ones.

The main method in SFPreviewForm is DisplayPage. This method displays the current page of the report and creates shape objects in the same size and position as each report object in the page. How does DisplayPage know what report objects appear on the page? By looking in the cursor created by DBFListener, of course.

```
lparameters tnPageNo
local lnPageNo, ;
  lcObject, ;
  loObject
with This

* If we haven't been initialized yet, do so now.

  if vartype(.oListener) = 'O'
    if not .lInitialized
      .InitializePreview()
    endif not .lInitialized

* Ensure we have a shape container with no shapes.

    .AddShapeContainer()

* Ensure a proper page number was specified.

    if between(tnPageNo, .nFirstPage, .nLastPage)
      lnPageNo = tnPageNo
    else
      lnPageNo = .nFirstPage
    endif between(tnPageNo, .nFirstPage, .nLastPage)

* Select the output cursor and create a shape around
* each report object on the specified page.

    select (.cOutputAlias)
    seek lnPageNo
    scan while PAGE = lnPageNo
      .AddObjectToContainer()
    endscan while PAGE = lnPageNo

* Set the current page number and draw the page.

    .nCurrentPage = lnPageNo
    .DrawPage()

* Flag whether we're on the first or last page.

    .lFirstPage = lnPageNo  = .nFirstPage
    .lLastPage  = lnPageNo >= .nLastPage

* Refresh the toolbar if necessary.

    .RefreshToolbar()

* If we don't have a listener object, we can't
* proceed.

  else
    messagebox('There is no listener object.', 16, ;
      .Caption)
  endif vartype(.oListener) = 'O'
endwith
```

This code starts by calling InitializePreview if the preview hasn't been initialized yet, and then calling AddShapeContainer to add the container used to hold the shapes to the form. We won't look at AddShapeContainer here; it simply removes any existing container and adds a new one from the class whose class name and library are specified in the cContainerClass and cContainerLibrary properties. DisplayPage then ensures that a valid page number was specified and spins through the rendered output cursor, adding a shape for each object on the current page to the container. It then sets the nCurrentPage property to the page number and calls DrawPage to display the preview image for the current page on the form. DisplayPage updates lFirstPage and lLastPage so the buttons in a toolbar can be properly enabled or disabled (for example, the Last Page button is disabled if lLastPage is .T.), and then refreshes the toolbar.

InitializePreview, which is called from DisplayPage the first time that method is called, ensures that certain properties are initialized properly. As the comments in this method indicate, one complication is that if you use a RANGE clause for a report run, such as RANGE 6, 7, the pages may be numbered 6 and 7 but when you call the listener's OutputPage method to draw the preview image on the form, the first page is 1, the second page is 2, and so forth. To overcome the potential mismatch between these numbering schemes, InitializePreview sets the nFirstPage and nLastPage properties to the first and last page numbers (6 and 7 in this example) and nPageOffset as the value to subtract from a "real" page number to get the output page number.

InitializePreview also puts the report page height and width into the nMaxWidth and nMaxHeight properties. These values are used to size the container used for the report preview; if they're larger than the form size, scrollbars will appear because the form's ScrollBars property is set to 3-Both. Note a couple of complications here. First, the page height and width values are in 960^{ths} of an inch while the form uses pixels. Fortunately, it's easy to convert from 960^{ths} of an inch to pixels: divide the value by 10, since the report engine renders at 96 DPI. The second complication is that if the DBFListener object isn't the lead listener for a report run, its GetPageWidth and GetPageHeight methods don't return valid values. Fortunately, _ReportListener handles this by setting the custom SharedPageWidth and SharedPageHeight properties to the appropriate values.

Finally, InitializePreview clears some properties used for finding text (we'll look at those later), opens the class library used for the shapes that'll be added to the form for the report objects, and flags that initialization has been done so this method isn't called a second time.

```
with This

* Set the starting and first page offset. Even though
* we may not have output the first page due a RANGE
* clause, the pages are numbered starting with 1 from
* an OutputPage point-of-view.

  .nFirstPage  = .oListener.CommandClauses.RangeFrom
  .nPageOffset = .nFirstPage - 1

* The Width and Height values are 1/10th of the
* values from the report because those values are in
* 960ths of an inch and the report engine uses a
* resolution of 96 DPI. Our listener may be a
* successor, so use the appropriate Shared properties
* if they exist. Also, get the last page number using
* either SharedOutputPageCount (which may not have
* been filled in if the listener is the lead listener
* and has no successor) or OutputPageCount, adjusted
* for the offset.

  if pemstatus(.oListener, 'SharedPageWidth', 5)
    .nMaxWidth  = .oListener.SharedPageWidth/10
    .nMaxHeight = .oListener.SharedPageHeight/10
    if .oListener.SharedOutputPageCount > 0
      .nLastPage = ;
        .oListener.SharedOutputPageCount + ;
        .nPageOffset
    else
      .nLastPage = .oListener.OutputPageCount + ;
        .nPageOffset
    endif .oListener.SharedOutputPageCount > 0
```

```
  else
    .nMaxWidth  = .oListener.GetPageWidth()/10
    .nMaxHeight = .oListener.GetPageHeight()/10
    .nLastPage  = .oListener.OutputPageCount + ;
      .nPageOffset
  endif pemstatus(.oListener, 'SharedPageWidth', 5)

* Clear the find settings.

  .ClearFind()

* Open the appropriate class library if necessary.

  if not '\' + upper(.cShapeLibrary) $ ;
    set('CLASSLIB')
    .lOpenedLibrary = .T.
    set classlib to (.cShapeLibrary) additive
  endif not '\' ...

* Flag that we've been initialized.

  .lInitialized = .T.
endwith
```

DrawPage, called from DisplayPage to draw the current preview page image on the form, calls the OutputPage method of the listener, passing it the page number (adjusted for the starting offset), the container used as the placeholder for the image, and the value 2, which indicates the output should go to a VFP control. DrawPage also calls HighlightObjects to highlight any report objects we want highlighted; I'll discuss this later. Note that the Paint event of the form also calls DrawPage because when the form is redrawn (such as during a resize), the placeholder container is redrawn and therefore the preview image is lost, so DrawPage restores it.

```
with This
  if vartype(.oListener) = 'O'
    .oListener.OutputPage(.nCurrentPage - ;
      .nPageOffset, .oContainer, 2)
    .HighlightObjects()
  else
    messagebox('There is no listener object.', 16, ;
      .Caption)
  endif vartype(.oListener) = 'O'
endwith
```

AddObjectToContainer, called from DisplayPage, adds a shape of the class specified in cShapeClass (the class library specified in cShapeLibrary was previously opened in InitializePreview) to the shape container for the current report object. The shape is sized and positioned based on the HEIGHT, WIDTH, TOP, and LEFT columns in the cursor, although, as we saw earlier, these values must be divided by 10 to convert them to pixels.

```
local lcObject, ;
  loObject
with This
  lcObject = 'Object' + transform(recno())
  .oContainer.AddObject(lcObject, .cShapeClass)
  loObject = evaluate('.oContainer.' + lcObject)
  with loObject
    .Width   = WIDTH/10
    .Height  = HEIGHT/10
    .Top     = TOP/10
    .Left    = LEFT/10
    .nRecno  = recno()
    .Visible = .T.
  endwith
endwith
return loObject
```

Handling events

The cShapeClass property is set to "SFReportShape" by default. SFReportShape is a subclass of Shape with code in its Click, RightClick, and DblClick events that call the OnObjectClicked method of the form, passing it the record number in the report contents cursor this shape represents and a numeric value indicating which event occurred (1 for Click, 2 for DblClick, and 3 for RightClick). This allows SFPreviewForm to receive notification whenever a report object is clicked.

OnObjectClicked handles a click on the shape representing a report object by raising the appropriate event for the click type. The benefit of using RAISEEVENT() is that any object can use BINDEVENT() to ObjectClicked, ObjectDblClicked, or ObjectRightClicked to implement the desired behavior without having to subclass SFPreviewForm. You could even have multiple behaviors if you wish, since multiple objects can bind to the same event. Any object that binds to these events will receive as a parameter a SCATTER NAME object for the current record in the report contents cursor.

```
lparameters tnRecno, ;
  tnClickType
local loObject
select (This.cOutputAlias)
go tnRecno
scatter memo name loObject
do case
  case tnClickType = 1
    raiseevent(This, 'ObjectClicked',      loObject)
  case tnClickType = 2
    raiseevent(This, 'ObjectDblClicked',   loObject)
  otherwise
    raiseevent(This, 'ObjectRightClicked', loObject)
endcase
```

There are several other methods in SFPreviewForm. Show instantiates a toolbar using the class and library names specified in the cToolbarClass and cToolbarLibrary properties if lShowToolbar is .T. The FirstPage, PreviousPage, NextPage, and LastPage methods call DisplayPage, passing the appropriate value to display the desired page. SaveFormPosition and SetFormPosition save and restore the size and shape of the preview form between report runs. I'll discuss the rest of the methods next month.

Live preview surface

Let's try it out. TestSFPreview.PRG uses DBFListener as the listener for the Customers report, then instantiates SFPreviewForm, sets its properties to the appropriate values, and tells it to display the first page.

```
loListener = newobject('DBFListener', ;
  'DBFListener.PRG')
report form Customers object loListener

* Show the report in our custom previewer.

loForm = newobject('SFPreviewForm', 'SFPreview.vcx')
with loForm
  .cOutputAlias = loListener.cOutputAlias
  .Caption      = 'Customer Report'
  .oListener    = loListener
  .FirstPage()
endwith
```

TestSFPreview.PRG then instantiates an object to handle clicks in the preview surface and binds the various click events to it. Finally, it displays the Debug Output window (because that's where click events will be echoed by the click handler class) and shows the preview form.

```
loHandler = createobject('ClickHandler')
bindevent(loForm, 'ObjectClicked',      loHandler, ;
  'OnClick')
bindevent(loForm, 'ObjectDblClicked',   loHandler, ;
  'OnDblClick')
```

```
bindevent(loForm, 'ObjectRightClicked', loHandler, ;
  'OnRightClick')

* Display the debug output window and the preview
* form.

activate window 'debug output'
loForm.Show(1)
```

Here's part of the definition of the click handler class (the OnDblClick and OnRightClick methods aren't shown because they're nearly identical to OnClick). The ObjType property of the passed object indicates what type of report object was clicked (for example, 5 means a label and 8 means a field) and Contents contains the contents in the case of a label or field.

```
define class ClickHandler as Custom
  procedure OnClick(toObject)
    do case
      case inlist(toObject.ObjType, 5, 8)
        debugout 'You clicked ' + ;
          trim(toObject.Contents)
      case toObject.ObjType = 7
        debugout 'You clicked a rectangle'
      case toObject.ObjType = 6
        debugout 'You clicked a line'
      case toObject.ObjType = 17
        debugout 'You clicked an image'
    endcase
  endproc
enddefine
```

When you run TestSFPreview.PRG, you'll see the preview form shown in **Figure 1**. Although this looks similar to the preview form that comes with VFP, try clicking on various report objects. You'll see information about the object echoed to the Debug Output window (I decided to send output there rather than WAIT WINDOW because the latter interfered with the DblClick event). This simple example doesn't do much, but imagine the possibilities: jumping to another section of the report or a different report altogether, launching a VFP form, providing a shortcut menu that displays different options depending on the particular report object that was right-clicked, supporting bookmarks, and so on.

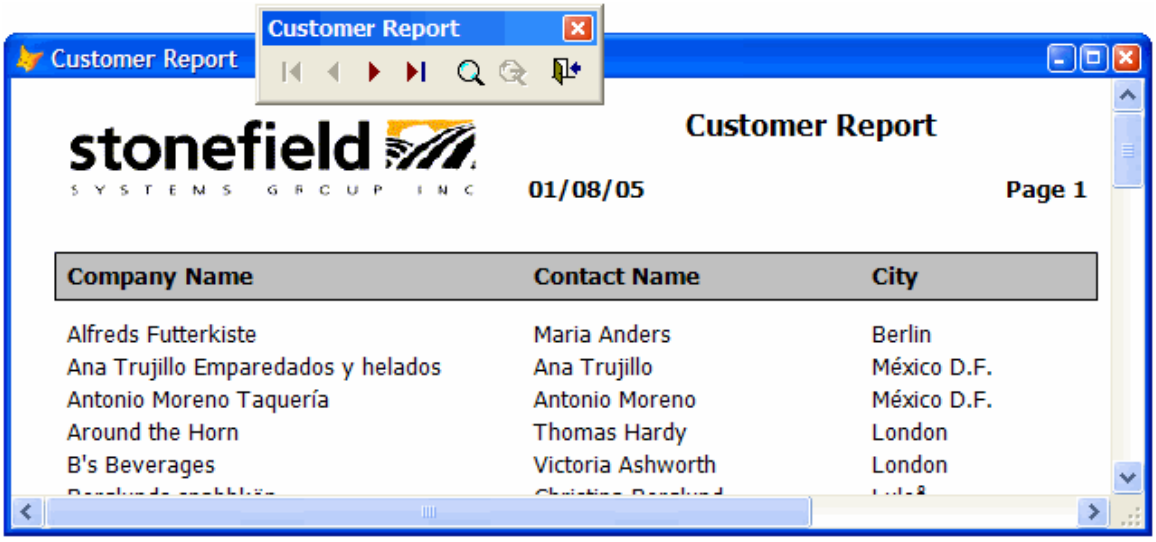

Figure 1. SFPreviewForm, combined with DBFListener, provides a live preview surface.

You may have noticed that the toolbar contains a couple of interesting looking buttons. These are used for finding text within the report. I'll discuss that topic next month.

Summary

By outputting the contents of a report run to a table or cursor, DBFListener provides us with information about where each object is rendered on a report, which can be used in a lot of ways. I hope you're starting to see the incredible possibilities the VFP 9 ReportListener class provides us.

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