Cool Uses for ReportListeners

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Web site: http://www.stonefield.com
Web site: http://www.stonefieldquery.com
Blog: http://doughennig.blogspot.com

Overview

You’ve probably seen the demos showing the new ReportListener in VFP 9: dynamically formatted text, rotated labels, and so forth. But did you know report listeners can be used for a lot more than that? This practical session will show you all kinds of cool uses for report listeners, including hyperlinking text in reports, providing a “live” preview surface that can handle click events, finding and highlighting text, and more.

Introduction

Before VFP 9, the reporting engine was monolithic: it handled everything–data handling, object positioning, rendering, previewing, and printing. The new reporting engine in VFP 9 splits responsibility for reporting between the reporting engine, which now just deals with data handling and object positioning, and a new VFP base class, ReportListener, which handles rendering and output. In addition, a new Xbase application, ReportPreview.APP, provides a VFP Form-based preview window with more capabilities than the window used in earlier versions.

Some of the ReportListener subclasses that come with VFP provide things like HTML and XML output. However, since ReportListener receives events through every step of report output, we can make subclasses do a lot more than just provide output to different file types. In this document, I'll explore some of the cool uses I've found for ReportListener.

Note: this document assumes at least some knowledge of ReportListener. For more information on this class, see the VFP documentation, my "What's New in Nine" book from Hentzenwerke Publishing (http://www.hentzenwerke.com), or Cathy Pountney's excellent white paper, "Visual FoxPro 9.0 Report Writer In Action," available from the VFP home page (http://msdn.microsoft.com/vfoxpro/).

Hyperlinking Reports

Wouldn't it be cool if you could tell VFP to add a hyperlink to a field in a report? Then the user could click on the hyperlink to navigate to some related information. For example, a report showing customers and their Web sites or email addresses would have live links; clicking on a Web site link would navigate their browser to that URL. Even more interesting would be the ability to navigate to something else within your application. For example, clicking on a company name in a report could bring up the customers data entry form with that company as the selected record.

Because the report preview window that ships with VFP doesn't support live, clickable objects in a report (although we'll see how to do something about that later in this document), the easiest way to implement this is using HTML, which natively supports hyperlinks.

VFP comes with a report listener that outputs HTML (the HTMLListener class built into ReportOutput.APP and in _ReportListener.VCX in the FFC folder), but I was sure it would require a lot of work to get it to support hyperlinks. However, I was pleasantly surprised to discover how little effort it required.

First, a little background. HTMLListener is a subclass of XMLDisplayListener, which is a subclass of XMLListener, which is a subclass of _ReportListener, the class I typically use as the parent class for my own listeners. When you use HTMLListener, either directly by instantiating it and using it as the listener for a report or by specifying OBJECT TYPE 5 in the REPORT command, it actually generates XML for the report (this is performed by its parent classes), then applies an XSL transform to the XML to generate the HTML. The XSLT to use is defined in the GetDefaultUserXSLTAsString method.

The default XSLT used by HTMLListener is very complex, and not being much of an XSL expert, I thought it might be an overwhelming task to figure out what to change to add support for hyperlinks. However, as I started poking through GetDefaultUserXSLTAsString, I discovered the following:

```
<xsl:when test="string-length(@href) &gt; 0">
  <A href="{@href}">
  <xsl:call-template name="replaceText"/>
  </A>
</xsl:when>
```

This XSL adds an anchor tag to the HTML if there's an HREF attribute on the current element in the XML. This is cool—it means HTMLListener already supports hyperlinks! However, searching for "HREF" turned up no hits in XMLDisplayListener or XMLListener, so how to add that attribute to an element, especially dynamically?

After doing some more poking around, I found that the attributes of a particular element were set in the GetRawFormattingInfo method of XMLListener. So, I subclassed HTMLListener and added the behavior I want to this method.

The following code, taken from HyperlinkListener.PRG, provides a listener that generates a hyperlink on an object in a report if that object's User memo contains the directive "*:URL =" followed by the expression to use as the URL.

```
define class HyperlinkListener as HTMLListener ;
  of home() + 'ffc\_ReportListener.vcx'
  QuietMode = .T.
```

```
    && default QuietMode to suppress feedback
  dimension aRecords[1]
    && an array of information for each record in FRX

* Before we run the report, go through the FRX and
* store information about any field with our expected
* directive in its USER memo into the aRecords array.

  function BeforeReport
    dodefault()
    with This
      .SetFRXDataSession()
      dimension .aRecords[reccount()]
      scan for atc('*:URL', USER) > 0
        .aRecords[recno()] = ;
          alltrim(strextract(USER, '*:URL =', ;
          chr(13), 1, 3))
      endscan for atc('*:URL', USER) > 0
      .ResetDataSession()
    endwith
  endfunc

* If the current field has a directive, add the URL
* to the attributes for the node.

  function GetRawFormattingInfo(tnLeft, tnTop, ;
    tnWidth, tnHeight, tnObjectContinuationType)
    local lcInfo, ;
      lnURL
    with This
      lcInfo = dodefault(tnLeft, tnTop, tnWidth, ;
        tnHeight, tnObjectContinuationType)
      lcURL  = .aRecords[recno('FRX')]
      if not empty(lcURL)
        .SetCurrentDataSession()
        lcInfo = lcInfo + ' href="' + ;
          textmerge(lcURL) + '"'
        .ResetDataSession()
      endif not empty(lcURL)
    endwith
    return lcInfo
  endfunc
enddefine
```

The BeforeReport event fires just before the report runs. It uses the SetFRXDataSession method to select the data session the FRX cursor is in, and then scans through the FRX and puts the URL expression for any object that has the directive into an array. It calls ResetDataSession at the end to restore the data session the listener is in.

The GetRawFormattingInfo method uses DODEFAULT() to perform the usual behavior, which generates the attributes for an XML element as a string. It then checks the appropriate array element (the data session for the FRX cursor was selected by code in XMLListener before this code executes) to see whether the current object in the report has the directive, and if so, adds an HREF attribute to the XML element. It calls SetCurrentDataSession to select the data session used by the report's data and uses TEXTMERGE() on the URL expression because the expression will likely contain something specific for each record, such as <<CustomerID>>.

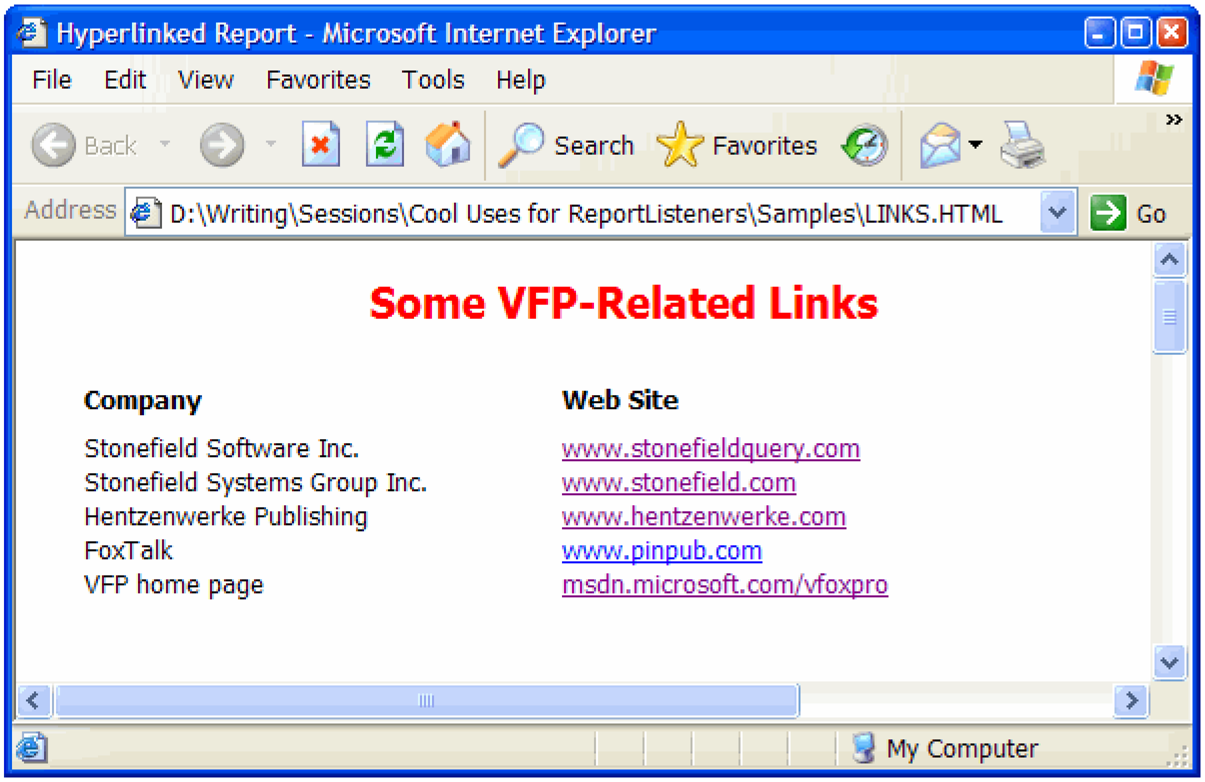
That's it! Let's look at some examples of how we can use this listener.

Example 1: Live links to URLs

Links.FRX is a simple example that shows how this listener works. It reports on the Links table, which has a list of company names and their Web sites. The website field in the report has "*:URL = http://<<trim(website)>>" in its User memo. Links.PRG runs this report, using HyperlinkListener as the report listener, and uses the _ShellExecute class in the FFC to display the HTML file in your default browser. **Figure 1** shows the results.

```
loListener = newobject('HyperlinkListener', ;
  'HyperlinkListener.prg')
loListener.TargetFileName = fullpath('Links.html')
loListener.PrintJobName   = 'Hyperlinked Report'
```

```
report form Links object loListener
loShell = newobject('_ShellExecute', ;
  home() + 'ffc\_Environ.vcx')
loShell.ShellExecute(loListener.TargetFileName)
```


Hyperlinked Report - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address D:\Writing\Sessions\Cool Uses for ReportListeners\Samples\LINKS.HTML
Go
My Computer

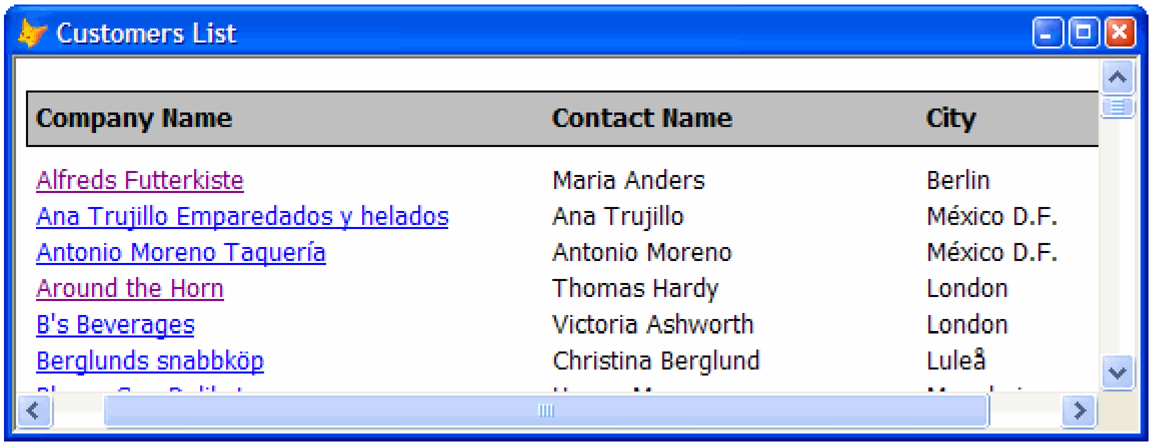
Some VFP-Related Links

Company	Web Site
Stonefield Software Inc.	www.stonefieldquery.com
Stonefield Systems Group Inc.	www.stonefield.com
Hentzenwerke Publishing	www.hentzenwerke.com
FoxTalk	www.pinpub.com
VFP home page	msdn.microsoft.com/vfoxpro

Figure 1. The HTML generated from the Links report has live hyperlinks.

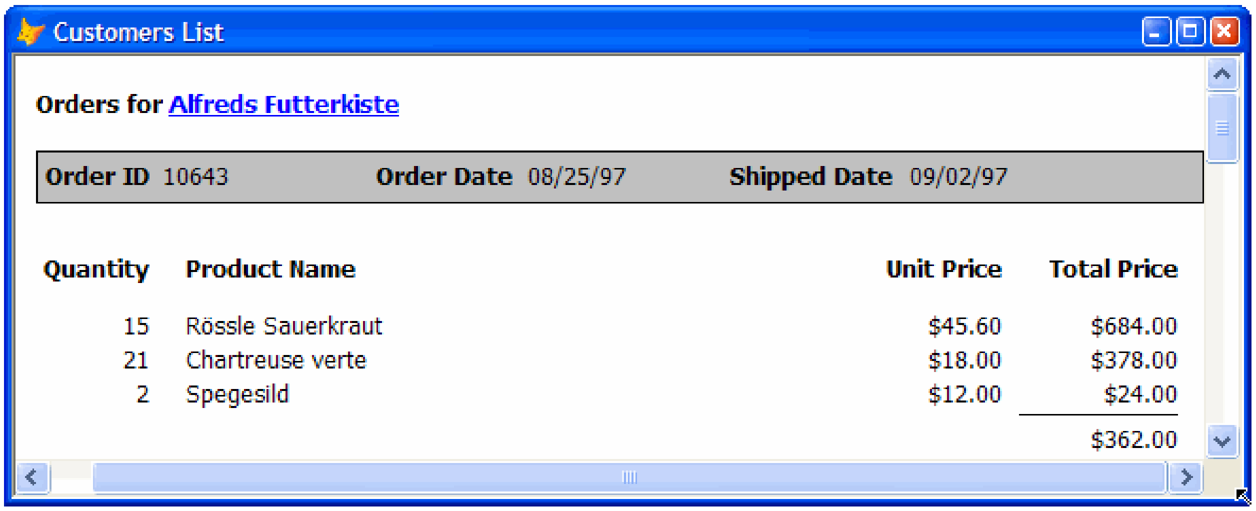
Example 2: Drilldown reports

HyperlinkReports.SCX is a more complex example. As you can see in **Figure 2**, it presents a list of customer information. However, this HTML is displayed in a Web Browser ActiveX control imbedded in a VFP form rather than a browser window. Clicking on a company name runs a report of orders for that customer and displays it in the form, as shown in **Figure 3**. The orders report also has a hyperlink that returns the display to the customer list. So, this form provides drilldown reports.



Company Name	Contact Name	City
Alfreds Futterkiste	Maria Anders	Berlin
Ana Trujillo Emparedados y helados	Ana Trujillo	México D.F.
Antonio Moreno Taquería	Antonio Moreno	México D.F.
Around the Horn	Thomas Hardy	London
B's Beverages	Victoria Ashworth	London
Berglunds snabbköp	Christina Berglund	Luleå

Figure 2. HyperlinkReports.SCX shows a customer report with each customer hyperlinked to their orders.



Orders for Alfreds Futterkiste

Order ID	10643	Order Date	08/25/97	Shipped Date	09/02/97

Quantity	Product Name	Unit Price	Total Price
15	Rössle Sauerkraut	$45.60	$684.00
21	Chartreuse verte	$18.00	$378.00
2	Spegesild	$12.00	$24.00
			$362.00

Figure 3. Clicking on a customer's link displays a report of the orders for that customer.

The Init method of the form uses the HyperlinkListener class to generate a hyperlinked HTML file from the HyperlinkCustomers report, and then calls the ShowReport method to display it in the Web Browser control. It also maintains a collection of HTML files generated by the form so they can be deleted when the form is closed.

```
with This

* Create a collection of the HTML files we'll create
* so we can nuke them all when we close.

  .oFiles = createobject('Collection')

* Create the customers report.

  .oListener = newobject('HyperlinkListener', ;
    'HyperlinkListener.prg')
  .oListener.TargetFileName = ;
    fullpath('HyperlinkCustomers.html')
  report form HyperlinkCustomers object .oListener
  .oFiles.Add(.oListener.TargetFileName)

* Display it.

  .ShowReport()
endwith
```

The ShowReport method simply tells the Web Browser control to load the current HTML file:

```
local lcFile
lcFile = This.oListener.TargetFileName
This.oBrowser.Navigate2(lcFile)
```

Rather than generating orders reports for every customer and hyperlinking to them, I decided to generate the reports on demand when a customer name is clicked. To do that, I needed to intercept the hyperlink click. Fortunately, that's easy to do: simply put code into the BeforeNavigate2 event of the Web Browser control.

To tell BeforeNavigate2 that this isn't a normal hyperlink, I used a convention of "vfps://," which stands for "VFP script," rather than "http://." The code in BeforeNavigate2 looks for this string in the URL to be navigated to, and if found, executes the code in the rest of the URL rather than navigating to it. For example, the User memo of the CompanyName field in HyperlinkCustomers.FRX has the following:

```
*:URL = vfps://Thisform.ShowOrdersForCustomer('
<<CustomerID>>')
```

The HyperlinkListener report listener will convert this to an anchor tag such as <a href="vfps://Thisform.ShowOrdersforCustomer('ALFKI')"> for the customer with a CustomerID of ALFKI. When you click on this hyperlink in the Web Browser control, BeforeNavigate2 fires and the code in that event strips off the "vfps://" part and executes the rest. It also sets the Cancel parameter, passed by reference, to .T. to indicate that the normal navigation should not take place (similar to using NODEFAULT in a VFP method). Here's the code for BeforeNavigate2:

```
LPARAMETERS pdisp, url, flags, targetframename, ;
  postdata, headers, cancel
local lcMethod
if url = 'vfps://'
  lcMethod = substr(url, 8)
  lcMethod = left(lcMethod, len(lcMethod) - 1)
    && strip trailing /
  &lcMethod
  cancel = .T.
endif url = 'vfps://'
```

The ShowOrdersForCustomer method, executed when you click on a company name, runs the HyperlinkOrders report for the specified customer, displays it in the Web Browser control, and adds the file name to the collection of files to be deleted when the form is closed.

```
lparameters tcCustomerID
with This
  .oListener.TargetFileName = ;
    fullpath(tcCustomerID + '.html')
  report form HyperlinkOrders object .oListener ;
    for Orders.CustomerID = tcCustomerID
  .ShowReport()
  .oFiles.Add(.oListener.TargetFileName)
endwith
```

The CustomerName field in the HyperlinkOrders report has "*:URL = vfps://Thisform.ShowCustomers()" in its User memo so clicking on this hyperlink in the report redisplays the customer list.

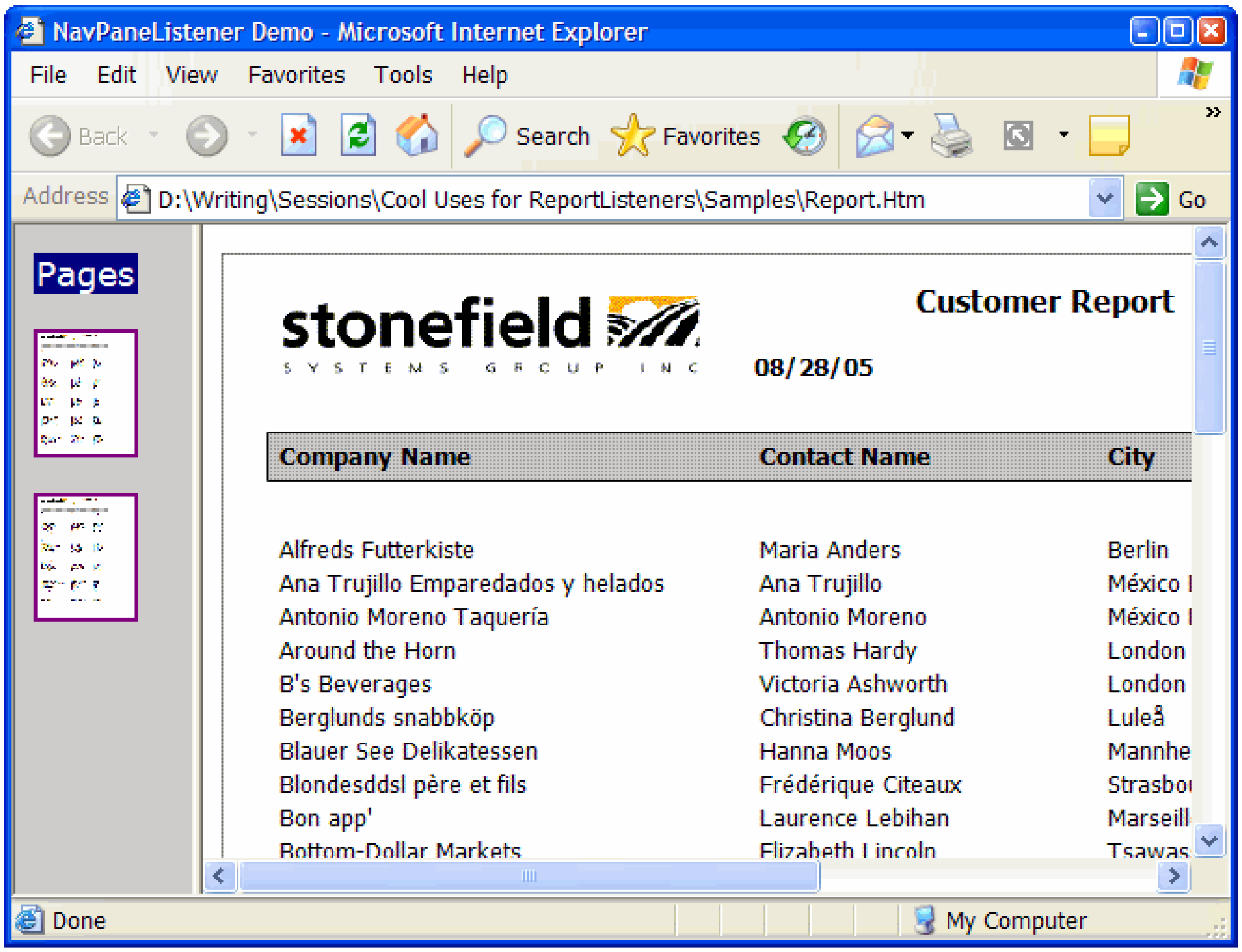
Example 3: Launching a VFP form

CustomerReport.SCX is similar to HyperlinkReports.SCX, albeit a little simpler. It too hosts a Web Browser control that displays the HTML from a report, EditCustomers.FRX, which looks the same as the previous example. However, clicking on a customer name in this form displays a maintenance form for the selected customer.

EditCustomers.FRX is a clone of the HyperlinkCustomers report used in the previous example, but has "*:URL = vfps://Thisform.EditCustomer('<<CustomerID>>')" in the User memo of the CompanyName field instead. The form's EditCustomer method, called from BeforeNavigate2 when a customer name is clicked, launches the Customers form, passing it the CustomerID for the selected customer. The Customers form is a simple maintenance form for the Customers table, with controls bound to each field and Save and Cancel buttons.

Report Table of Contents

MVP Fabio Vazquez has created another kind of listener that has hyperlinks, albeit for a completely different purpose. His NavPaneListener, available for download from http://ReportListener.com, provides an HTML report previewer with a table of contents for the report. As you can see in **Figure 4**, a thumbnail image of each page is shown at the left and the current page is shown at the right. Clicking on a thumbnail navigates to the appropriate page.


NavPaneListener Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address
Go
Pages
Done
My Computer

D:\Writing\Sessions\Cool Uses for ReportListeners\Samples\Report.Htm

Customer Report

08/28/05

Company Name	Contact Name	City
Alfreds Futterkiste	Maria Anders	Berlin
Ana Trujillo Emparedados y helados	Ana Trujillo	México I
Antonio Moreno Taquería	Antonio Moreno	México I
Around the Horn	Thomas Hardy	London
B's Beverages	Victoria Ashworth	London
Berglunds snabbköp	Christina Berglund	Luleå
Blauer See Delikatessen	Hanna Moos	Mannhe
Blondesddsl père et fils	Frédérique Citeaux	Strasbo
Bon app'	Laurence Lebihan	Marseill
Bottom-Dollar Markets	Elizabeth Lincoln	Tsawas

Figure 4. Fabio Vazquez's NavPaneListener creates an HTML report previewer with a table of contents.

Like HyperlinkListener, NavPaneListener is quite simple. Its OutputPage event, called as each page is to be output, simply generates a GIF file for the page by calling itself again with the appropriate parameters. tnDeviceType is initially -1, meaning no output, because the listener's ListenerType property is set to 2. In that case, OutputPage calls itself, passing the name and path for a GIF to generate (the cPath property defaults to the current directory) and a device type value which indicates a GIF file. On the second call, in addition to the normal behavior (generating the specified GIF file), OutputPage passes the name and path to the AddPage method of a collaborating object stored in the oNavigator property. (Note: I translated some of Fabio's code into English to make it more readable.)

```
procedure OutputPage(tnPageNo, teDevice, ;
  tnDeviceType)
  local lnDeviceType
  with This
    do case
      case tnDeviceType = -1  && None
        lnDeviceType = 103     && GIF
        .OutputPage(tnPageNo, .cPath + 'Page' + ;
          transform(tnPageNo) + '.gif', lnDeviceType)
      case tnDeviceType = 103
        .oNavigator.AddPage(teDevice)
    endcase
  endwith
endproc
```

After the report is done, the navigator object creates a couple of HTML documents, one that defines a frameset with the table of contents in the left frame and the contents to display in the right frame, and one that contains the table of contents as thumbnails of the GIF files hyperlinked to display the full-size GIF file in the content frame. The navigator object then automates Internet Explorer to display the frameset document.

Rendering to a Cursor

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.

Normally, when you use a ReportListener, it performs some type of visible output, such as HTML, reports with dynamic formatting, and so forth. However, the output from the ReportListener we're going to look at now doesn'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. 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. You can use this cursor or table for lots of things. I'll show you a couple of uses for it in this document.

"Live" Preview Surface

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 960ths of an inch while the form uses pixels. Fortunately, it's easy to convert from 960ths 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 (created by the GetReportObject method).

```
lparameters tnRecno, ;
  tnClickType
local loObject
loObject = This.GetReportObject(tnRecno)
do case
```

```
  case tnClickType = 1
    raiseevent(This, 'ObjectClicked',      loObject, ;
      This)
  case tnClickType = 2
    raiseevent(This, 'ObjectDblClicked',   loObject, ;
      This)
  otherwise
    raiseevent(This, 'ObjectRightClicked', loObject, ;
      This)
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 in the next section.

Check it out

Let's try it out. TestSFPreview1.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
```

TestSFPreview1.PRG then instantiates an object to handle clicks in the preview surface and binds the various click events to it. Finally, it displays the preview form.

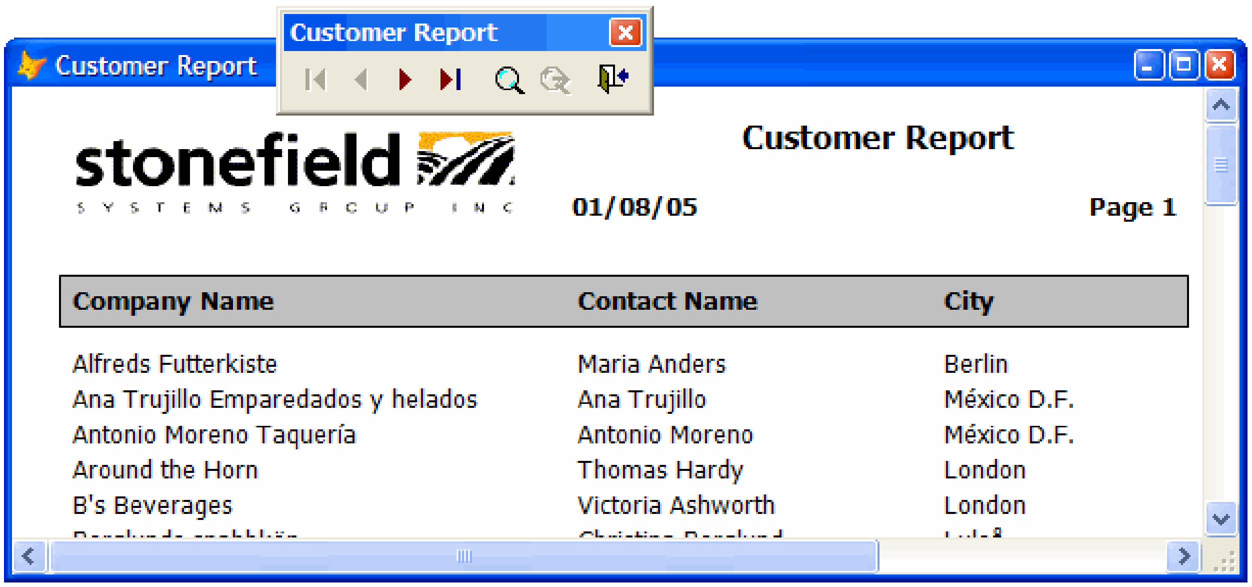
```
loHandler = createobject('ClickHandler')
bindevent(loForm, 'ObjectClicked',      loHandler, ;
  'OnClick')
bindevent(loForm, 'ObjectDblClicked',   loHandler, ;
  'OnDblClick')
bindevent(loForm, 'ObjectRightClicked', loHandler, ;
  'OnRightClick')

* Display the preview form.

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, toForm)
    do case
      case inlist(toObject.ObjType, 5, 8)
        messagebox('You clicked ' + ;
          trim(toObject.Contents))
      case toObject.ObjType = 7
        messagebox('You clicked a rectangle')
      case toObject.ObjType = 6
```

```
        messagebox('You clicked a line')
      case toObject.ObjType = 17
        messagebox('You clicked an image')
    endcase
  endproc
enddefine
```

When you run TestSFPreview1.PRG, you'll see the preview form shown in **Figure 5**. 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 displayed in a message box. 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.



Customer Report

01/08/05

Page 1

Company Name	Contact Name	City
Alfreds Futterkiste	Maria Anders	Berlin
Ana Trujillo Emparedados y helados	Ana Trujillo	México D.F.
Antonio Moreno Taquería	Antonio Moreno	México D.F.
Around the Horn	Thomas Hardy	London
B's Beverages	Victoria Ashworth	London

Figure 5. SFPreviewForm, combined with DBFListener, provides a live preview surface.

Finding Text

You may have noticed that the toolbar contains a couple of interesting looking buttons. These are used for finding text within the report; clicking on the Find button in the preview form's toolbar calls the FindText method of the preview form. That method prompts the user for some text to find, calls ClearFind to clear any existing find settings, tries to find the specified text in the output cursor created by DBFListener, and calls the HandleFind method to deal with the results.

```
local lcText, ;
  lcObject
with This
  lcText = inputbox('Find:', 'Find Text')
  if not empty(lcText)
    .ClearFind()
    select (.cOutputAlias)
    .cTextToFind = lcText
    locate for ;
      upper(This.cTextToFind) $ upper(CONTENTS)
    .HandleFind(found())
  endif not empty(lcText)
endwith
```

FindNext, called from the Find Next button in the toolbar, simply uses CONTINUE to find the next instance and also calls HandleFind to handle the results.

HandleFind deals with the results of the search. If a record was found containing the desired text, HandleFind calls the AddHighlightedItem method to add an object containing information about the shape to highlight to the oHighlightedItems collection (we won't look at this method). One complication of highlighting the found text is that if the preview form is sized smaller than page size, the highlighted text may be outside the visible area of the page. So, HandleFind calls the ScrollToObject method to scroll the form so the highlighted text is within the visible area. It then either calls DrawPage or DisplayPage, both of which I discussed in the previous section, depending on whether the first instance of the found text appears on the currently displayed page or on a different page. (In case you're wondering, ScrollToObject must be called before DrawPage or DisplayPage because the highlighting process, which we'll look at later, needs to have the highlighted area within the current viewable area.) If no matching record could be found, HandleFind beeps (using the Windows API MessageBeep function, declared in Init), clears the existing find information, and redraws the current page, thus clearing any previously highlighted text.

```
lparameters tlFound
local loObject
with This

* If we found the object, add it to the collection
* of highlighted objects and flag that the Find Next
* function can be used.

  if tlFound
    loObject = .GetReportObject(recno())
    .AddHighlightedItem(loObject)
    .lCanFindNext = .T.

* Scroll the form if the found object isn't currently
* visible. If the object is on the current page,
* redraw it. Otherwise, display the proper page.

    .ScrollToObject(loObject.Top, loObject.Height, ;
      loObject.Left, loObject.Width)
    if loObject.Page = .nCurrentPage
      .DrawPage()
      .RefreshToolbar()
    else
      .DisplayPage(loObject.Page)
    endif loObject.Page = .nCurrentPage

* We didn't find the text, so clear the find
* settings.

  else
    MessageBeep(16)
    .ClearFind()
    .DrawPage()
    .RefreshToolbar()
  endif tlFound
endwith
```

The ScrollToObject method does what its name suggests (I find that's usually the best way to name a method <g>)—it scrolls the form as necessary so the specified location is visible. This involves checking the various ViewPort* properties to see what part of the total image is currently displayed in the viewable area and calling SetViewPort to change the viewable area if necessary.

```
lparameters tnTop, ;
  tnHeight, ;
  tnLeft, ;
  tnWidth
local lnNewTop, ;
  lnNewLeft, ;
  lnVPos, ;
  lnHPos
with This
  lnNewTop  = .ViewPortTop
  lnNewLeft = .ViewPortLeft
  lnVPos    = tnTop  + tnHeight
  lnHPos    = tnLeft + tnWidth
  if not between(lnVPos, .ViewPortTop, ;
    .ViewPortTop + .ViewPortHeight)
    lnNewTop = lnVPos - .ViewPortHeight/2
  endif not between(lnVPos ...
  if not between(lnHPos, .ViewPortLeft, ;
    .ViewPortLeft + .ViewPortWidth)
    lnNewLeft = lnHPos - .ViewPortWidth/2
  endif not between(lnHPos ...
  if lnNewTop <> .ViewPortTop or ;
    lnNewLeft <> .ViewPortLeft
    .SetViewPort(lnNewLeft, lnNewTop)
  endif lnNewTop <> .ViewPortTop ...
endwith
```

Highlighting text

How is the found text highlighted? The DrawPage method, which I discussed earlier, calls the HighlightObjects method to ensure any objects on the current page that exist in the collection of items to highlight are drawn in a manner that makes them stand out.

```
local lcObject, ;
  lcRepObject, ;
  loRepObject
with This
  for each loObject in .oHighlightedItems
    if loObject.Page = .nCurrentPage
      lcRepObject = loObject.Name
      loRepObject = evaluate('.oContainer.' + ;
        lcRepObject)
      .HighlightObject(loRepObject)
    endif loObject.Page = .nCurrentPage
  next loObject
endwith
```

The actual work of highlighting a specific object is done in HighlightObject. Remember that the page displayed in the preview window is really a GDI+ image, so we can't do much with it. As we saw earlier, SFPreviewForm creates invisible shape objects (not really invisible, but without a border so they don't appear on the preview surface) for each rendered item in the current page of the report. So, to make the found text stand out, we simply have to make the object representing the text visible. You might think the code could simply do something like this:

```
Object.BorderWidth = 2
Object.BorderStyle = 1
Object.BorderColor = rgb(255, 0, 0)
```

However, that doesn't work for two reasons. First, even though it's transparent, the shape overlays the report image, blocking off the text the user wants to see. Second, the form continually flashes. It turns out that if something is changed when Paint fires (in SFPreviewForm, Paint calls DrawPage, which calls HighlightObjects, which calls HighlightObject, which would change the border of the shape if we used this code), it starts a vicious cycle: Paint fires, which changes the drawing surface, which causes Paint to fire, which changes the drawing surface, and so on. So, instead, we'll use GDI+ to draw a box around the shape.

The Init method of the form instantiates GpGraphics and GpPen objects from _GDIPlus.VCX in the FFC subdirectory of the VFP home directory. This class library, which provides an easy-to-use wrapper for GDI+ functions, was created by Walter Nicholls, and he discussed some of its classes in the August and September 2004 issues of FoxTalk. GDI+ needs a handle to the surface it draws on, so you normally call the CreateFromHWnd method of GpGraphics, passing it the HWnd property of the form. However, I ran into a snag using it with SFPreviewForm: when the ScrollBars property is set to anything but 0-None, none of the GDI+ drawing I did showed up on the form. Fortunately, Walter generously pointed out to me that drawing should actually occur on the child window of the form (the part inside the form that scrolls is actually another window), and that I could get a handle to the child window using the GetWindow Windows API function. A similar snag is that if you set the

ShowWindow property of the form to 2-As Top-level Form, the top-level form itself is another window, so Init uses GetWindow twice in this case.

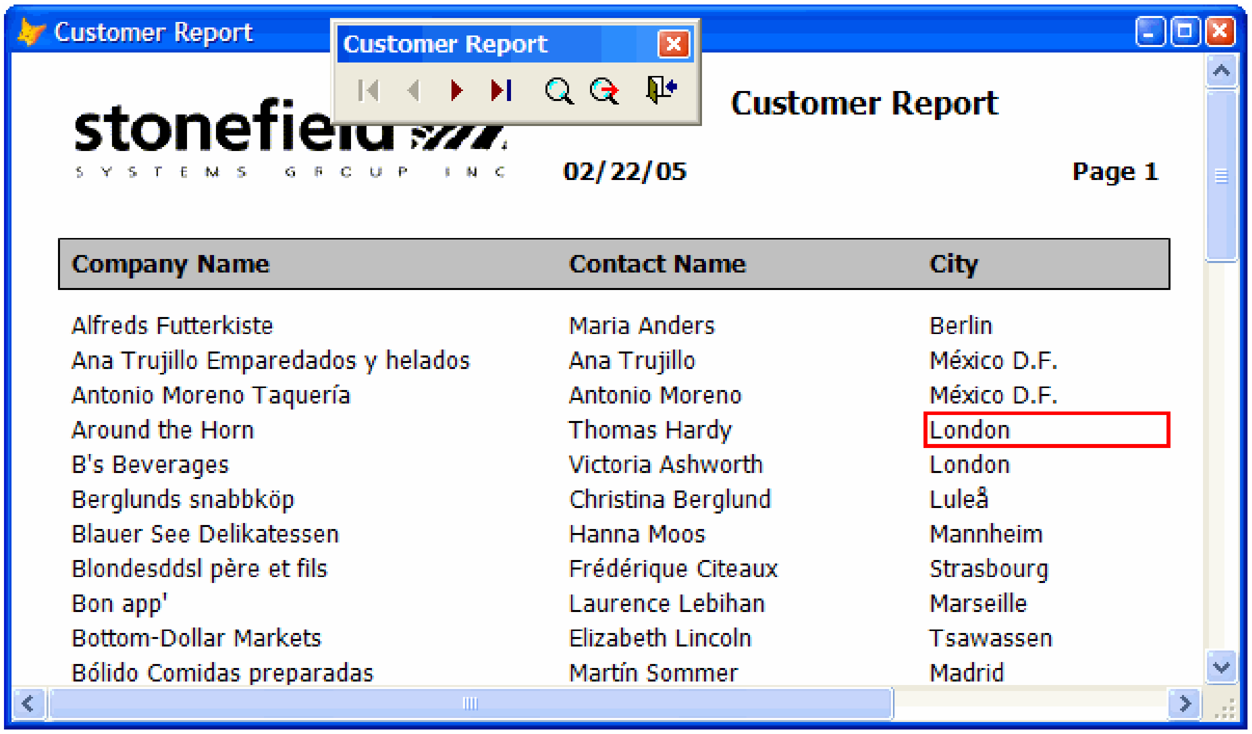
The GpPen object instantiated in Init needs to know what color and line width to use, so set the custom nHighlightColor and nHighlightWidth properties of the form to the RGB value for the desired color and width of the box; they're set by default to 255 (Red) and 2. Note that the RGB values used by VFP are somewhat different than those used by GDI+, so the code in Show rearranges the color value so it works with GDI+.

HighlightObject uses the GpGraphics and GpPen objects to draw a rectangle of the appropriate size and position for the shape object being highlighted.

```
lparameters toObject
local lnX1, ;
  lnY1
with This

* Figure out the upper left corner of the box.

  lnX1 = toObject.Left + .oContainer.Left
  lnY1 = toObject.Top  + .oContainer.Top

* Draw the box.

  .oGraphics.DrawRectangle(.oPen, lnX1, lnY1, ;
    toObject.Width, toObject.Height)
endwith
```

Check it out

Let's see how it works. Run TestSFPreview2.PRG to run a report and preview it using SFPreviewForm. Click the Find button in the toolbar and when prompted, enter something that appears multiple times, such as "London." You'll see a red box appear around the first instance of that text; see **Figure 6** for an example. Click the Find Next button and notice that the box around the first instance disappears and the second instance is now highlighted. As you continue to click Find Next, you may see the preview form automatically scroll or move to another page so the found instance is always visible. Once no more instances are found, you'll hear a beep and no text will be highlighted.


Customer Report

Customer Report

02/22/05

Page 1

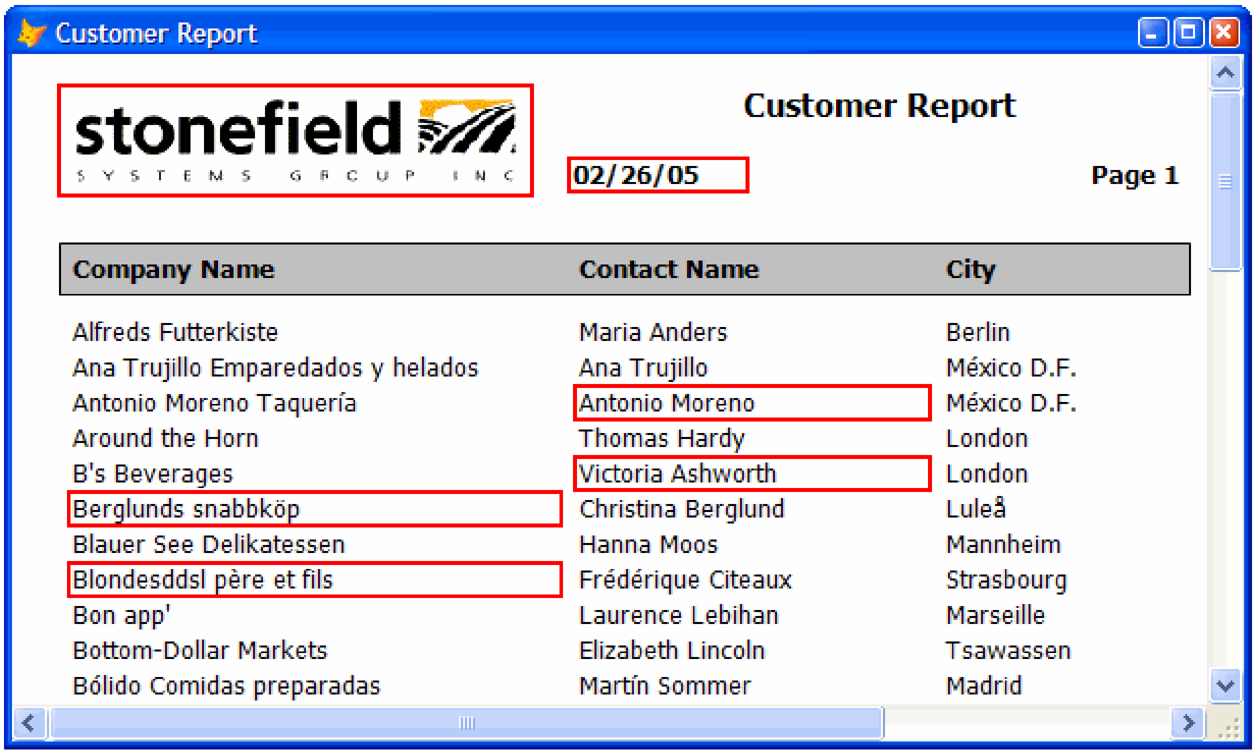
Company Name	Contact Name	City
Alfreds Futterkiste	Maria Anders	Berlin
Ana Trujillo Emparedados y helados	Ana Trujillo	México D.F.
Antonio Moreno Taquería	Antonio Moreno	México D.F.
Around the Horn	Thomas Hardy	London
B's Beverages	Victoria Ashworth	London
Berglunds snabbköp	Christina Berglund	Luleå
Blauer See Delikatessen	Hanna Moos	Mannheim
Blondesddsl père et fils	Frédérique Citeaux	Strasbourg
Bon app'	Laurence Lebihan	Marseille
Bottom-Dollar Markets	Elizabeth Lincoln	Tsawassen
Bólido Comidas preparadas	Martín Sommer	Madrid

Figure 6. SFPreviewForm highlights found text, adding an important feature to a report preview window.

One thing you might be curious about is why I used a collection to hold the highlighted items when only one item is highlighted at a time during a find. That's because I wanted to support the possibility of highlighting multiple items at a time. Perhaps you want the find functionality to find and highlight all instances of the found text rather than one at a time. Perhaps you want to support bookmarks, with each bookmarked item being highlighted.

Here's a simple example: like TestSFPreview1.PRG, TestSFPreview2.PRG binds the ObjectClicked and ObjectRightClicked events to the OnClick and OnRightClick methods of a click handler class, but with different results. When you click an object, that object becomes highlighted. To unhighlight the object, right-click it. **Figure 7** shows the results after I clicked on various objects.

```
loHandler = createobject('ClickHandler')
bindevent(loForm, 'ObjectClicked',      loHandler, ;
  'OnClick')
bindevent(loForm, 'ObjectRightClicked', loHandler, ;
  'OnRightClick')

* A class to handle object clicks.

define class ClickHandler as Custom
  procedure OnClick(toObject, toForm)
    toForm.AddHighlightedItem(toObject)
    toForm.DrawPage()
  endproc

  procedure OnRightClick(toObject, toForm)
    toForm.RemoveHighlightedItem(toObject)
    toForm.DrawPage()
  endproc
enddefine
```



Customer Report

stonefield SYSTEMS GROUP INC

Customer Report

02/26/05

Page 1

Company Name	Contact Name	City
Alfreds Futterkiste	Maria Anders	Berlin
Ana Trujillo Emparedados y helados	Ana Trujillo	México D.F.
Antonio Moreno Taquería	Antonio Moreno	México D.F.
Around the Horn	Thomas Hardy	London
B's Beverages	Victoria Ashworth	London
Berglunds snabbköp	Christina Berglund	Luleå
Blauer See Delikatessen	Hanna Moos	Mannheim
Blondesddsl père et fils	Frédérique Citeaux	Strasbourg
Bon app'	Laurence Lebihan	Marseille
Bottom-Dollar Markets	Elizabeth Lincoln	Tsawassen
Bólido Comidas preparadas	Martín Sommer	Madrid

Figure 7. Because it uses a collection, SFPreviewForm supports multiple highlighted objects.

Summary

The new ReportListener in VFP 9 can be used to provide features we could only dream of in earlier versions. Among the things I discussed in this document are hyperlinking objects within a report, drilldown reports, reports that launch some VFP form or other action, reports with navigation panes, having a "live" preview surface, finding text, and highlighting objects in a report. This truly shows the power of report listeners! I hope you're starting to see the incredible possibilities the VFP 9 ReportListener class provides us.

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, and the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro. Doug is co-author of the "What's New in Visual FoxPro" series (the latest being "What's New in Nine") 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). Doug formerly wrote the monthly "Reusable Tools" column in FoxTalk. He has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over the world. He is one of the administrators for the VFPX VFP community extensions Web site (http://www.codeplex.com/Wiki/View/aspx?ProjectName=VFPX). He has been a Microsoft Most Valuable Professional (MVP) since 1996.