Extending the VFP 9 Reporting System, Part II: Run-Time

Doug Hennig
Stonefield Software Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Voice: 306-586-3341
Fax: 306-586-5080
Email: dhennig@stonefield.com
Web: www.stonefield.com
Web: www.stonefieldquery.com

Overview

In addition to the design-time extensibility of VFP 9's reporting system discussed in Part I of this white paper, VFP 9 also provides the ability to extend the behavior of the reporting system when reports are run. In this document, you'll learn about VFP 9's report listener concept, how it receives events as a report is run, and how you can create your own listeners to provide different types of output besides the traditional print and preview.

Introduction

One of the biggest changes in VFP 9 is the incredible improvements made in the reporting system. There are several aspects to this, one of which we'll explore in this document: the ability to extend the behavior of the run-time reporting engine.

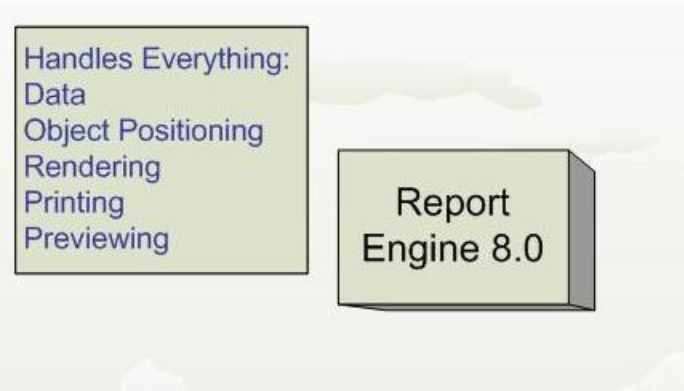
The VFP development team had several goals in mind when they worked on the run-time improvements, including:

- Handling more types of report output than just printing and previewing.
- Using GDI+ for report output. This provides many significant improvements, such as more accurate rendering, smooth scaling up and down of images and fonts, and additional capabilities such as text rotation.
- Providing a more flexible and extendible reporting system.

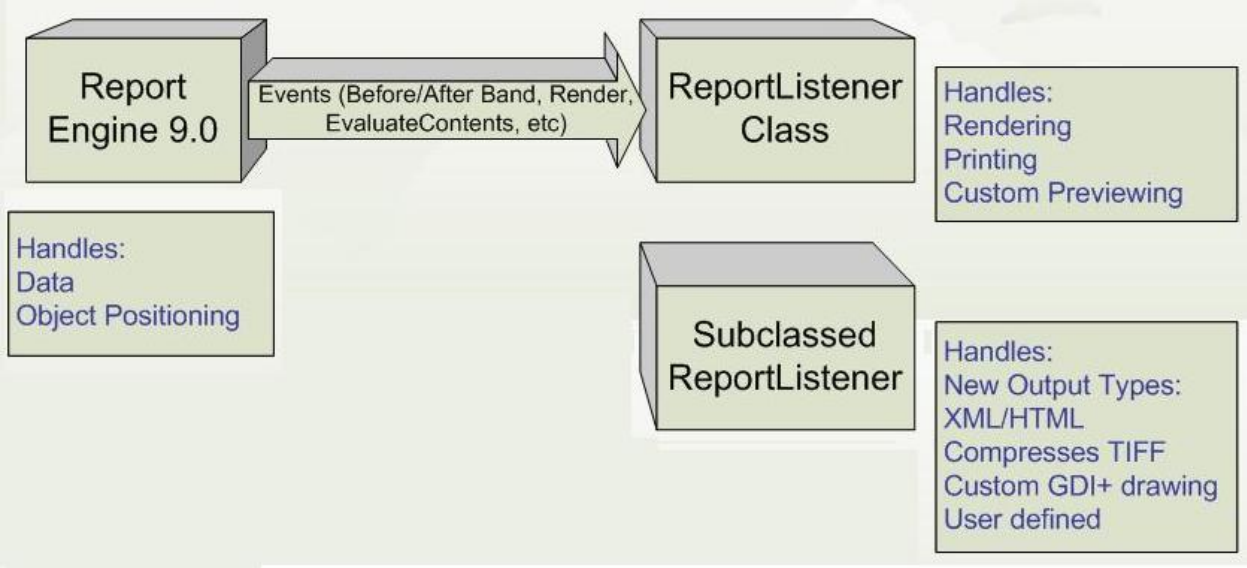
VFP 9 includes both the old report engine and the new one, so you can run reports under either engine as you see fit. However, once you see the benefits of the new report engine, you won't want to go back to old-style reporting.

Reporting Engine Architecture

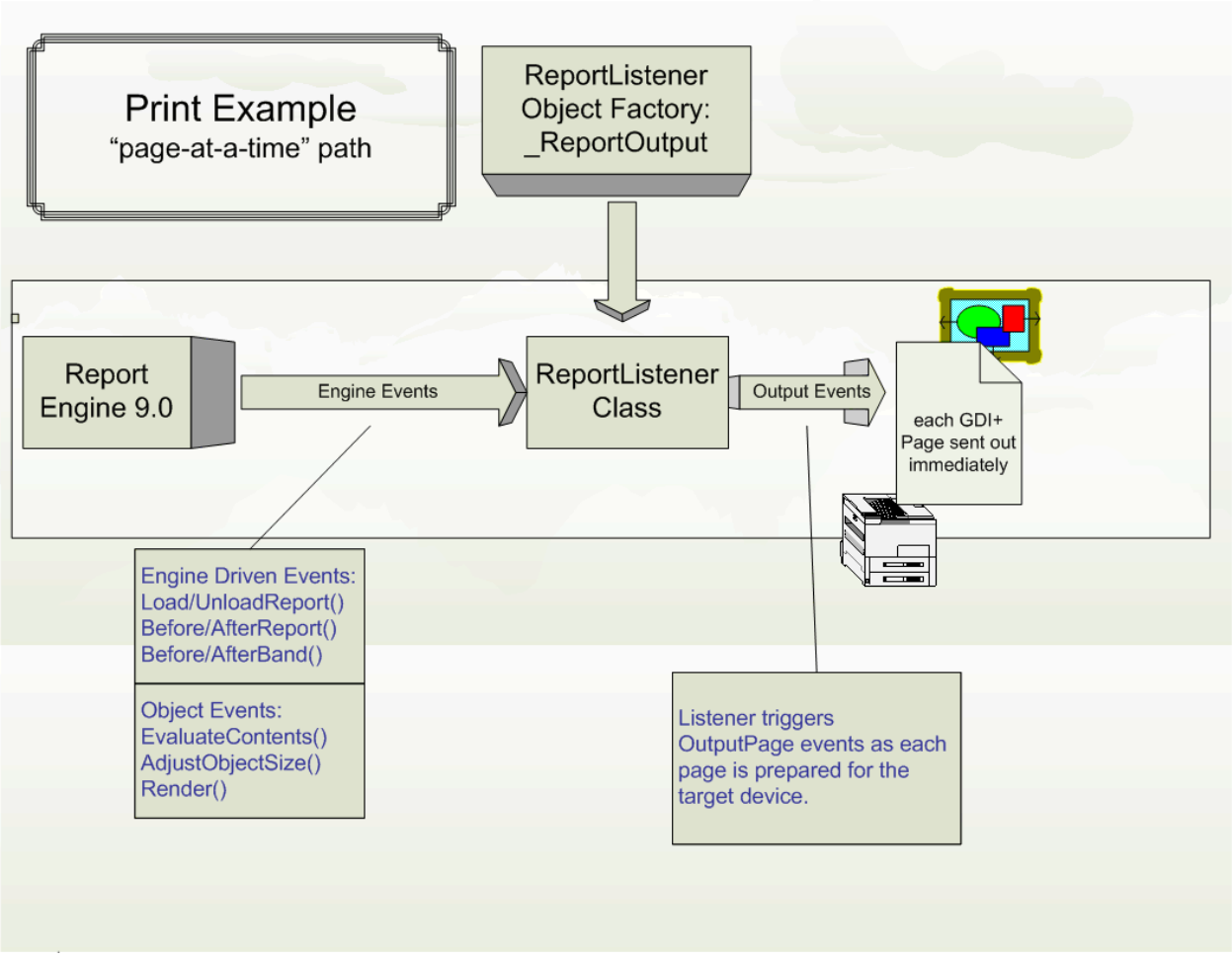
Before VFP 9, the report engine was rather monolithic; it handled everything and with a few exceptions (UDFs, expressions for OnEntry and OnExit of bands, etc.), you couldn't interact with it during a report run.



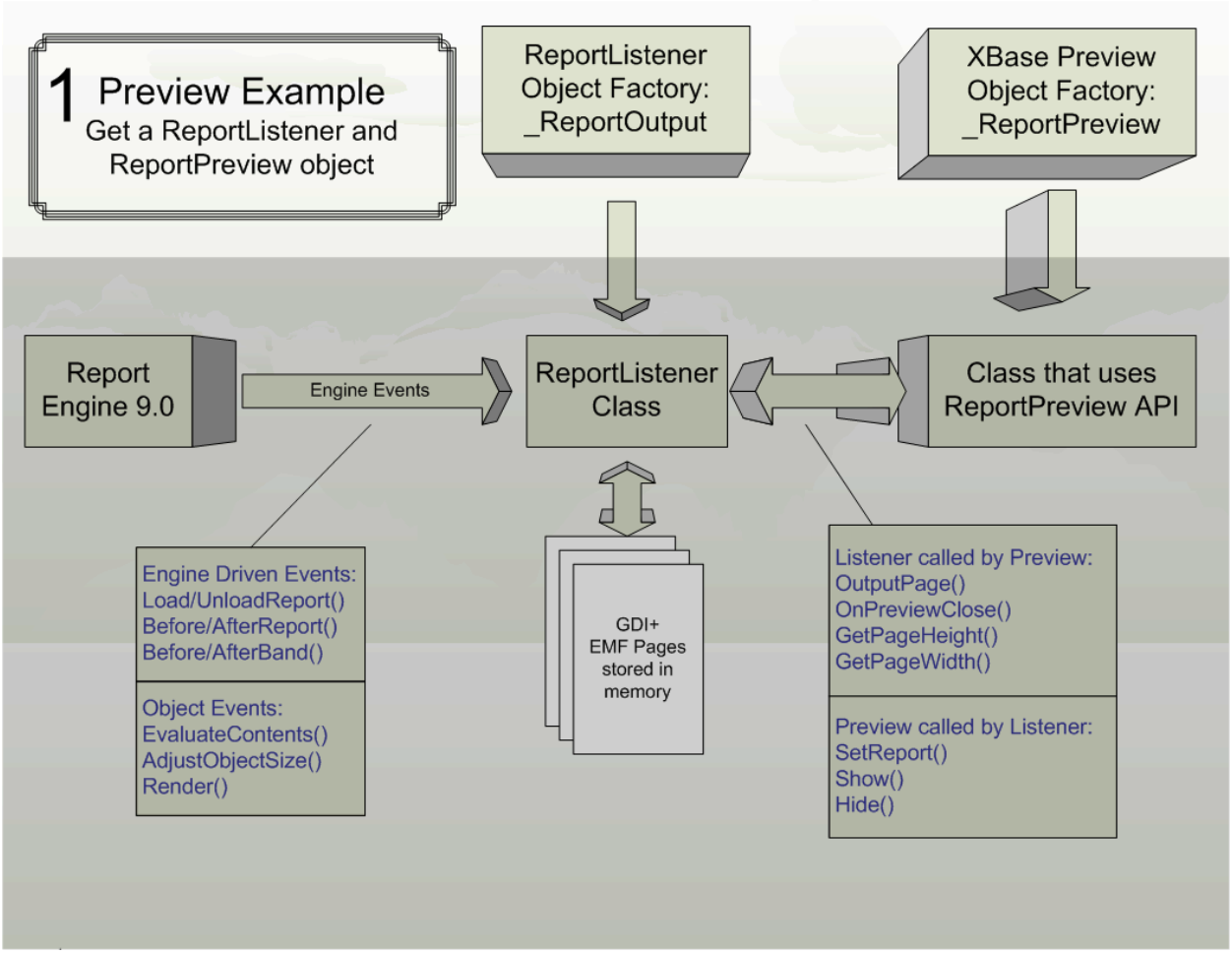
The new reporting engine in VFP 9 splits responsibility for reporting between the report engine, which now just deals with data-handling and object positioning, and a new object known as a report listener, which handles rendering and output. Because report listeners are classes, we can now interact with the reporting process in ways we could only dream of before.

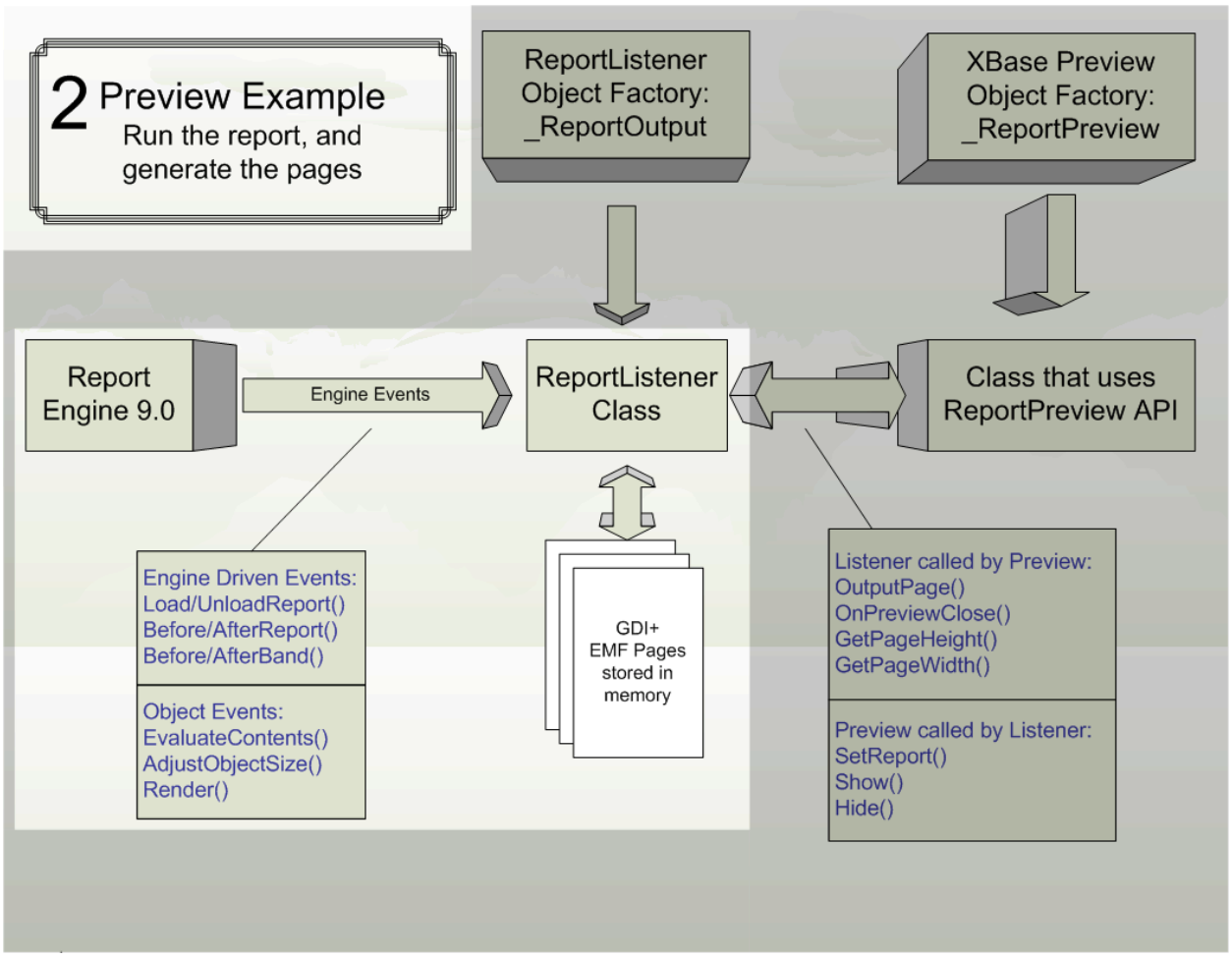


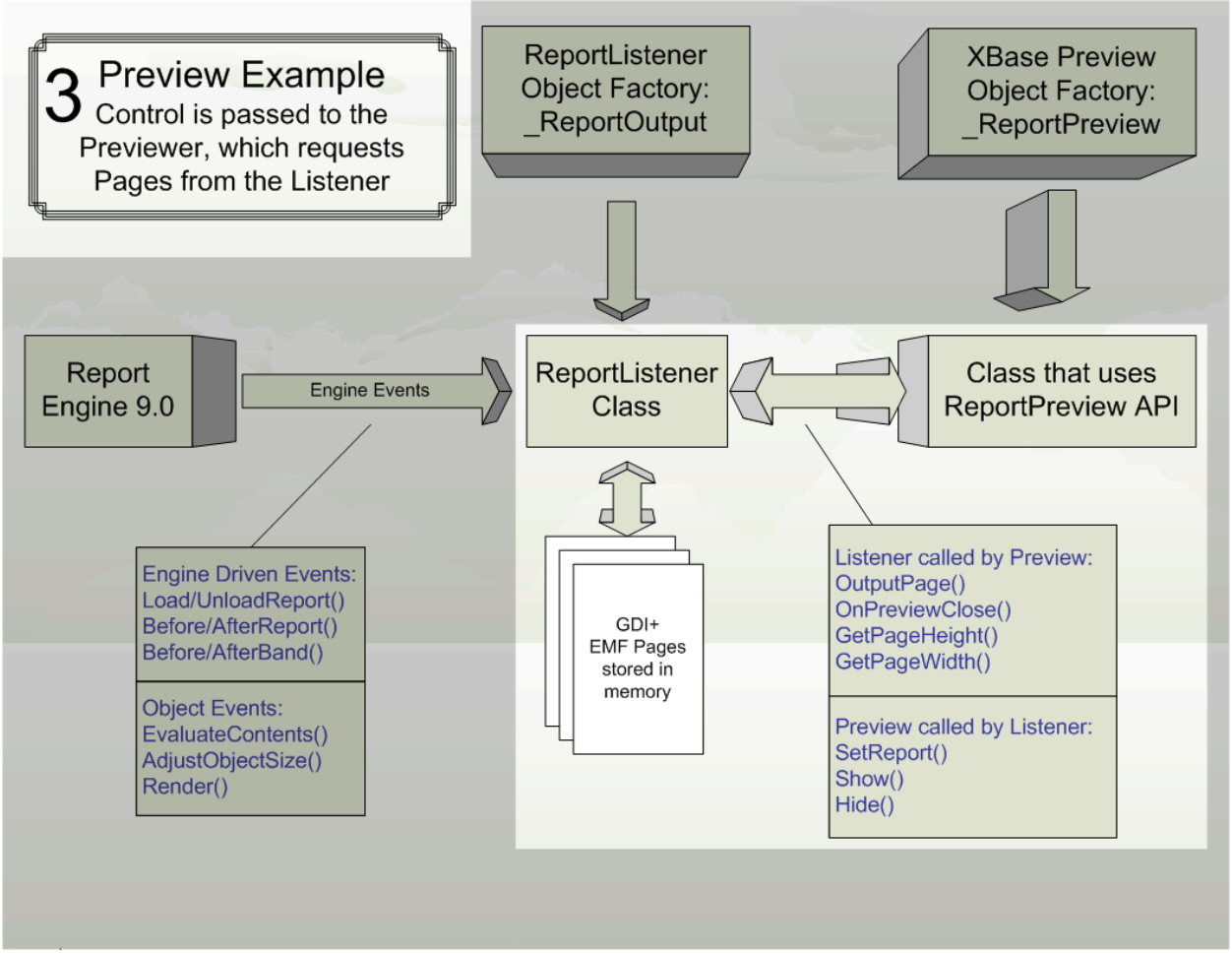
Report listeners produce output in two ways. “Page-at-a-time” mode renders a page and then outputs it. This is typically used when a report is printed.



In “all-pages-at-once” mode, the report listener renders all the pages and caches them in memory. It then outputs these rendered pages on demand. This is typically used when a report is previewed.




2 Preview Example
Run the report, and generate the pages
ReportListener Object Factory: _ReportOutput
XBase Preview Object Factory: _ReportPreview
Report Engine 9.0
Engine Events
ReportListener Class
Class that uses ReportPreview API
Engine Driven Events:
Load/UnloadReport()
Before/AfterReport()
Before/AfterBand()
Object Events:
EvaluateContents()
AdjustObjectSize()
Render()
GDI+ EMF Pages stored in memory
Listener called by Preview:
OutputPage()
OnPreviewClose()
GetPageHeight()
GetPageWidth()
Preview called by Listener:
SetReport()
Show()
Hide()



New Reporting Syntax

VFP 9 supports running reports using the old report engine; simply use the REPORT command as you did before (although, as we'll see in a moment, you can use a new command to override the behavior of REPORT). To get new-style reporting behavior, use the new OBJECT clause of the REPORT command. OBJECT supports two ways of using it: by specifying a report listener and specifying a report type. Microsoft refers to this as "object-assisted" reporting.

A report listener is an object that provides new-style reporting behavior. Report listeners are based on a new base class in VFP 9, ReportListener. We'll look at this class in more detail later. To tell VFP to use a specific listener for a report, instantiate the listener class and then specify the object's name in the OBJECT clause of the REPORT command. Here's an example:

```
loListener = createobject('MyReportListener')
report form MyReport object loListener
```

If you'd rather not instantiate a listener manually, you can have VFP do it for you automatically by specifying a report type:

```
report form MyReport object type 1
```

The defined types are 0 for outputting to a printer, 1 for previewing, 2 for "page-at-a-time" mode but not send the output to a printer, 3 for "all-pages-at-once" mode but not invoke the preview window, 4 for XML output, and 5 for HTML output. Other, user-defined, types can also be used.

When you run a report this way, the application specified in the new _REPORTOUTPUT system variable (ReportOutput.APP in the VFP home directory by default) is called to figure out which listener class to instantiate for the specified type. It does this by looking for the listener type in a listener registry table; we'll go into more detail on this later. If it finds the desired class, it instantiates the class and gives a reference to the listener object to the reporting engine. Thus, using OBJECT TYPE SomeType in the REPORT command is essentially the same as:

```
loListener = .NULL.
do (_ReportOutput) with SomeType, loListener
report form MyReport object loListener
```

You're probably thinking "But I've got tons of reports in my application. Do I have to find and modify every REPORT command in the entire application?" Fortunately, there's an easier way: SET REPORTBEHAVIOR 90 turns on object-assisted reporting by default. This means the REPORT command then behaves as if you've specified OBJECT TYPE 0 when you use the TO PRINT clause or OBJECT TYPE 1 when you use the PREVIEW clause. SET REPORTBEHAVIOR 80 reverts to VFP 8 and earlier behavior. If most or all of the reports in your application work just fine in object-assisted mode, use SET REPORTBEHAVIOR 90 at application startup. Because there are rendering differences between new and old-style reporting, some reports may need to be tweaked to work properly with new-style reporting, so either tweak them or use SET REPORTBEHAVIOR 80 to run just those reports.

ReportOutput.APP is primarily an object factory: it instantiates the appropriate listener for a report. It also includes some listeners that provide XML and HTML output. However, since it's just an Xbase application, you could substitute it with your own application by setting _REPORTOUTPUT accordingly.

Another new system variable, _REPORTPREVIEW, specifies the name of an Xbase application used as the preview window for reports. By default, this variable points to ReportPreview.APP in the VFP home directory, but you could substitute your own application if you wish to. For example, you may want a form that provides a list of reports at the left and a preview area at the right. When the user double-clicks on a report, it previews that report in the preview area. The ability to use an Xbase application as the preview window provides us with almost limitless control over the appearance and behavior of previewing. This document doesn't discuss ReportPreview.APP or the requirements of a replacement for it.

ReportListener

One of the keys to more flexible reporting and support for other types of output than just previewing and printing is the new ReportListener base class. During the run of a report, VFP exposes reporting events to ReportListeners as they happen. The base class ReportListener has some native behavior, but extensibility really kicks in when you create and use your own subclasses.

Let's take a look at the properties, events, and methods of ReportListener to understand its capabilities.

Properties

Property	Type	Description
AllowModalMessages	L	If .T, allows modal messages showing the progress of the report (the default is .F.).
CommandClauses	O	An Empty object with properties indicating what clauses were used in the REPORT command.
CurrentDataSession	N	The data session ID for the report's data.
CurrentPass	N	Indicates the current pass through the report. A report with _PageTotal will require two passes; others will only require one, so CurrentPass will always be 1 in that case.
DynamicLineHeight	L	.T. (the default) to use GDI+ line spacing, which varies according to font characteristics, or .F. to use old-style fixed line spacing.
FRXDataSession	N	The data session ID for the FRX cursor (a copy of the report file the reporting engine is running opened for a ReportListener's use).
GDIPlusGraphics	N	The handle for the GDI+ graphics object used for rendering. Read-only.
ListenerType	N	The type of report output the listener produces. The default is -1, which specifies no output, so you'll need to change this to a more reasonable value. See the discussion of the OutputPage method for a list of values.
OutputPageCount	N	The number of pages rendered.
OutputType	N	The output type as specified in the OBJECT TYPE clause of the REPORT command.
PageNo	N	The current page number being rendered.
PageTotal	N	The total number of pages in the report.
PreviewContainer	O	A reference to the display surface onto which the report will be rendered for previewing.

PrintJobName	C	The name of the print job as it appears in the Windows Print Queue dialog.
QuietMode	L	.T. (the default is .F.) to suppress progress information.
SendGDIPlusImage	N	1 (the default is 0) to sent a handle to an image for a General field to the Render method.
TwoPassProcess	L	Indicates whether two passes will be used for the report.

The CommandClauses property contains a reference to an Empty object with properties representing the various clauses of the REPORT command, plus a few other goodies.

Property	Type	Description
ASCII	L	.T. if the ASCII keyword was specified when outputting to a file.
DE_Name	C	The name of the DataEnvironment object for the report. Either the name specified with the NAME clause or the name of the report if not specified.
Environment	L	.T. if the ENVIRONMENT keyword was specified.
File	C	The name of the report to run.
Heading	C	The heading specified with the HEADING keyword.
IsDesignerLoaded	L	Indicates if the person designing the report has been drinking <g>. Seriously, .T. if the report is being run from within the Report Designer.
InScreen	L	.T. if the INSCREEN keyword was specified.
InWindow	C	The name of the window specified with the IN WINDOW keyword.
IsReport	L	.T. if this is a report or .F. if it's a label.
NoConsole	L	.T. if the NOCONSOLE keyword was specified.
NoDialog	L	.T. if the NODIALOG keyword was specified.
NoEject	L	.T. if the NOEJECT keyword was specified.

NoPageEject	L	.T. if the NOPAGEEJECT keyword was specified.
NoReset	L	.T. if the NORESET keyword was specified.
NoWait	L	.T. if the NOWAIT keyword was specified with the PREVIEW keyword.
Off	L	.T. if the OFF keyword was specified.
OutputTo	N	The type of output specified in the TO clause: 0 = no TO clause was specified, 1 = printer, 2 = file
PDSetup	L	.T. if the PDSETUP keyword was specified with the LABEL command.
Plain	L	.T. if the PLAIN keyword was specified.
Preview	L	.T. if the PREVIEW keyword was specified.
Prompt	L	.T. if the PROMPT keyword was specified.
RangeFrom	N	The starting page specified in the RANGE clause, or 1 if not specified.
RangeTo	N	The ending page specified in the RANGE clause, or -1 if not specified.
RecordTotal	N	The total number of records in the main cursor being reported on.
Sample	L	.T. if the SAMPLE keyword was specified with the LABEL command.
Summary	L	.T. if the SUMMARY keyword was specified.
ToFile	C	The name of the file specified with the TO FILE clause.
ToFileAdditive	L	.T. if the ADDITIVE keyword was specified when outputting to a file.
Window	C	The name of the window specified with the WINDOW keyword.

A special comment about datasession handling is in order. There are actually three datasessions involved during a report run. The first is the datasession the ReportListener is instantiated in. This will often be the default datasession. The second is the datasession the FRX cursor is open in. The FRXDataSession property contains the datasession ID for this cursor, so use SET DATASESSION TO This.FRXDataSession if you need access to the FRX. The third is the

datasession the report's data is in. If the report has a private datasession, this will be a unique datasession; otherwise, it'll be the default datasession. The CurrentDataSession property tells you which datasession to use, so if a ReportListener needs to access the report's data, you'll need to SET DATASESSION TO This.CurrentDataSession. Remember to save the ReportListener's datasession and switch back to it after selecting either the FRX or report data datasession.

Events

There are several types of events, so we'll look at each type separately.

Report Events

Report events are those that fire when something affects the report as a whole.

Event	Parameters	Description
LoadReport	None	Analogous to the Load event of a form in that it's the first event fired and returning .F. prevents the report from running. Since this event is fired before the FRX is loaded and the printer spool is opened, this is the one place where you can change the contents of the FRX on disk or change the printer environment before the report is run.
UnloadReport	None	Like the Unload event of a form, UnloadReport is fired after the report has been run. This is typically used for clean up tasks.
BeforeReport	None	Fires after the FRX has been loaded but before the report has been run.
AfterReport	None	Fires after the report has been run.
OnPreviewClose	tlPrint	Fires when the user closes the preview window or prints a report from preview.

Band Events

These events fire as a band is being processed.

Event	Parameters	Description
BeforeBand	tnObjCode tnFRXRecno	Fires before a band is processed. The first parameter represents the value of the OBJCODE field in the FRX for the specified band, and the second is the record number in the FRX cursor for the band's record.
AfterBand	tnObjCode	Fires after a band is processed. Same parameters as BeforeBand.

	tnFRXRecno	

Object Events

These events fire as a report object is being processed.

EvaluateContents(tnFRXRecno, toObjProperties): this event fires at the beginning of band processing for each field (but not label) object, and gives you the opportunity to change the appearance of the field. The first parameter is the FRX record number for the field object being processed and the second is an object containing properties about the field object. The properties this object contains are shown in the following table. You can change any of these properties to change the appearance of the field in the report. If you do so, set the Reload property of the object to .T. to notify the report engine that you've changed one or more of the other properties. Also, return .T. if other listeners can make more changes to the field. We'll see some practical examples of this method later.

Property	Type	Description
FillAlpha	N	The alpha, or transparency, of the fill color. Allows finer control that simply transparent or opaque. The values range from 0 for transparent to 255 for opaque.
FillBlue	N	The blue value of an RGB() color for the fill color.
FillGreen	N	The green value of an RGB() color for the fill color.
FillRed	N	The red value of an RGB() color for the fill color.
FontName	C	The font name.
FontSize	N	The font size.
FontStyle	N	A value representing the font style. Additive values of 1 (bold), 2 (italics), 4 (underlined), and 128 (strikethrough).
PenAlpha	N	The alpha of the pen color.
PenBlue	N	The blue value of an RGB() color for the pen color.
PenGreen	N	The green value of an RGB() color for the pen color.
PenRed	N	The red value of an RGB() color for the pen color.

Reload	L	Set this to .T. to notify the report engine that you've changed one or more of the other properties.
Text	C	The text to be output for the field object.

AdjustObjectSize(tnFRXRecno, toObjProperties): this event fires at the beginning of band processing for each shape object. It gives you the ability to change the shape, and is usually used when you want to replace the shape with a custom rendered object and need to size the object dynamically. The first parameter is the FRX record number for the shape object being processed and the second is an object containing properties about the shape object. The properties this object contains are shown in the following table. If you change Height or Width, set the Reload property of the object to .T. to notify the report engine that you've changed these properties.

Property	Type	Description
Height	N	The height of the object, from 0 to 64000. Changing this value to a greater value (a lesser one is ignored) causes other floating objects in the band to be pushed down and the band to stretch.
Left	N	The left position of the object. Read-only; provided for reference only.
Top	N	The top position of the object. Read-only; provided for reference only.
Width	N	The width of the object, from 0 to 64000. Changing this value doesn't alter report engine behavior, but the new value will be passed to the Render method so the listener could do something with it.
Reload	L	Set this to .T. to notify the report engine that you've changed one or more of the other properties.

Render(tnFRXRecno, tnLeft, tnTop, tnWidth, tnHeight, tnObjectContinuationType, tcContentsToBeRendered, tnGDIPlusImage): this method is the big one: it's called at least once for each object being rendered (it may be called more than once for objects that span bands or pages). As with the other object events, the first parameter is the FRX record number for the object being rendered. The next four parameters represent the position and size of the object. tnObjectContinuationType indicates whether a field, shape, or line object spans a band or page; it contains one of four possible values:

Value	Description
0	This object is complete; it doesn't continue onto the next band or page.

1	The object has been started, but will not finish on the current page.
2	The object is in the middle of rendering; it neither started nor finished on the current page.
3	The object has been finished on the current page.

tcContentsToBeRendered contains the text of a field or the filename of a picture if appropriate. For fields, the contents are provided in Unicode, appropriately translated to the correct locale using the FontCharSet information associated with the FRX record. Use STRCONV() to convert the string if you want to do something with it, such as storing it in a table. tnGDIPlusImage is used if a picture comes from a General field and the SendGDIPlusImage property is .T.; it contains the graphics handle for the image.

You can supply code in this method if you want to render an object differently than would otherwise be done. Note, however, that pretty much anything you'll need to do will require calling GDI+ API functions, so this isn't for the faint of heart. For information about GDI+, see http://msdn.microsoft.com/library/en-us/gdicpp/GDIPlus/GDIPlusReference/FlatGraphics.asp. We'll see an example that overrides Render later.

Methods

Event	Parameters	Description
CancelReport	None	Allows Xbase code to terminate a report early. Required so the ReportListener can do necessary cleanup to prevent crashes, close the print spooler, etc.
OutputPage	tnPageNo teDevice tnDeviceType tnLeft tnTop tnWidth tnHeight tnClipLeft tnClipTop tnClipWidth tnClipHeight	Outputs the specified rendered page to the specified device. The optional tnLeft through through tnClipHeight parameters allow the Listener to specify exactly what area on the target device should be used for rendering when the device type is a container. Discussed in more detail below.
IncludePageInOutput	tnPageNo	Indicates whether the specified page is included in the output or not.
SupportsListenerType	tnType	Indicates whether a listener supports the specified type of output.

GetPageWidth	None	Returns the page width during a report run.
GetPageHeight	None	Returns the page height during a report run.
DoStatus	tcMessage	Provides modeless feedback during a report run.
UpdateStatus	None	Updates the feedback UI.
ClearStatus	None	Removes the modeless feedback UI.
DoMessage	tcMessage tiParams tcTitle	Provides modal feedback during a report run if AllowModalMessages is .T; otherwise, calls DoStatus.

The OutputPage method warrants more discussion. The tnDeviceType parameter determines the type of output this method should perform; it also determines what type of parameter is expected for teDevice. The table below lists the types of output supported in the base class ReportListener. Subclasses could support other types of output.

tnDeviceType	Description	teDevice
0	Printer	Printer handle
1	Graphics device	GDI+ graphic handle
2	Xbase preview window	Reference to VFP control to output to
101	EMF file	File name
102	TIFF file	File name
103	JPEG file	File name
104	GIF file	File name
105	BMP file	File name
201	Multi-page TIFF	File name (the file must already exist).

There are four possible values for the ListenerType property, and each affects how OutputPage is called differently.

ListenerType	Description
0	OutputPage is called by the report engine after each page is rendered to output to a printer. It passes 0 (printer) for tnDeviceType and the GDI+ handle for the printer for teDevice.
1	OutputPage is called by a previewer to display the specified page after all rendering is complete.
2	OutputPage is called by the report engine after each page is rendered but no output is sent to the printer. It passes -1 for tnDeviceType and 0 for teDevice.
3	OutputPage must be called manually to output the specified page after all rendering is complete.

By the way, because report listeners use XBase code, it's now possible to trace code during report execution, something that wasn't possible before and was the source of a lot of frustration for those using UDFs in their reports.

Registering Listeners

Now that we know what a ReportListener looks like, we can create different subclasses that have the behavior we need. Before we do that, though, let's look at how to tell ReportOutput.APP about them.

Like ReportBuilder.APP (see my white paper entitled "Extending the VFP 9 Reporting System, Part I: Run-Time" for details on ReportBuilder.APP), ReportOutput.APP uses a registry table to define which listeners it knows about. Although this table is built into ReportOutput.APP, you can create a copy of it called OutputConfig.DBF using DO (_ReportOutput) WITH -100 (this mechanism may change in a future release). If ReportOutput.APP finds a table with this name in the current directory or VFP path, it'll use it as the a source of listeners it looks at when running a report. Here's the structure of this table:

Field Name	Type	Values	Description
OBJTYPE	I	100 for a listener record	Other record types are used as well; see the VFP documentation for details.
OBJCODE	I	Any valid	Listener type (e.g. 1 for preview)

		listener type	
OBJNAME	V(60)		Class to instantiate
OBJVALUE	V(60)		Class library class is found in
OBJINFO	M		Application containing the class library.

Note that ReportOutput.APP only looks for the first record with OBJTYPE = 100 and OBJCODE set to the desired listener type. So, you'll need to remove or unregister (set OBJCODE to another value such as by adding 100 to it) other listener records of the same type. InstallListener.PRG can handle this for you. Pass it the listener class, library, and type, and it'll take care of the messy details.

```
lparameters tcClass, ;
  tcLibrary, ;
  tnObjCode
local lcClass, ;
  lcLibrary

* Open the report listener table (create it if necessary).

if not used('OutputConfig')
  if not file('OutputConfig.DBF')
    do (_ReportOutput) with -100
  endif not file('OutputConfig.DBF')
  use OutputConfig
endif not used('OutputConfig')

* If the specified listener is already there, ensure it's enabled by setting
* ObjCode to the proper value. Otherwise, add a record for it.

lcClass   = upper(tcClass)
lcLibrary = upper(tcLibrary)
locate for ObjType = 100 and upper(ObjName) == lcClass and ;
  upper(ObjValue) == lcLibrary
if found()
  replace ObjCode with tnObjCode
else
  insert into OutputConfig ;
      (ObjType, ;
      ObjCode, ;
      ObjName, ;
      ObjValue) ;
    values ;
      (100, ;
      tnObjCode, ;
      tcClass, ;
      tcLibrary)
endif found()

* Disable any other listeners with the same ObjCode by setting their ObjCode
* value to an unused value.

replace ObjCode with ObjCode + 100 for ObjType = 100 and ;
```

```
  ObjCode = tnObjCode and not upper(ObjName) == lcClass

* Clean up and exit.

use
```

Note that you don't need to register a listener to use it; you can simply instantiate it manually and pass the reference to it to the OBJECT clause of the REPORT command. This mechanism is a little more work but gives you better control, doesn't require an external copy of ReportOutput.APP's registry table, and allows you to do things like chain report listeners together, as we'll see in a little while.

SFReportListener

Because we should never use VFP base classes, I created a subclass of ReportListener called SFReportListener, defined in SFReportListener.VCX. It provides a few utility methods that most listeners require. It also allows chaining of listeners by providing a Successor property that may contain an object reference to another listener and having all events call the same method in the successor object if it exists, using code similar to:

```
if vartype(This.Successor) = 'O'
  This.Successor.ThisMethodName()
endif vartype(This.Successor) = 'O'
```

One complication (there are several, actually) with successors is that if a successor calls its CancelReport method to cancel the report, the report doesn't actually cancel because only the CancelReport method of the "lead" listener (the one the report engine is actually talking to) cancels the report. So, in the Assign method for the Successor property, we'll use BINDEVENT() to ensure that when the successor's CancelReport method is called, so is ours.

```
lparameters toSuccessor
This.Successor = toSuccessor
if vartype(toSuccessor) = 'O'
  bindevent(toSuccessor, 'CancelReport', This, 'CancelReport', 1)
endif vartype(toSuccessor) = 'O'
```

SelectFRX switches datasessions to the one the FRX cursor is in, saves the FRX record pointer to a custom property, and optionally positions the FRX if a record number was specified.

```
lparameters tnFRXRecno
This.nDataSession = set('DATASESSION')
set datasession to This.FRXDataSession
This.nFRXRecno = recno()
if pcount() = 1
  go tnFRXRecno
endif pcount() = 1
```

UnselectFRX restores the FRX record pointer and switches back to the listener datasession.

```
go This.nFRXRecno
set datasession to This.nDataSession
```

GetReportObject uses these two methods to return an object for the specified record in the FRX. This makes it easier to examine information about any FRX object.

```
lparameters tnFRXRecno
local loObject
This.SelectFRX(tnFRXRecno)
scatter memo name loObject
This.UnSelectFRX()
return loObject
```

Because events like Render and EvaluateContents fire not just once for every record in the FRX but for every object that gets rendered (that is, they fire the number of records in FRX times the number of records in the data set being reported on), you want to minimize the amount of work done in these methods. For example, if you store a directive in the User memo that tells a listener how to process a report object, any code that parses the User memo would be called many times, even though it's really only needed once. So, I added a custom array property called aRecords that can contain any information about the records in the FRX you need. For example, the first time an object is rendered, you could check to see if there's an entry in aRecords for the object. If not, you do whatever is necessary (such as parsing User) and store some information into aRecords. Otherwise, you simply retrieve the information from aRecords.

To support this concept, the BeforeReport event dimensions aRecords to as many records as there are in the FRX so we don't have to redimension it as the report runs.

```
with This

* Dimension aRecords to as many records as there are in the FRX so we don't
* have to redimension it as the report runs.

  .SelectFRX()
  if alen(.aRecords, 2) > 0
    dimension .aRecords[reccount(), alen(.aRecords, 2)]
  else
    dimension .aRecords[reccount()]
  endif alen(.aRecords, 2) > 0
  .UnSelectFRX()

* Handle successor chains.

  if vartype(.Successor) = 'O'
    .Successor.CurrentDataSession =.CurrentDataSession
    .Successor.BeforeReport()
  endif vartype(.Successor) = 'O'
endwith
```

Because OutputPage gets called with a particular page number, and a successor won't necessarily know what page that is, this method stores the passed page number to a custom nOutputPageNo property, which other listeners can use, and then calls the successor's OutputPage method. The SFReportListenerProgressMeter class, for example, uses this so it can display what page is being output in a progress meter.

SFReportListenerDirective

SFReportListenerDirective is a subclass of SFReportListener. Its purpose is to support directives in the User memo that tell the listener how to process a report object. An example of a directive might be *:LISTENER ROTATE = -45, which tells the listener to rotate this object 45 degrees counter-clockwise. Because User might be used for a variety of purposes, directives supported by SFReportListenerDirective must start with *:LISTENER (those of you who used GENSCRNX in the FoxPro 2.x days will recognize this type of directive).

Different directives are handled by different objects. They don't necessarily have to be subclasses of ReportListener (some of the examples we'll see later are based on Custom) if they simply change properties of the object being rendered. Because you may use multiple directives for the same object, SFReportListenerDirective maintains a collection of directive handlers and calls the appropriate one as necessary.

The Init method creates the collection of directive handlers and fills it with several commonly-used handlers. Additional handlers can be added in a subclass or after this class has been instantiated by adding to the collection (note that the keyword used for the collection must be upper-cased).

```
with This
  .oDirectiveHandlers = createobject('Collection')
  loHandler = newobject('SFDynamicForeColorDirective', 'SFReportListener.vcx')
  .oDirectiveHandlers.Add(loHandler, 'FORECOLOR')
  loHandler = newobject('SFDynamicBackColorDirective', 'SFReportListener.vcx')
  .oDirectiveHandlers.Add(loHandler, 'BACKCOLOR')
  loHandler = newobject('SFDynamicStyleDirective', 'SFReportListener.vcx')
  .oDirectiveHandlers.Add(loHandler, 'STYLE')
  loHandler = newobject('SFDynamicAlphaDirective', 'SFReportListener.vcx')
  .oDirectiveHandlers.Add(loHandler, 'ALPHA')

* Dimension aRecords to the number of columns we need.

  dimension .aRecords[1, 2]
endwith
```

The EvaluateContents method checks to see if we've already checked the current report object for directives; in that case, the second column of the appropriate row in aRecords will be .T. If not, we call UpdateListenerDirectives to see if there are any directives for the report object and update aRecords accordingly. It then checks to see if the first column in aRecords contains a collection of directives for this report object (since you may have more than one directive for a given object). If so, each element in the collection contains the name of a directive handler object in the oDirectiveHandlers collection and the directive argument (for example, if the directive is *:LISTENER ROTATE = -45, this argument would be "-45"). We call the HandleDirective method of each handler we're supposed to, passing it the properties object passed in to us and the directive argument.

```
lparameters tnFRXRecno, ;
  toObjProperties
local loDirective, ;
```

```
  loHandler
with This

* For performance reasons, we want to minimize the amount of work we do in
* this method, so check our collection of records to see if we've already
* processed this one. If not, call UpdateListenerDirectives to update the
* collection.

  if not .aRecords[tnFRXRecno, 2]
    .UpdateListenerDirectives(tnFRXRecno)
  endif not .aRecords[tnFRXRecno, 2]

* If we have any listener directives for this record, call each one.

  if vartype(.aRecords[tnFRXRecno, 1]) = 'O'
    for each loDirective in .aRecords[tnFRXRecno, 1]
      loHandler = .oDirectiveHandlers.Item(loDirective.DirectiveHandler)
      loHandler.HandleDirective(This, loDirective.Expression, ;
        toObjProperties)
    next loDirective
  endif vartype(.aRecords[tnFRXRecno, 1]) = 'O'
endwith
```

UpdateListenerDirectives calls the GetReportObject method (defined in the parent class) to find the record for the report object in the FRX and return a SCATTER NAME object containing properties for each field in the FRX record. It then parses the User memo of the report object, looking for *:LISTENER directives. Any it finds are checked for validity by seeing if a handler for it exists in the oDirectiveHandlers collection, and is so, the directive is added to a collection object stored in the first column of the aRecords row for the report object. The second column of the aRecords row is set to .T. whether any directives found or not to indicate that this report object has been processed and doesn't need to be done again.

```
lparameters tnFRXRecno
local loFRX, ;
  laLines[1], ;
  lnLines, ;
  lnI, ;
  lcLine, ;
  lnPos, ;
  lcClause, ;
  lcExpr
  loHandler, ;
  loDirective
with This

* Flag that we've processed this record.

  .aRecords[tnFRXRecno, 2] = .T.

* Get the specified FRX record and process any lines in the User memo.

  loFRX   = .GetReportObject(tnFRXRecno)
  lnLines = alines(laLines, loFRX.User)
  for lnI = 1 to lnLines
    lcLine = alltrim(laLines[lnI])

* If we found a listener directive and it's one we support, add it and the
* specified expression to our collection (create the collection the first time
```

```
* it's needed).

    if upper(left(lcLine, 10)) = '*:LISTENER'
      lcLine   = substr(lcLine, 12)
      lnPos    = at('=', lcLine)
      lcClause = alltrim(left(lcLine, lnPos - 1))
      lcExpr   = alltrim(substr(lcLine, lnPos + 1))
      lnPos    = ascan(.aDirectiveHandlers, lcClause, -1, -1, 1, 15)
      try
        loHandler   = .oDirectiveHandlers.Item(upper(lcClause))
        lcExpr      = loHandler.ProcessExpression(lcExpr)
        loDirective = createobject('Empty')
        addproperty(loDirective, 'DirectiveHandler', lcClause)
        addproperty(loDirective, 'Expression',       lcExpr)

* Create a collection of all directives this record has.

        if vartype(.aRecords[tnFRXRecno, 1]) <> 'O'
          .aRecords[tnFRXRecno, 1] = createobject('Collection')
        endif vartype(.aRecords[tnFRXRecno, 1]) <> 'O'
        .aRecords[tnFRXRecno, 1].Add(loDirective)
      catch
      endtry
    endif upper(left(lcLine, 10)) = '*:LISTENER'
  next lnI
endwith
```

Directive Handlers

SFReportDirective is an abstract class that directive handlers can be subclassed from. It's a subclass of SFCustom, my Custom base class defined in SFCtrls.VCX, and just has two abstract methods: HandleDirective, which is called from the EvaluateContents method of SFReportListenerDirective to handle the directive, and ProcessExpression, which is called from the UpdateListenerDirectives method of SFReportListenerDirective to convert the directive argument from text into a format the handler can use.

SFDynamicStyleDirective is a directive handler that allow you to change the font style (that is, whether it's normal, bold, italics, or underlined) for a report object based on an expression that's dynamically evaluated for every record in the report's data set. Specify the directive in the User memo of a report object using the following syntax:

```
*:LISTENER STYLE = StyleExpression
```

where StyleExpression is an expression that evaluates to the desired style.

One complication: styles are stored in an FRX as numeric values, so to make it easier to specify styles, SFDynamicStyleDirective allows you to use #NORMAL#, #BOLD#, #ITALIC#, and #UNDERLINE# to specify the style. These values are additive, so #BOLD# + #ITALIC# would give bold italicized text. The ProcessExpression method takes care of converting the style text into the appropriate numeric values (the cnSTYLE_* constants in this code are defined in SFReporting.H).

```
lparameters tcExpression
```

```
local lcExpression
lcExpression = strtran(tcExpression, '#NORMAL#',    transform(cnSTYLE_NORMAL))
lcExpression = strtran(lcExpression, '#BOLD#',      transform(cnSTYLE_BOLD))
lcExpression = strtran(lcExpression, '#ITALIC#',    transform(cnSTYLE_ITALIC))
lcExpression = strtran(lcExpression, '#UNDERLINE#', ;
  transform(cnSTYLE_UNDERLINE))
return lcExpression
```

Here's an example of a directive (taken from the SHIPVIA field in TestDynamicFormatting.FRX) that displays a report object in bold under some conditions and normal under others:

```
*:LISTENER STYLE = iif(SHIPVIA = 3, #BOLD#, #NORMAL#)
```

The HandleDirective method evaluates the expression. If the expression was valid, it sets the FontStyle property of the properties object to the dynamic style and sets Reload to .T. so the report engine knows the report object has changed.

```
lparameters toListener, ;
  tcExpression, ;
  toObjProperties
local lnStyle
lnStyle = evaluate(tcExpression)
if vartype(lnStyle) = 'N'
  toObjProperties.FontStyle = lnStyle
  toObjProperties.Reload    = .T.
endif vartype(lnStyle) = 'N'
```

SFDynamicAlphaDirective is very similar to SFDynamicStyleDirective, but it sets the PenAlpha property of the report object to the specified value. Specify the directive using the following syntax:

```
*:LISTENER ALPHA = AlphaExpression
```

SFDynamicColorDirective is also very similar to SFDynamicStyleDirective, but it deals with the color of the report object instead of its font style. As with styles, colors must be specified as RGB values, so SFDynamicColorDirective supports colors to be specified as text, such as #RED#, #BLUE#, and #YELLOW#. Specify the directive using the following syntax:

```
*:LISTENER FORECOLOR = ColorExpression
```

where ColorExpression is an expression that evaluates to the desired color.

The code in the HandleDirective method is similar to that in SFDynamicStyleDirective, but it calls SetColor rather than setting the FontStyle property. SetColor is abstract in this method; it's implemented in two subclasses of SFDynamicColorDirective: SFDynamicBackColorDirective and SFDynamicForeColorDirective. Here's the code from SFDynamicBackColorDirective to show how the color is set:

```
lparameters toObjProperties, ;
  tnColor
with toObjProperties
  .FillRed   = bitand(tnColor, 255)
```

```
  .FillGreen = bitrshift(bitand(tnColor, 0x00FF00), 8)
  .FillBlue  = bitrshift(bitand(tnColor, 0xFF0000), 16)
endwith
```

The code in the ProcessExpression method of SFDynamicColorDirective is also very similar to that in SFDynamicStyleDirective; it converts the color text into the appropriate RGB values.

TestDynamicFormatting.FRX shows how these two directive handlers (and SFReportListenerDirective) work. It prints records from the sample Northwind Orders table that comes with VFP. The SHIPPEDDATE field has the following in User:

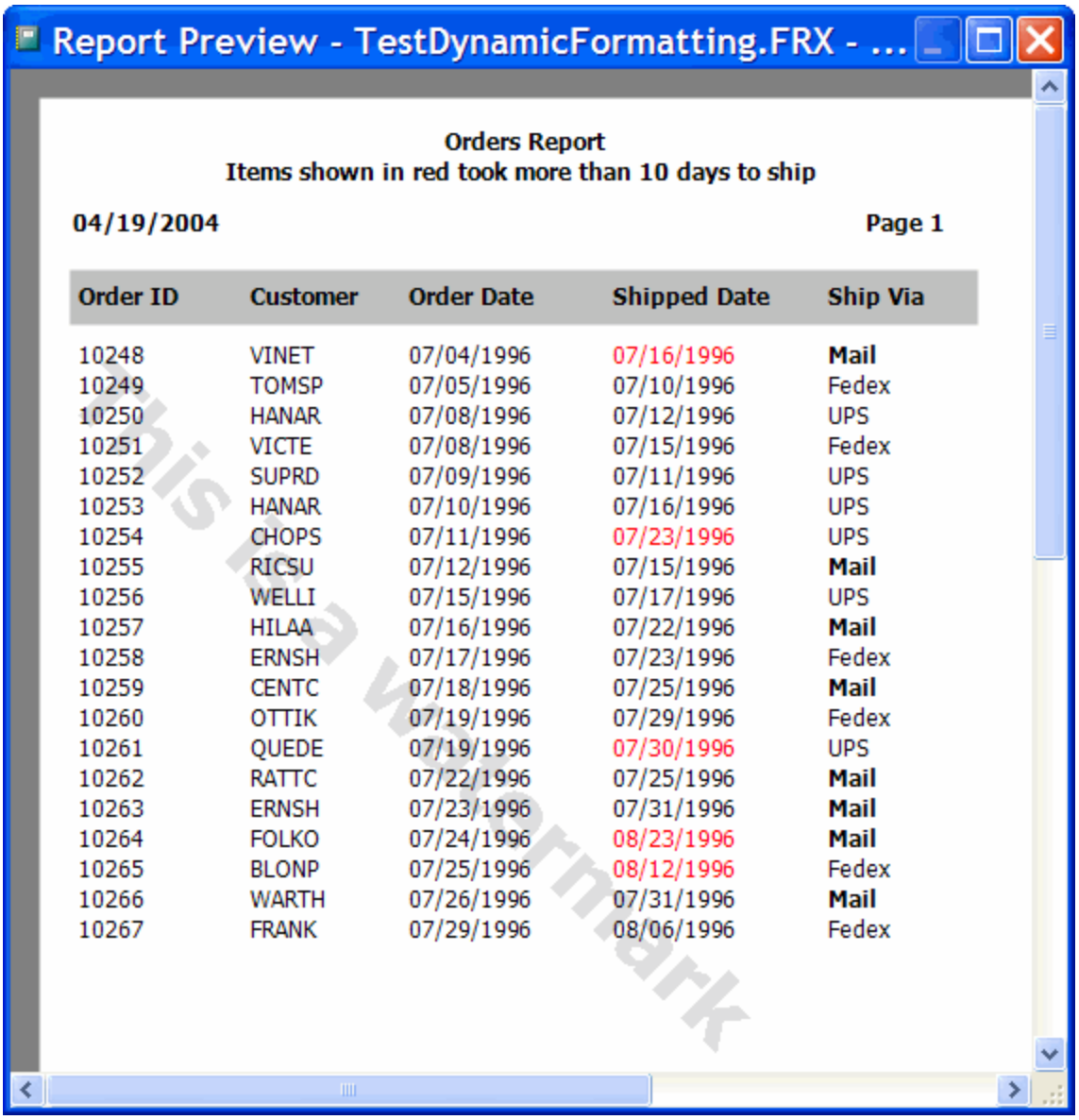
```
*:LISTENER FORECOLOR = iif(SHIPPEDDATE > ORDERDATE + 10, #RED#, #BLACK#)
```

This tells the listener to display this field in red if the date the item was shipped is more than 10 days after it was ordered or black if not. The SHIPVIA field displays in bold if the shipping method is 3 or normal if not, as we saw earlier when we discussed SFDynamicStyleDirective. The expression for the SHIPVIA field shows the use of another new feature in VFP 9: the ICASE() function. This function is like IIF() except it's an immediate CASE rather than an immediate IF. This expression displays "Fedex" if SHIPVIA is 1, "UPS" if it's 2, and "Mail" if it's 3.

```
icase(SHIPVIA = 1, 'Fedex', SHIPVIA = 2, 'UPS', SHIPVIA = 3, 'Mail')
```

The following code (taken from TestDynamicFormatting.PRG) shows how to run this report with SFReportListenerDirective as its listener. This code also uses SFRotateDirective, which we'll look at in a moment, and shows how multiple listeners can be used for a report by chaining them.

```
use _samples + 'Northwind\orders'
loListener = newobject('SFRotateDirective', 'SFReportListener')
loListener.Successor = newobject('SFReportListenerDirective', ;
  'SFReportListener')
report form TestDynamicFormatting.FRX preview object loListener next 20
```



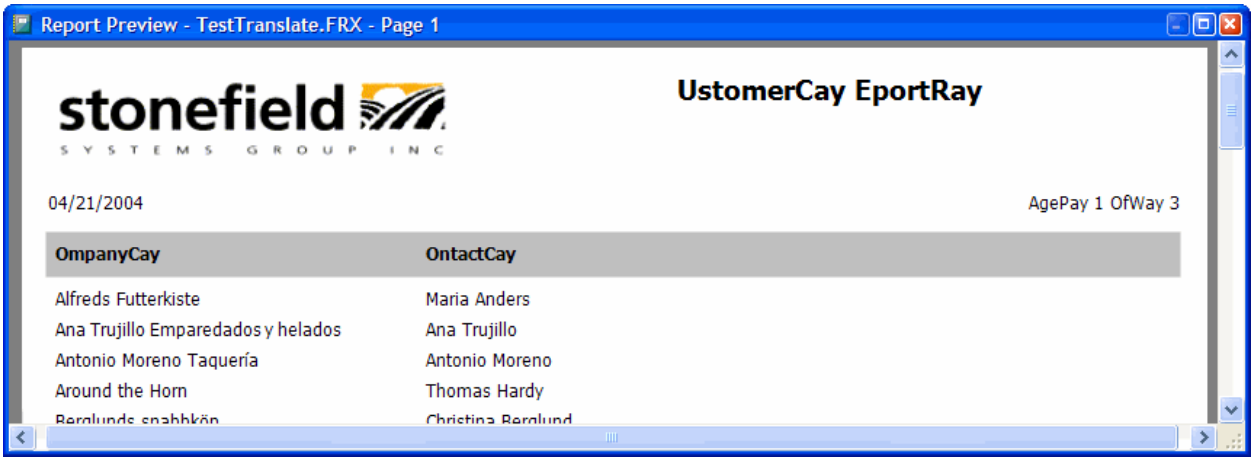
SFTranslateDirective allows you to create multi-lingual reports by defining that certain fields should be translated. Its HandleDirective method opens a STRINGS table that contains each string in its own record and columns for each language. It assumes a global variable called gcLanguage contains the language to use for the report; you could of course substitute any other mechanism you wish. It looks up the text in the current field in STRINGS and finds the appropriate translation from the column for the desired language. If the text is different, it's written to the Text property of the properties object and Reload is set to .T. so the report engine will use the new string.

```
lparameters toListener, ;
  tcExpression, ;
  toObjProperties
local lcText, ;
  lcNewText, ;
  lnI, ;
```

```
  lcWord

* Open the strings table if necessary.

if not used('STRINGS')
  use STRINGS again shared in 0
endif not used('STRINGS')

* Try to find the current text for the object and replace it with the text in
* the desired language (stored in the global variable gcLanguage).

store toObjProperties.Text to lcText, lcNewText
for lnI = 1 to getwordcount(lcText)
  lcWord = getwordnum(lcText, lnI)
  if seek(upper(lcWord), 'STRINGS', 'ENGLISH')
    lcNewText = strtran(lcNewText, lcWord, trim(evaluate('STRINGS.' + ;
      gcLanguage)))
  endif seek(upper(lcWord), 'STRINGS', 'ENGLISH')
next lnI
if not lcNewText == toObjProperties.Text
  toObjProperties.Text   = lcNewText
  toObjProperties.Reload = .T.
endif not lcNewText == toObjProperties.Text
```

To use this listener, simply place *:LISTENER TRANSLATE into the User memo of any field objects you want translated and set gcLanguage to the desired language. Note that since EvaluateContents is only called for field objects, you have to use them instead of label objects, even for text that doesn't change. TestTranslate.PRG shows how to add SFTranslateDirective to the collection of directive handlers recognized by SFReportListenerDirective. This sample uses Pig Latin because I couldn't remember enough of my high school French to create any meaningful messages <g>.

```
use _samples + 'Northwind\customers'
loListener = newobject('SFReportListenerDirective', 'SFReportListener')
loHandler  = newobject('SFTranslateDirective',      'SFReportListener.vcx')
loListener.oDirectiveHandlers.Add(loHandler, 'TRANSLATE')
gcLanguage = 'PigLatin'
report form TestTranslate.FRX preview object loListener
```



SFRotateDirective is another directive handler, except it's based on SFReportListener rather than SFReportDirective because it doesn't just simply change the properties of the report object via the properties object. Instead, it overrides the Render method to rotate the report object.

To specify that a report object should be rotate, put a directive in the User memo using the following syntax:

```
*:LISTENER ROTATE = Angle
```

where Angle is the angle to rotate (clockwise angles are specified as positive values, counter-clockwise angles as negative). This doesn't support dynamic expressions because I couldn't see a use for that, but a simple change in the SFRotateDirective class would add that ability.

The Init method declares some GDI+ functions needed by Render and dimensions aRecords to 2 columns (the BeforeReport code in SFReportListener will then redimension aRecords to that many columns and as many rows as there are records in the FRX).

```
declare integer GdipRotateWorldTransform in GDIPlus.DLL ;
  integer graphics, single angle, integer enumMatrixOrder_order
declare integer GdipSaveGraphics in GDIPlus.DLL ;
  integer graphics, integer @state
declare integer GdipRestoreGraphics in GDIPlus.DLL ;
  integer graphics, integer state
declare integer GdipTranslateWorldTransform in GDIPlus.DLL ;
  integer graphics, single dx, single dy, integer enumMatrixOrder_order

* Dimension aRecords to the number of columns we need.

dimension This.aRecords[1, 2]
```

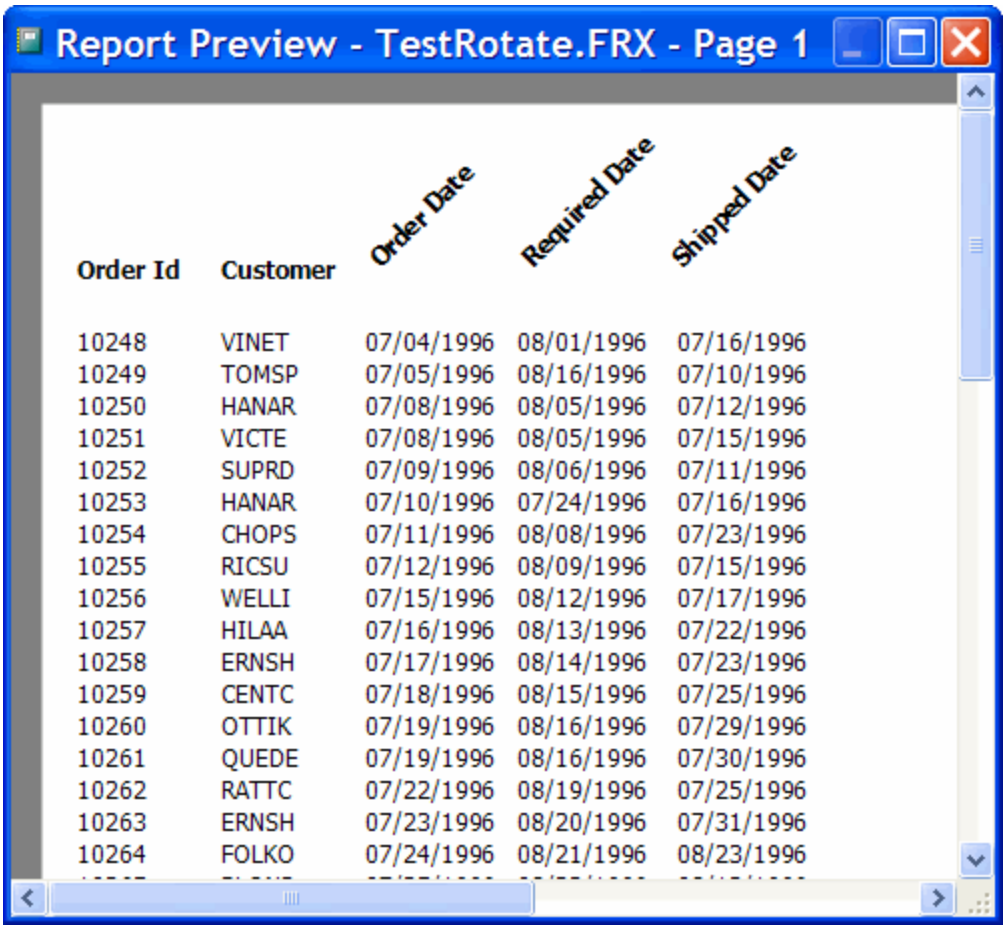
The Render method starts by checking if we've looked for directives in this report object before or not. If not, it calls the HandleListenerDirectives to do that (we won't look at the code in that method; it's similar to, albeit simpler, than that in SFReportDirective). If a rotation angle was specified, Render uses GDI+ methods to change the drawing angle for the object, uses DODEFAULT() to do the rendering, restores the GDI+ settings, and issues NODEFAULT so the rendering isn't done a second time when the method exits.

```
lparameters tnFRXRecno, ;
  tnLeft, ;
  tnTop, ;
  tnWidth, ;
  tnHeight, ;
  tnObjectContinuationType, ;
  tcContentsToBeRendered, ;
  tnGDIPlusImage
local llHandled, ;
  lnAngle, ;
  lnState
with This

* If we haven't determined if this record has a rotate directive or not, do
so.

  if not .aRecords[tnFRXRecno, 2]
```

```
    .UpdateListenerDirectives(tnFRXRecno)
  endif not .aRecords[tnFRXRecno, 2]

* If we're supposed to rotate this object, do so.

  llHandled = .F.
  lnAngle   = .aRecords[tnFRXRecno, 1]
  if lnAngle <> 0

* Save the current state of the graphics handle.

    lnState = 0
    GdipSaveGraphics(.GDIPlusGraphics, @lnState)

* Move the 0,0 point to where we'd like it to be so when we rotate, we're
* rotating around the appropriate point.

    GdipTranslateWorldTransform(.GDIPlusGraphics, tnLeft, tnTop, 0)

* Change the angle at which the draw will occur.

    GdipRotateWorldTransform(.GDIPlusGraphics, lnAngle, 0)

* Restore the 0,0 point.

    GdipTranslateWorldTransform(.GDIPlusGraphics, -tnLeft, -tnTop, 0)

* Explicitly call the base class behavior to do the drawing.

    dodefault(tnFRXRecNo, tnLeft, tnTop, tnWidth, tnHeight, ;
      tnObjectContinuationType, teContentsToBeRendered, tnFirstLine, ;
      tnLastLine, toImage)

* Put back the state of the graphics handle.

    GdipRestoreGraphics(.GDIPlusGraphics, lnState)

* We've already done the drawing, so don't let it happen again.

    nodefault

* Flag that we've handled rendering.

    llHandled = .T.
  endif lnAngle <> 0
endwith
return llHandled
```

We saw the use of the this in the TestDynamicFormatting example. TestRotate.FRX is another sample report that show how this works. The column headings for the date fields have rotate directives so the date fields can be placed closer together. The following code (taken from TestRotate.PRG) shows how to run this report with SFRotateDirective as its listener:

```
use _samples + 'Northwind\orders'
loListener = newobject('SFRotateDirective', 'SFReportListener')
report form TestRotate.FRX preview object loListener next 20
```



SFReportProgressMeter

If you want progress information during the run of a report, SFReportProgressMeter will do the trick. Its Init method instantiates a progress form class. BeforeReport sets a custom lRendering property to .T. and AfterReport sets it to .F. so we can tell the difference between the rendering and output phases of a report run. Because either one could be used for status updates, both DoStatus and UpdateStatus call the custom UpdateProgressMeter method. This method displays different messages (using the custom cProgressMeter* properties) and updates the meter differently based on whether lRendering is .T. or not:

```
lparameters tcMessage
local lcMessage
with This

* If we're rendering pages, get the current record number in the data set. If
* this is the first time we've been called, get the total number of records to
* process and set the title of the progress meter.

  if .lRendering
    if .nProgress = 0
      .oThermometer.SetMaximum(.CommandClauses.RecordTotal)
      .oThermometer.SetTitle(.cProgressMeterRenderingTitle)
    endif .nProgress = 0
```

```
    .nProgress = recno()

* If we're not rendering, we're outputting pages. Get the current page number.
* If this is the first time we've been called, get the total number of pages
* and set the title of the progress meter.

  else
    if .nProgress = 0
      .oThermometer.SetMaximum(.OutputPageCount)
      .oThermometer.SetTitle(.cProgressMeterOutputtingTitle)
    endif .nProgress = 0
    .nProgress = .nOutputPageNo
  endif .lRendering

* Get the message to display if one wasn't passed.

  do case
    case vartype(tcMessage) = 'C'
      lcMessage = tcMessage
    case .lRendering
      lcMessage = textmerge(.cProgressMeterRenderingMessage)
    otherwise
      lcMessage = textmerge(.cProgressMeterOutputtingMessage)
  endcase

* Update the thermometer.

  .oThermometer.Update(.nProgress, lcMessage)
endwith
```

TestGraphicOutput.PRG, which we'll look at in the next section, shows how to use this listener class.

SFReportListenerGraphic

The OutputPage method of ReportListener supports outputting pages to graphics files. To make this easier, I created SFReportListenerGraphic as a subclass of SFReportListener. It has two custom properties: cFileName, which should be set to the name of the file to create, and nFileType, which can either be set to the number representing the file type or left at 0, in which case SFReportListenerGraphic will set it to the proper value based on the extension of the filename in cFileName.

If ListenerType is 2 (the default for this class), OutputPage is called after each page is rendered. In that case, OutputPage will handle outputting to the specified file. If ListenerType is 3, pages are only output when OutputPage is specifically called, so AfterReport goes through the rendered pages and calls OutputPage for each one. Notice that if a multi-page TIFF file is specified, the first page must be output as a single-page TIFF file, and then subsequent pages will be appended to it by outputting them as a multi-page TIFF file. Here's the code from AfterReport; OutputPage is similar but slightly simpler. In this code, LISTENER_DEVICE_TYPE_* are constants defined in SFReporting.H.

```
local lcBaseName, ;
  lcExt, ;
  lnI, ;
```

```
  lcFileName
with This
  dodefault()
  if .ListenerType = 3
    if .nFileType = 0
      .GetGraphicsType()
    endif .nFileType = 0
    lcBaseName = addbs(justpath(.cFileName)) + juststem(.cFileName)
    lcExt      = justext(.cFileName)
    for lnI = 1 to .OutputPageCount
      do case
        case .nFileType <> LISTENER_DEVICE_TYPE_MULTI_PAGE_TIFF
          lcFileName = forceext(lcBaseName + padl(lnI, 3, '0'), lcExt)
          .OutputPage(lnI, lcFileName, .nFileType)
        case not file(.cFileName)
          .OutputPage(lnI, .cFileName, LISTENER_DEVICE_TYPE_TIFF)
        otherwise
          .OutputPage(lnI, .cFileName, .nFileType)
      endcase
      .DoStatus('Page ' + transform(lnI) + ' of ' + ;
        transform(.OutputPageCount))
    next lnI
    .ClearStatus()
  endif .ListenerType = 3
endwith
```

TestGraphicOutput.PRG shows how SFReportListenerGraphic works. It combines the effects of multiple listeners to render the report properly (this uses the same TestDynamicFormatting.FRX we saw earlier), display a progress meter, and output to graphics files.

```
use _samples + 'Northwind\orders'
loListener = newobject('SFReportListenerGraphic', 'SFReportListener')
loListener.cFileName = fullpath('TestReport.gif')
loListener.Successor = newobject('SFRotateDirective', 'SFReportListener')
loListener.Successor.Successor = newobject('SFReportListenerDirective', ;
  'SFReportListener')
loListener.Successor.Successor.Successor = ;
  newobject('SFReportListenerProgressMeter', 'SFReportListener')
report form TestDynamicFormatting.FRX object loListener range 1, 6

* Display the results.

declare integer ShellExecute in SHELL32.DLL ;
  integer nWinHandle, ;                         && handle of parent window
  string cOperation, ;                          && operation to perform
  string cFileName, ;                           && filename
  string cParameters, ;                         && parameters for executable
  string cDirectory, ;                          && default directory
  integer nShowWindow                           && window state
ShellExecute(0, 'Open', loListener.cFileName, '', '', 1)
```

SFReportListenerWaterMark

Under some conditions, you may want to add a watermark to a report. For example, if the user indicates that this report run should be a draft version rather than the final version (perhaps not all data entry has been completed), you'd want a "Draft" watermark added. In a demo version of your application, you might want to add "Demo Version" to all reports. However, you wouldn't

want to have two versions of each report, one with and one without watermarks. Instead, it'd be handy to add watermarks dynamically.

SFReportListenerWatermark does just that. The secret to adding a watermark dynamically is to save a copy of the FRX to a temporary file, add the watermark to the FRX, run the report, and then restore the copy of the FRX. The proper place to do these tasks is in LoadReport and UnloadReport. Here's the code for LoadReport: it saves the FRX and then adds the watermark:

```
local lnSelect
with This
  .RealReport = forceext(addbs(sys(2023)) + sys(2015), 'FRX')
  erase (.RealReport)
  erase (forceext(.RealReport, 'FRT'))
  copy file (.CommandClauses.File) to (.RealReport)
  copy file (forceext(.CommandClauses.File, 'FRT')) to ;
    (forceext(.RealReport, 'FRT'))
  lnSelect = select()
  select 0
  use (.CommandClauses.File) again shared alias _FRXFILE
  insert into _FRXFILE ... (rest omitted for brevity)
  use
  select (lnSelect)
  dodefault()
endwith
```

UnloadReport restores the FRX:

```
with This
  set datasession to .FRXDataSession
  set safety off
  use in FRX
  copy file (.RealReport) to (.CommandClauses.File)
  copy file (forceext(.RealReport, 'FRT')) to ;
    (forceext(.CommandClauses.File, 'FRT'))
  erase (.RealReport)
  erase (forceext(.RealReport, 'FRT'))
  dodefault()
endwith
```

Run TestWatermark.PRG to see how this works.

What About PDF?

Of course, the question you're asking now is "What about PDF output?" As of this writing, Microsoft has no plans to include PDF output with VFP 9. However, there are several ways you can get PDF output from VFP (now or in VFP 9):

- Use the Adobe or another PDF writer.
- Use a VFP-specific third-party tool that supports PDF output, such as Mind's Eye Report Engine, XFRX, or FRX2Any.

- In VFP 9, you can output to a TIFF file and then use the freeware GhostScript utility to convert it to a PDF file.

Because this was written early in the VFP 9 beta, this is still up in the air. Stay tuned, though–I can guarantee there will be some great solutions available soon.

Other Ideas

The sky's the limit in terms of what you can do with report listeners. Here some other ideas; I'm sure you can think of lots more.

- Have lines in the detail band display with alternating colors (like the old banded computer paper).
- Create a report listener that chains reports together so they can be run with one command.
- Create custom rendering objects, such as pie or bar charts.

Summary

Microsoft has done an incredible job of opening up the VFP reporting engine, both at design-time and run-time. By passing report events to Xbase ReportListener objects, they've allowed us to react to these events to do just about anything we wish, from providing custom feedback to users to providing different types of output to dynamically changing the way objects are rendered.

Note: since this document was written during the first beta phase of VFP 9, many of the details described in this document may be different in the release version. Be sure to check my Web site (www.stonefield.com) for updates to this document and the accompanying samples.

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 CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro. Doug is co-author of "What's New in Visual FoxPro 8.0", "The Hacker's Guide to Visual FoxPro 7.0", and "What's New in 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 writes 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 North America. He has been a Microsoft Most Valuable Professional (MVP) since 1996.