Making the Most of VFP 9 SP2 Reports (Part II)

Cathy Pountney
Memorial Business Systems, Inc.
cathy@frontier2000.com
www.frontier2000.com
www.mbs-intl.com
www.cathypountney.blogspot.com
Twitter: frontier2000

Part I of this 2-part series showed you how to use the new reporting features introduced in VFP 9 SP2. This session steps it up several notches by showing you how to add your own features. Not only will you learn how to implement those features on your reports, but you'll also learn how to make the features available in the Report Builder's UI. In addition, you'll see a "best practices" methodology that makes it easy for developers to share their enhancements with one another. You don't want to miss this session. Come be a part of history in the making!

The VFP 9 SP2 Enhancements

There's so much new stuff in VFP 9 SP2 that it's hard to figure out where to start. I guess the best place is to talk about the back-story with early Report Listeners and transition into the better concepts introduced with VFP 9 SP2.

Problems with early VFP 9 Report Listeners

When VFP 9 was first released, we were introduced to Report Listeners and all their power. I, like many other developers, started subclassing Report Listeners and creating all kinds of reporting features. We could even chain multiple Report Listener classes together to take advantage of multiple features on any given report. This was great and we were all excited about the endless possibilities.

However, there were issues with the new power given to us. First, we had to remember which features were used on a given report so we could remember to instantiate the required Report Listeners. Second, if we chained too many Report Listeners together, performance started to suffer exponentially.

Some developers took the approach of creating one giant Report Listener with all their custom features. This got rid of the serious performance issue with lots of Report Listeners, however, it had its own set of flaws. First, a lot of wasted cycles were being used to process code that wasn't necessary on the majority of reports. Rarely would a given report use all your custom features. Second, sharing features between developers was difficult. To use your custom features, another developer would have to dissected your Report Listener class, and reassemble the parts in his or her own Report Listener class. Of course, the other developer had to be very careful that your features don't interfere with theirs.

The VFP 9 SP2 Solution

VFP 9 SP2 introduces a wonderful solution to the use of Report Listeners for implementing custom features in reports. Customization is abstracted out of the Report Listener class and placed within custom classes. Each custom feature (a.k.a. special effect) should have its own custom class. To run a report using a custom feature, instantiate the special effect Report Listener (fxListener), register the required custom class(es), and then sit back and let the Report Listener do its job. Scattered throughout the processing of the Report Listener class, calls are made to the registered custom classes. Those custom classes are where all the work is done to implement any given feature.

The beauty of this solution is that it's extremely easy to share custom features amongst developers. If I create five different custom features, each residing in their own custom class, all I have to do is give you the class libraries holding those classes and you can take advantage of the features. You don't have to figure out how to merge my code into your Report Listener class.

The FFC

Microsoft implemented this solution without making any changes to the core EXE of Visual FoxPro. Instead, everything is done through the FFC. The FFC (FoxPro Foundation Classes)

refers to a set of class libraries that reside in the FFC subdirectory under the VFP 9 HOME() directory. These class libraries are full of classes that provide all kinds of functionality. These classes were created by Microsoft and shipped with VFP 9. You can use these classes in your application and take advantage of all the coding already done by the Microsoft Fox Team.

While the FFC has been included since VFP 6, it took on new importance with VFP 9. Several of the key reporting components were implemented via the FFC. In SP2, the importance of the FFC has taken on an even bigger role in reporting. The _ReportListener class library has a new report listener class called fxListener and a new custom class called fxAbstract. These two classes work together to provide many of the new features in SP2, such as dynamic formatting and advanced attribute properties.

The MemberData Cursor

To maintain backwards compatibility, Microsoft didn't want to modify the structure of the FRX file. Yet with all the new features introduced in SP2, the new data had to be stored somewhere in the FRX. Their solution was to introduce the "MemberData" concept to reports.

In VFP forms, a MemberData property exists to hold additional information. The information is stored as an XML string. Various components of VFP read the XML, parse it out, and do special things with the information discovered. Other components do the reverse and build an XML string and write it back to the MemberData property.

In reports, the STYLE field wasn't being used which made it the perfect candidate for the place to store an XML string packed full of information. All the necessary information about a given custom feature can be stored in the Member Data cursor within the Report Designer. The cursor is then converted to an XML string and stored in the STYLE field. When reports are run, the process is reversed to obtain all the necessary information for a given feature. It's quite an ingenious concept for storing new information without altering the FRX structure.

The _ReportListener Class Library

Several new classes have been added to the _ReportListener class library in VFP 9 SP2. **Figure 1** shows the new class hierarchy. A new special effect Report Listener called fxListener has been added and this class is instrumental to implementing many of the new features available in the Report Builder. There's also a new class called fxAbstract and several subclasses of this class which are all called by the fxListener class to assist with implementation of the new features.

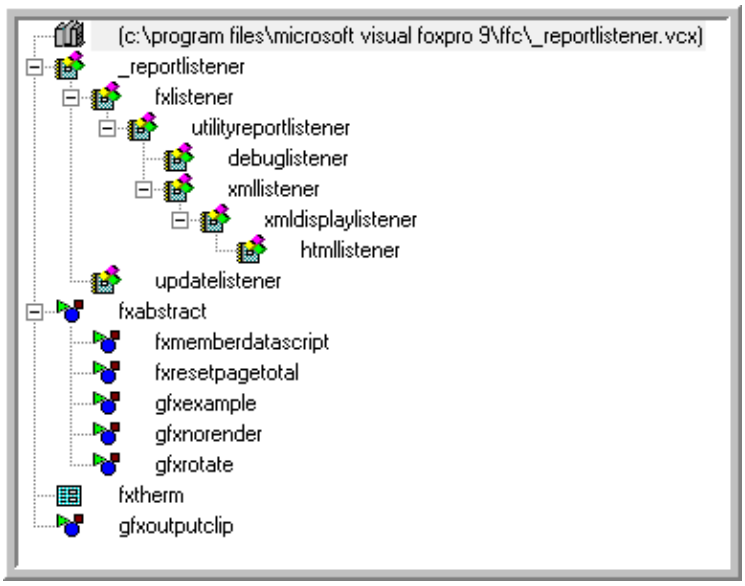


Figure 1. The _ReportListener class library contains several new classes in SP2.

The fxListener class

The fxListener Report Listener works by maintaining two collections that contain references to a number of special effects custom classes needed for running reports. Various methods in fxListener make calls to the custom classes in the collection to implement special behaviors. This means I can code my ABC feature in a custom class and you can code your XYZ feature in a different custom class. To take advantage of both features, all you have to do is tell the fxListener Report Listener to add my and your custom classes to its collections. Everything else falls into place from there.

In addition to the special effects custom classes you might tell the fxListener about, the fxListener also instantiates a few of its own that are required to implement some of the new features available via the Report Builder. For example, the dynamic properties feature and the Rotation feature are provided by special effects custom classes.

It's important to note that this new special effects concept is meant for just that, special effects. For situations where you need to create a completely different type of output, you should still create a new Report Listener to handle the new output. For example, if you're brave and have the knowledge to create PDF output straight from FoxPro, create a new Report Listener to handle this output (and then please post this Report Listener on the community VFPX site so the rest of us FoxPro developers can benefit from it).

The fxAbstract class

The fxAbstract class is the abstract class for providing special effects. This class isn't meant to be called directly, but rather subclassed for each special effect feature. It contains a method called ApplyFX.

The ApplyFX method is the method the SendFX method of the fxListener class calls. It accepts several parameters with the first being a reference to the running Report Listener.

The second parameter is a token representing the method in the Report Listener that called the SendFX method. The next twelve parameters are placeholders for passing information from the Report Listener into the ApplyFX method.

Running Reports with Custom Features

It's worth nothing that because these custom features are implemented through the fxListener class, it's mandatory you run your reports using this Report Listener or one derived from it. Running reports using any of the following direct or indirect methods causes VFP to automatically instantiate the appropriate Report Listener.

```
*----------------------------
*-- INDIRECT USE OF fxListener
*----------------------------

*-- Traditional syntax
SET REPORTBEHAVIOR 90
REPORT FORM MySpecialReport PREVIEW

*-- Using the TYPE clause
REPORT FORM MySpecialReport OBJECT TYPE 1

*-- Instantiate a Listener
LOCAL loRL
DO (_ReportOutput) WITH 1, loRL && 1 = Preview, 4 = XML, 5 = HTML
REPORT FORM MySpecialReport OBJECT m.loRL

*--------------------------
*-- DIRECT USE OF fxListener
*--------------------------

loRL = NEWOBJECT('fxListener', HOME() + '\FFC\_ReportListener')
loRL.ListenerType = 1 && Preview
REPORT FORM MySpecialReport OBJECT m.loRL

*----------------------
*-- YOUR OWN fxListener
*----------------------

loRL = NEWOBJECT('swFox_Listener', 'swFox_Listeners')
* (swFox_Listener is based on fxListener)
loRL.ListenerType = 1 && Preview
REPORT FORM MySpecialReport OBJECT m.loRL
```

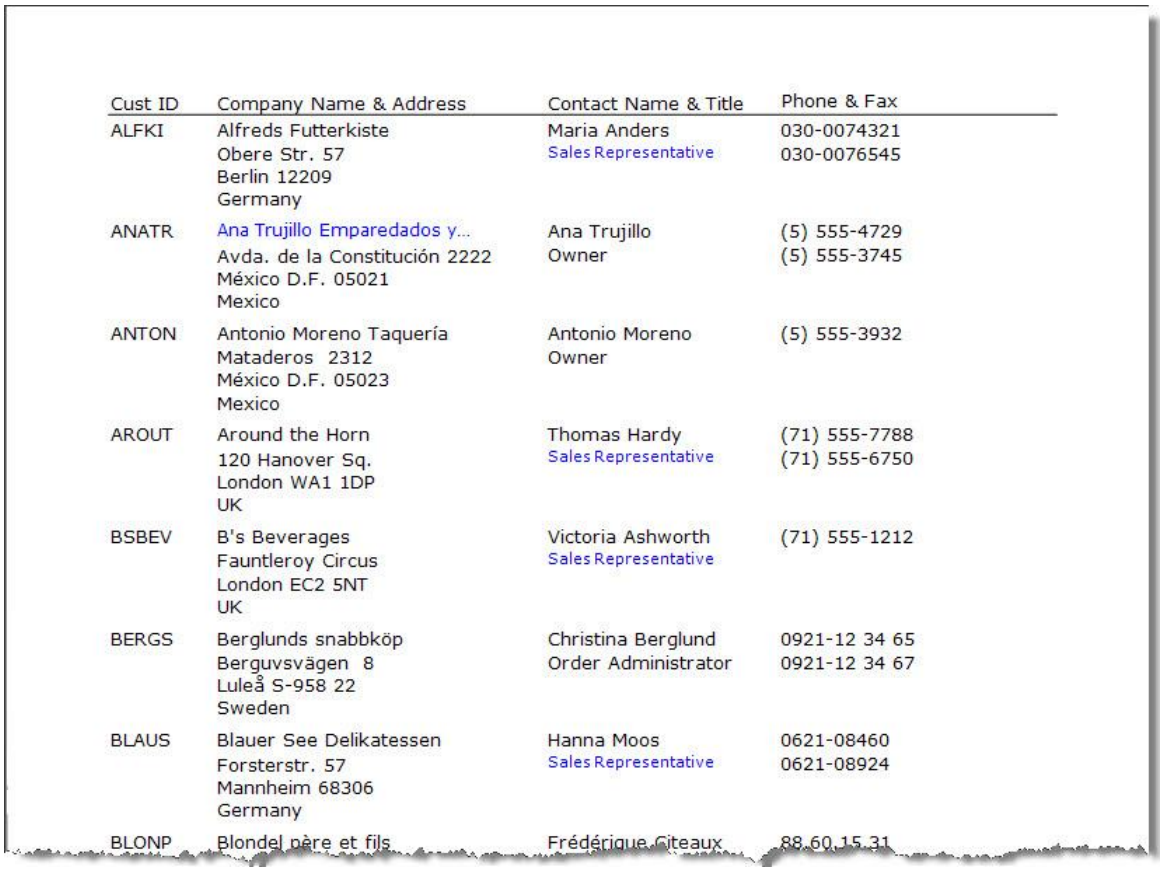
It's important to remember that if you're trying to use the new SP2 features and running through a Report Listener that was created prior to SP2, the new features probably won't work. That's because the old Report Listener isn't based on fxListener and probably isn't based on one of its derivatives either. You need to redefine your Report Listener to subclass from fxListener or one of its derivatives.

Create Your Own Custom Features

In Part I of this session, you learned about all the new attribute properties provided by Microsoft, such HTML-related properties, Render Suppression properties, and Document properties. Those are all great, but wouldn't it be nice if you could create your own attribute properties and custom features? Well, Microsoft's implementation allows just that. You can create as many attribute properties as you want and your only limitation is your creativity.

You can implement your own attribute properties by following the same concepts Microsoft has. Start by adding the new attribute properties to the desired report objects. Next, create your own classes derived from fxAbstract and add whatever code is necessary to implement the specific special effect features. Finish up by registering your classes with the running Report Listener.

The sample report shown in **Figure 2** demonstrates a custom feature I created called "Reduce Font." The sample code provided with this session contains all the classes and reports associated with this feature. This feature does three things; it reduces the font size of a string whenever the string doesn't fit within the defined width, it honors a minimum font size so as not to reduce the font too small, and it changes the font color to blue when the size has been reduced.



Cust ID	Company Name & Address	Contact Name & Title	Phone & Fax
ALFKI	Alfreds Futterkiste Obere Str. 57 Berlin 12209 Germany	Maria Anders Sales Representative	030-0074321 030-0076545
ANATR	Ana Trujillo Emparedados y... Avda. de la Constitución 2222 México D.F. 05021 Mexico	Ana Trujillo Owner	(5) 555-4729 (5) 555-3745
ANTON	Antonio Moreno Taquería Mataderos 2312 México D.F. 05023 Mexico	Antonio Moreno Owner	(5) 555-3932
AROUT	Around the Horn 120 Hanover Sq. London WA1 1DP UK	Thomas Hardy Sales Representative	(71) 555-7788 (71) 555-6750
BSBEV	B's Beverages Fauntleroy Circus London EC2 5NT UK	Victoria Ashworth Sales Representative	(71) 555-1212
BERGS	Berglunds snabbköp Berguvsvägen 8 Luleå S-958 22 Sweden	Christina Berglund Order Administrator	0921-12 34 65 0921-12 34 67
BLAUS	Blauer See Delikatessen Forsterstr. 57 Mannheim 68306 Germany	Hanna Moos Sales Representative	0621-08460 0621-08924
BLONP	Blondel père et fils	Frédérique Citeaux	88.60.15.31

Figure 2. Create some new attribute properties to reduce the font size of large strings.

Add the attribute properties to the report

To start creating this feature, select the Add… button on the Advanced tab of the desired report object. Set the Type to "Yes or No", enter ReduceFont.Enabled as the Property Name, and leave the Yes option checked. Then hit the Add button to save your new entry. This is the new attribute property that controls whether or not the Reduce Font feature should be used on this object.

Optionally, if you'd like to set a minimum font size, add another attribute property. Leave the Type as the default of Expression, enter "ReduceFont.MinSize" as the Name, and enter your desired minimum font size as the Value.

Just in case you're curious, here's what happens behind the scenes. A new record is added to the MemberData cursor containing information about the new attribute properties. That MemberData cursor is then converted to XML and saved in the Style field of the FRX. The following is an example of what is saved for the two new attribute properties added to the report.

```
<VFPData>
   <reportdata name="Microsoft.VFP.Reporting.Builder.AdvancedProperty" type="R"
script="" execute="1" execwhen="ReduceFont.Enabled" class="" classlib="" declass="5"
declasslib="" penrgb="" fillrgb="" pena="" filla="" fname="" fsize="" fstyle=""/>
   <reportdata name="Microsoft.VFP.Reporting.Builder.AdvancedProperty" type="R"
script="" execute="9" execwhen="ReduceFont.MinSize" class="" classlib="" declass="1"
declasslib="" penrgb="" fillrgb="" pena="" filla="" fname="" fsize="" fstyle=""/>
</VFPData>
```

The Name attribute contains a namespace reserved by the Report Builder for Advanced Properties. The Type attribute contains an "R" indicating this is from the Report Builder and the ExecWhen attribute contains the name of the attribute properties we added (ReduceFont.Enabled and ReduceFont.MinSize). The DEClass attribute contains a "5" for the first attribute property indicating the Execute attribute contains a Yes/No value. The DEClass attribute contains a "1" for the second attribute property indicating the Execute attribute contains an expression. The Execute attribute contains the values we entered (A 1 in the first attribute property indicates Yes.)

Create the required classes

The next step is to create the special effect classes. A class library called cpfx exists in the sample code provided with this session and it contains all the classes for this feature.

The first class is called cpFxAbstract. This is a subclass of the fxAbstract class found in the FFC. Two of the methods contain code that build collections based on information from the FRX. The other two methods perform tasks that are commonly needed in implementing custom special effects. All of the special effect features shown in this session start as subclasses from this class to ensure consistency and better performance.

The fxReduceFont_AdvProp class contains the code to implement this custom feature. The ApplyFX method (see **Listing 1**) contains a CASE statement which checks to see what

method called it. If BeforeReport made the call, the code sets the CallEvaluateContents property to 2. The code then creates a collection to remember which records in the FRX use the ReduceFont.Enabled and ReduceFont.MinSize attribute properties. Maintaining this information in a collection is more efficient than switching back to the FRX for every object the report processes.

If you plan to check for calls made by the EvaluateContents or AdjustObjectSize methods, be sure to set the CallEvaluateContents and/or CallAdjustObjectSize properties to 2 in code called by the BeforeReport method. If you fail to do this, you'll be left scratching your head and wondering why your code isn't working. The fxListener class does not call the EvaluateContents or AdjustObjectSize methods by default and you have to override the properties to ensure the methods get called.

If the EvaluateContents method made the call, the code checks to see if the current object being processed is contained in the collection which means the Reduce Font feature should be applied. If found, it calls a new method called ReduceFont. The code in ReduceFont (see **Listing 2**) determines whether the string fits, and if not, it processes through a loop reducing the font size by 1 each time until the string fits (or the minimum font size is reached.)

Listing 1. ApplyFX Method of fxReduceFont_AdvProp class.

```
LPARAMETERS m.toListener, m.tcMethodToken, ;
   m.tP1, m.tP2, m.tP3, m.tP4, m.tP5, m.tP6,  m.tP7, m.tP8, ;
   m.tP9, m.tP10, m.tP11, m.tP12

DO CASE
   CASE m.tcMethodToken = 'BEFOREREPORT'

      *-- Make sure the EvaluateContents method gets fired
      m.toListener.CallEvaluateContents = 2 && Always

      *-- Create the collection objects in the Report Listener
      This.BuildCollectionFromMemberData(m.toListener, ;
         'oPropEnabled', '', 'ReduceFont.Enabled', .t.)
      This.BuildCollectionFromMemberData(m.toListener, ;
         'oPropMinSize', '', 'ReduceFont.MinSize')

   CASE m.tcMethodToken = 'EVALUATECONTENTS'

      *-- If this feature is used, apply it
      IF This.oPropEnabled.GetKey(TRANSFORM(m.tP1)) > 0 AND ;
            This.oPropEnabled.Item(TRANSFORM(m.tP1)).Execute = '1'
         This.ReduceFont(m.toListener, m.tP1, @tP2)
      ENDIF
ENDCASE

RETURN
```

Listing 2. ReduceFont method of the fxReduceFont_AdvProp class.

```
LPARAMETERS toListener, tnFRXRecno, toObjProperties
```

```
LOCAL lnSmallest, lnFontSize, lnWidth, ;
   lnMaxWidth, lcText, lnDecimals, ;
   lcStyle, lcFRXRecNo

*-- Convert the RecNo to character
lcFRXRecNo = TRANSFORM(m.tnFRXRecNo)

*-- What is the smallest font size to use (default to 4)
IF This.oPropMinSize.GetKey(m.lcFRXRecNo) > 0
   lnSmallest = VAL(This.oPropMinSize.Item(m.lcFRXRecNo).Execute)
ELSE
   lnSmallest = 4
ENDIF

*-- Prep for loop
lcStyle = This.ConvertFontStyleToCodes(m.toObjProperties.FontStyle)
lnMaxWidth = This.oFRX.Item(m.lcFRXRecno).Width && FRUs from FRX
lnFontSize = m.toObjProperties.FontSize
lnDecimals = SET("Decimals")
SET DECIMALS TO 3

*-- Add an extra character to make sure it fits. Otherwise, it can be
*-- just a tiny bit too close and doesn't reduce correctly.
lcText = RTRIM(m.toObjProperties.Text) + 'W'

*-- Change the font, if necessary
DO WHILE m.lnFontSize > m.lnSmallest

   *-- Using lnFontSize, how wide would the text be (in FRUs)?
   lnWidth = m.toListener.FRXCursor.GetFRUTextWidth(m.lcText, ;
         m.toObjProperties.FontName, m.lnFontSize, m.lcStyle )
   IF m.toListener.FRXCursor.ScreenDPI <> 96
      *-- NOTE: GetFRUTextWidth takes into account the current display DPI
      *--       but the rest of the native reportlistener assumes 96 DPI.
      *--       If the display isn't 96, we need to massage the figure
      lnWidth = m.lnWidth * m.toListener.FRXCursor.ScreenDPI / 96
   ENDIF

   *-- If the text fits with this font,
   *-- get out
   IF m.lnWidth <= m.lnMaxWidth
      EXIT
   ENDIF

   *-- Reduce the font size so we can try again
   lnFontSize = m.lnFontSize - 1

ENDDO

*-- Change the font, if needed
IF m.toObjProperties.FontSize <> m.lnFontSize
   toObjProperties.FontSize = m.lnFontSize
   toObjProperties.Reload = .T.
```

```
   *-- Just for demo purposes, change the color to make these more obvious
   *-- and to demonstrate how to change font colors.
   toObjProperties.PenRed = 0
   toObjProperties.PenGreen = 0
   toObjProperties.PenBlue = 255
ENDIF

*-- Restore decimals
SET DECIMALS TO &lnDecimals
```

Run the report

At this point you have the new Reduce Font attribute properties defined in the report and the new fxReduceFont_AdvProp class defined to handle the processing. The last step is to run the report with the fxListener class, being sure to register your fxReduceFont_AdvProp class prior to issuing the REPORT FORM command. This can be done with the following code.

```
LOCAL  loRL

*-- Instantiate the Report Listener
DO (_ReportOutput) WITH 1, loRL

*-- Add the special effect to the Report Listener collection
*-- PARAMETERS for AddCollectionMember:
*-- m.tcClass, m.tcClassLib,m.tcModule,m.tlSingleton, m.tlInGFX, m.tlRequired
loRL.AddCollectionMember('fxReduceFont_advprop', 'CPfx', '', .t., .f., .f.)

*-- Run the report
REPORT FORM 210_ReduceFont_AdvProp OBJECT m.loRL

*-- Clean up
*-- PARAMETERS for RemoveCollectionMember
*-- m.tcName, m.tlInGFX, m.tlNameIsClass
loRL.RemoveCollectionMember('fxReduceFont_advprop', .f., .t.)
```

As you can see, it's fairly simple to add and implement your own attribute properties. For each attribute you want to implement, create a class derived from fxAbstract and add the attribute property to any report objects.

Wouldn't it be nice, though, if you could make your new attribute properties show up on the Advanced tab of all report objects instead of having to add it manually each time (and remember the exact naming convention each time)? Of course it would. In the next few sections, I explain what the Report Builder's configuration table is and how it allows you to do just that.

The Report Builder's configuration table

Some of the new features in Visual FoxPro 9.0 Service Pack 2 are implemented with help from the Report Builder's configuration table. This table was originally designed as a lookup table which maps events in the Report Builder to specific event handler classes. In

Service Pack 2, its role has been expanded to assist with the new Advanced Properties feature. By default, the table is built into ReportBuilder.app which renders it read-only. However, you can create a copy of this table and tell Visual FoxPro to use your version instead of the one built into ReportBuilder.app.

At this point you're probably thinking, "Hmm ... I didn't know about this table." If that's the case, you are not alone. This table isn't very discoverable through the user interface and most developers aren't even aware of its existence. From the Report Builder invoke the Properties dialog of the report or any object on the report. Next, right-click on any unused portion of the dialog and select Options… from the right-click menu. This brings up the Report Builder Options dialog shown in **Figure 3**. Alternatively, you can also invoke the Report Builder Options dialog by entering the following in the command window (I told you it wasn't very discoverable)!

```
DO (_reportbuilder)
```

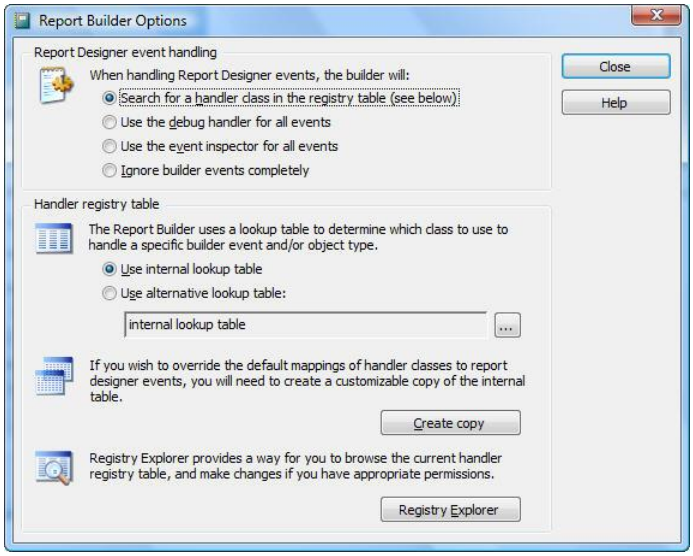


Figure 3. Access the Report Builder Options dialog from a Properties dialog or by running the Report Builder application via the command window.

The first thing you need to do is create an editable copy of this table. Select the Create copy button which invokes a standard Windows Save As dialog. After selecting the directory and filename, you're prompted as to whether you want to use this copy as the current configuration table. Answer yes to have the new copy registered as the alternative Handler Registry lookup table. The scope of this change depends on what you name the new file, where you store the file, and VFP's search path. Each time a given VFP session needs to access the Report Builder's configuration table, it does the following to locate a valid file:

- Checks to see if it has already saved the name of the file which can occur from:
    - A programmatic assignment.
    - A manual selection in the Report Builder Options dialog.

- A previous search for the Report Builder's configuration table during this VFP session which resulted in successfully finding a valid file.

- Checks the current FoxPro configuration file using SYS(2019), (default is CONFIG.FPW) and looks for an entry of REPORTBUILDER_REGISTRY.
- Looks for a table named "ReportBuilder.dbf" in the FoxPro's search path.
- Looks for a table named "ReportBuilder.dbf" in the same directory as ReportBuilder.app.
- Looks for a table named "frxBuilder.dbf" in the same directory as the class library of the class currently executing code. This is considered the "internal lookup table."

The name of the Report Builder's configuration table is stored in a member of _Screen.ReportBuilderData. To access this information, use _Screen.ReportBuilderData.Get('registry').

If you want the new Report Builder configuration table to persist as the default for new VFP sessions, accept the suggested file name of "ReportBuilder.dbf" and store it in a directory that exists in your VFP path. If you want to name the file something else or you want to store it in a path other than your search path, use the CONFIG.FPW file or a command line in your startup program to assign the file as the Report Builder's configuration file as follows.

```
*-- Configuration File (CONFIG.FPW)
REPORTBUILDER_REGISTRY = 'c:SomeDirectoryOrPath\ReportBuilder_Custom.dbf'

*-- Startup Program
DO (_ReportBuilder) WITH 3, 'c:SomeDirectoryOrPath\ReportBuilder_Custom.dbf'
```

Now that you have an editable copy, you can modify it as needed. You can, of course, simply BROWSE the table. However, a Registry Explorer utility is included in VFP, but it's buried within the Report Builder Options dialog. Invoke the Report Builder Options dialog as previously described and select the Registry Explorer button near the bottom of the dialog. This invokes the Registry Explorer as shown in **Figure 4**. Alternatively, you can enter the following into the command window.

```
DO (_REPORTBUILDER) WITH 7
```

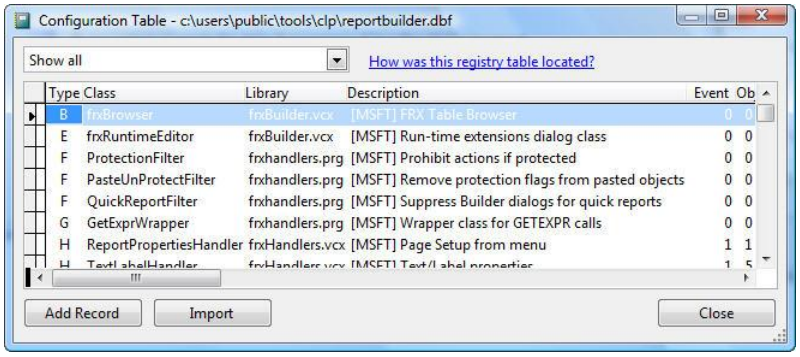


Figure 4. Use the Registry Explorer utility to modify the Report Builder Configuration table.

The Registry Explorer displays a grid of all the records in the Report Builder Configuration table. It contains the following columns:

- **Type:** The Handler Type. Click on the header of the Type column for a list of valid values.
- **Class:** Specifies the name of the class to run.
- **Library:** Specifies the class library the class resides in.
- **Description:** Specifies a description of this record. All records natively supplied by Microsoft begin with [MSFT]. You should adopt a similar naming convention and prefix your records with a meaningful identifier. This is helpful when you have records from multiple developers mixed in your Report Builder Configuration table.
- **Event:** Specifies the event type the handler class is registered to handle. Use a value of -1 to represent all events. A detailed list of all valid event types can be found in the Visual FoxPro 9.0 help file. Look in the Index for the subject "Report Builder Events." This invokes a help page titled "Understanding Report Builder Events." You can also click on the header of this column for a list of valid values.
- **ObjType:** Specifies the OBJTYPE (from the FRX) that this class handles. Use a value of -1 to represent all OBJTYPEs. A report called 90frx.frx resides in the HOME() + 'Tools\FileSpec' directory. Run this report to see a complete list of valid OBJTYPE values. You can also click on the header of this column for a list of valid values.
- **ObjCode:** Specifies the OBJCODE (from the FRX) that this class handles. Use a value of -1 to represent all OBJCODEs. Run 90frx.frx to see a complete list of valid OBJCODE values. You can also click on the header of this column for a list of valid values.
- **Native:** Indicates this event type and object type combination should be handled by the native Report Builder processing. This seems a bit strange at first to create a record that does nothing but the native behavior, but there is a valid use for this. You can create one record that has custom behavior for all OBJTYPEs, and then use this flag on a different record to override the custom behavior and revert back to the native behavior for one specific OBJTYPE value.
- **Debug:** Indicates this event type and object type combination is to use the DebugHandler class instead of the class listed in the class and class library columns. This is useful for debugging a custom handler.
- **Order:** Indicates the order in which the records are displayed on the screen. If two or more records have the same order number, they are ordered back-to-back, but there's no guarantee which one gets displayed first.

Use the combobox in the upper-left corner of the Registry Explorer to filter the list of records. Use the Add Record button to add a new record to the table. Use the Import button

to import records from another table which gives you the ability to take advantage of features designed by other developers willing to share their work.

Implement Global Attribute Properties

Now that we've discussed the Report Builder Configuration table and learned how to modify it, let's discuss how to use this table to create your own attribute properties in all your reports.

To add new attribute properties globally, invoke the Report Builder's Registry Explorer. Next, select "Show only Advanced Property definitions" from the combobox in the upper-left corner. This step isn't necessary, but it makes things a lot cleaner when you're only looking at Advanced Properties. Also notice that some of the columns disappear when you use this filter because they aren't applicable to Advanced Properties.

Select the Add Record command button and a new record is added to the bottom of the grid. Enter "P" for the Type and enter the name you want to assign to this attribute property. If you want to assign a default value, enter it in the Default Value column, being sure to remember the value is stored as a character string. You can also enter a description of this attribute property. It's not necessary, but I strongly recommend you always enter something descriptive. I also recommend you prefix all your descriptions with the same string so you can identify which records are yours and which ones came from someone else.

Next, enter the Edit Type. The valid values are:

- **1:** The attribute property accepts an expression by invoking the Expression Builder.
- **2:** The attribute property accepts a text or an XML string by invoking a text editing window.
- **3:** The attribute property accepts a string by invoking an InputBox() dialog.
- **4:** The attribute property accepts a file by invoking the Open dialog.
- **5:** The attribute property accepts a Yes or No value by toggling between the values. A Yes is stored as "1" and a No is stored as "0".

Enter the OBJTYPE to indicate which type of object this attribute property applies to. Use -1 to indicate all OBJTYPEs. Enter the OBJCODE to indicate which types of objects within the OBJTYPE this attribute property applies to. Use -1 to indicate all OBJCODEs. The last column to enter is the Order Number which indicates the order this attribute property appears on the screen.

If you want to apply an attribute property to more than one OBJTYPE or OBJCODE, but not all, you have two options. The first option is to add a new record for each OBJTYPE/OBJCODE you want to apply the attribute property to. The second option is to add a new record using -1 as the OBJTYPE/OBJCODE. Then add an additional record for each OBJTYPE/OBJCODE you do NOT want to apply the attribute property to, checking the "Native" checkbox.

Figure 5 shows the two attribute property records needed to implement the Reduce Font feature globally. These new properties now appear on every field object on every report.

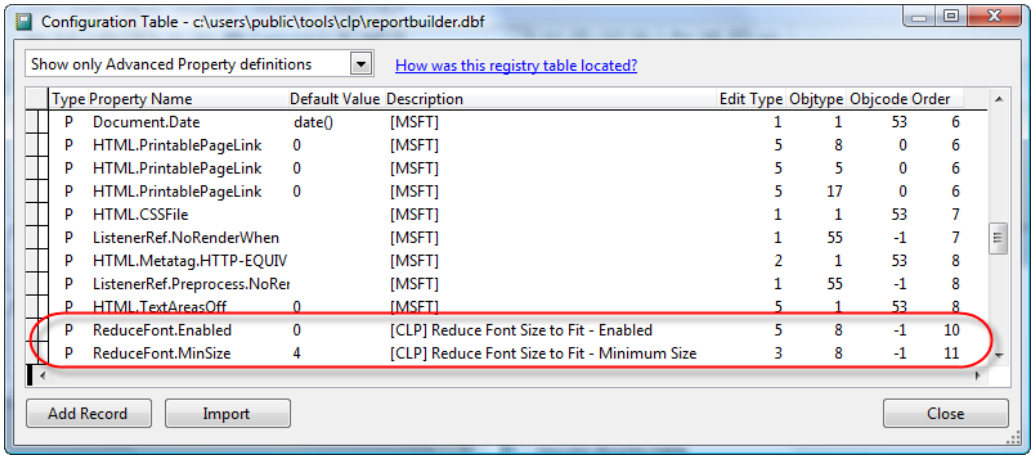


Figure 5. Add two new attribute properties to implement the Reduce Font feature.

Make Custom Features More Accessible

So far we've learned how to implement new special effect features two different ways; using individual attribute properties on report objects and using global attribute properties on all report objects. Now let's learn a third way to implement special effect features on reports. That's what I love about FoxPro … there's always more than one way to skin a Fox!

The Advanced tab, which shows a grid of attribute properties, is okay but it's certainly not the best user interface. In this section I discuss how to add your own page to the pageframe on the Properties dialog of a report object. You can use this new page to add checkboxes, textboxes, spinners, comboboxes, and anything else you want. These objects can be linked to the new special effect features you want to implement.

Creating the abstract page

The Report Builder user interface uses containers called "panels" on the Property dialogs. Various groups of objects reside on a panel. The panel has code to ensure the objects on the page are linked to pieces of data in the FRX which are refreshed and saved as needed. A page on the pageframe of a Properties dialog has one or more panels dropped onto it.

Because we're going to want to implement lots of custom special effect features in our applications, the best approach is to create an abstract panel class which all others are based on. The sample code with this session includes a class called cpfxpanel in the cpfx class library. It's based on the Container class and has a shape (to provide a visual border) and a label as shown in **Figure 6**.

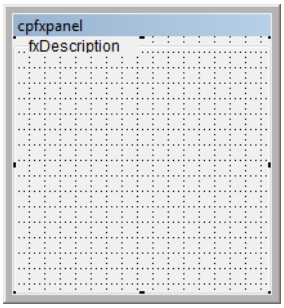


Figure 6. Add visual objects to your abstract panel class.

Next, add a property called Event. The Report Builder application looks for the existence of this property and if it exists, a reference to the event handler object is stored in the property. If the property isn't found, no harm is done, but it's nice to have this reference available if you need it.

Now add four methods to the class; LoadFromFRX, SaveToFRX, GetFromString, and AddToString. The LoadFromFRX and SaveToFRX methods don't need any code at the abstract level. The AddToString method is used to consolidate data the user has entered on a panel into a single string so it can be saved to an individual field in the MemberData cursor (see **Listing 3**). The GetFromString method does the opposite and extracts the data for display on a panel from the string (see **Listing 4**). Remember, the MemberData cursor is nothing more than a cursor built from an XML string stored in the STYLE field of the FRX.

Listing 3. AddToString method of the cpFxPanel class.

```
LPARAMETERS tcString, tcKey, txValue, tcSeparator

RETURN EVL(m.tcString, '') + EVL(m.tcSeparator, This.cSeparator) + ;
   m.tcKey + '=' + TRANSFORM(m.txValue)
```

Listing 4. GetFromString method of the cpFxPanel class.

```
LPARAMETERS tcString, tcKey, txDefault, tcSeparator

LOCAL  lcType, ;
       lcSeparator, ;
       lcReturn, ;
       lxReturn

lcType = VARTYPE(m.txDefault)
lcSeparator = EVL(m.tcSeparator, This.cSeparator)
lcReturn = STREXTRACT(EVL(m.tcString, ''), m.lcSeparator + m.tcKey + ;
   '=', m.lcSeparator, 1, 2)

DO CASE
   CASE EMPTY(m.lcReturn)
      lxReturn = m.txDefault
   CASE m.lcType = 'N'
      lxReturn = VAL(m.lcReturn)
   CASE m.lcType = 'C'
```

```
      lxReturn = m.lcReturn
   CASE m.lcType = 'D'
      lxReturn = CTOD(m.lcReturn)
   CASE m.lcType = 'L'
      lxReturn = (m.lcReturn = TRANSFORM(.t.))
ENDCASE   RETURN m.lxReturn
```

You also need to create a property called cSeparator and set it to the character(s) you want to use as the separator in the string. The sample in this session uses two vertical bars (||).

The Reduce Font feature revisited

To demonstrate implementing a custom feature through the user interface, I'll explain how to hook up the same Reduce Font feature previously discussed. This gives you a good comparison of some of the subtle differences in this approach as opposed to attribute properties on the Advanced tab.

The Panel

Start by creating the panel for the Reduce Font feature. Create a class based on your new abstract panel class. (The new class is called fxReduceFont_Panel in the sample code.) Add a checkbox to turn the feature on or off and add a label and spinner for the minimum font size as shown in **Figure 7**.

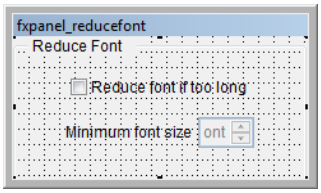


Figure 7. Add a checkbox, label, and spinner to the panel for the Reduce Font feature.

Now you need to add code for linking these controls to data in the FRX. The Report Builder calls SaveToFRX to save data to the FRX. This is where you put code that takes the values from the controls on the panel and saves them to the FRX. The code in **Listing 5** shows how to do this. If the Reduce Font feature is turned on, it makes sure a record exists in the MemberData. If the feature isn't turned on and a record exists, it deletes it. The code stores CPFX as the namespace in the Name field and REDUCEFONT as the special effect feature name in the ExecWhen field. It also uses the Execute field to store the minimum font size.

When attribute properties were added through the Advanced tab, we had to add two records to the MemberData cursor; one to turn the special effect feature on or off, and another one for the minimum font size. That's because the grid on the Advanced tab only gives us the chance to enter one value so we had to create two records to hold the information. By skipping the Advanced tab and using our own code to maintain the MemberData cursor, we can use as many of the fields as we want to store anything we want. Now we only need one record in the MemberData cursor to implement the Reduce Font feature.

Listing 5. SaveToFRX method of the fxReduceFont_Panel class.

```
*-- Data is stored in the following fields
*-- Name: CPFX
*-- ExecWhen: REDUCEFONT
*-- Execute: Minimum Font Size
SELECT MemberData

LOCAL  llReduceFont, ;
       lnMinFont
llReduceFont = This.chkReduceFont.Value
lnMinFont = This.spnMinFont.Value

LOCATE FOR Type = 'R' AND ALLTRIM(Name) == 'CPFX' AND ;
   ALLTRIM(ExecWhen) = 'REDUCEFONT'

DO CASE
   CASE FOUND() AND m.llReduceFont
      *-- Found record, update with values
      REPLACE Execute WITH TRANSFORM(m.lnMinFont) ;
         IN memberdata
   CASE FOUND()
      *-- Found but don't need, so delete it
      DELETE IN MemberData
   CASE m.llReduceFont
      *-- Didn't find and need it, so add it
      INSERT INTO memberdata (Type, Name, ExecWhen, Execute) ;
         VALUES ('R', 'CPFX', 'REDUCEFONT', ;
            TRANSFORM(m.lnMinFont))
   OTHERWISE
      *-- Didn't find it, don't need it, do nothing
ENDCASE
```

The flip side of the SaveToFRX method is the LoadFromFRX method. This method gets called by the Report Builder to load information from the FRX into the controls on the panel. The code for the Reduce Font feature is shown in **Listing** 6. It simply looks for the REDUCEFONT record in the MemberData cursor and if it finds it, sets the checkbox to checked and the minimum font spinner to whatever is saved in the Execute field. If it doesn't find the record, it sets the checkbox to unchecked and disables the minimum font spinner.

Listing 6. LoadFromFRX method of the fxReduceFont_Panel class.

```
*-- Data is stored in the following fields
*-- Name: CPFX
*-- ExecWhen: REDUCEFONT
*-- Execute: Minimum Font Size
SELECT MemberData

*-- Get the Reduce Font record
LOCATE FOR Type = 'R' AND ALLTRIM(Name) == 'CPFX' AND ;
   ALLTRIM(ExecWhen) = 'REDUCEFONT'
IF FOUND()
```

```
   This.chkReduceFont.Value = .t.
   This.spnMinFont.Value = VAL(Execute)
ELSE
   This.chkReduceFont.Value = .F.
   This.spnMinFont.Value = 4
ENDIF

*-- Enable/Disable the spnMinFont control
This.spnMinFont.Enabled = (This.chkReduceFont.Value)
```

Now you can create a page class (based on VFP's base Page class), set the caption to whatever you want, and drop the Reduce Font panel onto the new page. That's it for creating the page.

Register the new page

Of course, now that you have a new page, you have to tell the Report Builder to display the page. Invoke the Registry Explorer for the Report Builder's Configuration table and add a new record as shown in **Figure 8**. The Type is "T", the Class and Class Library corresponds to the page you created, and the Event is 1. The ObjType is 8 which indicates Field objects only and the ObjCode is -1. These settings tell the Report Builder to display this new page on Field objects only. Other objects, such as labels or pictures, won't have this new page which is what you want because the Reduce Font feature is only applicable to Field objects.

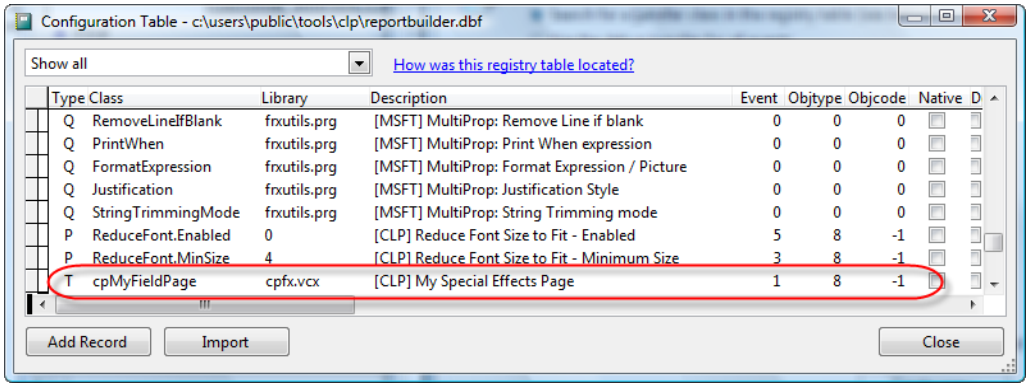


Figure 8. Add a new record to the Report Builder's Configuration table to display a new page on the pageframe of a Properties dialog.

As one would expect, if you modify or create a record that affects a Property dialog currently displayed, you have to close the Property dialog and re-open it before you see the changes. There's also some occasional quirkiness in Visual FoxPro after adding new records. If you encounter strange behavior, close Visual FoxPro and reopen it to test your newly added page.

Edit any report and bring up the Properties dialog for any Field object on the report. You should now see your new page as shown in **Figure 9**. This new page should not be visible for objects other than Field objects.

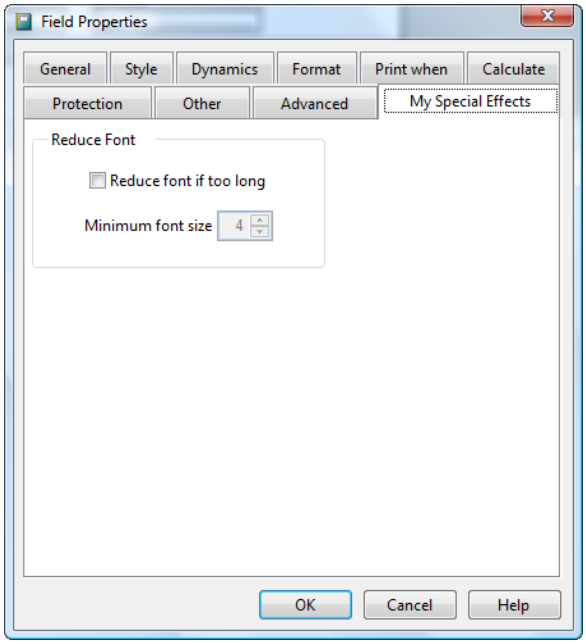


Figure 9. Implement custom special effects by adding a new page to the Properties dialog.

Create the class

Now that you have the user interface implemented for the Reduce Font feature, it's time to revisit the class for implementing the feature. Previously, we created a class called fxReduceFont_AdvProp which contains code in the ApplyFX and ReduceFont methods. Implementing the Reduce Font feature through the new user interface page is almost the same, but slightly different.

Copy the fxReduceFont_AdvProp class into a new class called fxReduceFont_UI_Version. Change the code in ApplyFX as indicated by the highlighted lines in **Listing 7** and change the code in ReduceFont as indicated by the highlighted lines in **Listing 8**. As you can see, there are very few changes required.

Listing 7. The ApplyFX method of the fxReduceFont_UI class.

```
LPARAMETERS m.toListener, m.tcMethodToken, ;
   m.tP1, m.tP2, m.tP3, m.tP4, m.tP5, m.tP6,  m.tP7, m.tP8, ;
   m.tP9, m.tP10, m.tP11, m.tP12

DO CASE
   CASE m.tcMethodToken = 'BEFOREREPORT_FX'
      m.toListener.CallEvaluateContents = 2 && Always
      This.BuildCollectionFromMemberData(m.toListener, ;
         'oFXReduceFont', 'CPFX', 'REDUCEFONT', .t.)

   CASE m.tcMethodToken = 'EVALUATECONTENTS'
      IF This.oFXReduceFont.GetKey(TRANSFORM(m.tP1)) > 0
         This.ReduceFont(m.toListener, m.tP1, @tP2)
      ENDIF
ENDCASE
```

```
RETURN
```

Listing 8. The ReduceFont method of the fxReduceFont_UI class.

```
LPARAMETERS toListener, tnFRXRecno, toObjProperties

*-- Data is stored in the following fields
*-- Name: CPFX
*-- ExecWhen: REDUCEFONT
*-- Execute: Minimum Font Size

LOCAL lnSmallest, lnFontSize, lnWidth, ;
   lnMaxWidth, lcText, lnDecimals, ;
   lcStyle, lcFRXRecNo

*-- Convert the RecNo to character
lcFRXRecNo = TRANSFORM(m.tnFRXRecNo)

*-- What is the smallest font size to use (default to 4)
IF This.oFXReduceFont.GetKey(m.lcFRXRecNo) > 0
   lnSmallest = VAL(This.oFXReduceFont.Item(m.lcFRXRecNo).Execute)
ELSE
   lnSmallest = 4
ENDIF

*-- Prep for loop
lcStyle = This.ConvertFontStyleToCodes(m.toObjProperties.FontStyle)
lnMaxWidth = This.oFRX.Item(m.lcFRXRecno).Width && FRUs from FRX
lnFontSize = m.toObjProperties.FontSize
lnDecimals = SET("Decimals")
SET DECIMALS TO 3

*-- Add an extra character to make sure it fits. Otherwise, it can be
*-- just a tiny bit too close and doesn't reduce correctly.
lcText = RTRIM(m.toObjProperties.Text) + 'W'

*-- Change the font, if necessary
DO WHILE m.lnFontSize > m.lnSmallest

   *-- Using lnFontSize, how wide would the text be (in FRUs)?
   lnWidth = m.toListener.FRXCursor.GetFRUTextWidth(m.lcText, ;
         m.toObjProperties.FontName, m.lnFontSize, m.lcStyle )
   IF m.toListener.FRXCursor.ScreenDPI <> 96
      *-- NOTE: GetFRUTextWidth takes into account the current display DPI
      *--       but the rest of the native reportlistener assumes 96 DPI.
      *--       If the display isn't 96, we need to massage the figure
      lnWidth = m.lnWidth * m.toListener.FRXCursor.ScreenDPI / 96
   ENDIF

   *-- If the text fits with this font,
   *-- get out
   IF m.lnWidth <= m.lnMaxWidth
      EXIT
```

```
   ENDIF

   *-- Reduce the font size
   lnFontSize = m.lnFontSize - 1

ENDDO

*-- Change the font, if needed
IF m.toObjProperties.FontSize <> m.lnFontSize
   toObjProperties.FontSize = m.lnFontSize
   toObjProperties.Reload = .T.
ENDIF

*-- Restore decimals
SET DECIMALS TO &lnDecimals
```

Now that you have everything in place it's just a matter of setting the new options on various report objects and running the report using the same technique previously shown.

The Watermark features

Now that you're an expert at changing the Report Builder user interface and implementing special effects, let's discuss how to implement one more special effect feature – Watermarks! In fact, we're going to implement two different kinds of watermarks: text and graphical.

Using text as a watermark

Text watermarks (see **Figure 10**) are commonly used on reports. For example, you might see "DRAFT" on a document that's not quite finished, or you might see "COPY" to indicate a copy of the original, or you might see "Confidential" on financial or medical records containing confidential information. GDIPlus can be used to accomplish all of these requirements.

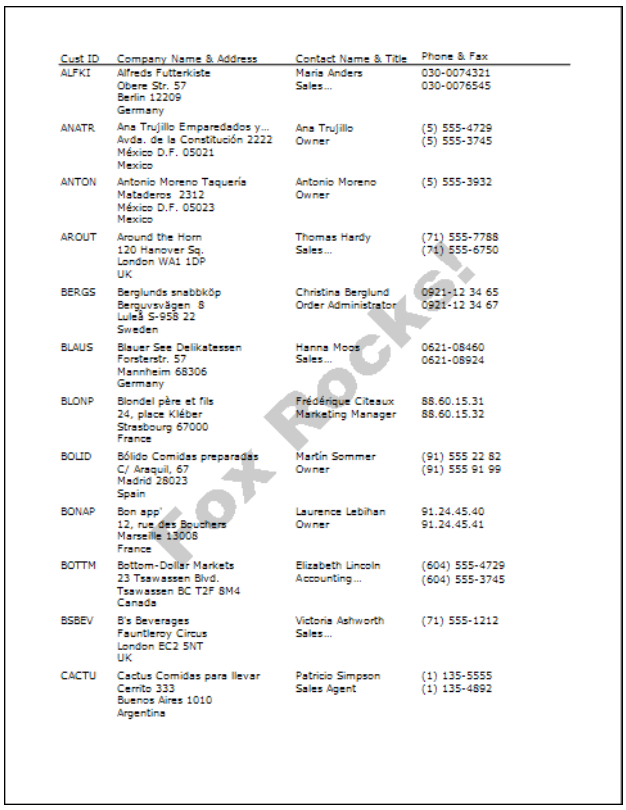


Figure 10. Use GDIPlus to add a Text Watermark to your reports.

Create the user interface for Text Watermarks

Start by creating a panel (based on your abstract panel class) with a text control for the text string to print and a spinner control for the percentage (see **Figure 11**.) The percentage is used to control how dark the text string prints; 100% is solid black.

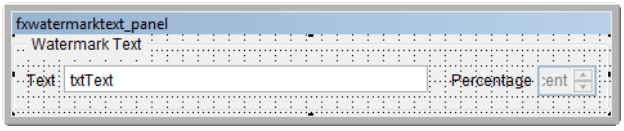


Figure 11. Create a panel with a text control and a spinner control for the Text Watermark special effect feature.

Use the ExecWhen field to store WATERMARKTEXT as the feature name, the Execute field to store the text string, and the Class field to store the percentage. The code shown in **Listing 9** and **Listing 10** saves and restores the data between the panel and the fields in the MemberData cursor.

Listing 9. The SaveToFRX method of the fxWatermarkText_Panel class.

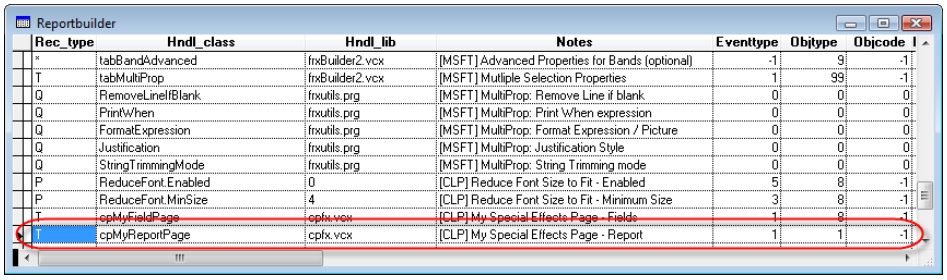
```
SELECT MemberData
LOCATE FOR Type = 'R' AND ALLTRIM(Name) == 'CPFX' AND ;
   ALLTRIM(ExecWhen) == 'WATERMARKTEXT'

DO CASE
```

```
    CASE FOUND() AND NOT EMPTY(This.txtText.Value)
       *-- Found record, update with values
       REPLACE Execute WITH This.txtText.Value, ;
              Class WITH TRANSFORM(This.spnPercent.Value) ;
          IN memberdata
    CASE FOUND()
       *-- Found but don't need, so delete it
       DELETE IN MemberData
    CASE NOT EMPTY(This.txtText.Value)
       *-- Didn't find and need it, so add it
       INSERT INTO memberdata (Type, Name, ExecWhen, ;
              Execute, Class) ;
          VALUES ('R', 'CPFX', 'WATERMARKTEXT', ;
              This.txtText.Value, TRANSFORM(This.spnPercent.Value))
    OTHERWISE
       *-- Didn't find it, don't need it, do nothing
ENDCASE
```

Listing 10. The LoadFromFRX method of the fxWatermarkText_Panel class.

```
SELECT MemberData

*-- Get the Reduce Font record
LOCATE FOR Type = 'R' AND ALLTRIM(Name) == 'CPFX' AND ;
   ALLTRIM(ExecWhen) == 'WATERMARKTEXT'
IF FOUND()
   This.txtText.Value = Execute
   This.spnPercent.Value = VAL(Class)
ELSE
   This.txtText.Value = ''
   This.spnPercent.Value = 20
ENDIF

This.spnPercent.Refresh()
```

Once you have the panel defined, create a page class and drop the panel on the page. Now you have to tell the Report Builder about this page. Invoke the Registry Explorer and add a record as shown in **Figure 12**. Set the EventType to 1, the ObjType to 1, and the ObjCode to -1. This tells the Report Builder to add this page to the Properties dialog of the Report.



Rec_type	Hndl_class	Hndl_lib	Notes	Eventtype	Objtype	Objcode
*	tabBandAdvanced	frxBuilder2.vcx	[MSFT] Advanced Properties for Bands (optional)	-1	9	-1
T	tabMultiProp	frxBuilder2.vcx	[MSFT] Multiple Selection Properties	1	99	-1
Q	RemoveLineIfBlank	frxutils.prg	[MSFT] MultiProp: Remove Line if blank	0	0	0
Q	PrintWhen	frxutils.prg	[MSFT] MultiProp: Print When expression	0	0	0
Q	FormatExpression	frxutils.prg	[MSFT] MultiProp: Format Expression / Picture	0	0	0
Q	Justification	frxutils.prg	[MSFT] MultiProp: Justification Style	0	0	0
Q	StringTrimmingMode	frxutils.prg	[MSFT] MultiProp: String Trimming mode	0	0	0
P	ReduceFont.Enabled	0	[CLP] Reduce Font Size to Fit - Enabled	5	8	-1
P	ReduceFont.MinSize	4	[CLP] Reduce Font Size to Fit - Minimum Size	3	8	-1
T	cpMyFieldPage	cpfx.vcx	[CLP] My Special Effects Page - Fields	1	8	-1
T	cpMyReportPage	cpfx.vcx	[CLP] My Special Effects Page - Report	1	1	-1

Figure 12. Add a record to the Report Builder Configuration table so your new page appears on the Report Properties dialog.

Create the special effect class for Text Watermarks

Next, create a class based on your fxAbstract class to implement the Text Watermark feature. Unlike previous examples, we don't need to process every object on the report for this special effects feature. Instead, we only need to do some initialization up front and then process this special effects feature once for each page. The code shown in **Listing 11** shows what to add to the ApplyFx method.

When ApplyFX is called from the BeforeReport method, it does some standard processing to build a collection for this feature as well as build another collection of FRX records using this feature. It then calls another method, Watermark_Init, to prepare for GDIPlus. When ApplyFX is called from the BeforeBand method, the code looks to see if this is the Page Header band. If so, it calls the Watermark_Print method do to the actual printing of the text string on the page.

Listing 11. The ApplyFx method of the fxWatermarkText class.

```
LPARAMETERS m.toListener, m.tcMethodToken, ;
   m.tP1, m.tP2, m.tP3, m.tP4, m.tP5, m.tP6,  m.tP7, m.tP8, ;
   m.tP9, m.tP10, m.tP11, m.tP12

LOCAL  ln

DO CASE
   CASE m.tcMethodToken = 'BEFOREREPORT_FX'
      *-- Create the collection
      m.toListener.CallAdjustObjectSize = 2 && Always
      This.BuildCollectionFromMemberData(m.toListener, ;
         'oFXWatermark', 'CPFX', 'WATERMARKTEXT', .t.)

      *-- Create all the graphic objects
      This.Watermark_Init(m.toListener)

   CASE m.tcMethodToken = 'BEFOREBAND'
      *-- If processing the Page Header band .. look for watermark
      IF m.tP1 = 1 AND This.oFXWatermark.GetKey(This.cReportKey) > 0
         This.Watermark_Print(m.toListener, m.tP1, @tP2)
      ENDIF
ENDCASE


RETURN
```

The Watermark_Init method starts by iterating through the oFRX collection to find the report header record. It then remembers this record number so later on we don't have to keep iterating through the oFRX collection on each page. It then instantiates several classes in the _GDIPlus class library in the FFC directory. This class library contains several classes that are wrappers to GDIPlus API calls. **Listing 12** shows which ones are needed to print the watermark text.

Listing 12. Watermark_Init method of the fxWatermarkText class.

```
LPARAMETERS m.toListener

*-- Find the RECNO of the report record and remember it
FOR ln = 1 TO This.oFRX.Count
   IF This.oFRX.Item[m.ln].ObjType = 1
      This.cReportKey = This.oFRX.GetKey[m.ln]
      EXIT
   ENDIF
ENDFOR

*-- Create a color object
This.AddProperty('oColor', ;
   NEWOBJECT('gpColor', HOME() + 'FFC\_GDIPLUS'))

*-- Create a brush object
This.AddProperty('oBrush', ;
   NEWOBJECT('gpSolidBrush', HOME() + 'FFC\_GDIPLUS'))

*-- Create a Rectangle object
This.AddProperty('oRectangle', ;
   NEWOBJECT('gpRectangle', HOME() + 'FFC\_GDIPLUS', '', 1,1,1,1))

*-- Create a text format object
This.AddProperty('oTextFormat', ;
   NEWOBJECT('gpStringFormat', HOME() + 'FFC\_GDIPLUS'))

*-- Create a font object
This.AddProperty('oFont', ;
   NEWOBJECT('gpFont', HOME() + 'FFC\_GDIPLUS'))
```

The Watermark_Print method (see **Listing 13**) prints the actual text string on the page. It seems like a lot of code, but it's fairly straight-forward once you understand GDIPlus. First, create a Color object which determines the color of the text. Default it to black and apply the percentage entered on the report. Next, create a Brush object using the Color object just created. Think of this as picking up a paintbrush and dipping it into the paint.

The next thing to do is create a Text Format object, setting the vertical and horizontal alignment to center. This ensures the text string prints centered within the region we tell it. Then get the text string from the MemberData cursor.

Now that we have the text string, start calculating where to print it and what size font to use. Get the paper size from the Report Listener object and create a Rectangle object leaving 1" margins on all four sizes. Then start a loop that increments the font size, and create a Font object. Once we have the Font object, call the MeasureStringA method on the FFCGraphics object of the Report Listener, passing in the text string, the Font object we just created, and the Rectangle and Text Format objects we previously created. One of the parameters this method returns is the number of lines needed to print the text of the given font size in the given rectangle. As long as the number of lines isn't greater than 1, increment the font size and repeat the loop. Once the return value is greater than one,

decrement the font size back down and get out of the loop. Now that we know the maximum font size, create the final Font object using this value.

The next thing to do is prepare for printing the text in the middle of the page at a rotated angle of 45°. Start by calling the Save method of the FFCGraphics object on the Report Listener to remember the current settings. Then make three more calls to the FFCGraphics object to define the rotation: TranslateTransform, RotateTransform, and TranslateTransform again. These three methods work together to prepare for the rotation. Then call the DrawStringA method of the FFCGraphics object on the Report Listener, passing it the text string, the Font object, the Rectangle object, the Text Format object, and the Brush object. Wrap up by calling the Restore method of the FFCGraphics to restore all the settings.

There's a bug in the shipped code for the MeasureStringA method of the GPGraphics class in FFC_GDIPlus.vcx. Look for variable references to lcRect and change them to lcRectF. There should be two places where the variable is used (be sure to fix this bug before running your example or the sample provided to avoid an infinite loop of error messages).

Listing 13. Watermark_Print method of the fxWatermarkText class.

```
LPARAMETERS m.toListener, m.nBandObjCode, m.nFRXRecNo

*-- Data is stored in the following fields
*-- Name: CPFX
*-- ExecWhen: WATERMARKTEXT
*-- Execute: Text String
*-- Class: Percentage

LOCAL  lnPercent, ;
       lcText, ;
       lnPageWidth, ;
       lnPageHeight, ;
       lcFontName, ;
       lnFontSize, ;
       ln, ;
       lnChars, ;
       lnLines, ;
       lnLen, ;
       lnX, ;
       lnY, ;
       lnHandle, ;
       lnSaveGraphicsHandle

*-- Set the Color
lnPercent = VAL(This.oFXWatermark.Item(This.cReportKey).Class) * 255/100
This.oColor.Set(0,0,0,m.lnPercent) && black

*-- Define the Brush object
This.oBrush.Create(This.oColor)

*-- Define the text format object
This.oTextFormat.Create()
```

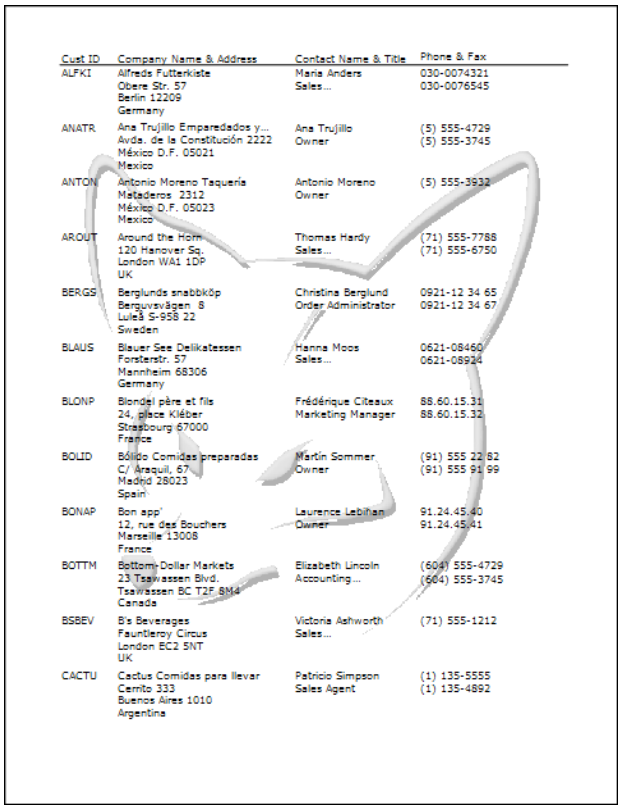
```
This.oTextFormat.Alignment = 1 && Center horizontal
This.oTextFormat.LineAlignment = 1 && Center vertical

*-- Get the text
lcText = This.oFXWatermark.Item(This.cReportKey).Execute

*-- Get the page width and height
lnPageWidth = m.toListener.SharedPageWidth
lnPageHeight = m.toListener.SharedPageHeight

*-- Create the rectangle object (page size less 1" margins)
This.oRectangle.Create(960, 960, lnPageWidth-960-960, lnPageHeight-960-960)

*-- Figure out the largest font size we can use and still fit in rectangle
lcFontName = 'Arial Black'
lnFontSize = 0
DO WHILE .t.

   *-- Increase font size by 2 points
   lnFontSize = lnFontSize + 2

   *-- Define the font object
   *-- The 3rd parameter indicates the style (0=Normal)
   *-- The 4th parameter indicates the unit of measure (3=point or 1/72 inch)
   This.oFont.Create(m.lcFontName, m.lnFontSize, 0, 3)

   *-- How many lines did it take to print with this font size?
   *-- (Note - There's a bug in the shipped code for the MeasureStringA
   *-- method of the GPGraphics class in FFC\_GDIPlus.vcx. Look for variable
   *-- references to lcRect and change them to lcRectF. There should be two
   *-- places where the wrong variable is used.)
   lnChars = 0
   lnLines = 0
   lnLen = m.toListener.FFCGraphics.MeasureStringA(m.lcText, This.oFont, ;
      This.oRectangle, This.oTextFormat, @lnChars, @lnLines)

   *-- More than 1 line, get out
   IF m.lnLines > 1
      lnFontSize = lnFontSize - 2
      EXIT
   ENDIF
ENDDO

*-- Define the font object using the calculated size
This.oFont.Create(m.lcFontName, m.lnFontSize, 0, 3)
*-- Get the middle of the page
lnX = lnPageWidth / 2
lnY = lnPageHeight / 2

*-- save the current state of the graphics handle
lnHandle = 0
m.toListener.FFCGraphics.Save(@lnHandle)
lnSaveGraphicsHandle = lnHandle
```

```
*-- now move the 0,0 point to where we'd like it to be so that when
*-- we rotate we're rotating around the appropriate point
m.toListener.FFCGraphics.TranslateTransform(lnX, lnY, 0)

*-- now change the angle at which the draw will occur
m.toListener.FFCGraphics.RotateTransform(-45, 0)

*-- restore the 0,0 point
m.toListener.FFCGraphics.TranslateTransform(-lnX, -lnY, 0)

*-- Draw the text
m.toListener.FFCGraphics.DrawStringA(m.lcText, ;
   This.oFont, This.oRectangle, This.oTextFormat, This.oBrush)

*-- Put back the state of the graphics handle
m.toListener.FFCGraphics.Restore(lnSaveGraphicsHandle)
```

It seems like a lot of code at first, but once you walk through it and understand it, it's not that complicated. This Text Watermark example is pretty bare bones and you can certainly vamp it up in your application by letting the developer choose the font name and style, color, rotation angle, position on the page, and anything else you can think of. The possibilities are endless.

Using a graphic file as a watermark

Graphic watermarks are commonly used to print company logos on reports. Similar to the Text Watermark example, you can use GDIPlus to print graphic images on your reports as shown in **Figure 13**.



Cust ID	Company Name & Address	Contact Name & Title	Phone & Fax
ALFKI	Alfreds Futterkiste Obere Str. 57 Berlin 12209 Germany	Maria Anders Sales...	030-0074321 030-0076545
ANATR	Ana Trujillo Emparedados y... Avda. de la Constitución 2222 México D.F. 05021 Mexico	Ana Trujillo Owner	(5) 555-4729 (5) 555-3745
ANTON	Antonio Moreno Taquería Mataderos 2312 México D.F. 05023 Mexico	Antonio Moreno Owner	(5) 555-3932
AROUT	Around the Horn 120 Hanover Sq. London WA1 1DP UK	Thomas Hardy Sales...	(71) 555-7788 (71) 555-6750
BERGS	Berglunds snabbköp Berguvsvägen 8 Luleå S-958 22 Sweden	Christina Berglund Order Administrator	0921-12 34 65 0921-12 34 67
BLAUS	Blauer See Delikatessen Forsterstr. 57 Mannheim 68306 Germany	Hanna Moos Sales...	0621-08460 0621-08924
BLONP	Blondel père et fils 24, place Kléber Strasbourg 67000 France	Frédérique Citeaux Marketing Manager	88.60.15.31 88.60.15.32
BOLID	Bólido Comidas preparadas C/ Araquil, 67 Madrid 28023 Spain	Martín Sommer Owner	(91) 555 22 82 (91) 555 91 99
BONAP	Bon app' 12, rue des Bouchers Marseille 13008 France	Laurence Lebihan Owner	91.24.45.40 91.24.45.41
BOTTM	Bottom-Dollar Markets 23 Tsawassen Blvd. Tsawassen BC T2F 8M4 Canada	Elizabeth Lincoln Accounting...	(604) 555-4729 (604) 555-3745
BSBEV	B's Beverages Fauntleroy Circus London EC2 5NT UK	Victoria Ashworth Sales...	(71) 555-1212
CACTU	Cactus Comidas para llevar Cerrito 333 Buenos Aires 1010 Argentina	Patricio Simpson Sales Agent	(1) 135-5555 (1) 135-4892

Figure 13. Use GDIPlus to add a Graphic Watermark to your reports.

Create the user interface for Graphic Watermarks

Start by creating a panel (based on your abstract panel class) similar to the one shown in **Figure 14**. The panel has a control for choosing a file for use as the graphic image. It also has controls to let the developer choose the size and location of the image.

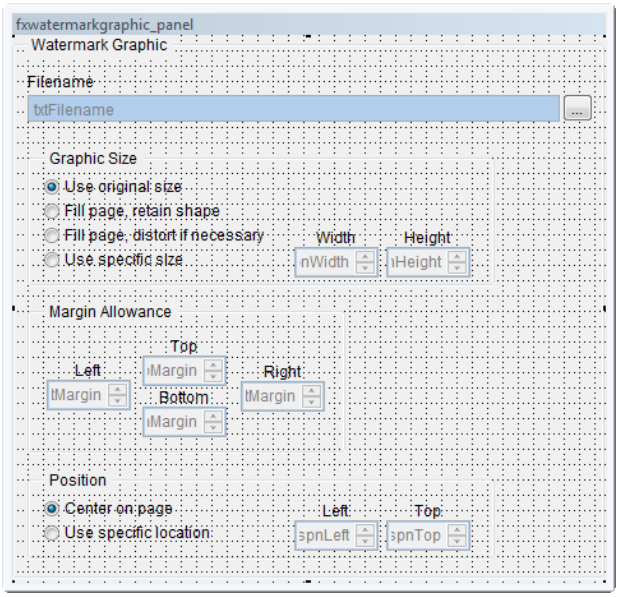


Figure 14. Create a panel with controls for choosing the image file and the size and location of the image.

Use the ExecWhen field to store WATERMARKGRAPHIC as the feature name and the Execute field to store the name of the graphic file. The Class field can be used to store the remainder of the information on the panel. The code shown in **Listing 14** and **Listing 15** saves and restores the data between the panel and the fields in the MemberData cursor.

Listing 14. SaveToFRX method of the fxWatermarkGraphic_Panel class.

```
*-- Data is stored in the following fields
*-- Name: CPFX
*-- ExecWhen: WATERMARKGRAPHIC
*-- Execute: Graphic Filename
*-- Class:   SIZE= (1=original, 2=fill/retain shape, 3=fill, 4=specific)
*--       WIDTH=
*--       HEIGHT=
*--       TOPMARGIN=
*--       BOTTOMMARGIN=
*--       LEFTMARGIN=
*--       RIGHTMARGIN=
*--       POSITION=
*--       LEFT=
*--       TOP=

LOCAL  lcFile, ;
       lcString

lcFile = This.txtFilename.Value
lcString = ''
lcString = This.AddToString(m.lcString, 'SIZE', This.opgSize.Value)
lcString = This.AddToString(m.lcString, 'WIDTH', This.spnWidth.Value)
lcString = This.AddToString(m.lcString, 'HEIGHT', This.spnHeight.Value)
lcString = This.AddToString(m.lcString, 'TOPMARGIN', This.spnTopMargin.Value)
lcString = This.AddToString(m.lcString, 'BOTTOMMARGIN', ;
                            This.spnBottomMargin.Value)
lcString = This.AddToString(m.lcString, 'LEFTMARGIN', This.spnLeftMargin.Value)
lcString = This.AddToString(m.lcString, 'RIGHTMARGIN', ;
                            This.spnRightMargin.Value)
lcString = This.AddToString(m.lcString, 'POSITION', This.opgPosition.Value)
lcString = This.AddToString(m.lcString, 'LEFT', This.spnLeft.Value)
lcString = This.AddToString(m.lcString, 'TOP', This.spnTop.Value)

SELECT MemberData
LOCATE FOR Type = 'R' AND ALLTRIM(Name) == 'CPFX' AND ;
   ALLTRIM(ExecWhen) == 'WATERMARKGRAPHIC'

DO CASE
   CASE FOUND() AND NOT EMPTY(m.lcFile)
      *-- Found record, update with values
      REPLACE Execute WITH m.lcFile, ;
              Class WITH m.lcString ;
         IN memberdata
   CASE FOUND()
      *-- Found but don't need, so delete it
      DELETE IN MemberData
```

```
   CASE NOT EMPTY(m.lcFile)
      *-- Didn't find and need it, so add it
      INSERT INTO memberdata (Type, Name, ExecWhen, Execute, Class) ;
         VALUES ('R', 'CPFX', 'WATERMARKGRAPHIC', m.lcFile, m.lcString)
   OTHERWISE
      *-- Didn't find it, don't need it, do nothing
ENDCASE
```

Listing 15. LoadFromFRX method of the fxWatermarkGraphic_Panel class.

```
*-- Data is stored in the following fields
*-- Name: CPFX
*-- ExecWhen: WATERMARKGRAPHIC
*-- Execute: Graphic Filename
*-- Class:   SIZE= (1=original, 2=fill/retain shape, 3=fill, 4=specific)
*--       WIDTH=
*--       HEIGHT=
*--       TOPMARGIN=
*--       BOTTOMMARGIN=
*--       LEFTMARGIN=
*--       RIGHTMARGIN=
*--       POSITION=
*--       LEFT=
*--       TOP=

SELECT MemberData

LOCATE FOR Type = 'R' AND ALLTRIM(Name) == 'CPFX' AND ;
   ALLTRIM(ExecWhen) == 'WATERMARKGRAPHIC'
IF FOUND()
   This.txtFilename.Value = Execute
   This.opgSize.Value = This.GetFromString(Class, 'SIZE', 1)
   This.spnWidth.Value = This.GetFromString(Class, 'WIDTH', 1.00)
   This.spnHeight.Value = This.GetFromString(Class, 'HEIGHT', 1.00)
   This.spnTopMargin.Value = This.GetFromString(Class, 'TOPMARGIN', 1.00)
   This.spnBottomMargin.Value = This.GetFromString(Class, 'BOTTOMMARGIN', 1.00)
   This.spnLeftMargin.Value = This.GetFromString(Class, 'LEFTMARGIN', 1.00)
   This.spnRightMargin.Value = This.GetFromString(Class, 'RIGHTMARGIN', 1.00)
   This.opgPosition.Value = This.GetFromString(Class, 'POSITION', 1)
   This.spnLeft.Value = This.GetFromString(Class, 'LEFT', 1.00)
   This.spnTop.Value = This.GetFromString(Class, 'TOP', 1.00)
ELSE
   This.txtFileName.Value = ''
   This.opgSize.Value = 1
   This.spnWidth.Value = 1.00
   This.spnHeight.Value = 1.00
   This.spnTopMargin.Value = 1.00
   This.spnBottomMargin.Value = 1.00
   This.spnLeftMargin.Value = 1.00
   This.spnRightMargin.Value = 1.00
   This.opgPosition.Value = 1
   This.spnLeft.Value = 1.00
   This.spnTop.Value = 1.00
ENDIF
```

```
This.Refresh()
```

Once you have the panel defined, add it to the same page class you added your Text Watermark panel. You already added a record for this page to the Report Builder Configuration table with the Registry Explorer so nothing further is required for the user interface.

Create the special effect class for Graphic Watermarks

Next, create a class based on your fxAbstract class to implement the Graphic Watermark feature. The code in the ApplyFx method is so similar to the Text Watermark example that it's not worth repeating here. The only thing you have to change is the reference from WATERMARKTEXT to WATERMARKGRAPHIC.

The Watermark_Init method is also similar in that it starts by iterating through the oFRX collection to find the report header record and then instantiates a few classes in the _GDIPlus class library in the FFC directory (see **Listing 16**). The difference, however, is that fewer GDIPlus classes are needed.

Listing 16. Watermark_Init method of the fxWatermarkGraphic class.

```
LPARAMETERS m.toListener

LOCAL  ln

*-- Find the RECNO of the report record and remember it
FOR ln = 1 TO This.oFRX.Count
   IF This.oFRX.Item[m.ln].ObjType = 1
      This.cReportKey = This.oFRX.GetKey[m.ln]
      EXIT
   ENDIF
ENDFOR

*-- Create a Rectangle object
This.AddProperty('oRectangle', ;
   NEWOBJECT('gpRectangle', HOME() + 'FFC\_GDIPLUS', '', 1,1,1,1))

*-- Create an image object
This.AddProperty('oImage', ;
   NEWOBJECT('gpImage', HOME() + 'FFC\_GDIPLUS'))
```

The Watermark_Print method (see **Listing 17**) starts by loading the Image object with the graphic file. It then does a bunch of processing to figure out what size the graphic should appear based on the original size of the graphic and the options entered on the report. Once the position and size is calculated, the Rectangle object is created accordingly. The last thing that happens is the DrawImageScaled method of the FFCGraphics object on the Report Listener is called, passing the Image object and the Rectangle object.

Listing 17. Watermark_Print method of the fxWatermarkGraphic class.

```
LPARAMETERS m.toListener, m.nBandObjCode, m.nFRXRecNo
```

```
LOCAL  lnActPictWidth, ;
       lnActPictHeight, ;
       lnPageWidth, ;
       lnPageHeight, ;
       lnSize, ;
       lnTopMargin, ;
       lnBottomMargin, ;
       lnLeftMargin, ;
       lnRightMargin, ;
       lnPrtPictWidth, ;
       lnPrtPictHeight, ;
       lnWRatio, ;
       lnHRatio, ;
       lnRatio, ;
       lnPosition, ;
       lnLeft, ;
       lnTop, ;
       lcString

*-- If the file doesn't exist, get out
IF NOT FILE(This.oFXWatermark.Item(This.cReportKey).Execute)
   RETURN
ENDIF

*-- Create the image
This.oImage.CreateFromFile(This.oFXWatermark.Item(This.cReportKey).Execute)

*-- Get the picture width and height
lnActPictWidth = This.oImage.ImageWidth
lnActPictHeight = This.oImage.ImageHeight

*-- Get the string for the rest of the properties
lcString = This.oFXWatermark.Item(This.cReportKey).Class

*-- Get the page width and height
lnPageWidth = m.toListener.SharedPageWidth
lnPageHeight = m.toListener.SharedPageHeight

*-- Calculate size to print graphic
lnSize = This.GetFromString(m.lcString, 'SIZE', 1)
lnTopMargin = This.GetFromString(m.lcString, 'TOPMARGIN', 1.00) * 960
lnBottomMargin = This.GetFromString(m.lcString, 'BOTTOMMARGIN', 1.00) * 960
lnLeftMargin = This.GetFromString(m.lcString, 'LEFTMARGIN', 1.00) * 960
lnRightMargin = This.GetFromString(m.lcString, 'RIGHTMARGIN', 1.00) * 960
DO CASE
   CASE m.lnSize = 1 && Actual Size
      lnPrtPictWidth = m.lnActPictWidth
      lnPrtPictHeight = m.lnActPictHeight
      lnTopMargin = 0
      lnBottomMargin = 0
      lnLeftMargin = 0
      lnRightMargin = 0
   CASE m.lnSize = 2 && Fill Page, retain shape
```

```
      lnWRatio = (m.lnPageWidth - m.lnLeftMargin - m.lnRightMargin) ;
         / m.lnActPictWidth
      lnHRatio = (m.lnPageHeight - m.lnTopMargin - m.lnBottomMargin) ;
         / m.lnActPictHeight
      lnRatio = MIN(m.lnWRatio, m.lnHRatio)
      lnPrtPictWidth = m.lnActPictWidth * m.lnRatio
      lnPrtPictHeight = m.lnActPictHeight* m.lnRatio
   CASE m.lnSize = 3 && Fill Page, distort
      lnPrtPictWidth = m.lnPageWidth - m.lnLeftMargin - m.lnRightMargin
      lnPrtPictHeight = m.lnPageHeight - m.lnTopMargin - m.lnBottomMargin
   CASE m.lnSize = 4 && Specific Size
      lnPrtPictWidth = This.GetFromString(m.lcString, 'WIDTH', 0.00) * 960
      lnPrtPictHeight = This.GetFromString(m.lcString, 'HEIGHT', 0.00) * 960
      lnTopMargin = 0
      lnBottomMargin = 0
      lnLeftMargin = 0
      lnRightMargin = 0
ENDCASE

*-- Calculate Position
lnPosition =  This.GetFromString(m.lcString, 'POSITION', 1)
DO CASE
   CASE m.lnSize = 3 && Fill page, distort
      lnLeft = m.lnLeftMargin
      lnTop = m.lnTopMargin
   CASE m.lnPosition = 1 && Center
      lnLeft = (m.lnPageWidth - m.lnPrtPictWidth) / 2
      lnTop = (m.lnPageHeight - m.lnPrtPictHeight) / 2
   OTHERWISE && Specific location
      lnLeft = This.GetFromString(m.lcString, 'LEFT', 0.00) * 960
      lnTop = This.GetFromString(m.lcString, 'TOP', 0.00) * 960
ENDCASE

*-- Create the rectangle
This.oRectangle.Create(m.lnLeft, m.lnTop, m.lnPrtPictWidth, m.lnPrtPictHeight)

*-- Draw the image
m.toListener.FFCGraphics.DrawImageScaled(This.oImage, This.oRectangle)
```

That's it! You can now add a graphic image as a watermark on any of your reports. You can even combine the Text Watermark and Graphic Watermark special effects features on the same report.

Data Drive Custom Features

So far, the examples shown have required you to register your custom special effect classes with the running Report Listener. That is fine for demonstration purposes, but in a production app that's cumbersome. I don't know about you, but my memory isn't exactly what it used to be. Having to remember which special effects are used on a particular report is too much for me to remember! A better solution is to data drive all your custom features so they're registered automatically when needed.

Create the table

The first step in data driving custom features is to create a table (it's called Report_Features in the sample code) with five fields; Namespace, Property, Class, ClassLib, and gfx. The first four fields are character fields and the last field is a logical field where .T. indicates this is a GFX feature and .F. indicates this is an FX feature. As you create new special effect features, add a record to this table to describe the new attribute property and the class needed to implement the special effect feature.

Create the program

The next step in data driving custom features is to create the code that references the data driving table, compares it against your report, and figures out what classes to instantiate and register. One approach is to subclass the fxListener class and put the code in the BeforeReport method. Another approach is to create a stub program for running all reports which is what I've done for this session.

The basic concept is to call this stub program (see **Listing 18**) to run all your reports instead of issuing a REPORT FORM command directly. The stub program accepts parameters to indicate what report to run, any clauses you want on the REPORT FORM command, and optionally a reference to an existing Report Listener. If no Report Listener is passed, one is created for you.

Listing 18. RunReport.prg (Stub for running reports with data driven custom features).

```
*-- Run Reports:
*-- Instantiate any classes needed to provide features that
*-- have been added to the report's Advanced Properties.
LPARAMETERS tlLabel, tcReportName, tcClauses, tnType, toListener

*-- tlLabel         .t. means it's a Label not a Report
*-- tcReportName The name of the frx
*-- tcClauses    Any clauses you want appended to the REPORT FORM command
*-- tnType          Send the numeric type to indicate which Report Listener you want
instantiated
*--toListener       If you have already instantiated a Report Listener, send it here

LOCAL  lnSelect, ;
       lnFX, ;
       lcReportLabel, ;
       loFeatures, ;
       lcClass, ;
       lcFeature, ;
       llUseFX

*-- Get the listener
IF NOT VARTYPE(m.toListener) = 'O'
   m.toListener = InstantiateListener(m.tnType)
ENDIF

*-- Get out if no Report Listener instantiated
IF VARTYPE(m.toListener) <> 'O'
```

```
   RETURN
ENDIF

*-- Does the listener have the ability to use special effects
llUseFX = PEMSTATUS(m.toListener, 'AddCollectionMember', 5)
IF m.llUseFX

   *-- Initialize
   loFeatures = CREATEOBJECT('Collection')
   lnSelect = SELECT(0)
   USE Report_Features IN 0
   USE (FORCEEXT(m.tcReportName, 'FRX')) ALIAS FRX IN 0

   *-- Look for special effects used in the FRX
   SELECT FRX
   SCAN FOR NOT EMPTY(Style)

      *-- Look for fx data in the STYLE field
      lnFX = XMLTOCURSOR(Style, 'tmpFX')
      IF m.lnFX = 0
         LOOP
      ENDIF

      *-- Instantiate each fx, if not already processed
      SELECT tmpFX
      SCAN FOR Type == 'R'
         SELECT Report_Features
         LOCATE FOR ALLTRIM(NameSpace) == UPPER(ALLTRIM(tmpFX.Name)) AND ;
                    ALLTRIM(Property) == UPPER(ALLTRIM(tmpFX.ExecWhen))
         IF NOT FOUND()
            LOOP
         ENDIF

         *-- Determine if the feature has already been added
         lcClass = ALLTRIM(Report_Features.Class)
         IF m.loFeatures.GetKey(m.lcClass) > 0
            LOOP
         ENDIF
         loFeatures.Add(m.lcClass)

         *-- Instantiate the applicable fx/gfx class
         m.toListener.AddCollectionMember( ;
                m.lcClass, ;
                ALLTRIM(Report_Features.ClassLib), ;
                '', .t., Report_Features.Gfx, .f.)
      ENDSCAN
   ENDSCAN

   USE IN Report_Features
   USE IN tmpFX
   USE IN FRX
   SELECT (m.lnSelect)
ENDIF
```

```
*-- Run the report
lcReportLabel = IIF(m.tlLabel, 'LABEL', 'REPORT')
&lcReportLabel FORM (m.tcReportName) &tcClauses OBJECT m.toListener

*-- Clean up
IF m.llUseFX
   FOR EACH lcFeature IN m.loFeatures FOXOBJECT
      m.toListener.RemoveCollectionMember(m.lcFeature, .f., .t.) &&  m.tcName,
m.tlInGFX, m.tlNameIsClass
   ENDFOR
ENDIF

RETURN

**************************
FUNCTION InstantiateListener
**************************
LPARAMETERS tnType

LOCAL  lcClass

DO CASE
   CASE INLIST(m.tnType, 0, 1, 2, 3)
      *-- 0=Print
      *-- 1=Preview
      *-- 2=Render Page by Page (no output)
      *-- 3=Render all Pages at once (no output)
      m.lcClass= 'FXListener'
   CASE m.tnType = 4 && XML
      m.lcClass= 'XMLListener'
   CASE m.tnType = 5 && HTML
      m.lcClass= 'HTMLListener'
   OTHERWISE && Just use the base listener
      lcClass = ''
ENDCASE

IF EMPTY(m.lcClass)
   loListener = CREATEOBJECT('ReportListener')
ELSE
   loListener = NEWOBJECT(m.lcClass, HOME() + 'FFC\_ReportListener')
ENDIF
loListener.ListenerType = m.tnType

RETURN m.loListener
```

The first thing the stub does is instantiate a Report Listener (if one isn't passed in by reference.) Next, the data driving table and the FRX are both opened. A SCAN/ENDSCAN is then used to loop through the FRX looking for records with information in the STYLE field.

Each time information is found in the STYLE field, the XML string is converted to a temporary cursor for processing. The temporary cursor is then scanned for records related to custom features, and an attempt is made to find that particular feature in the data

driving table. If a match is found, the indicated class is instantiated and registered with the Report Listener.

After the entire FRX is processed, the tables are closed and the report is run using a REPORT FORM command. Finally, some cleanup is done and the custom features are unregistered.

That's it. Now you don't ever have to remember what custom features are used on any given report. Simply run all your reports by calling the stub program and everything magically falls into place.

Share Custom Features

The FoxPro Community has always been one of the most giving and sharing development communities around. When I started realizing the potential of what we as a community can do with these custom features on reports, my head started spinning with ideas. I wanted to harness the potential into a concept where we can easily share our custom features with each other.

Enter stage right … VFPX, also known as CodePlex (http://www.codeplex.com/Wiki/View.aspx?ProjectName=VFPX). This is a place where several FoxPro-related open-source projects reside. These projects have been created by other FoxPro developers who generously donated their time to the community. Many are a collaboration of several people; developers, testers, technical writers, etc. If you've never looked at this site, I encourage you to do it right now. You'll be amazed at all the tools and utilities available to you, all free of charge!

After the Southwest Fox conference is over and the dust has settled, I plan to create a new project on VFPX called fxReports. The concept is simple. You'll be able to download some core code I've created, along with instructions to walk you through using custom report features in your application. The download also contains two utilities: Unwrap & Merge and Wrap & Share.

The instructions

I'd like to create a form that's clear and simple and basically walks you through using custom report features (see the preliminary form shown in **Figure 15**.) I plan to have this form explain how to prepare your system for using custom features, how to use features created by others, and of course, how to create your own features and share them with the FoxPro Community.

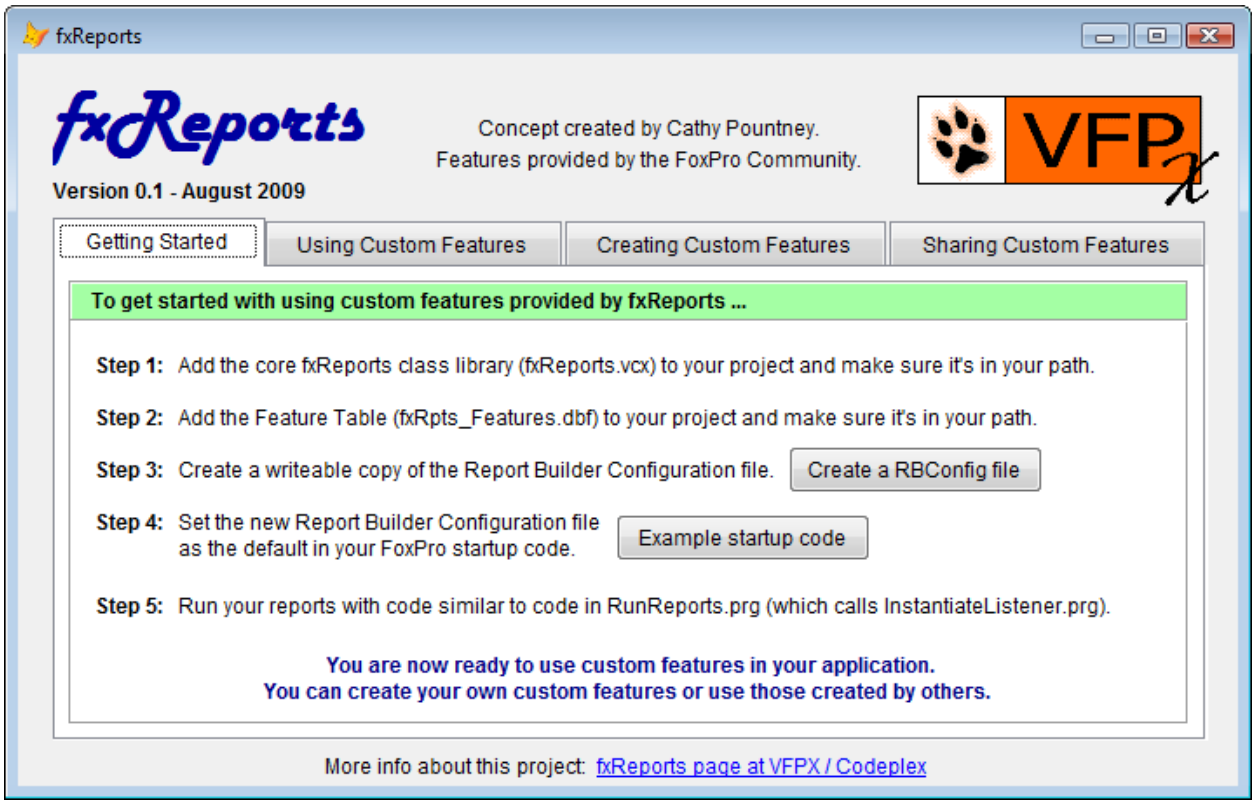


Figure 15. Run this simple form to walk you through using custom report features in your application.

Unwrap & Merge

The Unwrap & Merge utility is quite simple to run (see **Figure 16**.) First you tell it where your Report Builder Configuration table is. Second, you tell it where your data driving table is. Third, you tell it what zip file you want to process. Finally, you tell it where you want the new features installed.

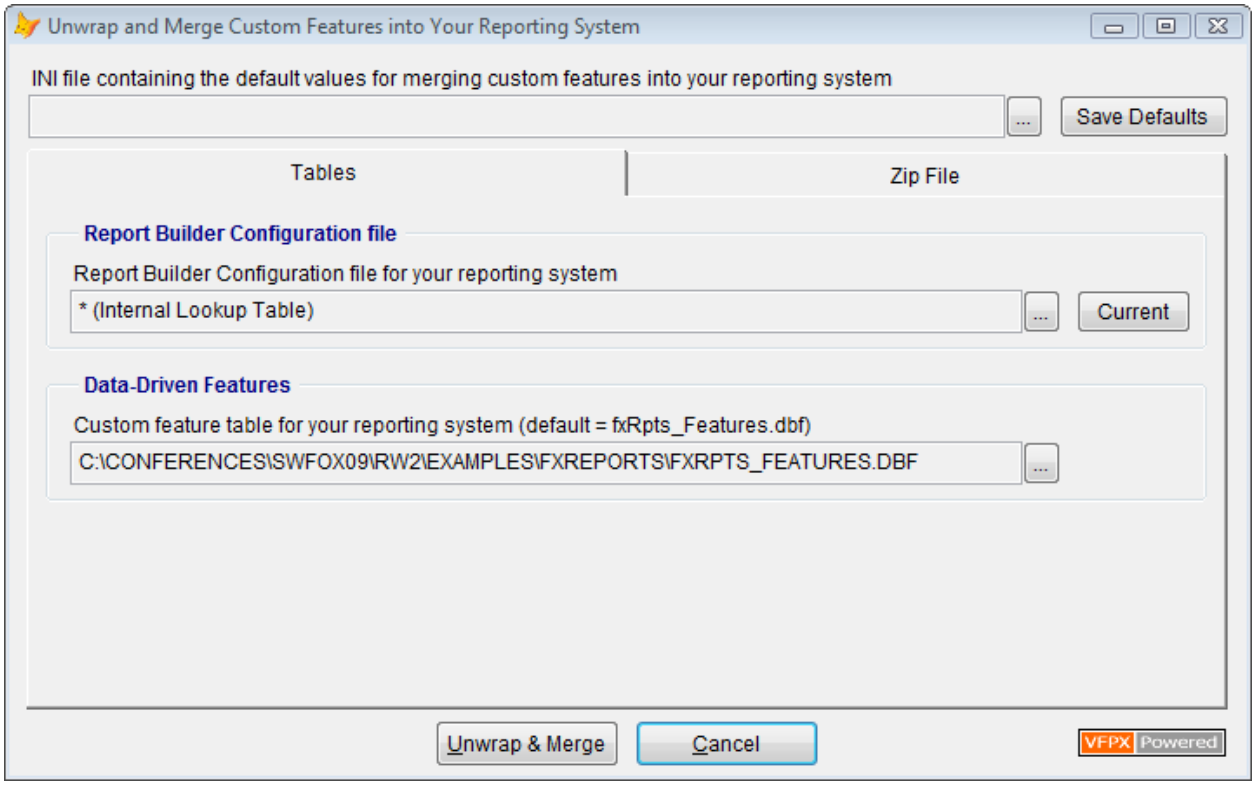


Figure 16. Use the Unwrap & Merge utility to install custom report features.

Once you've given it those four pieces of information you tell it to "Unwrap & Merge". What it does behind the scenes is install all the files needed for the feature(s) you're installing. It also appends the appropriate records to your Report Builder Configuration table and your data driving table.

That's it. Once the utility is done, you can start using the new features on your reports immediately. Really! It's that simple!

Wrap & Share

Once you've gotten the hang of using custom report features you can start creating your own. And of course, I encourage you to share all your hard work with other FoxPro developers. Thus, the Wrap & Share utility (see **Figure 17**.)

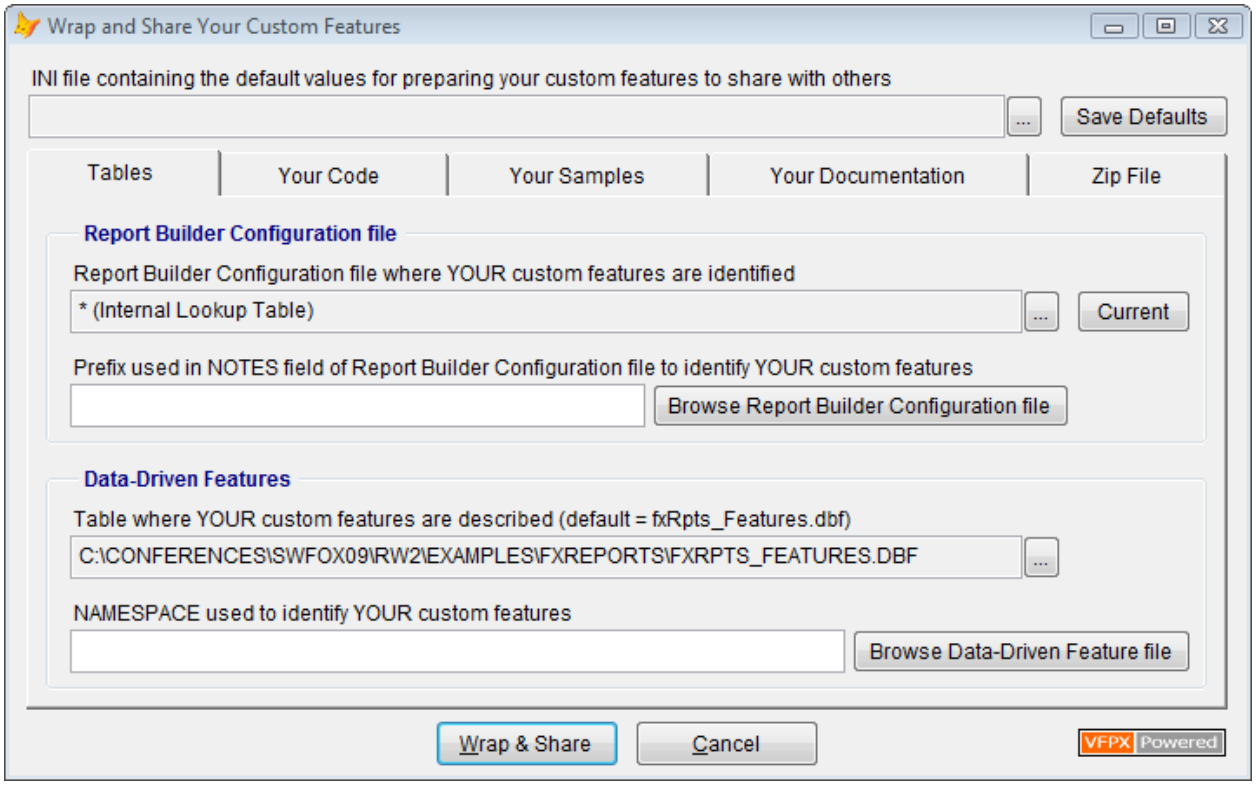


Figure 17. Use the Wrap & Share utility to package up and share your custom report features with others.

This utility is designed to package up everything needed to implement your custom report feature so it can be shared with others. The first tab is where you tell it about your Report Builder Configuration table and the unique prefix you used on all the records you added. You also use this tab to indicate where your data driving table is and what the unique namespace is for your custom features. Remember, you may have records from other developers merged into your Report Builder Configuration table and data driving table, so you have to identify which ones are yours.

The second tab is where you'll indicate all the files needed for your custom report feature. Most likely you'll have a class library with several classes. You can also include any programs, graphic files, or any other type of file.

The third tab is where you can indicate any samples you've put together to demonstrate this custom feature. It's not required, but I encourage everyone to put together a sample of how to use the feature. Like the samples, I'm also encouraging everyone to provide documentation with their custom report features. The fourth tab is used to for indicating the documentation files.

The final tab is simply a place to tell the utility what to call the zip file and where to put it.

Once you've entered all the information, simply hit the "Wrap & Share" button and it will do the rest. It sucks out the appropriate records from your Report Builder Configuration table and puts them into another table. It does the same with the records from the data driving

table. It then zips up all the files for your code, samples, and documentation. Of course, it does this all in a manner that meshes with how the Unwrap & Merge utility works. Once everything is zipped up, you can share it with other developers.

Where to share

At this point, I'm still working on a place for everyone to upload their "Wrap & Share" zip files. VFPX isn't exactly the right place to have several people posting things all the time. There are some other projects on VFPX with this same type of issue. I've been in discussions with a few other developers on how to resolve this and we hope to have it hashed out soon.

The concept we want to implement is this. Have a central repository to host the downloads. We want a process where we can authorize people to upload files to this central repository. Obviously, we need to make sure the uploads are legit and are really related to the VFPX project. Once approved, new files become available for others to download and use in their applications. We'd also like to have something in place for tracking bugs, comments, etc. about the downloads.

At the time of this writing, this central repository concept hasn't been fully hashed out, but we hope to have it figured out and implemented soon. Once we get if figured out, the details will be on VFPX and I'll post something on my blog.

Summary

Most people already know how passionate I am about reports. With the ability to create our own custom report features in VFP 9 SP2, I'm over the top with excitement. I've only scratched the surface in this session. The possibilities of what we'll be able to do in our reports are endless. Personally, I have a file folder full of little scraps of paper with my chicken-scratching, each describing an idea for a custom report feature. If we all put our heads together, create custom report features, and share them with each other, we'll conquer the world. Okay .. maybe it's not THAT big of an invention .. but it's close! ☺

Biography

Cathy Pountney has been developing software for almost three decades and is proud to have earned the Microsoft Visual FoxPro MVP Award seven years in a row. She is equally proud to have had the opportunity to work as a subcontractor onsite in Redmond with the Microsoft Fox Team in 2001. Cathy enjoys writing articles for various Fox magazines as well as writing books. She authored The Visual FoxPro Report Writer: Pushing it to the Limit and Beyond and co-authored Visual FoxPro Best Practices for the Next Ten Years and Making Sense of Sedna and SP2. Cathy participates in her local FoxPro user group (GRAFUG) and speaks at other user groups when time permits. She has spoken at numerous conferences including GLGDW, Essential Fox, Advisor DevCon, DevTeach, and of course, her favorite, Southwest Fox. For the past several years, Cathy has worked for Memorial Business Systems writing software for the cemetery and funeral home industry, which proves ... Fox is NOT dead!

Email: *cathy@frontier2000.com*
My website: *www.frontier2000.com*
MBS's website: *www.mbs-intl.com*
Blog: www.cathypountney.blogspot.com
Twitter: frontier2000