A More Flexible Report Designer

Doug Hennig

This month, Doug presents a way to make the VFP Report Designer more flexible, such as customizing the pages of the properties dialogs without having to change ReportDesigner.APP.

The FoxPro reporting system was fairly stagnant for many years. The Report Designer and the reporting engine were black boxes built into the VFP executable that provided only limited internal access through things such as user-defined functions called from various places.

However, that all changed in VFP 9. Thanks to well-known VFP gurus Lisa Slater Nicholls and Colin Nicholls, as well as VFP team member Richard Stanton, both the reporting engine and Report Designer became much more open and accessible. The Report Designer raises design-time events you can create event handlers for and the ReportListener base class provides customization points for almost every step in the report run process.

One of things that really excited me about the changes in the reporting system in VFP 9 is that while the design surface is still built into VFP9.EXE, most of the dialogs you can open are pure VFP code. Even better, Microsoft includes the source code for these dialogs: if you unzip XSource.ZIP in the Tools\XSource subdirectory of the VFP program folder, you'll find a Tools\XSource\VFPSource\ReportBuilder folder that contains the source code for ReportBuilder.APP, the add-on for the Report Designer that provides the dialogs and an event handling framework.

Because we have the source code for the Report Designer dialogs, we can finally do one thing I've always wanted to do: customize the properties dialogs for the various report objects. Because I expose the VFP Report Designer to my users, I want to simplify the dialogs, which were designed mostly for developers, and add custom controls for special effects I implement using ReportListener classes.

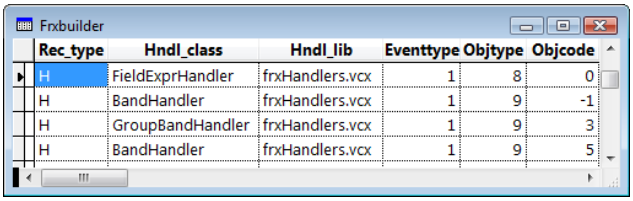
While working on a customized version of ReportBuilder.APP, I came up with some ideas to make customizing the ReportDesigner even easier and more flexible. I eventually implemented these ideas into a project I call FRXTabs, which is the focus of this article.

Before we get into FRXTabs, first let's cover some background into how the Report Designer works.

How ReportBuilder.APP displays a dialog

When you do something in the Report Designer, such as double-clicking a text box, an event is fired. When an event occurs, the Report Designer looks at the content of the system variable _ReportBuilder for the name of an event handler application, and then passes information about the event to that application. By default, _ReportBuilder contains the path for ReportBuilder.APP, which provides the dialogs displayed in the Report Designer and other event handling behaviors as well.

Each event raised in the Report Designer is handled by a class. ReportBuilder.APP looks in a report "registry" table to see which class to instantiate for that event. By default, the registry table is FRXBuilder.DBF, built into ReportBuilder.APP. **Figure 1** shows part of this table.


Frxbuilder

Rec_type	Hndl_class	Hndl_lib	Eventtype	Objtype	Objcode
H	FieldExprHandler	frxHandlers.vcx	1	8	0
H	BandHandler	frxHandlers.vcx	1	9	-1
H	GroupBandHandler	frxHandlers.vcx	1	9	3
H	BandHandler	frxHandlers.vcx	1	9	5

Figure 1. FRXBuilder.DBF is the report builder registry table.

When the user double-clicks a text box to display its properties dialog, event type 1 occurs for the object (see the "Understanding Report Builder Events" topic in the VFP help for a list of the event types). Text boxes have OBJTYPE = 8 and OBJCODE = 0 in the FRX (you can use REPORT FORM HOME() + 'Tools\Filespec\90FRX' to see a list of the values for these fields for the various report objects). So,

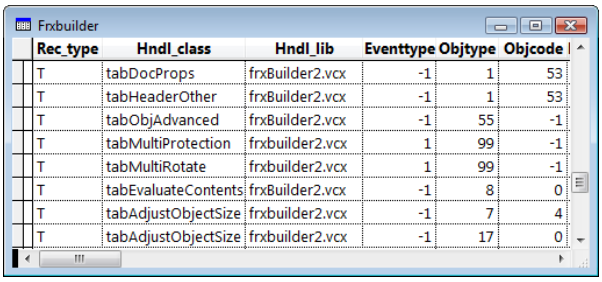
ReportBuilder.APP looks in the registry table for a record with REC_TYPE = "H" (for event handler), EVENTTYPE = 1, OBJTYPE = 8, and OBJCODE = 0. Once it finds such a record, ReportBuilder.APP instantiates the class specified in the HNDL_CLASS and HNDL_LIB fields and calls its Execute method. In the case of a text box, that's FieldExprHandler in FRXHandlers.VCX, as you can see in **Figure 1**. If you open FieldExprHandler in the Class Designer, you'll see the familiar Field Properties dialog.

Most of the dialogs displayed in the Report Designer are subclasses of FRXHandlerForm, defined in FRXBuilder.VCX. FRXHandlerForm provides the basics for a properties dialog, and each subclass implements the specifics for the type of report object it maintains. Typically, these subclasses have a pageframe with individual tabs for sets of properties.

Extending dialogs

VFP 9 SP2 provides a mechanism to add additional pages to the pageframe of a particular dialog without having to modify the class and rebuild ReportBuilder.APP: records with REC_TYPE = "T" in the registry table. When an event occurs, FRXHandlerForm.LoadFromFRX, which is called from Execute, looks for "T" records for the current event and object type. For example, when you double-click a text box, the "H" record for that event causes FieldExprHandler to be instantiated. LoadFromFRX looks for "T" records with the following criteria: EVENTTYPE = 1 (a properties dialog event) or -1 (meaning any event); OBJTYPE = 8, -1 (meaning any object), or 55 (a report layout object); and OBJCODE = 0 or -1. If it finds any such records, it adds the classes specified in HNDL_CLASS and HNDL_LIB as pages of the pageframe.

As you can see in **Figure 2**, the new pages added to Report Designer dialogs in SP2, such as the Dynamics page, are actually implemented as "T" records rather than added to the dialog classes. For example, one of the "T" records for a text box specifies TabEvaluateContents in FRXBuilder2.VCX, which represents the Dynamics page.


Frxbuilder

Rec_type	Hndl_class	Hndl_lib	Eventtype	Objtype	Objcode
T	tabDocProps	frxBuilder2.vcx	-1	1	53
T	tabHeaderOther	frxBuilder2.vcx	-1	1	53
T	tabObjAdvanced	frxBuilder2.vcx	-1	55	-1
T	tabMultiProtection	frxbuilder2.vcx	1	99	-1
T	tabMultiRotate	frxbuilder2.vcx	1	99	-1
T	tabEvaluateContents	frxBuilder2.vcx	-1	8	0
T	tabAdjustObjectSize	frxbuilder2.vcx	-1	7	4
T	tabAdjustObjectSize	frxbuilder2.vcx	-1	17	0

Figure 2. "T" records in the report builder registry table add additional tabs to dialogs.

There are only a couple of requirements for the class specified in a "T" record. First, it must be a subclass of Page since it'll be a page in the pageframe in the dialog. If you want, it can be a subclass of Pge in FRXControls.VCX, itself a subclass of Page, but that's not a requirement. Second, it must have LoadFromFRX (called when the dialog is displayed) and SaveToFRX (called when the user selects a different page or clicks OK) methods. LoadFromFRX typically loads information from the report object's FRX record and populates the values of the controls on the page. SaveToFRX typically saves the values of the controls in the page into the FRX record.

Adding your own handlers and pages

Now that you know how the native report builder displays dialogs and adds additional pages to the existing dialog classes, it should be obvious how you can create your own dialogs and pages: overriding the HNDL_CLASS and HNDL_LIB values in the appropriate "H" record substitutes your dialog class for the native one and adding "T" records adds additional pages to the dialog. There are a couple of ways you can do this:

- Alter the contents of the existing FRXBuilder.DBF. Of course, since FRXBuilder.DBF is built into ReportBuilder.APP, that means rebuilding the APP.
- Create a copy of FRXBuilder.DBF, alter the contents of the copy, and then tell the native ReportBuilder.APP to use the new table rather than the built-in one.

To create a copy of FRXBuilder.DBF, open a report in the Report Designer, bring up any properties dialog (for example, double-click a text box), right-click, and choose Options from the shortcut menu. That displays the Report Builder Options dialog shown in **Figure 3**. Click the "Create copy" button to create a copy of the registry table. You only have to do this once.

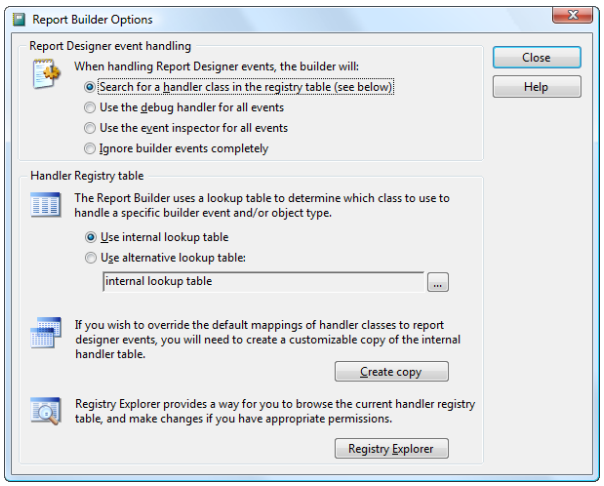


Figure 3. The Report Builder Options dialog has a function to create a copy of the report builder registry table.

To tell ReportBuilder.APP to use the new table, use the following code:

```
do (_reportbuilder) with 3, "TablePath.DBF"
```

where TablePath is the name and path for the registry table copy. Typically, you'll add this code to your startup program.

Note that there's a bug in ReportBuilder.APP: when called in this way, it sets DELETED off but doesn't restore it again afterward. So, if you have DELETED set on, remember to SET DELETED ON after using the line of code shown above.

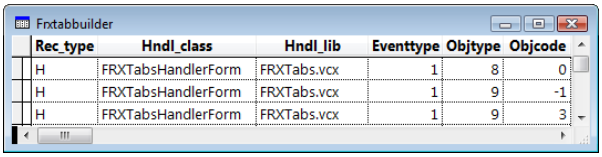
FRXTabs

In the native report builder, each of the dialogs is represented by its own, single-purpose form class. For example, FieldExprHandler in FRXHandlers.VCX is the properties dialog for text boxes and PictureBoundHandler, also in FRXHandlers.VCX, is the properties dialog for images. As I mentioned earlier, each of these is a subclass of FRXHandlerForm in FRXBuilder.VCX, customized for the needs of the particular dialog.

Although you can add your own pages to the dialogs using "T" records, there isn't an easy way to alter existing pages of the dialogs without modifying the classes and rebuilding ReportBuilder.APP. I wanted a more flexible mechanism. So I created FRXTabs.

FRXTabs uses a single, generic class for all properties dialogs. Pages and controls are added to this class at runtime rather than design time, which is the key to the flexibility FRXTabs provides.

The generic dialog class is FRXTabsHandlerForm in FRXTabs.VCX. FRXTabsHandlerForm is a subclass of the same FRXHanderForm the native builder dialogs use, with just a few methods overridden. To tell ReportBuilder.APP to use FRXTabsHandlerForm for most of the events rather than the native dialogs, change the "H" records in the registry table to specify FRXTabsHandlerForm. I've already done this in FRXTabBuilder.DBF that comes with FRXTabs so you don't have to. Compare the records in **Figure 4** with those in **Figure 1**.

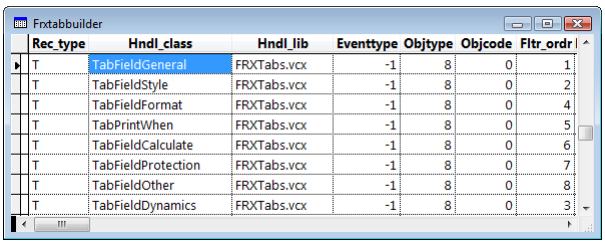


Rec_type	Hndl_class	Hndl_lib	Eventtype	Objtype	Objcode
H	FRXTabsHandlerForm	FRXTabs.vcx	1	8	0
H	FRXTabsHandlerForm	FRXTabs.vcx	1	9	-1
H	FRXTabsHandlerForm	FRXTabs.vcx	1	9	3

Figure 4. The "H" records in FRXTabHandler specify FRXTabsHandlerForm for most dialogs.

FRXTabBuilder.DBF has two fields that don't appear in FRXBuilder.DBF: LOADFRX, which contains code to be executed when the dialog is displayed, and FILTER, a memo field which can contain a filter expression that can suppress a page in the dialog if you want that page to appear conditionally, such as only for certain users. We'll see how these fields are used in the next section.

To specify what pages appear in a specific dialog, add "T" records to the registry table for each page for each dialog type. The FLTR_ORDR column contains the order in which the pages appear. For example, **Figure 5** shows the "T" records for the properties dialog for a text box. Eight pages are specified for EVENTYPE = 1, OBJTYPE = 8, and OBJCODE = 0: General, represented by the TabFieldGeneral class, is the first page (FLTR_ORDR is 1), followed by Style (TabFieldStyle), Dynamics (TabFieldDynamics), Format (TabFieldFormat), Print When (TabPrintWhen), Calculate (TableFieldCalculate), Protection (TabFieldProtection), and Other (TabFieldOther). In addition, a "T" record for the Advanced Page specifies TabObjectAdvanced as the class to use, EVENTTYPE = -1 (any event), and OBJTYPE = 55 and OBJCODE = -1 (any layout object). Specifying a page this way means it's added to every dialog without having to create one "T" record per dialog.



Rec_type	Hndl_class	Hndl_lib	Eventtype	Objtype	Objcode	Fltr_ordr
T	TabFieldGeneral	FRXTabs.vcx	-1	8	0	1
T	TabFieldStyle	FRXTabs.vcx	-1	8	0	2
T	TabFieldFormat	FRXTabs.vcx	-1	8	0	4
T	TabPrintWhen	FRXTabs.vcx	-1	8	0	5
T	TabFieldCalculate	FRXTabs.vcx	-1	8	0	6
T	TabFieldProtection	FRXTabs.vcx	-1	8	0	7
T	TabFieldOther	FRXTabs.vcx	-1	8	0	8
T	TabFieldDynamics	FRXTabs.vcx	-1	8	0	3

Figure 5. The "T" records for a particular event and report object type specify the pages that appear.

FRXTabsHandlerForm

Let's see how FRXTabsHandlerForm works. LoadFromFRX, called when the form is displayed,

has two jobs to do: execute a modified version of the parent class (FRXHandlerForm) code and execute any custom code for the specific dialog that's specified in the LOADFRX memo in the registry record.

If you look at the native builder dialogs, such as FieldExprHandler, you see they use DODEFAULT() plus some custom code specific to that dialog. We need to do something similar, so I copied the code in each of those dialogs, pasted it into the LOADFRX memo of the appropriate "H" record in FRXTabBuilder.DBF, and then modified it so it'll work when called by EXECSCRIPT() rather than as the method of a form. For example, I replaced "This" with "poForm" (poForm contains a reference to This, as we'll see when we look at the code for LoadFromFRX) because "This" can only be used in method code. I also added #INCLUDE FRXBuilder.H since the code contains constants defined in that or other include files. See the comments at the start of each LOADFRX memo for the exact changes made to that dialog's code.

FRXTabsHandlerForm.LoadFromFRX starts by calling the new LocateHandlerClass method to find the "H" record in the registry table that caused the dialog to be launched. Next, it calls the new ParentLoadFromFRX method. That method contains a modified copy of the code from FRXHandlerForm.LoadFromFRX; rather than altering the code in FRXHandlerForm. LoadFromFRX, it seemed to me a better approach to copy its code, make the necessary changes, and then call that modified code. That way, the native ReportBuilder.APP can be used without any changes. We'll look at ParentLoadFromFRX in a moment.

Next, if there's any custom code in the LOADFRX memo in the registry record for the dialog, poForm is set to a reference to This so it can be used in the LOADFRX code as discussed earlier, and the LOADFRX code is retrieved, including the #DEFINES in any #INCLUDE files, using the new GetHandlerLoadFromFRXScript. That code is executed using EXECSCRIPT() after a complication is dealt with: the native report builder #INCLUDE files have a duplicate #DEFINE statement, so that line must be removed.

```
local loEvent, ;
    llCode, ;
    lnRecno, ;
    llReturn, ;
    lcCode, ;
    loException as Exception
private poForm
loEvent  = This.FRXEvent
llCode   = ;
    This.LocateHandlerClass(;
    loEvent.eventType, ;
    loEvent.ObjType, loEvent.ObjCode) and ;
    not empty(FRXRegistry.LoadFRX)
lnRecno  = recno('FRXRegistry')
llReturn = This.ParentLoadFromFRX()
if llReturn and llCode
    go lnRecno in FRXRegistry
    poForm = This
    lcCode = ;
        This.GetHandlerLoadFromFRXScript(;
        FRXRegistry.LoadFRX)
    lcCode = strtran(lcCode, ;
        '#define DEFAULT_MBOX_TITLE_LOC ' + ;
        "Report Builder"')
        && remove a duplicate #DEFINE
    try
        llReturn = execscript(lcCode)
    catch to loException
        messagebox(loException.Message + ;
            c_CR + 'LoadFRX script for ' + ;
            trim(FRXRegistry.Notes), ;
            MB_ICONSTOP, ;
            DEFAULT_MBOX_TITLE_LOC)
    endtry
endif llReturn ...
return llReturn
```

As I mentioned earlier, ParentLoadFromFRX contains a modified copy of the code in FRXHandlerForm.LoadFromFRX. The reason we have to use a modified copy rather than issuing DODEFAULT() from our LoadFromFRX method is that we want to deal with "T" records a little differently: we want pages added in the order specified in FLTR_ORDR rather than the order specified in the registry table, and we want to remove records with a duplicate FLTR_ORDR. The latter allows us to have records that are both specific for a certain OBJTYPE and general for an OBJCODE, such as records for all bands and specific for group header bands.

The modification to the FRXHandlerForm.LoadFromFRX code is relatively simple and well-documented. Essentially, we are going to create a cursor of the desired "T" records rather than using records from the registry table directly. Here's the specific code that replaces the SELECT FRXRegistry statement just prior to a SCAN loop processing the "T" records for the current dialog:

```
*** DH 11/30/2009: create a cursor of "T"
*** records so they can be processed in
*** FLTR_ORDR order
*** select frxRegistry
*** curRec = recno()
local lcTabs, lcClass, lcLibrary, lnOrder
lcTabs = '__FRXRegistry'
select * from FRXRegistry ;
    where REC_TYPE = HANDLREG_EXTRATAB and ;
    inlist(EVENTTYPE, ;
        This.frxEvent.EventType, -1) and ;
    (inlist(OBJTYPE, ;
        This.frxEvent.ObjType, -1) or ;
(OBJTYPE = FRX_OBJTYPE_LAYOUTCONTROLS and ;
    This.frxcursor.IsLayoutControl( ;
    This.frxEvent.ObjType))) and ;
    inlist(OBJCODE, This.frxEvent.ObjCode, ;
        -1) and ;
    (empty(FILTER) or evaluate(FILTER)) and ;
    not deleted() ;
    order by FLTR_ORDR ;
```

```
    into cursor (lcTabs) readwrite
delete from (lcTabs) where FLTR_ORDR in ;
    (select FLTR_ORDR from (lcTabs) ;
    group by FLTR_ORDR ;
    having count(FLTR_ORDR) > 1) and ;
    (inlist(OBJTYPE, -1, ;
    FRX_OBJTYPE_LAYOUTCONTROLS) or ;
    OBJCODE = -1)
*** DH 11/30/2009: end of new code
```

There are a few other changes to the LoadFromFRX code in ParentLoadFromFRX but they aren't important to this discussion.

In summary, FRXTabsHandlerForm provides a generic dialog class that's used for the properties dialog for every report object because the "H" records for every object specify FRXTabsHandlerForm rather than the native class. Any custom LoadFromFRX code that a specific native dialog has is reproduced in the dialog's LOADFRX memo in the registry table and executed when the dialog starts. The various pages of each dialog are loaded at runtime because the "T" records specify them. Thus, we have a data-driven, generic dialog that's both more flexible than the native dialogs and much easier to maintain.

Replacing native pages

As you can likely guess, the biggest job in implementing FRXTabs was creating classes that replace each of the pages in the native dialog classes. Actually, it wasn't as big a job as you'd think. It turns out that each page in the native dialog doesn't consist of individual controls but rather an instance of a container class of controls. So, it was basically a matter of creating page classes and dropping on them the same container classes used in the pages of the native dialogs.

I started by creating TabBase, the parent class for all of the page classes in FRXTabs. TabBase is a subclass of Pge in FRXControls.VCX, with the addition of an About method for documentation purposes.

I then created subclasses of TabBase for each page in each dialog. Let's look at a specific example: the General page of the properties dialog for text boxes.

I created a subclass of TabBase called TabFieldGeneral. I considered dropping on it instances of PanelFieldExpr, PanelFieldPositioning, and PanelAbsolutePositioning, the three container classes that appear in the General page of the text box properties dialog. However, the instances of those classes in the native properties dialog, FieldExprHandler, have a few visual changes, such as Height and Width, so instead I copied the instances from FieldExprHandler and pasted them into TabFieldGeneral. You can see the result in Figure 6. I then set the include file for the class to FRXBuilder.H and put the following code into Init:

```
This.Caption       = UI_TAB_GENERAL_LOC
This.Name          = 'pageGeneral'
This.HelpContextID = ;
    UI_CONTROL_PROPS_GENERAL_HELP_ID
```

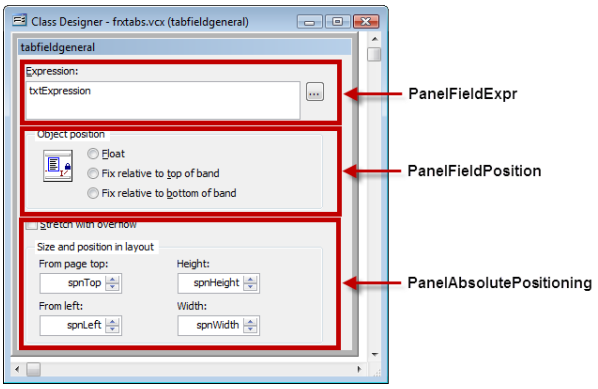


Figure 6. TabFieldGeneral contains instances of three existing container classes.

That's all that's required to create a class that's the same as the corresponding page in a properties dialog. However, while I was at it, I decided to fix a few issues.

For example, if you look very closely at the Format page of the text box properties dialog after selecting Numeric, you may notice that the "CR if positive" checkbox doesn't quite align with the other checkboxes in its column; it's one pixel too far to the right as you can see in the enlarged image in Figure 7.

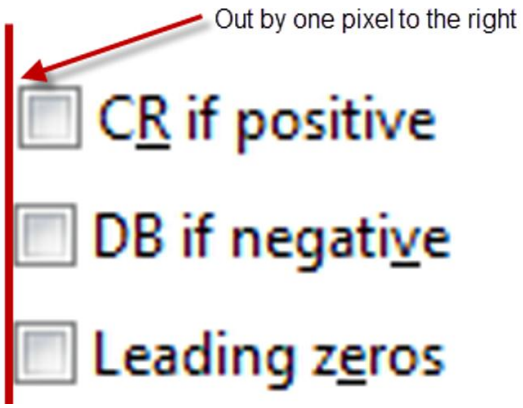


Figure 7. The "CR if positive" checkbox doesn't align with the others.

Figure 8 shows that in Windows Vista and higher, in which case Segoe UI is used as the font for controls, the "SET DATE format" option is cut off.

Figure 8. The "SET DATE format" option is cut off.

While the logical thing to do is to fix the issues with these and other controls in the native container classes (PanelFieldFormat in FRXPanels.VCX in the case of these two issues), one of my goals was to not make any changes in those classes so FRXTabs can be used without rebuilding ReportBuilder.APP. So, I made the necessary changes to the instances in the FRXTabs classes instead. For example, in TabFieldFormat, used for the Format page of the text box properties dialog, I moved chkCRIfPositive one pixel to the left and set chkSetDate.AutoSize to .T. to resolve the two issues I mentioned here. To see which changes I made, check the comments in the About method in each class.

One other issue to note is that while in general ReportBuilder.APP is localizable (translate the strings defined in FRXBuilder_LOC.H to the desired language and rebuild ReportBuilder.APP), the controls associated with rotation have hard-coded captions. So, I created FRXTabs.H, which contains constants for the captions of those controls, used it as the include file for the appropriate classes, and in the Init method of those controls set Caption to the appropriate constant. Thus, to localize FRXTabs, translate the strings in FRXTabs.H to the desired language and recompile FRXTabs.VCX.

Using FRXTabs

To use FRXTabs, download it from the Subscriber downloads page for this article and unzip it into any folder. FRXTabs consists of FRXTabs.VCX/VCT, which contains all of the replacement dialog and tab classes; FRXTabBuilder.DBF/FPT, which is a modified report registry table that specifies FRXTabHandlerForm as the handler for most events and has the "T" records that define the pages for every dialog; FRXTabs.H, the #INCLUDE file mentioned in the previous section; and FixPaths.PRG, discussed next.

Before using FRXTabs, you need to ensure that the paths it uses internally (specifically, the location of the ReportBuilder source code files) are correct for your system. I provided a program to handle that for you; simply DO FIXPATHS.PRG.

You can use FRXTabs as is without doing any customization. The benefit you'll get is that I fixed all of the visual issues in ReportBuilder.APP. To do so, simply use this from the Command window or in your applications:

```
do (_reportbuilder) with 3, ;
    "FRXTabBuilder.DBF"
```

However, the real advantage of FRXTabs is the ability to customize the dialogs as you see fit. For example, I never use the "SET DATE format" and "British date" options in the Format page of the text box properties dialog, so their presence simply clutters the dialog. Wouldn't it be nice to get rid of them and any other options you never use? With FRXTabs, it's really easy:

- Subclass TabFieldFormat, the class used for the Format page. Let's call the subclass MyFieldFormat in MyFRXTabs.VCX.
- In MyFieldFormat, set chkSetDate.Visible and chkBritishDate.Visible to .F. Note that there are several copies of those two checkboxes but you can't see them because the containers in the class are sized quite narrow. You'll have to find the controls in the Properties window. You may also want to move some other controls to account for the empty space of these two invisible controls.
- Edit the "T" record that has HNDL_CLASS = "TabFieldFormat", setting HNDL_CLASS to "MyFieldFormat" and HNDL_LIB to "MyFRXTabs.VCX".

Figure 9 shows what the Format page looks like when you use MyFieldFormat. Not only are "SET DATE format" and "British date" gone, "CR if positive" is aligned with the other checkboxes because MyFieldFormat is a subclass of TabFieldFormat, which fixes that visual issue.

Of course, you can add custom pages to any dialog by adding "T" records to FRXTabBuilder.DBF just as you would with the native FRXBuilder.DBF.

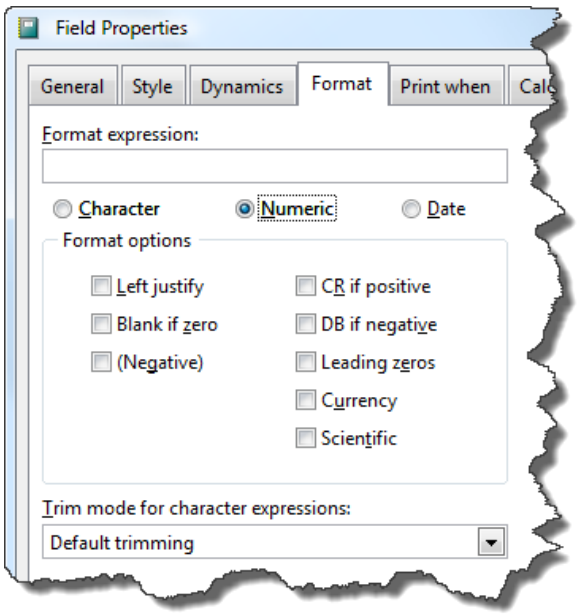


Figure 9. You can easily customize existing pages using FRXTabs.

Fixes in ReportBuilder.APP

Although one of the goals of FRXTabs is to extend ReportBuilder.APP without making any changes to it, there are a couple of places you might want to change in the report builder source code.

- As I mentioned earlier, passing 3 to ReportBuilder.APP to specify a registry table causes DELETED to be set off. Here's the fix for that: just before the SET DELETED OFF statement in FRXBuilder.PRG, add this code:

```
*** DH 11/26/2009: this sets DELETED off when
*** called with "register table" (3) so we'll
*** save and restore the current value
local lSetDeleted
lSetDeleted = set('DELETED') = 'ON'
*** DH 11/26/2009: end of new code
```

  Add this just after the SET ESCAPE ON statement later in the code:

```
*** DH 11/26/2009: restore DELETED
if m.lSetDeleted
    set deleted on
endif m.lSetDeleted
*** DH 11/26/2009: end of new code
```

- In my blog (http://tinyurl.com/ykukhhe), I discussed the fix for a bug that causes Print Environment to be set on accidentally. You might as well fix that one too.

Note that if you're using Windows Vista or later versions, you may wish to copy the source code from Tools\XSource\VFPSource\ReportBuilder to another folder and make the changes there, since everything under the VFP program folder is read-only due to Windows security.

After making these changes, rebuild ReportBuilder.APP and copy it to the VFP home directory.

Summary

FRXTabs is an add-on for the VFP Report Designer that allows you to easily customize the various properties dialogs. You can alter or remove controls, add new controls, remove or rearrange the order of pages, and so forth. FRXTabs gives you both the power of subclasses and the advantages of data-driven design to provide you with almost complete control over the appearance and behavior of the report designer dialogs.

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; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of "Making Sense of Sedna and SP2," the "What's New in Visual FoxPro" series (the latest being "What's New in Nine"), "Visual FoxPro Best Practices For The Next Ten Years," and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).