Extending the VFP 9 Reporting System, Part I: Design-Time

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

Overview

Among the new and improved features in the reporting system of VFP 9 is the ability to extend the Report Designer to provide easier-to-use, more powerful, and more flexible report writing to your development team and even your end-users. In this document, you will learn about the new Report Builder application, how it captures and handles events raised by the Report Designer, and how you can create your own handlers to extend the VFP Report Designer in ways you never thought possible.

Introduction

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

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

- Simplifying and improving the UI. In VFP 8 and earlier, there were a lot of dialogs related to reporting. Some of them had somewhat unusual interfaces, and some spawned yet other dialogs. You can see an example of the clunky interface when you double-click on a text object: that action brings up a properties dialog for the object, but that dialog doesn't allow you to change the font or color of the object. In VFP 9, there are only a few dialogs, because all of the properties for an object are now in one place.

- New features. Features such as report and object protection, DataEnvironment reuse, absolute positioning, and design-time labels are now available in VFP 9.

- Extendible. VFP has always had a very extendible IDE, but VFP 9 blows the lid off extensibility! You can now replace some or all Report Designer dialogs and completely change both the UI and behavior of report events if you wish.

Design-Time Report Events

Earlier versions of VFP provided very little means of customizing the Report Designer, other than perhaps the appearance of the window it appeared in. One of the design goals for VFP 9 is to provide a mechanism to hook into the behavior of the Report Designer so its appearance and behavior can be customized as much as possible. This was implemented by having events in the Report Designer, such as opening a report or adding an object, passed to an Xbase component, which can take any action necessary when events occur.

The new system variable _REPORTBUILDER points to an application (referred to in this document as "the report builder application") that receives notification about events from the Report Designer. When a design-time report event occurs and _REPORTBUILDER points to an existing application, the Report Designer creates a private data session and opens a copy of the FRX currently being edited in that data session, then calls the report builder application. By default, _REPORTBUILDER is set to ReportBuilder.APP in the VFP home directory, but you can substitute another application if you wish. If you specify a non-existent application, no error occurs but events are not raised in that case. Any application specified by _REPORTBUILDER must be modal in nature because the Report Designer expects to call it and receive a return value indicating what happened.

The Report Designer passes the following parameters to the report builder application when an event occurs:

Parameter	Type	Description
ReturnFlags	N	Passed by reference with an initial value of -1. Used to return values to the Report Designer.
EventType	N	An integer representing the event that occurred.
CommandClauses	O	An Empty object with properties indicating what clauses were used in the CREATE/MODIFY REPORT command.
DesignerSessionID	N	The data session ID of the Report Designer.

ReturnFlags is used to return values back to the Report Designer. The possible return values are shown in the table below, along with constants representing these values that may be defined in FOXPRO.H in the release version of VFP 9. These values are bit flags that can be summed if desired.

Value	Constant	Description
1	FRX_REPBLDR_HANDLE_EVENT	The event has been handled by the report builder application so the usual action the Report Designer would normally is suppressed (sort of like using NODEFAULT).
2	FRX_REPBLDR_RELOAD_CHANGES	The report builder application made changes in the FRX cursor so the Report Designer should reload the changes into its internal copy of the FRX.

EventType contains a value identifying the event that occurred. The possible values, along with constants representing them and the type of event (a report event, an object event, or a band event), are shown in the following table. The table also indicates whether the event can be suppressed by adding 1 to the ReturnFlags parameter. “Must delete record” means that since a newly-created object’s record has already been added to the FRX, to suppress the creation of an object, you must delete its record in the FRX cursor and set the ReturnFlags parameter to 3 (the event was handled and changes should be reloaded).

Value	Constant	Description	Type	Can Suppress
1	FRX_BLDR_EVENT_PROPERTIES	A Properties dialog (for the report, a band, or an object) is being invoked, either by double-clicking or by a menu action.	Report, Object, Band	Yes

2	FRX_BLDR_EVENT_OBJECTCREATE	An object or band is being created.	Object, Band	Yes (must delete record)
3	FRX_BLDR_EVENT_OBJECTCHANGE	An object or band is being moved or resized (although it looks like this may only fire for bands).	Object, Band	Yes
4	FRX_BLDR_EVENT_OBJECTREMOVE	An object or band is being removed.	Object, Band	Yes
5	FRX_BLDR_EVENT_OBJECTPASTE	One or more objects are being pasted into the report from the clipboard. This event is fired once for each object being pasted.	Object	Yes (must delete record)
6	FRX_BLDR_EVENT_REPORTSAVE	The report is being saved.	Report	No
7	FRX_BLDR_EVENT_REPORTOPEN	The report is being opened.	Report	No
8	FRX_BLDR_EVENT_REPORTCLOSE	The report is being closed.	Report	Yes
9	FRX_BLDR_EVENT_DATAENV	The data environment is being opened.	Report	Yes
10	FRX_BLDR_EVENT_PREVIEWMODE	The report preview mode is being invoked.	Report	Yes
11	FRX_BLDR_EVENT_OPTIONALBANDS	The Optional Bands dialog is being invoked.	Report	Yes
12	FRX_BLDR_EVENT_DATAGROUPING	The Data Grouping dialog is being invoked.	Report	Yes
13	FRX_BLDR_EVENT_VARIABLES	The Variables dialog is being invoked.	Report	Yes
14	FRX_BLDR_EVENT_EDITINPLACE	Ctrl-E was pressed on a label object.	Object	Yes
15	FRX_BLDR_EVENT_SETGRIDSCALE	The Grid Scale dialog is being invoked.	Report	Yes
16	FRX_BLDR_EVENT_OBJECTDROP	One or more objects are being created by a drag-and-drop operation from the	Object	Yes (must delete record)

		DataEnvironment, a DBC, or the Project Manager. This event fires once for each object being created, and if a label is created, once for each label as well.		
17	FRX_BLDR_EVENT_IMPORTDE	“Load data environment” was selected from the Report menu.	Report	Yes
18	FRX_BLDR_EVENT_REPORT	The report is to be printed.	Report	Yes
19	FRX_BLDR_EVENT_QUICKREPORT	“Quick Report” was selected from the Report menu.	Report	Yes

CommandClauses has the properties shown in the following table.

Property	Type	Description
AddTableToDE	L	.T. if the Add Table to DataEnvironment option was turned on in the Quick Report dialog.
Alias	L	.T. if the Add Alias option was turned on in the Quick Report dialog or the ALIAS clause was specified in the CREATE REPORT FROM command.
FieldList	O	A collection of numeric field numbers representing the fields specified in the Quick Report dialog or the FIELDS clause of the CREATE REPORT FROM command.
File	C	The file name of the FRX open in the Report Designer. This file may not actually exist if CREATE REPORT/LABEL was used.
Form	L	.T. if a form layout was chosen in the Quick Report dialog or the FORM clause was specified in the CREATE REPORT FROM command; .F. if a column layout was chosen or COLUMN was specified.
From	C	Contains the table name specified in the CREATE REPORT FROM command.
InScreen	L	.T. if the IN SCREEN clause was specified in the CREATE/MODIFY REPORT/LABEL command.
InWindow	C	The name of the window specified in the IN <window> clause of the CREATE/MODIFY REPORT/LABEL command.
IsCreate	L	.T. if the command was CREATE REPORT/LABEL or .F. for MODIFY

		REPORT/LABEL.
IsReport	L	.T. if the command was CREATE/MODIFY REPORT or .F. if it's CREATE/MODIFY LABEL.
NoEnvironment	L	.T. if the NOENVIRONMENT clause was specified in the MODIFY REPORT/LABEL command.
NoOverwrite	L	.T. if the NOOVERWRITE clause was specified in the CREATE REPORT FROM command.
NoWait	L	.T. if the NOWAIT clause was specified in the CREATE/MODIFY REPORT/LABEL command.
Protected	L	.T. if the PROTECTED clause was specified in the CREATE/MODIFY REPORT/LABEL command.
Save	L	.T. if the SAVE clause was specified in the CREATE/MODIFY REPORT/LABEL command.
Titles	L	.T. if the Titles option was turned on in the Quick Report dialog or the TITLES clause was specified in the CREATE REPORT FROM command.
Width	N	The number of columns specified in the CREATE REPORT FROM command.
Window	C	The window name specified in the WINDOW <name> clause of the CREATE/MODIFY REPORT/LABEL command.

The report builder application runs within a private datasession that contains the FRX cursor. The Report Designer passes its own datasession ID to the report builder application in case it needs to access the tables open in the DataEnvironment of the Report Designer.

The FRX cursor the Report Designer creates for the report builder application has the alias "FRX". The record pointer is on the record for the object the event occurs for; this may be the report header record (the first record in the cursor) if it's a report event rather than an object event. The records for any objects selected in the Report Designer have the CURPOS field set to .T. There's one slight complication with this: since CURPOS is used by the report header record to store the value of the Show Position setting, you should ignore this record when looking at records with CURPOS set to .T. For example, to count the number of selected objects, use:

```
count for CURPOS and recno() > 1
```

ReportBuilder.APP

By default, _REPORTBUILDER is set to ReportBuilder.APP in the VFP home directory. This application provides a framework for handling design-time report events, plus provides a new set of more attractive and functional dialogs that replace the native ones used by the Report Designer. ReportBuilder.APP can be distributed with your applications to provide its behavior in a runtime environment.

In addition to being called automatically by the Report Designer, you can call ReportBuilder.APP manually to change its behavior for the current VFP session (it doesn't write settings to any external location, such as a table, INI file, or the Windows Registry, so, with one exception we'll see in a moment, state isn't preserved from session to session).

- If you call it with no parameters or pass it 1, it'll display an options dialog in which you can change the behavior of the report builder application. You can also right-click in the Properties dialog and choose Options to launch the Options dialog.

- Pass 2 and optionally the name of an FRX file to browse a report.

- Pass 3 and the name of a DBF file to use as the handler registry (we'll discuss this later).

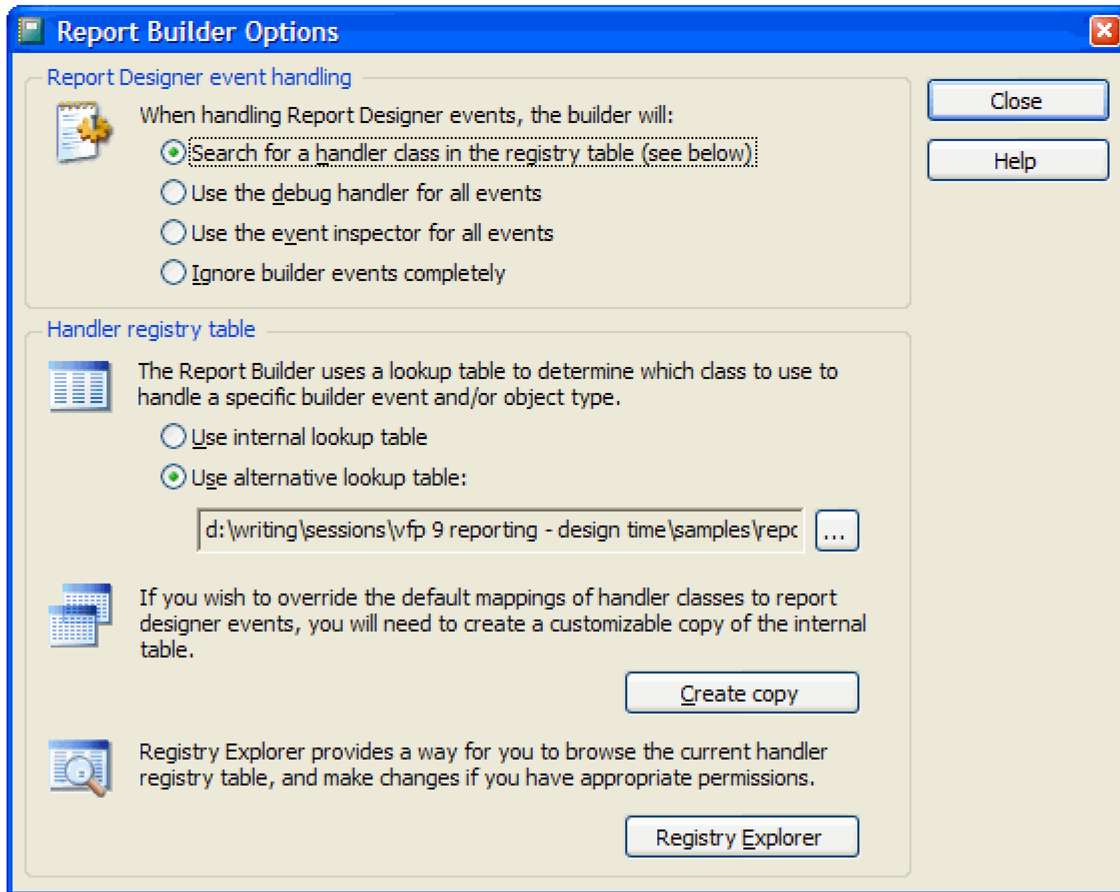
- Pass 4 and a numeric value to set the "handle mode". The numeric values match the buttons in the When handling Report Designer events, the builder will setting shown in the options dialog. For example, DO (_REPORTBUILDER) WITH 4, 3 tells ReportBuilder to use the event inspector.

- Pass 5 and optionally the name and path of a file to create a copy of the internal handler registry table (if you don't pass the second parameter, you'll be prompted for the table to create).

The Options dialog has the following features:

- You can define what happens when ReportBuilder.APP receives a report event. The choices are to search for a handler class in the handler registry table (the default behavior), use a "debug" handler for events (displays a dialog showing the FRX in a grid and allowing modifications to it and other settings), use an "event inspector" handler (displays information about the event and the FRX in a MESSAGEBOX() window), or ignore report events.

- You can specify what handler registry table should be used, or make a copy of the internal one (we'll discuss the handler registry later). You can also browse the selected registry table.



There are several ways you can extend the functionality of the Report Designer.

- You can replace ReportBuilder.APP with your own report builder application by changing _REPORTBUILDER.

- You can wrap ReportBuilder.APP by changing _REPORTBUILDER to your own application and having your application call back to ReportBuilder.APP. For those of you who used FoxPro 2.x, this may remind you of the GENSCRNX approach.

- You can register event handling objects in ReportBuilder.APP's registry table. I suspect this will be the most popular choice because ReportBuilder.APP provides a report builder framework and lets you simply focus on handlers for report events.

Registering Report Event Handlers

If you want to create your own report event handlers that will be called by ReportBuilder.APP when a report event fires, there are two steps required. The first is to create a class that implements the desired behavior. It must have an Execute method that accepts an event object as a parameter, because that's what ReportBuilder.APP expects. The second is to register the handler in ReportBuilder.APP's handler registry table. We'll look at the second task first, then spend the rest of this document discussing the first.

When a report event occurs, ReportBuilder.APP looks in a handler registry table to find a handler for the event. By default, it uses the handler registry table built into the APP file. This table provides handlers for many report events so the new Xbase Report Designer dialogs are used rather than the native ones. However, if you want to register your own handlers, you have to tell ReportBuilder.APP to use a different handler registry table. There are several ways to do this:

- DO (_REPORTBUILDER) and click on the Create Copy button to write the internal handler registry table to an external one. By default, this copy will be named ReportBuilder.DBF. When ReportBuilder.APP starts, it looks for a table called ReportBuilder.DBF in the current directory or VFP path. If it finds such a table, it uses that table rather than the internal one. So, simply creating this copy means ReportBuilder.APP will use it without having to do anything else.
- DO (_REPORTBUILDER) and click on the Choose button to select the table to use.
- As mentioned earlier, DO (_REPORTBUILDER) WITH 3, <DBF to use> to specify the desired handler registry table without any UI.

Once you've specified an external handler registry table, you can manually add or edit records in that table or, in the ReportBuilder.APP Options dialog, click on the Explore Registry button and edit the records in the resulting dialog.

The handler registry table has the following structure:

Field Name	Type	Values	Description
REC_TYPE	C(1)	E, F, G, H, or X	Specifies the type of record (discussed later): • H: report event handler record • F: report event filter record • X: exit handler record • G: GetExpression wrapper record • E: run-time extension editor class
HNDL_CLASS	C(25)		Class name.
HNDL_LIB	C(25)		Class library.
EVENTTYPE	I	0 - 19 or -1	Report event type ID (-1 means any event).
OBJTYPE	I	0 - 10 or -1	Report object type (-1 for any type).

OBJCODE	I	0 - 26 or -1	Report object code (-1 for any code).
NATIVE	L	.T. or .F.	.T. to force the report event to be passed back to the Report Designer for native behavior.
DEBUG	L	.T. or .F.	.T. to force the debug handler to be used for this event/object combination.
FLTR_ORDR	C(1)	" ", "1", "2", ..	For filters and exit handlers only, specifies the order in which they are applied.
NOTES	C(35)		Not used by ReportBuilder.APP.

When a report event occurs, ReportBuilder.APP looks for the handler class to instantiate by looking for a record where EVENTTYPE matches the report event ID (or is -1), OBJTYPE matches the OBJTYPE column of the selected FRX record (or is -1), and OBJCODE matches the OBJCODE column of the selected FRX record (or is -1). Because ReportBuilder.APP uses the first handler record it finds that meets its conditions, you may need to delete or disable built-in handlers if you wish to implement your own. One way you can disable a handler record without deleting it is to change EVENTTYPE to an invalid value. I like to add 100 to EVENTTYPE to disable a record because I can easily re-enable it by subtracting 100.

As you can see in the table, REC_TYPE registers different types of records. Here are the different types available:

- A report event handler handles a report event.
- A report event filter is a class that gets an earlier crack at the report event than a handler does. While only a single handler is instantiated, ReportBuilder.APP instantiates the classes specified in all filter records, in FLTR_ORDER order, and calls their Execute methods. Upon return from Execute, if any filter object's AllowToContinue property is .F., no further processing happens and the Report Designer is informed that the event has been handled.

  As you can see from this description, although they can both respond to report events, there's a big difference between event handlers and filters:

  - Filters are instantiated on every report event, while a handler is only instantiated for the event it's registered for in the handler registry table.
  - All filters are instantiated on an event, while only a single handler is.

  This means filters are good for the behavior you want to occur on multiple, possibly all, events, while handlers are specific for one type of event.

- Exit handlers are similar to a combination of filters and event handlers in that after the other processing is done, ReportBuilder.APP runs all registered exit handlers by instantiating the appropriate classes, in FLTR_ORDER order, and calling their Execute methods. These handlers are really just intended to perform any post-event cleanup behavior.

- GetExpression wrappers provide a wrapper or replacement for the GETEXPR dialog.

- Run-time extension editors replace the dialog displayed when you click on the Edit Settings button for the Run-time Extension property in the Other page of the Properties dialog for an object.

The Report Event Handling Process

When ReportBuilder.APP is called because a report event occurred, it does the following:

- Runs any registered filters as described earlier. Processing stops if any of them have AllowToContinue set to .F.

- Tries to find the handler for the event using the following search pattern:

    REC_TYPE="H", EVENTTYPE, OBJTYPE, OBJCODE

    REC_TYPE="H", EVENTTYPE, OBJTYPE, -1

    REC_TYPE="H", EVENTTYPE, -1, -1

    REC_TYPE="H", -1, -1, -1

- If a handler is found, ReportBuilder.APP instantiates it and calls its Execute method. If no handler is found, the event is passed back to the Report Designer and native behavior is used instead.

- If a handler was found, ReportBuilder.APP runs all registered exit handlers as described earlier.

Handler Interfaces

Report event filters, event handlers, exit handlers, GetExpression wrappers, and run-time extension editors can be based on any class and have only a single required method. The method signatures are:

- Filters: Execute(toEvent). ReportBuilder.APP passes this method a reference to an event object, which we'll look at later, with properties containing information about the event. To prevent further processing, set the custom AllowToContinue property to .F. in this method; in that case, the ReturnFlags property of the event object should be set appropriately.

- Event handlers: Execute(toEvent). This method also receives an event object. The return value is not important, but the ReturnFlags property of the event object is.
- Exit handlers: Execute(). The return value is not important.
- GetExpression wrapper: GetExpression(tcDefaultExpr, tcDataType, tcCalledFrom, toEvent). tcDefaultExpr is the default expression for the dialog and tcDataType is the data type of the expression. tcCalledFrom indicates where this method was called from: "PrintWhenExpression", "FieldExpression", "OLEBoundField", "OLEBoundExpression", "BandGroupOnExpression", "VariableValueToStore", or "VariableInitialValue". toEvent is the same event object passed to other handlers. The return value is the expression.
- Run-time extension editors: same signature as event handlers.

Event Object

ReportBuilder.APP passes an event object to the methods of handlers. This object includes as properties the parameters passed by the Report Designer to ReportBuilder.APP, plus some other useful information.

Property	Type	Description
BuilderPath	C	Path of ReportBuilder.APP.
CommandClauses	O	The same CommandClauses object passed to ReportBuilder.APP from the Report Designer.
DefaultRecno	I	The record pointer in the FRX cursor.
DefaultSessionID	I	The data session of the Report Designer (the fourth parameter passed in from the Report Designer).
EventType	I	The event type (the second parameter passed in from the Report Designer).
FRXCursor	O	A helper object containing useful functions for interacting with the FRX cursor.
FRXSessionID	I	The data session in which the FRX cursor is open (the default session when the event handler is instantiated).
HandleMode	I	Indicates how ReportBuilder.APP handles events: 1 means search for a handler class in the registry table, 2 means use the debug handler, 3 means use the Event Inspector, and 4 means ignore builder events.

MultiSelect	L	.T. if multiple objects are selected in the Report Designer.
ObjCode	I	The value of the OBJCODE field of the selected record in the FRX cursor.
ObjType	I	The value of the OBJTYPE field of the selected record in the FRX cursor.
Protected	L	.T. if the Report Designer was launched with the PROTECTED keyword.
ReturnFlags	I	The value of this property is returned to the Report Designer in the first parameter passed in from the Report Designer. It's initially set to 1 (FRX_REPBLDR_HANDLE_EVENT); set it to 3 (FRX_REPBLDR_HANDLE_EVENT + FRX_REPBLDR_RELOAD_CHANGES) if your class makes changes to the FRX cursor that need to be reloaded into the layout.
SelectedObjectCount	I	The number of selected objects in the report layout, determined by counting CURPOS = .T. in the FRX cursor (not counting the header record).
SessionData	O	A reference to a Name-Value pair manager object used to store data between ReportBuilder.APP invocations.
UniqueID	C	The value of the UNIQUEID field of the selected record in the FRX cursor.
UnitConverter	O	A reference to an FRX unit converter object (the FRXUnitConverter class in FRXBuilder.VCX built into ReportBuilder.APP). This class has methods to convert between FRU (FoxPro Report Units, 1/10000 inch) and other units.
UsingInternalRegistry	L	.T. if the internal registry table is used.

It has several public methods; however, some are used by ReportBuilder.APP rather than an event handler. The ones useful for an event handler are:

Method	Parameters	Returns	Description
GetEventTypeText	tiEvent	C	Returns the name of a given event type.
GetExpression	tcDefaultExpr [, tcDataType [, tcCalledFrom]]	C	Displays a Get Expression dialog and returns the selected expression.
GetExtensionEditor	None	O	Returns a reference to an run-time extension editor class as specified in the handler registry table.

GetTargetTypeText	tiObjType, tiObjCode	C	Returns the name of a given object type/object code. This actually just wraps the GetTargetTypeText() method of the FRXCursor object.
SetHandledByBuilder	tlNoDefault	.T.	Pass .T. to add 1 to ReturnFlags.
SetReloadChanges	tlReload	.T.	Pass .T. to add 2 to ReturnFlags.

FRXCursor Helper Object

This object, referenced in the FRXCursor property of the event object, provides methods you may find useful when working with an FRX.

Method	Parameters	Returns	Description
BinStringToInt	tcBytes	N	Returns the numeric equivalent of binary data in a string of bytes.
BinToInt	tcValue	I	Converts a binary string into an integer.
CreateBandCursor	[tcFRXAlias]	L	Creates a cursor with the alias “Bands” containing useful information about the bands in the report.
CreateCalcResetOnCursor	[tcFRXAlias]	L	Creates a cursor with the alias “Reset_On” containing information for each option in the Calculation Reset combobox.
CreateDefaultPrintEnvCursor	[tcFRXAlias [, tcDestAlias]]	L	Creates a 1-record cursor named “DefPrnEnv” with the same structure as the FRX, with the default printer environment data loaded into EXPR, TAG, and TAG2.
CreateGroupCursor	[tcFRXAlias]	L	Creates a cursor with the alias “Groups” containing useful information about the report data grouping.
CreateMemberDataCursor	[tcFRXAlias [, tcMDAlias]]	L	Creates a 1-record cursor of report MemberData attributes, extracting and converting the XML from the currently selected FRX record’s STYLE field.
CreateObjectCursor	[tcFRXAlias [, tiOption]]	L	Creates a cursor with the alias “Objects” containing useful information about the layout objects in the report. Calls

			CreateBandCursor() if necessary. The values for tiOption are: 0 (the default): all objects in the FRX, ignoring grouped items 1: only selected (CURPOS = .T.) records 2: all objects, showing grouped items as single record 3: grouped item breakdown Some records may be deleted. Use SET DELETED ON or manually ignore deleted records.
CreateVariableCursor	[tcFRXAlias]	L	Creates a cursor with the alias "Vars" containing useful information about the report variables defined in the report.
FRUToPixels	tnFRU	I	Returns the corresponding pixels for a given number of FRU (FoxPro Report Units, 1/10000 inch).
GetBandFor	tcUniqueID [, tlStart]	O	Returns an object containing information about the start or end band that surround the given report object.
GetFRUTextHeight	tcText, tcTypeFace, tiSize [, tcStyle]	N	Returns the height in FRU of given text and font spec.
GetFRUTextWidth	tcText, tcTypeFace, tiSize [, tcStyle]	N	Returns the width in FRU of a given text and font spec.
GetFRXTimeStamp	ttDateTime	I	Returns a FoxPro time stamp for a given datetime.
GetObjectsInBand	tcBandID [, tlRecnos]	O	Returns a collection of UNIQUEID values or record numbers (depending on the value of the second parameter) for each object contained in the specified band.
GetReportAttribute	tcAttrib [, tiAlternate]	Value	Returns a given attribute of the report header. Some attributes have associated information that can be obtained by passing an optional

			second parameter of 1.
GetSelectedObjectCount	None	I	Returns the number of currently selected objects (CURPOS=.T.) in the FRX cursor.
GetTargetTypeText	tiObjType, tiObjCode	C	Returns the name of a given object type/object code.
GetTimeStampString	tiTimeStamp	C	Converts a FoxPro time stamp into a readable string.
HasBand	tiObjCode	L	Returns .T. if the report contains a band of the given type.
HasDetailHeader	tcDetailBandID	L	Returns .T. if the specified detail band record has an associated detail header record.
HasProtectionFlag	tcBytes, tiFlag	L	Returns .T. if the given binary data (bytes) has a specific protection flag set.
InsertBand	tiObjCode	-	Inserts a band of a given type into the FRX cursor. Assumes the FRX cursor is selected and is positioned to the correct location for an insert.
InsertDetailBand	None	-	Inserts a Detail band record into the FRX cursor. Assumes the FRX cursor is selected and is positioned to the appropriate record to have the detail record inserted.
InsertDetailHeaderFooter	None	-	Inserts Detail Header/Footer records into the FRX cursor. Assumes the FRX cursor is selected, is positioned to the detail band record to have Header/Footer records added to it, and that the detail band doesn't already have header/footer records.
InsertSummaryBand	tlNewPage, tlPageHeader, tlPageFooter	-	Inserts a Summary band into the FRX cursor. Assumes the FRX cursor is selected and doesn't already have a Summary band.
InsertTitleBand	tlNewPage	-	Inserts a Title band into the FRX cursor. Assumes the FRX cursor is selected and doesn't already have a Title band.
IntToBin	tiValue	C	Converts an integer into a binary string.

IntToBinString	tiValue	C	Returns a string of bytes of the binary version of an integer.
PixelsToFRU	tnPixels	N	Returns the corresponding FRU for a given number of pixels.
StoreMemberDataCursor	[tcFRXAlias [, tcMDAlias]]	L	Takes a 1-record cursor of report MemberData attributes, converts the attributes in the MemberData XML, and stores it in the currently selected FRX record's STYLE field.
SynchObjectPositions	None	L	Resets non-deleted objects in the FRX relative to the start of the band they are in. Assumes Bands and Objects cursors have been created, current selected table is the FRX cursor, and does not need to restore the record pointer.

Creating Report Event Handlers

Because most report event handlers will have common requirements, I created a base class handler called SFReportEventHandler (a subclass of SFCustom, my Custom base class defined in SFCtrls.VCX), defined in SFReportBuilder.VCX. Its Init method instantiates an SFReportEventUtilities object into the oUtilities property. Like FRXCursor, this object has utility methods for dealing with report objects and events.

The Execute method of SFReportEventHandler saves the passed-in event object to its oEvent property and the oEvent property of the SFReportEventUtilities object, then calls the OnExecute method, which is abstract in this class. In a subclass, I won't override the Execute method, but will instead put the appropriate code into the OnExecute method.

```
lparameters toEvent
with This
  .oEvent             = toEvent
  .oUtilities.oEvent = toEvent
  .OnExecute()
  .oEvent             = .NULL.
  .oUtilities.oEvent = .NULL.
endwith
```

To handle a particular report event, create a subclass of SFReportHandler and register it in the handler registry table. To make it easy to do the latter, I created a program called InstallHandler.PRG. Pass it the class and library for the handler, the event number, and optionally the object type and code. It adds a record to the registry table (expected to be named ReportBuilder.DBF; change the USE and INSERT INTO statements to use a different table name) if it doesn't exist, and disables any other handlers for the same event.

```
lparameters tcClass, ;
  tcLibrary, ;
```

```
  tnEventType, ;
  tnObjType, ;
  tnObjCode
local lcClass, ;
  lnObjType, ;
  lnObjCode

* Open the report builder table.

use ReportBuilder

* If the specified handler is already there, ensure it's enabled by
* setting EventType to the proper value. Otherwise, add a record for
* it, using defaults for the object type and code if they weren't
* passed.

lcClass = padr(upper(tcClass), len(Hndl_Class))
locate for upper(Hndl_Class) == lcClass
if found()
  replace EventType with tnEventType
else
  lnObjType = iif(vartype(tnObjType) = 'N', tnObjType, -1)
  lnObjCode = iif(vartype(tnObjCode) = 'N', tnObjCode, -1)
  insert into ReportBuilder ;
      (Rec_Type, ;
      Hndl_Class, ;
      Hndl_Lib, ;
      EventType, ;
      ObjType, ;
      ObjCode) ;
    values ;
      ('H', ;
      tcClass, ;
      tcLibrary, ;
      tnEventType, ;
      lnObjType, ;
      lnObjCode)
endif found()

* Disable any other handlers for the same event by setting their
* EventType code to an used value.

replace EventType with EventType + 100 ;
  for EventType = tnEventType and ;
  not upper(Hndl_Class) == lcClass

* Clean up and exit.

use
```

Report Templates

When you create a new report, you get a blank report. Wouldn't it be nice if VFP would automatically add certain common elements you want in every report? In other words, we'd like to have a report that's used as the template for all new reports.

Back in the FoxPro 2.6 days, we actually had this capability (although it was undocumented): since FoxPro always created a new report with the name Untitled if you didn't specify a name, if

you had a report called Untitled.FRX, VFP would open it, but prompt you for a new name when you saved the new report for the first time. Unfortunately, this trick doesn't work in VFP. First, if you don't specify a name, you get a default name of ReportN, where N is a number that increments from 1 every time a report is created in a particular VFP session. Second, even if a report named Report1.FRX exists and N will be 1 because this is the first time you've used CREATE REPORT since starting VFP, VFP doesn't open that report but gives you a blank report.

Now, with design-time report events, we can have report templates. There isn't an event that fires when a report is created, but there is when one is opened, so we simply have to check if the report is a new one or not (the utility method IsNewReport returns .T. if that's the case). If it's a new report, we ZAP existing records in the FRX, APPEND FROM a template FRX file, and set the return flag to indicate that the event was handled and the FRX was changed. Here's the code in the OnExecute method of SFNewReportHandlerBasic:

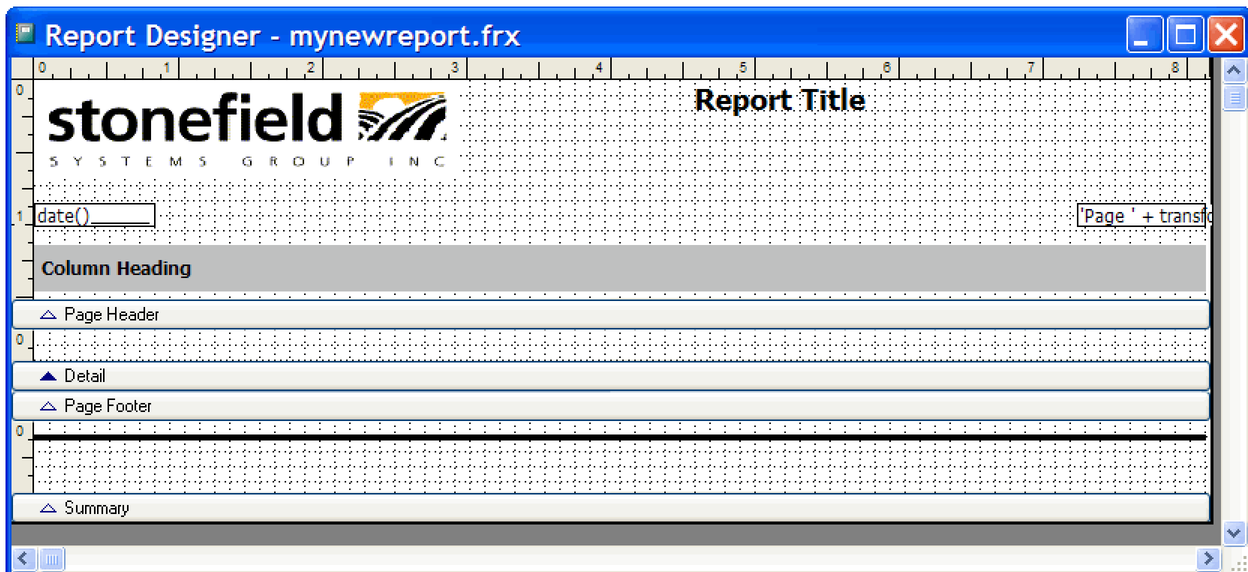
```
local lnSelect, ;
  lnRecno
if This.oUtilities.IsNewReport()
  lnSelect = select()
  select FRX
  zap
  lnRecno = recno()
  append from (This.cTemplateReport)
  This.oEvent.ReturnFlags = FRX_REPBLDR_HANDLE_EVENT + ;
    FRX_REPBLDR_RELOAD_CHANGES
  go lnRecno
  select (lnSelect)
endif This.oUtilities.IsNewReport()
```

The custom property cTemplateReport contains the name of the template FRX to use. By default, it contains Template.FRX. To register this class, run InstallNewReportHandlerBasic.PRG.

Let's get even fancier: how about asking the user which template they'd like to use? SFNewReportHandlerFancy is a subclass of SFNewReportHandlerBasic with the following code in OnExecute:

```
local loForm
if This.oUtilities.IsNewReport()
  loForm = newobject('SFSelectTemplateForm', 'SFReportBuilder.vcx')
  loForm.Show()
  if vartype(loForm) = 'O'
    This.cTemplateReport = loForm.cTemplate
  endif vartype(loForm) = 'O'
endif This.oUtilities.IsNewReport()
return dodefault()
```

This code uses the SFSelectTemplateForm class to display a list of available templates. This list comes from Templates.DBF, which has fields containing the name of the template FRX, a descriptive name for the template, and a memo containing comments about the template. To register this class, run InstallNewReportHandlerFancy.PRG.



Custom Dialog for New Fields

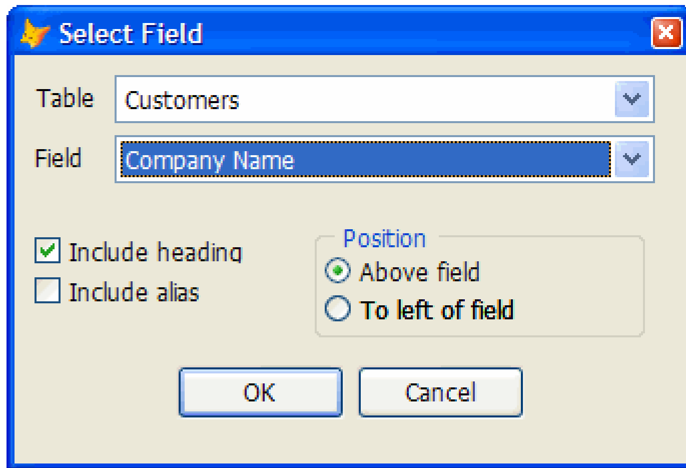
One of the things I've always wanted to do was replace the dialog that appears when a user adds a new field to a report. I want a dialog that is both simpler (it displays descriptive names for tables and fields) and more powerful (it doesn't require the tables to be in the DataEnvironment or open and it has options for adding a label to go along with the field). Since I can now take over the "new field" report event, I can finally create the dialog I want.

SFNewTextBoxHandler is registered as the handler for new fields with this line of code (taken from TestNewField.PRG):

```
do InstallHandler with 'SFNewTextBoxHandler', 'SFReportBuilder.vcx', 2, 8, 0
```

TestNewField.PRG also creates a meta data object that has collections of tables and fields read from a meta data table. We won't look at that object here; feel free to examine it yourself.

When you run TestNewField.PRG, it automatically creates a new report, but nothing else appears different. However, since SFNewTextBoxHandler is now the handler for new fields, when you add a field, you'll get the dialog shown below rather than the usual one.



This dialog displays descriptive names for the tables and fields in the SQL Server Northwind database, which, of course, aren't in the DataEnvironment or open in VFP (they don't need to be since this information comes from meta data). It also allows you to indicate whether a label is created or not, and if so, whether it should be placed in the page header band above the field or in the detail band to the left of the field.

As with other subclasses of SFReportEventHandler, it's the OnExecute method of SFNewTextBoxHandler that does the work. We won't go over all the code in this method, just the more interesting stuff. The first thing this method does is display the dialog shown above. Then, if the user clicks on OK, it retrieves some information about the selected field from the meta data (the data type, size, etc.). It then uses the SFReportEventUtilities object to retrieve the default font, size, and style from the header record in the FRX. Then, it uses the FRXCursor helper object to determine the height and width of the field in FRUs, and updates the field's record in the FRX accordingly.

```
lcFontName  = .oUtilities.GetReportHeaderValue('FONTFACE')
lnFontSize  = .oUtilities.GetReportHeaderValue('FONTSIZE')
lnFontStyle = .oUtilities.GetReportHeaderValue('FONTSTYLE')
lcFontStyle = .oUtilities.GetFontStyle(lnFontStyle)

* Determine the width and height for the textbox.

lnWidth  = .oEvent.FRXCursor.GetFRUTextWidth(lcText,  lcFontName, ;
  lnFontSize, lcFontStyle)
lnHeight = .oEvent.FRXCursor.GetFRUTextHeight(lcText, lcFontName, ;
  lnFontSize, lcFontStyle)

* Update the properties of the new textbox.

replace EXPR  with lcField, ;
  NAME      with lcName, ;
  WIDTH     with lnWidth, ;
  HEIGHT    with lnHeight ;
  PICTURE   with iif(empty(lcPicture), '', '"' + lcPicture + '"'), ;
  FILLCHAR  with lcType, ;
  FONTFACE  with lcFontName, ;
  FONTSIZE  with lnFontSize, ;
  FONTSTYLE with lnFontStyle, ;
```

```
  USER      with 'EXPR=' + lcExpr + chr(13) + chr(10) ;
  in FRX
```

The next thing it does is determine where to put the label object for the field. If it's supposed to go in the page header, the SFReportEventUtilities object is asked to find a label in the page header band that has "*:TEMPLATE" in its USER memo. If such an object exists, it's used as the template for the new label (font, style, vertical position, etc.). If "REMOVE" also appears, the template object is removed from the report.

```
if lnPosition = 1
  loBand     = .oUtilities.GetBandObject(FRX_OBJCOD_PAGEHEADER)
  loTemplate = .oUtilities.FindTemplateObject(loBand, ;
    FRX_OBJTYP_LABEL, '*:TEMPLATE', .T.)
  if vartype(loTemplate) = 'O'
    if '*:TEMPLATE REMOVE' $ upper(loTemplate.User)
      .oUtilities.RemoveReportObject(loTemplate.UniqueID)
      loTemplate.User = strtran(loTemplate.User, ;
        '*:TEMPLATE REMOVE', '*:TEMPLATE')
    endif '*:TEMPLATE REMOVE' $ upper(loTemplate.User)
    loObject = loTemplate
    lnVPos   = loObject.VPos
  else
    lnVPos = loBand.Stop - lnHeight - BAND_SEPARATOR_HEIGHT_FRUS
  endif vartype(loTemplate) = 'O'
```

If the label is supposed to go in the same band as the field, the SFReportEventUtilities object is asked to find a band in the report, then like the previous code, to look in that band for a label with "*:TEMPLATE" in its USER memo and use it as the template.

```
else
  loBand     = .oEvent.FRXCursor.GetBandFor(FRX.UniqueID)
  loTemplate = .oUtilities.FindTemplateObject(loBand, ;
    FRX_OBJTYP_LABEL, '*:TEMPLATE', .T.)
  if vartype(loTemplate) = 'O'
    if '*:TEMPLATE REMOVE' $ upper(loTemplate.User)
      .oUtilities.RemoveReportObject(loTemplate.UniqueID)
      loTemplate.User = strtran(loTemplate.User, ;
        '*:TEMPLATE REMOVE', '*:TEMPLATE')
    endif '*:TEMPLATE REMOVE' $ upper(loTemplate.User)
    loObject = loTemplate
  endif vartype(loTemplate) = 'O'
endif lnPosition = 1
```

Finally, a new label is added to the report.

```
with loObject
  .UniqueID  = sys(2015)
  .TimeStamp = This.oEvent.FRXCursor.GetFRXTimeStamp(datetime())
  .Name      = ''
  .Expr      = '"' + lcCaption + '"'
  .Height    = lnHeight
  .Width     = lnWidth
  .VPos      = lnVPos
  .HPos      = lnHPos
  .ObjType   = FRX_OBJTYP_LABEL
endwith
insert into FRX from name loObject
```

This is the most complicated event handler we've seen, because it has to deal with more things in the FRX. Doing this type of work requires a fair bit of knowledge about the structure of an FRX, so be sure to check out the 90FRX report in the Tools\FileSpec subdirectory of the VFP home directory for documentation on the FRX.

By the way, this sample shows another cool new feature in VFP 9: design-time labels. Notice that when you add a field to the report, it displays the caption rather than the field name in the field object. That's because the code above fills in the NAME column of the field object in the FRX with the caption for the field. When you use CREATE or MODIFY REPORT with the PROTECTED keyword, the Report Designer will display the contents of the NAME column rather than the EXPR column for field objects. This means you can display nice descriptive names for fields rather than the actual field names. Of course, they'll see the actual field names in the Properties dialog, but at least they'll see the nice names on the design surface.

Generating Cursors on the Fly

Since we can now create a report from meta data using the SFNewTextBoxHandler handler, what happens when we try to preview the report? It won't work, because the cursors haven't been opened. Ah, but what if we hooked into the preview event and generated the cursors before the report is run?

SFPreviewHandler gets registered as the handler for events 10 (previewing) and 18 (printing) by InstallPreview.PRG. It's OnExecute method is fairly simple: it generates a SQL SELECT statement by looking at the fields in the report (we won't look at the code for the CreateSQLStatement method here), instantiates an object that opens a connection to the SQL Server Northwind database, sends the SQL SELECT statement to SQL Server to create a cursor, and tells the report engine to cancel the preview or print if we failed for some reason.

```
local lcSelect, ;
  loConnection, ;
  llOK

* Create the SQL SELECT statement we need for the report.

wait window 'Retrieving data...' nowait
lcSelect  = This.CreateSQLStatement()

* Create a connection object and try to connect to the SQL Server Northwind
* database. If we succeeded, execute the SQL SELECT statement in the report's
* datasession.

loConnection = newobject('SFConnectionMgr', 'SFConnection.vcx')
loConnection.cConnectString = 'driver=SQL Server;server=(local);' + ;
  'database=Northwind;trusted_connection=yes'
if loConnection.Connect()
  llOK = loConnection.ExecuteStatement(lcSelect, ;
    This.oEvent.DefaultSessionID, sys(2015))
endif loConnection.Connect()

* If we failed to connect or create the cursor, display a warning and flag
* that we've handled the event so the preview stops.
```

```
wait clear
if not llOK
  messagebox(loConnection.cErrorMessage)
  This.oEvent.ReturnFlags = FRX_REPBLDR_HANDLE_EVENT
endif not llOK
```

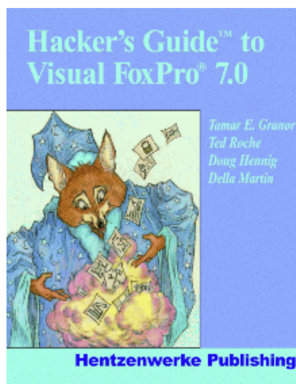
Summary

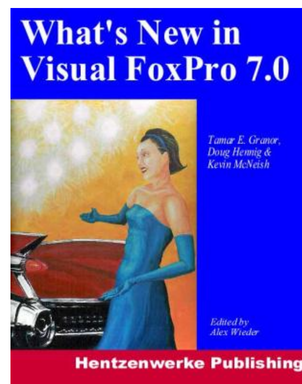
The VFP team certainly met their goals in the improvements they made to the reporting system. The dialogs available in ReportBuilder.APP are more attractive, easier to use, and more capable than those in earlier versions. The ability to hook into design-time report events means you can create customized report designers that are more powerful, flexible, and easier to use, both for your development team and your end users.

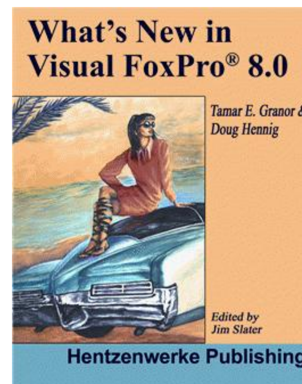
Note: since this document was written during the beta of VFP 9, many of the details described in this document may be different in the release version. Be sure to check my Web site (www.stonefield.com) for updates to this document and the accompanying samples.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT), the award-winning Stonefield Query, and the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro. Doug is co-author of “What’s New in Visual FoxPro 8.0”, “The Hacker’s Guide to Visual FoxPro 7.0”, and “What’s New in Visual FoxPro 7.0”. He was the technical editor of “The Hacker’s Guide to Visual FoxPro 6.0” and “The Fundamentals”. All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). Doug writes the monthly “Reusable Tools” column in FoxTalk. He has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He has been a Microsoft Most Valuable Professional (MVP) since 1996.







Copyright © 2004 Doug Hennig. All Rights Reserved