Building and Using VFP Developer Tools

By Doug Hennig

Introduction

In addition to being the best darn database management system available, VFP also gives us the ability to highly customize the interactive development environment (IDE). You can modify the menu, install new builders and wizards, implement developer toolbars, change the behavior of the Project Manager, and do lots of other things to make your IDE more productive. Even better is that all of this customization is done in the language we all know and love, so we don't have to have some C++ guru create add-ins for us. Even relatively inexperienced VFP developers can customize their environment to work the way they want.

This document will show you how to create simple tools that increase productivity for you and your development team. We'll look at modifying the VFP wizards and builders to provide additional or customized features, and creating your own builders using the BuilderD technology included with VFP 6.

Modifying VFP's Wizards and Builders

If you're like me, you probably haven't used the builders and wizards that come with VFP very much because they don't quite work the way you want them to. Maybe they don't have enough flexibility or maybe you just don't like the way they do something. Until VFP 6, there was no way to change the behavior of the builders and wizards because Microsoft didn't provide the source code. However, now we get the source code for not only most of the builders and wizards, but also the Class Browser, Component Gallery, and Coverage Profiler.

Source code for builders and wizards can be found in XSOURCE.ZIP in the TOOLS\XSOURCE directory under the VFP home directory. When you unzip this file, it creates a VFPSOURCE directory under which all the source code exists. Wizards are found in subdirectories of the WIZARDS directory and builders in subdirectories of BUILDERS (although some common files located in WIZARDS are used by some builders).

So, now that we have source code, we can change the builders and wizards to do what we want, right? Well, a better approach is to create new builders and wizards that use the majority of the classes and programs of the originals but override their behavior by subclassing classes and either cloning and modifying PRGs or creating wrapper PRGs. This document will detail exactly how to do this for each of the builders and wizards we replace.

Once you've created your replacement builder or wizard, how do you tell VFP to use it rather than the native one? Builders are registered in BUILDER.DBF and wizards in WIZARD.DBF, both of which are in the WIZARDS subdirectory of the VFP home directory. These tables have the same structure, which is shown in Table 1.

Field	Description
NAME	The descriptive name of the builder or wizard.
DESCRIPT	A description for the builder or wizard.
BITMAP	This field appears to be unused.
TYPE	What type of object the builder or wizard is for. In the case of BUILDER.DBF, it's generally the base class for an object (although there are also records with MULTISELECT, AUTOFORMAT, and RI in this field). For WIZARDS.DBF, it's things like FORM, REPORT, and QUERY.
PROGRAM	The name of the APP file containing the builder or wizard.
CLASSLIB	The class to instantiate in the APP file.
CLASSNAME	The class library of the main class in the APP file.
PARMS	Parameters to pass to the builder or wizard.

Table 1. Structure of BUILDER.DBF and WIZARD.DBF

When you invoke a builder, VFP calls the program specified in the _BUILDER system variable (BUILDER.APP by default). BUILDER.APP looks at the environment it's in (for example, which object the builder is being invoked for), looks for a record that matches the environment in BUILDER.DBF (for example, it looks for the base class of the object in the TYPE field), and invokes the registered builder. Wizards are treated the same way, except the system variable is _WIZARD, which points to WIZARD.APP, which looks in WIZARD.DBF.

To tell VFP to use a different builder or wizard, insert a new record into BUILDER.DBF or WIZARD.DBF that describes how to run your builder or wizard. If you want the choice of running the old builder or wizard as well as your new one, leave the record for the original builder or wizard alone. If you want to replace the old one, rather than deleting the original record, simply change its TYPE value to something that will never be recognized (I use a suffix of "X", such as "XGRID"); that way, you can simply restore the former builder by changing its TYPE value back.

We'll take a look at creating replacements for the Grid Builder, Referential Integrity Builder, Upsizing Wizard, and the WWW Search Page Wizard.

Creating a More Useful Grid Builder

The VFP Grid Builder provides a quick way to populate the columns of a grid and create the visual appearance you want. However, I can think of several things I don't like about the builder:

- It doesn't size the columns it creates properly. You have to manually size them to fit the data.

- The control type combo box on the Layout page of the builder lists VFP base classes and classes already existing in columns; there's no way to add your own class to this list.

- The columns and headers it creates are VFP base classes. You might want to substitute your own classes (which must be defined programmatically) to, for example, sort on a column by clicking on the header.

To create a replacement for the VFP Grid Builder, I first created the SFGRIDBLDR project (it's in the GRID subdirectory when you unzip the sample files that accompany this document) and added the following files: BUILDER.VCX (in the BUILDERS\BUILDERS subdirectory of the VFP wizard source directory), GRIDBLDR.VCX (in the BUILDERS\GRIDBLDR directory), THERM.VCX, WIZCTRL.VCX, WIZMOVER.VCX (all in WIZARDS\WZCOMMON), DUMMY.PRG, and WBGRID.PRG (both in BUILDERS). How did I know which files to add to this project? Simple: I just looked at the contents of the GRIDBLDR project.

Next, I subclassed the GridBuilder class into SFGridBuilder (in SFGRIDBLDR.VCX) and overrode the ResetColumns method, which sizes the columns appropriately for the selected field (I only implemented this idea, not the other two listed above). The code for that method is listed below. There are a couple of interesting things to note here. First, I expected to write a lot of complex code to figure out how wide a column should be, based on the width of the field, the font and font size of the grid, etc. Interestingly, it turns out that a method to do this (SetColWidth) already existed in the builder, but rather than passing this method the size of the field, the builder passed it a different value. The other thing is that currently the assignment of the calculated width to loColumn.Width is commented out. For reasons I haven't tracked down yet, changing the width of the column at this point seems to cause a problem in that when the builder is closed, the column width is set to 0. Instead, the width is stored in wbaCols. (In case you're wondering, yes, wbaCols is a public array. I didn't create the builder, so don't blame me for doing it this way!) So, the effect is that when a column is added to the grid, it isn't sized properly immediately, but is as soon as you do anything else (add another column, go to another page in the builder, close the builder, etc.).

```
local lnRows, ;
   lnI, ;
   lcField, ;
   loColumn, ;
   lcHeader, ;
   loHeader
dodefault()
lnRows = alen(wbaCols, 1)
for lnI = 1 to lnRows
```

```
    lcField = wbaCols[lnI, 2]
    if not empty(lcField)
       loColumn = evaluate('wbaControl[1].' + ;
          wbaCols[lnI, 7])
       wbaCols[lnI, 1] = This.SetColWidth(fsize(lcField), ;
          loColumn)
*      loColumn.Width  = wbaCols[lnI, 1]
    endif not empty(lcField)
next lnI
```

Finally, I created GRIDMAIN.PRG (using GRIDMAIN.PRG from the BUILDERS\GRIDBLDR directory for ideas) in the directory for my builder and made it the main program in the project. This program adds SFGRIDBLDR.VCX to the open class libraries using SET CLASSLIB so it can find our SFGridBuilder class. GRIDMAIN also auto-registers itself in the VFP BUILDER table if the APP is run directly.

To see this builder in action, build and run SFGRIDBLDR.APP to register it as the builder for grids. If you look at BUILDER.DBF after running this APP, you'll find that it's "deregistered" the original grid builder by changing the TYPE column from "GRID" to "XGRID" and added a new record for itself with "GRID" as the TYPE. Next, create a form, drop a grid on it, and invoke the builder. The new builder won't look any different than the old one, but when you add fields to the grid, you'll see that they're properly sized.

Creating a Better Referential Integrity Builder

I have a few problems with the Referential Integrity (RI) Builder that comes with VFP:

- When you click on the OK button to save the RI rule changes, it asks you not once but twice to confirm that you want to go ahead and do this. Not to be snippy, but I believe the purpose of the Cancel button is to allow me to back out; I clicked on the OK button because it was OK, so I don't need a "are you really, really sure you want to do this" confirmation dialog.

- You can't run the RI Builder unless you've packed the database first.

- I really don't need it to back up my current stored procedures to a file called RISP.OLD, which I always delete anyway (or forget to delete and then end up backing it up by mistake).

- The code it generates has at least one bug that prevents it from functioning correctly. For details on the bug, see my white paper "Error Handling in VFP", available from the Technical Papers page of www.stonefield.com.

- The code it generates is bulky and poorly commented. Trying to understand what this code does is a laborious process, and it's possible in a complex database to have RI code that exceeds the 64K limit of compiled programs in VFP. If you purchased "Effective Techniques for Application Development with Visual FoxPro 6.0" by Jim Booth and Steve Sawyer (available from Hentzenwerke Publishing, www.hentzenwerke.com), you have a copy of a faster, leaner, and better routine for maintaining RI. Steve's NEWRI routine is data-driven, so it determines at runtime rather than generation time

which rules need to be enforced. The result is clear, tight code rather than the unwieldy bulk produced by the VFP RI Builder.

- The only rules it supports are ignore, cascade, and restrict. What about other options you might want to use, such as nullify (set the foreign key in the child to .NULL.) or assign a new value (set the foreign key to, for example, a default value)?

Fixing these items is relatively easy because we have the source code for the RI Builder. I created SFRIBUILDR.APP, a replacement for the VFP RIBUILDR.APP. I didn't implement additional rules (the last item in the list above) but I did handle all the rest. I first created the SFRIBUILDR project (it's in the RI subdirectory when you unzip the sample files available from the Subscriber Download Site) and added the VFP RIBUILDR.VCX (located in the BUILDERS\RIBUILDR subdirectory of the VFP wizard source directory).

Next, I subclassed the VFP RIBuildr class into SFRIBuildr in SFRIBUILDR.VCX. I overrode the Load method to not give an error if there are any deleted records in the database (you can see where I commented out the existing code). I overrode the Click method of the OK button to not display confirmation dialogs, not copy the current stored procedures to RISP.OLD, and to fix the bug in the generated code. Also, if it detects NEWRI.PRG on your system (in the same directory as the SFRIBUILR.APP), it will place that code into the stored procedures of the database rather than generating any code. This change required a new method, RIMakeNewTr, to create triggers with a different name (__RI_Handler) than the ones used by the RI Builder (__RI_<action>_<table>, such as __RI_Delete_Customer). Because there's quite a bit of code in these methods, it isn't shown here; however, if you look at it in the source code provided, you'll see that I really only commented out a few lines and added a few other lines.

Finally, I copied RIMAIN.PRG from the BUILDERS\RIBUILDR directory to the directory for my builder, added it to the project, and made it the main program. I modified this PRG to point to the library where my subclass of the VFP RI builder class can be found (SFRIBUILDR.VCX) and to auto-register the builder in the VFP BUILDER table if the APP is run directly.

Building and executing SFRIBUILDR.APP results in this file being registered as the RI Builder instead of the usual RIBUILDR.APP. To see this in action, do the following:

- Move to the WIZARDS\RI directory of the sample files for this document.

- If you have Steve's NEWRI.PRG, copy it into this directory.

- Run COPYDEMO.PRG. This program will copy the VFP TESTDATA database to a DATA subdirectory so we don't touch the original database.

- Run DELETETEST.PRG. This will demonstrate a flaw in the code generated by the VFP RI Builder. The first browse shows several orders for the ALFKI customer, then tries to delete that customer. Because there's a cascade delete rule from CUSTOMER to ORDERS but a restrict rule from ORDERS to ORDITEMS, the customer shouldn't be deleted. However, notice that the error dialog appears six times (not just once as you'd

expect) and then another browse shows that the customer has been deleted (CUST_ID is .NULL.).

- Run COPYDEMO.PRG again since we've messed up the data.

- Open the DATA\TESTDATA database exclusively.

- Build and run SFRIBUILDR.APP to register it as the RI Builder.

- Open the Database Designer, choose the Edit Referential Integrity function, and then simply click OK in the RI Builder dialog. Notice no confirmation dialogs appeared. Choose the Edit Stored Procedures and notice that the code is different (if you have NEWRI.PRG, this code is placed in the stored procedures; otherwise, the RIDelete and RIUpdate methods have a bug fix implemented). If you have Steve's NEWRI.PRG, modify the CUSTOMER table and note the name of the trigger for each method. Also, you won't find an RISP.OLD file.

- Run DELETETEST.PRG again. This time, you only get the error message once and the ALFKI customer isn't deleted.

Creating a Better Upsizing Wizard

Jim Falino was frustrated with some of the behavior of the SQL Server Upsizing Wizard, so he created a version that has the behavior he wanted.

- The wizard creates a timestamp column in any table with at least one numeric, float, general, or memo field. Jim didn't want the wizard to automatically create any timestamp columns, so he removed this behavior.

- The wizard automatically creates a clustered index for any table with a primary key. While this may occasionally be desirable, Jim decided that the default should be to not create a clustered index, so he added a NONCLUSTERED clause when creating a primary key.

- Since database object names must be unique in SQL Server, you can't upsize things with non-unique names. For example, you can't upsize primary key indexes if you used the same primary key index name for more than one table (such as using "ID" for the primary key name for every table). Jim's solution is to rename primary key indexes to UQ_<table name>, since the name is unimportant in SQL Server.

- The available tables list box isn't sorted; tables appear in the same order as they appear in the DBC. In a DBC with hundreds of tables, it can be a pain finding a particular table. Jim set the SortLeft property of the SuperMover object (which indicates if the left list box should be sorted) to .T.

- The wizard can create a remote view with the same definition as each of the tables in the DBC. However, it names the view the same as the original table, and it renames the table to a unique name. Since some people prefer using a naming convention for view

names (for example, “V” plus the underlying table), Jim changed the wizard to do this instead.

- Jim fixed a problem in which the wizard sometimes rearranges the field order of compound indexes.

- VFP columns of Numeric data type upsize to Float data type, even though SQL Server has a Numeric data type as well. This causes VFP Numeric columns of more than two decimal places to be truncated and VFP Numeric columns with no decimal places upsize to two decimal places.

The changes Jim made were specific for his requirements; you could either use them if your requirements are the same or use them as a template for your own changes. Details on the changes Jim made can be found in his article in the July 1999 issue of FoxTalk (this article is also available online at www.pinpub.com), or you can contact him at jim@garpac.com.

Creating a Better WWW Search Page Wizard

In their book “Visual FoxPro 6 Enterprise Development” (Prima Tech, ISBN 0-7615-1381-7), Rod Paddock and John Petersen describe how to modify the WWW Search Page Wizard to eliminate the limitation of outputting only five fields to a Web page. Rather than modifying the wizard, you might want to subclass it instead. To do so, follow a technique similar to the way I’ve subclassed the builders. The code for the WWW Search Page Wizard is in the WIZARDS\WZINTNET directory of the VFP wizard source directory.

Creating Your Own Wizards

Rod Paddock and John Petersen also provide some detail in their book on using the VFP wizard classes to create your own wizards. The advantages of doing so are that someone has already built all the “engine” stuff for managing the wizard process and that your wizard will have the same look and feel as the other VFP wizards.

Creating Builders with BuilderD

While looking at the FoxPro Foundation Classes (FFC) that come with VFP 6, one of the things I noticed is that they all have custom properties called Builder and BuilderX, and BuilderX is set to = HOME() + “Wizards\BuilderD,BuilderDForm” in each class. I knew what these properties are for (they tell VFP the name of the builder for the class), but why does every class specify the same builder? Even more interesting, invoking the builder for each class shows a similar builder form but with different options for each one (Figure 1 shows the builder for the _HyperLinkLabel class, for example). How the heck does that happen when the same class is being used?

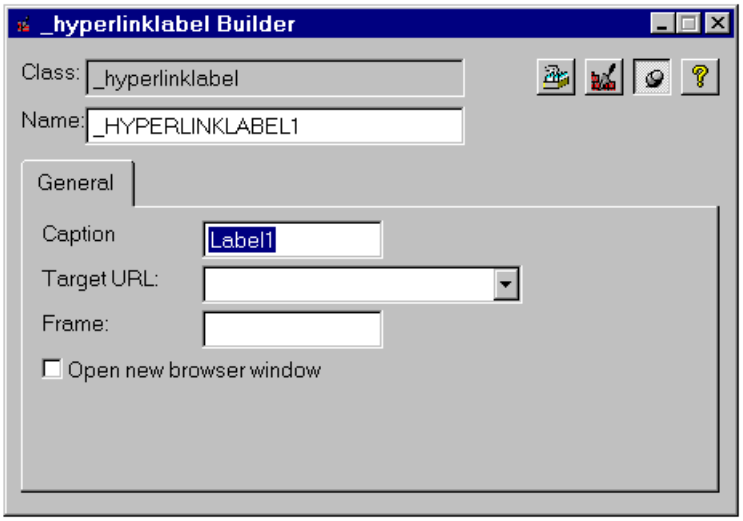

Figure 1. The builder for the _HyperLinkLabel class.

First a little background. As you're probably aware, VFP builders can be called a variety of ways, but the most common is probably by right-clicking on an object and choosing Builder from the context menu. This causes BUILDER.APP to be executed. BUILDER.APP checks to see if the selected object (we'll call this the "target object") has a BuilderX property, and if so, runs the program or instantiates the class specified in that property (if the builder is a class, it should be specified as the class library, a comma, and the class name). If it doesn't have a BuilderX property but has a Builder property, the builder runs the program or instantiates the class specified in that property (we'll see why there are two properties to specify the builder in a moment). If neither of these properties exist, it uses the default builder for the base class of the object as I discussed earlier.

You can create custom Builder and BuilderX properties in your classes (even in your base classes) and then fill them in with the name of the appropriate builder for each specific class. The reason for having two properties is that BuilderX specifies a custom builder for the specific class, while Builder is intended for a builder for a set of common classes such as all comboboxes or grids. As we'll see later, we can click on a button in the builder specified in the BuilderX property to bring up the builder specified in the Builder property.

So, we can easily specify our builders in Builder and BuilderX properties. But isn't it a lot of work to create your own builders, especially for classes that may not be used much?

BuilderD

You may be aware of the BuilderB technology created by Ken Levy to make creating builders faster and easier. BuilderB is a set of classes you can subclass to create your own builders. You add a control to the builder subclass for each property of the target object you want the builder to maintain. Although BuilderB makes creating builders much easier, it's kind of a drag to have to create a new builder subclass for each class you want a builder for. Fortunately for us lazy types, Ken enhanced BuilderB by making it data-driven. The new technology, called BuilderD (the "D" stands for "Dynamic"), is available from Ken's Web site

(www.classx.com) and is also included with VFP 6 (BUILDERD.VCX in the WIZARDS subdirectory of the VFP home directory).

BuilderD consists of several classes, but the main one is BuilderDForm; this is the data-driven builder form (notice this is the class specified in BuilderX for the FFC classes). As you can see in Figure 2, this form has textboxes for the class and the name of the target object, buttons that provide functionality like bringing up the Class Browser and displaying help, and a pageframe with a couple of pages, but no controls to manage the values of properties.

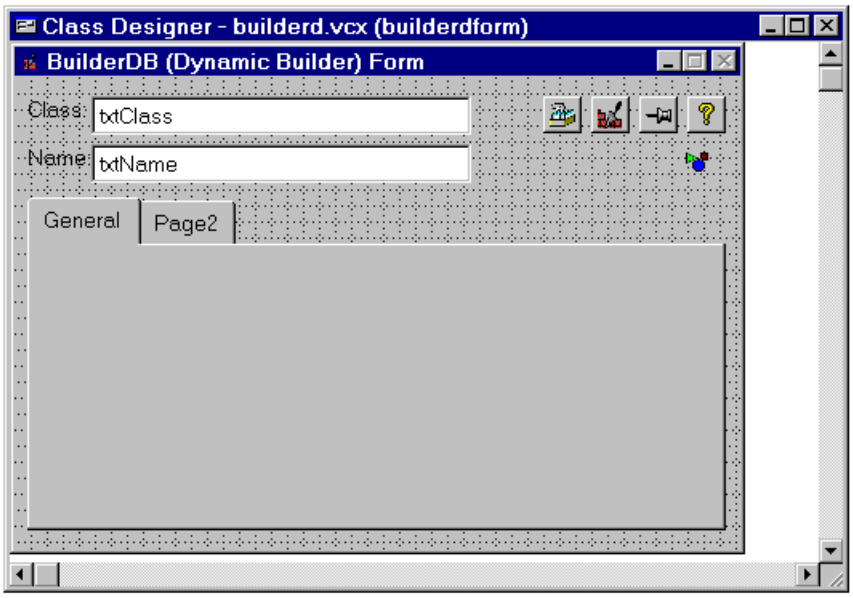

Figure 2. BuilderDForm.

Here's how this form is populated with the appropriate controls when it's instantiated:

- The Init method calls the SetObject method, which calls the AddObjects method (this code is actually in BuilderBaseForm, the parent class of BuilderDForm).

- The AddObjects method calls the AddObjects method of the oBuilderDB object on the form, which comes from the BuilderDB class.

- BuilderDB.AddObjects is a complex method, but the basics are that it opens the builder definition table (by default, BUILDERD.DBF located in the WIZARDS subdirectory of the VFP home directory, but another table can be specified by changing the cBuilderTable property), finds a record for the class of the target object, finds all the records linked to that record, and uses information in those records to create controls on one or more pages of the pageframe. These controls are based on classes in BUILDERD.VCX, such as BuilderCheckBox and BuilderTextBox, that know how to be bound to properties of the target object.

The result of these steps is a builder that can manage one or more properties of the target object. A specific builder can actually do a lot more than that, such as putting code into

methods of the target object or even the object's containers, but we'll take the simple approach for now.

BuilderD Table

Let's take a closer look at the builder definition table, BUILDERD.DBF, because understanding its structure is the key to creating your own builders. Table 2 shows the structure of BUILDERD.DBF; in this table, "property control" means the control in the builder that maintains a specific property of the target object.

Field	Purpose
TYPE	Defines the type of record. This contains either "CLASS" if this is a class record or "PROPERTY" if it's a property record.
ID	The identifier of the record, frequently the name of the class or property the record is for.
LINKS	A list of the ID values of other records (separated by carriage returns) linked to this record. This field is discussed in more detail below.
TEXT	The caption for the builder form if this is a "CLASS" record or the caption for the property control if it's a "PROPERTY" record.
DESC	The status bar text for the property control.
CLASSNAME	For a "CLASS" record, the name of the class this builder is for. BuilderDB looks for a record containing the target object's class in this field. For a "PROPERTY" record, the class to instantiate for the property control. If this field is blank, a default class from BUILDERD.VCX will be used (BuilderCheckBox for logical properties and BuilderTextBox for all others). Any class you specify should be a subclass of a BuilderD class because those classes have special properties and methods used by BuilderDForm.
CLASSLIB	The class library containing the class specified in CLASSNAME. This property can contain an expression (such as HOME() + "WIZARDS\BUILDERD.VCX") rather than a constant; in that case, put parentheses around the expression. For a "PROPERTY" record, if this field is blank and CLASSNAME is specified, BUILDERD.VCX is assumed. For a "CLASS" record, BuilderDB looks for a record containing the same value in this field (after evaluating it if necessary) as the ClassLibrary property of the target object; leave this blank to create a builder

	for a class without worrying about what class library it's in.
MEMBER	Blank for "CLASS" records. For "PROPERTY" records, the name of the target object property maintained by the property control this record defines. If blank, ID must contain the name of the property.
HELPFILE	The name of the CHM file containing help for this class. If blank, the current help file is used.
HELPID	The ID for the help topic.
TOP	The Top setting for the property control. If 0, BuilderDForm will place the control below the previous one (the first control on a page is placed at the value specified in BuilderDB.nTop).
LEFT	The Left setting for the property control. If 0, BuilderDForm will place the control near the left edge of the page it's on (specified by BuilderDB.nLeft).
HEIGHT	The Height setting for the property control. If 0, the default Height for the control is used.
WIDTH	The Width setting for the property control. If 0, the default Width for the control is used.
ROWSRCTYPE	If the class to use for the property control (specified in CLASSNAME) is a combobox, the RowSourceType setting for the combobox.
ROWSOURCE	If the property control is a combobox, the RowSource setting for the combobox. For example, if ROWSRCTYPE is 1 (Value), ROWSOURCE will contain a comma-delimited list of values for the combobox.
STYLE	If the property control is a combobox, the Style setting for the combobox.
VALIDEXPR	An expression used to validate the value of the property.
READONLY	.T. if the property control is read-only.
UPDONCHNG	.T. if the property control's value is written to the target object's property as it's changed (that is, from the InteractiveChange method).
UPDATED	The datetime the record was last modified (not used by BuilderD but for information only).

COMMENT	Comments about the record (not used by BuilderD).
USER	User information for the record (not used by BuilderD).

Table 2. The structure of BUILDERD.DBF.

Tables 3 and 4 show the records that make up a couple of builders, the ones for the FFC _HyperLinkBase and _HyperLinkLabel classes. I haven't shown all fields in BUILDERD.DBF because of space considerations; only those fields pertinent to the discussion are shown.

TYPE	ID	LINKS	CLASSNAME	CLASSLIB
CLASS	_HyperLinkBase	cTarget cFrame lNewWindow	_HyperLinkBase	(HOME()+"FFC_Hyperlink.vcx")
CLASS	_HyperLinkLabel	Caption _HyperLinkBase	_HyperLinkLabel	(HOME()+"FFC_Hyperlink.vcx")

Table 3. The record for the builders for the _HyperLinkBase and _HyperLinkLabel classes.

TYPE	ID	TEXT	CLASSNAME	ROWSRCTYPE	ROWSOURCE
PROPERTY	cTarget	Target URL:	BuilderComboBox	1	www.microsoft.com/vfoxpro
PROPERTY	cFrame	Frame:			
PROPERTY	lNewWindow	Open new browser window			
PROPERTY	Caption	Caption			

Table 4. Records defining the properties managed by the _HyperLinkBase and _HyperLinkLabel builders.

In the record in Table 3 for _HyperLinkBase, we see that CLASSNAME and CLASSLIB specify which class this is the builder for (notice that CLASSLIB contains an expression that'll be evaluated at runtime rather than a hard-coded value), and LINKS lists the records that specify the properties of this class the builder will manage. The cTarget PROPERTY record shown in Table 4 indicates that this property will be managed by a BuilderComboBox control with a RowSource containing the former URL of the Microsoft VFP Web site (it's now msdn.microsoft.com/vfoxpro). cFrame will be managed by a BuilderTextBox object (because it's a character property and the class isn't defined) and lNewWindow will have a BuilderCheckBox object (because it's a logical property).

Seems simple so far, right? Well, the LINKS field can actually get more complicated. First, if an ID specified in the LINKS field for a CLASS record doesn't have a matching record, that ID is assumed to be the name of a property. Thus, you could actually create a builder in a single record by simply specifying the properties it manages in the LINKS field of the CLASS record. Of course, you'd have to live with captions for the properties being the same as the property name, no status bar text, and default classes and sizes for the properties, but that's not too bad for a quick and dirty builder. A second complication is that an ID specified in the LINKS field for a CLASS can point to another CLASS record rather than to a PROPERTY record. In that case, this class "inherits" all of the links of the specified class. You can see this in the _HyperLinkLabel record in Table 3; one of its links is _HyperLinkBase, so not only does the builder for this class manage the Caption property (specifically listed in its LINKS field), but also the cTarget, cFrame, and lNewWindow properties because those are specified in the LINKS field for _HyperLinkBase. Thirdly, a PROPERTY record can be linked to another PROPERTY record. In that case, the record "inherits" all non-blank fields from the linked record. Finally, LINKS can contain the caption for the page in the builder pageframe if you specify it as @<caption> (for example, @Properties to use "Properties" as the page caption).

Creating a Builder

Let's check it out. Start by creating a subclass of the VFP CheckBox class called TestCheck in TEST.VCX. Add a custom property called BuilderX to this class and set its value to = HOME() + "Wizards\BuilderD,BuilderDForm". Then right-click on the class and choose Builder from the context menu. Oops, we get a "There are no registered builders of this type" error. That makes sense, since we haven't defined one yet (although wouldn't it be nice if it automatically created one for us—more on this later). Do the following:

- In the Command window, type USE HOME() + "WIZARDS\BUILDERD", then BROWSE.
- Choose "Append New Record" from the Table menu.
- Enter "CLASS" for TYPE, "TestCheck" for ID, "Enabled", "AutoSize", and "Caption" (pressing Enter after each one) for LINKS, "My Test Builder" for TEXT, "TestCheck" for CLASSNAME, and "TEST.VCX" for CLASSLIB.
- Close the browse window and type USE in the Command window to close the table.
- Right-click on the TestCheck class and choose Builder.

Cool, huh? Your very own builder, created in just about one minute! Uncheck Enabled and enter a different Caption, and watch these properties change instantly. Notice that we didn't bother creating records for the properties; Enabled and Caption records already existed in BUILDERD.DBF, so we just reused them, and since no AutoSize record existed, BuilderD just assumed we wanted to manage that property.

Let's build another one and see how little we can enter and still get a working builder. Create a subclass of the VFP TextBox class called TestText in TEST.VCX. Again, add a BuilderX property and set it to = HOME() + "Wizards\BuilderD,BuilderDForm". Create a record in BUILDERD.DBF and just specify TYPE ("CLASS"), ID ("TestText"), LINKS

("ReadOnly"), and CLASSNAME ("TestText"; we can skip CLASSLIB but CLASSNAME is required). Close the table, then bring up the builder for the class. Voila—a working builder. Check out the Builder button in the builder; it brings up another builder (one specified in Builder if that property existed and was filled in, or the default builder for the base class of this class, which is the VFP Text Box Builder in this case). Thus, even though we can have specific builders for a class, we can still access more generic builders as well.

Pre-Built Builders

I recently created a couple of builders using BuilderD, one for my SFGrid class and the other for SFPageFrame. The SFGrid builder maintains the DeleteMark and RecordMark properties (you could easily add other properties you might frequently change) but since they're easily changed in the Property Sheet, they weren't the real reason I created the builder. The real reason was because I like to use SFGridTextBox objects (a subclass of SFTextBox with a few properties set differently to improve their appearance in a grid) in the columns of a grid, and it's a pain to replace the generic TextBox objects with SFGridTextBox objects. So I created a button (SFGridTextBoxButton in SFBUILDERS.VCX) that does this automatically. Here's the code from the Click method of this button:

```
local loColumn, ;
   loControl, ;
   lcName
for each loColumn in Thisform.oObject.Columns
   for each loControl in loColumn.Controls
      if upper(loControl.Class) = 'TEXTBOX'
         lcName = loControl.Name
         loColumn.RemoveObject(lcName)
         loColumn.NewObject(lcName, 'SFGridTextBox', ;
            'SFCtrls.vcx')
      endif upper(loControl.Class) = 'TEXTBOX'
   next loControl
next loColumn
wait window 'SFGridTextBox added to each column' ;
   timeout 2
```

BuilderDForm has a reference to the target object stored in its oObject property, so any control on the builder form can reference or change things about the target object. In this code, Thisform.oObject.Columns references the Columns collection of the grid being affected by the builder.

One complication: how do I get this button on the builder form? I could create a record for it in the BUILDERD table, but that would add the button on a page of the pageframe and I'd rather have the button appear with the other builder buttons. So, I created a "button loader" class (SFBuilderButtonLoader in SFBUILDERS.VCX) which adds a button to the builder form and then returns .F. so it doesn't actually get instantiated. SFBuilderButtonLoader looks in the USER memo field of the current record in the BUILDERD table (the record that caused the class to be instantiated) for the class name and library of the button to add to the form (note in the code below that BUILDERD is open with the alias "BUILDER"). For SFGridTextBoxButton, for example, I created a record in the BUILDERD table with "SFGridTextBoxButton" as the ID but "SFBuilderButtonLoader" as the class, and entered "SFBuilders, SFGridTextBoxButton" in the USER memo. This tells SFBuilderButtonLoader

to add an SFGridTextBoxButton to the form. Here's the code from the Init method of SFBuilderButtonLoader:

```
local lcClass, ;
   lnPos, ;
   lcLibrary, ;
   lnTop, ;
   lnLeft, ;
   loControl

* If the BUILDERD table is open and positioned to the
* record for the button to be loaded (which it should
* be), we'll add the button by getting the class and
* library for the button from the USER memo.

if used('BUILDER') and ;
   lower(BUILDER.CLASSNAME) = lower(This.Class)
   with Thisform
      lcClass = BUILDER.USER
      lnPos   = at(',', lcClass)
      if lnPos > 0
         lcLibrary = left(lcClass, lnPos - 1)
         lcClass   = substr(lcClass, lnPos + 1)
      else
         lcLibrary = ''
      endif lnPos > 0

* Add the button. If we succeeded, find the first open
* "slot" in the button area on the form.

      .NewObject(lcClass, lcClass, lcLibrary)
      if type('.' + lcClass + '.Name') = 'C'
         lnTop  = .cmdClassBrowser.Top + ;
            .cmdClassBrowser.Height + 5
         lnLeft = .cmdClassBrowser.Left
         for each loControl in .Controls
            if loControl.Left = lnLeft and ;
               loControl.Top = lnTop
               lnLeft = lnLeft + ;
                  .cmdClassBrowser.Width + 6
               if lnLeft + .cmdClassBrowser.Width > .Width
                  lnLeft = .cmdClassBrowser.Left
                  lnTop  = lnTop + .cmdClassBrowser.Height + 5
               endif lnLeft + .cmdClassBrowser.Width > .Width
            endif loControl.Left = lnLeft ...
         next loControl

* Set the button position to the located slot and make
* it visible.

         with .&lcClass
            .Top     = lnTop
            .Left    = lnLeft
            .Visible = .T.
         endwith
      endif type('.' + lcClass + '.Name') = 'C'
   endwith
endif used('BUILDER') ...
return .F.
```

The SFGrid record in the BUILDERD table has DeleteMark, RecordMark, and SFGridTextBox in its LINKS column, so this builder gets these controls. I can now bring up my BuilderD builder for any SFGrid object, use the VFP Grid Builder (by clicking on the Builder button in the BuilderD form) to quickly create the columns in the grid, and then change those columns to use SFGridTextBox objects by clicking on my Add SFGridTextBox button.

For SFPageFrame, I wanted a similar function: a button that adds code to the RightClick method of each page in the pageframe; this allows right-clicking on a page to provide a context menu for the pageframe or (more likely) entire form. As with the SFGrid builder, I created a record in the BuilderD table that instantiates an SFBuilderButtonLoader object that then adds an SFCodePageButton object to the builder form. Here's the code from the Click method of SFCodePageButton that adds the desired code to each page:

```
local lcCode, ;
   loPage, ;
   lcCurrentCode
lcCode = 'This.Parent.ShowMenu()' + chr(13)
for each loPage in Thisform.oObject.Pages
   lcCurrentCode = loPage.ReadMethod('RightClick')
   if not lcCode $ lcCurrentCode
      loPage.WriteMethod('RightClick', lcCurrentCode + ;
         iif(empty(lcCurrentCode), '', chr(13)) + lcCode)
   endif not lcCode $ lcCurrentCode
next loPage
wait window 'Code added to each page' timeout 2
```

To add these new builders to your system, copy SFBUILDERS.VCX and VCT (found with the source files for this document) to some directory, then open the NEWBUILDERS table provided with the sample files and change the CLASSLIB fields to include a path to SFBUILDERS.VCX. Open the BUILDERD table and APPEND FROM the NEWBUILDERS table, add Builder and BuilderX properties to your classes, and enter = HOME() + "Wizards\BuilderD,BuilderDForm" into BuilderX.

Building a Builder Builder

As easy as it is to create a builder using BuilderD, wouldn't it be even better if there was a visual front-end to BUILDERD.DBF, along the same lines as the forms we provide to our users? What I'm talking about is something that builds builders; yes folks, a builder builder. In fact, we'll use BuilderD itself to build the builder builder (does that make it a builder builder builder?).

As an aside, about a decade ago, I worked for a company that had a pretty cool communications setup. We had a communications server (fast modems were fairly expensive back then) on our LAN that had the remote control software pcAnywhere installed on it. We had a program called LANAssist (essentially a LAN version of what pcAnywhere does) that allowed me to control that server from my workstation, so I could fire up pcAnywhere on it, and have it dial into a client's communication server. I would then use LANAssist on that server to take over any user's workstation on the client's LAN. What got really hard to follow was: when I type DIR, am I getting a directory of my machine, our communication server, the client's communication server, or the client workstation I was

controlling? It took serious concentration to remember what machine your keyboard was actually affecting. I think you'll find this analogous to what we'll be doing here: "is this builder managing my object or is it managing the builder that manages my object?" Consider yourself warned <g>.

The source code available with this document includes SFBUILDERS.VCX, which contains subclasses of BuilderD classes that provide the functionality we need. There isn't enough room here to go over all the code in these subclasses, so I'll hit the high points.

SFBuilderBuilderForm

The first class we'll look at it is SFBuilderBuilderForm. This class, subclassed from BuilderDForm (the builder form class for BuilderD), is the builder builder, and what you'll specify in the BuilderX property of a class (you'll enter "<directory>\SFBuilders, SFBuilderBuilderForm"). This subclass has some additional buttons added:

- An "add property control" button that adds a new property control to the builder and displays a builder for that property control so you can specify what property the control is bound to, what caption to use, and its size and position.

- An "edit builder caption" button that brings up a builder so you can change the caption for your builder.

- A "save" button that updates the builder definition records in BUILDERD.DBF for the builder we're working on.

- An "export" button that writes the builder definition records to another table. This allows you to ship this table to someone else, and they simply have to APPEND FROM it into BUILDERD.DBF to get a copy of the builder you created.

The Load method of SFBuilderBuilderForm changes the cProgramPath property to point to the WIZARDS subdirectory of the VFP home directory. This is necessary because the normal behavior of Load (which is first executed using DODEFAULT) is to set this property to the directory this class' class library is located in. Since we may not want to install SFBUILDERS.VCX in the WIZARDS directory but we still need to access files in that directory, this is a necessary step.

The Init of SFBuilderBuilderForm uses DODEFAULT() to do the normal behavior, then calls a custom method called CreateBuilderRecords. This method adds records to BUILDERD.DBF (if they don't already exist) that define the builders used to maintain our builder. Init then ensures that the correct number of pages is displayed and that any custom properties the builder manages but don't exist in the target object are added to the target object (more about this later).

The RightClick method of the form class calls a custom ShowMenu method, which instantiates an _ShortcutMenu object (from the FFC) and calls the ShortcutMenu method to populate the object, then activates the menu. This provides us with context menus for both the form and for property controls (so we can, for example, right-click on a control and select a function to edit or remove it).

Both the “add property control” button’s Click method and the “Add property control” function in the context menu call the form’s AddPropertyControl method to add a new property control. This method checks how many controls are on the last page of the pageframe and adds a new page if necessary (the nMaxObjects property of the oBuilderDB object on the form defines how many controls we’ll put on a page). It then adds BuilderLabel (for the caption of the property) and BuilderTextBox objects to the page and binds the text box to the Tag property of the target object by setting its cProperty property to “Tag” (I had to pick *some* property, and figured that something as global as Tag would be the best choice). Note that the BuilderTextBox class it adds doesn’t come from BUILDERD.VCX, but is instead the BuilderTextBox class in SFBUILDERS.VCX (which, as you may expect, is subclassed from the BuilderD version). The reason for using a subclass is that I’ve added code to the RightClick method of my subclass so we can have a context menu for property controls (there’s a subclass of BuilderCheckBox in SFBUILDERS.VCX for the same reason). Why did I name these classes the same as their parent classes and not “SFsomething”? That’s because when an existing builder definition is loaded from BUILDERD.DBF, oBuilderDB.AddObjects creates BuilderTextBox and BuilderCheckBox objects as the property controls. With SFBUILDERS.VCX earlier in the class search chain that BUILDERD.VCX (done with SET CLASSLIB TO SFBUILDERS, BUILDERD), I ensure that my subclasses are used instead of the BuilderD classes. Finally, AddPropertyControl calls the custom EditObject method, passing it a reference to the new control, to display a builder for the new property control. EditObject simply instantiates BuilderDForm (yes, we’re using the regular BuilderD form for our property control builder), passing it a reference to the control, and tiles it below and to the right of the current builder form.

Both the “edit builder caption” button’s Click method and the “Edit builder caption” function in the context menu call the form’s EditBuilderCaption method. This method simply calls EditObject, passing a reference to the builder form, so it displays a builder for the form, which simply provides a way to edit the caption of the form.

The Save method is called from both the “save” button’s Click method and the “Save” function in the context menu. This method spins through all the property controls in the form and creates or updates a record for that control in BUILDERD.DBF. The Export method (called from both the “export” button’s Click method and the “Export” function in the context menu) also calls Save, but passes it the name of a table to export to which it obtained from you using GETFILE().

Right-clicking on a property control displays a context menu with functions to edit the control (it simply calls EditObject, passing it a reference to the control), remove the control (which calls RemovePropertyControl to remove the control and any associated label object), or reset the value of the property the control manages to its default value (by calling the DefaultReset method; the code for this method is in BuilderBaseForm).

Other methods in SFBuilderBuilderForm are just support methods. For example, FindCaption locates the label object that provides the caption for a property control by looking for the object with a TabIndex value of one less than the control’s TabIndex value (pretty low-tech, I admit, but I didn’t want to change any code in any BuilderD classes, so this was the only way I could think of). FindClass and FindProperty find a specific CLASS or PROPERTY record in BUILDERD.DBF.

If you recall from earlier, the AddObjects method of the oBuilderDB object on the form is the data-driven engine of BuilderD; it reads builder definition records from the BUILDERD table and adds controls to the builder form. This method was overridden in SFBuilderBuilderForm to provide additional functionality: it adds a record for the class the builder is for to the BUILDERD table if one doesn't already exist and it adds SFBUILDERS.VCX before BUILDERD.VCX in the SET CLASSLIB command. The first change is needed because, otherwise, BuilderD would give you an error that there are no builders registered for this class; we want to auto-register a builder for the class the first time a builder is invoked for it. The second change ensures that our BuilderTextBox and BuilderCheckBox classes are used rather than BuilderD's versions, so we get a context menu when we right-click on a property control.

Other Builder Classes

As I mentioned earlier, the BuilderCheckBox and BuilderTextBox classes in SFBUILDERS.VCX are subclasses of the same named BuilderD classes. In addition to providing right-click behavior, these two classes have nTop and nLeft properties, with assign methods for each that set the custom lMoved property to .T. when these properties are changed. This allows us to detect when a property control was moved (the Left and Top controls in the property control builder are bound to nLeft and nTop rather than Left and Top directly). Only if a property control is moved do we store its Left and Top values to the BUILDERD table.

Another class in SFBUILDERS.VCX is SFPropertyCaption. This control is used to manage the caption for a property control. Why have a special class for that? Why not just use a normal BuilderTextBox object (which this is subclassed from)? The reason is that if the property control is a BuilderCheckBox, it has a Caption property, so the control manages that property. However, if the property control is a BuilderTextBox, it doesn't have a Caption property but instead has an associated label object, so the control must manage the Caption property of that object. Since BuilderD classes are intended to manage the properties of a single object, SFPropertyCaption had to override a few methods to allow it to handle this situation.

The SFBuilderPropertyComboBox class is a subclass of BuilderComboBox. It presents a list of all of the writable properties of the target object (obtained using AMEMBERS() to get a list of all properties and then checking PEMSTATUS() of each one to eliminate those that are read-only). Although its main goal is to change the cProperty property (which determines which property is being managed) of the property control it's managing (are you getting a headache thinking about these multiple levels of management?), it has a couple of interesting behaviors. First, if the property control is a BuilderTextBox but you've just changed the property it manages to a logical one (such as Enabled), it removes the BuilderTextBox and its associated label object, and puts a BuilderCheckBox in its place. It does the opposite if you change from a logical property to one of another type. Second, if you enter the name of a property that doesn't exist, you'll be prompted to create the property. If you agree, the builder uses AddProperty to add the new property to the target object. I don't really recommend doing this in an object dropped on a form, because this amounts to instance programming; if you really need a new property, you should consider using a subclass instead. However, this is a quick way to create a new property in a class *and* have the builder manage it in one step.

SFBuilderAddButton is a button class (subclassed from BuilderCommandButton) that's added to the builder form for a property control using the SFBuilderButtonLoader class. The Click method of SFBuilderAddButton simply calls the AddPropertyControl method of the builder form the property control is on (there's that multiple levels thing again) to create a new property control, then switches the current builder form to manage the new property control rather than the one you brought it up to manage in the first place. This means you can click on the Add Property Control button to add a new property control to the builder, and in the builder that comes up to manage it, create another one. This is a fast way to add multiple property controls in a hurry.

Trying it Out

Let's see SFBuilderBuilderForm in action; it's actually a lot easier to use than it was to describe. Create a subclass of the VFP TextBox class called MyTestText in TEST.VCX. Add a BuilderX property and set it to "SFBuilders,SFBuilderBuilderForm", then bring up the builder for the class. Notice that even though we didn't create any records in the BUILDERD table, we still got a builder form anyway (of course, there are no controls on the form, but we'll change that in a moment). This is because SFBuilderBuilderForm automatically created a CLASS record for the class in BUILDERD when it didn't find one.

Click on the Add Property Control button. You'll notice that a textbox and label appear on the builder form, but then another builder form appears on top of the original builder form. This new form is the builder for the property control we just added. In the Property combobox, select "Tooltiptext" and change the Caption to "Tool Tip Text" and the Width to 250. Move this builder form aside and notice that the property control on the original builder form has been changed accordingly. Click on the Add Another Property button in the second builder form and notice that another property control was added to the original builder form and this builder form now maintains it. Select "Statusbartext" for the Property and change the Caption to "Status Bar Text" and the Width to 250. Add another property control and select "Readonly" for the Property and change the Caption to "Read-Only"; this time, notice the property control in the original builder changes to a checkbox. Close the new builder form. Figure 3 shows the builder we've built and Figure 4 shows the property control builder.

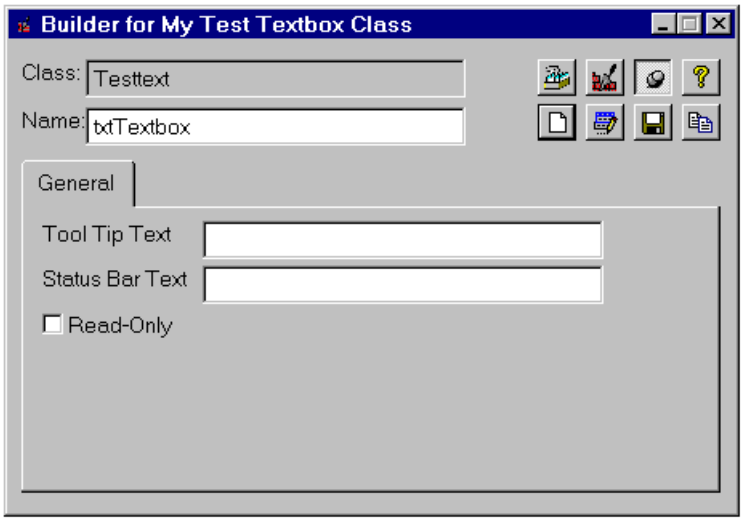

Figure 3. The BuilderD builder we've created for MyTestText objects.

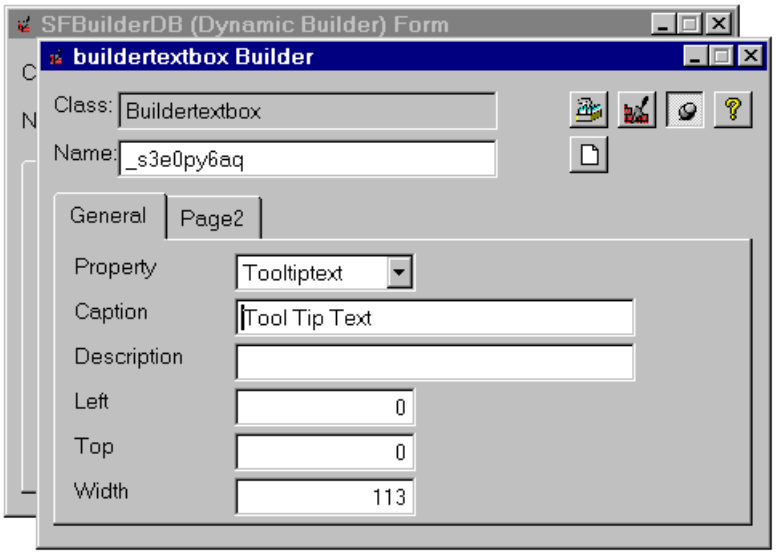

Figure 4. The property control builder.

To edit one of the property controls, right-click on it and select Edit Property Control from the context menu; the same property control builder you saw a moment ago appears. To remove the property control, choose Remove Property Control from the menu. To reset the value of this property in the target object to the default value, choose Reset to Default.

Let's change the Caption of the builder form to something more suitable. Click on the Edit Builder Caption button and enter "Builder for My Test Textbox Class" for the Caption in the SFBuilderBuilder Builder form that appears. Close this second builder.

If you close the builder without saving, the next time you bring up the builder for any MyTestText object, you'll have a builder with no properties again. To save the builder definition in the BUILDERD table, click on the Save Builder button. If you want to export the

builder definition to another table, click on the Export Builder button and enter a filename in the dialog that appears. You can then send this table to someone else so they can import it into their BUILDERD table and have access to the builder you created.

Summary

Because of VFP's open IDE, anyone can easily customize their development environment so they and their team can be more productive. I hope this document gives you some ideas about how you can improve your IDE.

Acknowledgements

I would like to acknowledge the following people who directly or indirectly helped with the information in this document: Steven Black, Jim Falino, Ken Levy, Rod Paddock, John Petersen, and Steve Sawyer.

Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341
Fax: (306) 586-5080
Email: dhennig@stonefield.com
World Wide Web: www.stonefield.com

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit and Stonefield Query. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP).