

Creating and Using Real World Builders – Made Easy

Session rsch1

*Richard A. Schummer
President*

*White Light Computing, Inc.
42759 Flis Dr.*

Sterling Heights, MI 48314

Voice: 586.254.2530

Fax: 586.254.2539

*E-mails: raschummer@whitelightcomputing.com
rick@rickschummer.com*

*Web sites: www.whitelightcomputing.com
www.rickschummer.com*

Overview

Builders are a handy way to set attributes on objects without opening up the Visual FoxPro Property Sheet or writing a line of code. How many times a day do you find yourself jumping to the Property Sheet, moving to the correct tab, and searching down the seemingly endless list to find that one property that you need to tweak? Dozens, hundreds, or does it just feel like a thousand? Builders are yet another shortcut to increasing your productivity in ways you may have not imagined. Right-click on the object, select Builder... from the menu, and let a builder do the work for you.

Builder technology has been around since Visual FoxPro 3.0, yet to this day when the topic of builders comes up in conversation amongst developers, it is usually met with blank stares. There are a number of native builders that ship with Visual FoxPro. Some are cool, some are okay, while others seem to be a little more than useless. The most important part of this technology is not the alternative property sheet user interfaces that ship with VFP, but the fact that they are extendible, even replaceable, and most of all easy to create.

This session will demonstrate builder technology inside of Visual FoxPro, how to leverage the existing builders (especially the cool ones included in VFP 8 and 9), and create and register your own builders (traditional and non-traditional ones). We will step through creating several builders from scratch, using the “old-fashion” builder techniques, using BuilderB techniques, and finally showing how the data driven BuilderD technique will knock your socks off!

Session attendees will learn how to...

1. What a VFP builder is and how it can increase their productivity.
2. What VFP builders to use and which ones to avoid.
3. How VFP builder technology is integrated into the development environment and how it can be exploited for their benefit.
4. How to create and register their own builders.
5. How to leverage builder technology without creating a “traditional” builder.
6. What the Builder and BuilderX properties are and how they can be useful.
7. What BuilderB technology is and how they can take advantage of it.
8. What BuilderD technology is and how they can exploit it
9. What are Property Editors and how are they implemented.

Skill Level/Prerequisites

Introductory to Expert developers, session will have something for all levels. Familiarity with the Form and Class Designer is a must.

What is a builder and why would I use one?

The builder technology was introduced with the debut of Visual FoxPro 3. Similar in some ways to the wizards that are built into Visual FoxPro, they assist us in various facets of application development. They are additional tools of our trade that shortcut operations we perform on the source code that is the foundation of the custom database applications we craft.

A builder is most commonly implemented as a user interface to the properties of an object. Developers that have used one of the native builders included with Visual FoxPro see the natural use, which is the extension or replacement of the Property Sheet. You can have a builder create properties, read the values in existing properties, and set the values in properties. You will see several examples of this type of builder throughout this whitepaper.

A builder can also read and write code in methods. This may not be apparent if you have only used the Visual FoxPro native builders that are available when using the Form or Class Designer. One example of this is the native Referential Integrity builder. It creates stored procedure code in the database for all the persistent relationships that you enforce a referential integrity rule. You can have a builder create methods as well. We have several code generation ideas presented in this whitepaper and demonstrate techniques of how this can be accomplished.

Wizards step you through a process to create something from scratch. Builders can also do this, but the one thing that distinguishes a builder from a wizard is that builders are re-entrant. This means that builders figure out what the existing settings are for an existing object and present those settings to the developer, or leverage those settings to do something.

I want to get the creative juices flowing even more. Think of builders as any tool that does something to assist you to develop code faster and more efficiently, removes repetitive tasks you perform, or runs a process that manually takes a lot of time and is performed regularly. Builders can add objects to a container; remove objects, transfer property settings and code from one object to another. A builder can write programs, create tables, generate stored procedures (in VFP and remote data), create reports, and even populate metadata.

A builder can be used to reduce exposure to specific properties and provide a set of valid settings. This is especially handy when you have new developers in your company or temporary contractors that need to learn your framework or the exact way you do development. An example of this situation that I have run into is a small company that used a commercial framework. They hired developers with little to no Visual FoxPro experience. The new developers could come up to speed on building forms very quickly following a set of instructions that stepped them through the entire process of creating a form (minus all the business logic). This made them productive, but was a manual process that allowed rookie developers to make mistakes. A builder could have done the majority of the steps and reduced the number of errors.

Custom properties are common in the classes and forms that we create. One of the disadvantages of custom properties is valid settings cannot be integrated into the Visual FoxPro Property Sheet (prior to VFP 9). We can do this quite easily using a builder. This is a key benefit since we often forget what we intended the property settings to be, or want an easy way to communicate to other developers what values can be assigned to these properties.

The last one I will discuss in this section is that builders can generate code, code headers, comments, and declare common variables we use like method or function return values. This can be a tremendous productivity booster.

What are the native builders included in VFP? *(Example: BUILDERDEMOVFP8.SCX)*

There are 15 native Visual FoxPro Builders that ship with Visual FoxPro 9 (see **Table 2**). The Member Data Editor is new in Visual FoxPro 9. The Cursor Adapter, Data Environment, and XML Web Service builders are all new in Visual FoxPro 8. These are the first builders added to the product since the Application Builder was added in Visual FoxPro 6.0.

We want to make sure you differentiate builders and wizards. The primary difference is that the wizards are executed once to get a desired result; builders are re-entrant, meaning that they read the settings and present information as it already exists on the object. Wizards may have default values for properties, and will step you through the process of creating what ever it is designed to create. If you run the wizard again, you can step through the same process, but nothing you previously created will be used in that process. Builders, if designed correctly, will read all the settings it is familiar with and present them for the developer to change. Thus you only need to change the items you want reflected in the object.

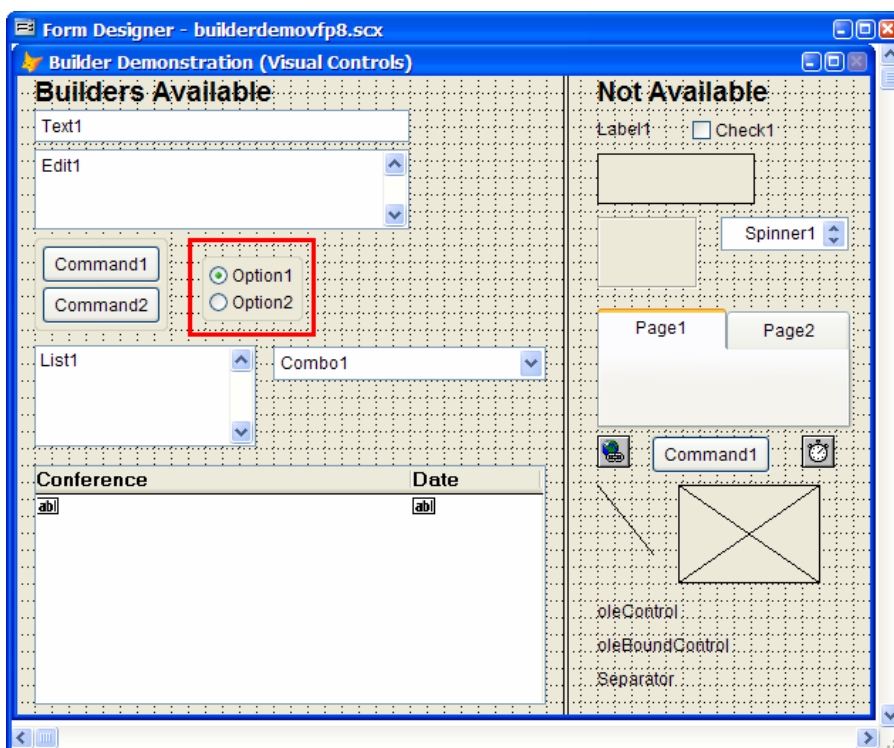


Figure 1. This form delineates the various controls that have builders (left side) and those that do not (right side).

The fundamental way the builders are started is to select the object (or objects) that you want to build. The context menu (displayed when you right-click on the selected object) will have a Builder... item. There are other ways to start a builder discussed in the section "How do I access Builders?"

Microsoft has obviously not created a builder for each and every object available in Visual FoxPro. In fact, Visual FoxPro 9 now has a total of 44 baseclasses, (see **Table 1**) so this means that there is lots of opportunity for developers to create builders. That said, there are a number of builders already created that we can leverage to increase our productivity (see Table 2).

Table 1. List of native Visual FoxPro baseclasses

ActiveDoc (deprecated)	Custom	Label	ReportListener
Checkbox	DataEnvironment	Line	Separator
Collection	EditBox	ListBox	Session
Column	Empty	OleBoundControl	Shape
ComboBox	Exception	OleControl	Spinner
CommandButton	Form	OptionButton	TextBox
CommandGroup	Formset	OptionGroup	Timer
Container	Grid	Page	Toolbar
Control	Header	PageFrame	XMLAdapter
Cursor	HyperLink	ProjectHook	XMLField
CursorAdapter	Image	Relation	XMLTable

Table 2. List of Visual FoxPro Native Builders

Builder	Description
Application	Displays a dialog to simplify creating and modifying forms, reports, complex controls, similar to the Project Manager. Additionally you can set up some of the build properties available on the Build Version dialog. If you are using the VFP provided framework or install the application builder metadata (almost required) you will have additional functionality.
AutoFormat	Displays a dialog that allows you to apply a set of styles to selected controls of the same type. Styles can be optionally applied to borders, colors, fonts, layout, and 3D effects.
Combo Box	Displays a dialog to set properties for a combo box including the RowSourceType and RowSource, Style, special effects, use of incremental searching, number of columns, and how it is bound to data. Only supports RowSourceType of Array, fields, and none.
Command Group	Displays a dialog for you to set properties for a command group. If you use command groups it definitely simplifies adding addition command buttons to the group, change captions, and select the layout (vertical or horizontal), as well as controlling the spacing between buttons.
CursorAdapter	Displays a dialog to build CursorAdapter objects quicker than typing all the settings in the property sheet. (new in VFP 8)
DataEnvironment	Displays a dialog to build dataenvironment objects that use the capabilities of cursoradapter objects. Allows you to select a datasource (native VFP data, ADO, ODBC, or XML), then add predefined or new cursoradapter objects. Can call the cursoradapter builder from the dataenvironment builder. (new in VFP 8)
Edit Box	Displays a dialog to set properties for an edit box. There is a plethora of common format settings to choose, the standard special effect, bordering, alignment, and the ControlSource
Form	Displays a dialog to add fields as new controls to a form. Additionally you can set the controls to follow any of the VFP provided styles commonly available in the form wizard.
Grid	Displays a dialog to set properties for a grid after selecting the RecordSource (free or database contained table). You can pick a VFP wizard based style, adjust column widths, Captions, change the control in the column, and set up parent child relationship based grids.
List Box	Displays a dialog to set properties for a list box including the RowSourceType and RowSource, Style, special effects, use of incremental searching, number of columns, and how it is bound to data. Only supports RowSourceType of Array, fields, and none.
Member Data Editor	Displays a dialog to allow developers to edit meta data associated with class members. Resulting XML metadata is stored in the new _memberdata property and interpreted in the development environment to configure the Property Sheet. (new in VFP 9)
Option Group	Displays a dialog for you to set properties for an option group. It definitely simplifies adding addition option buttons to the group, change captions, and select the layout (vertical or horizontal), as well as controlling the spacing between buttons. Additionally you can quickly change between standard and graphical option buttons.
Referential Integrity	Helps you to ensure referential integrity of tables by generating trigger

Builder	Description
	code to control rules on how records are inserted, updated, or deleted in related tables. Options are based on VFP database and the persistent relations set up before running the RI builder.
Text Box	Displays a dialog for you to set properties for a text box including the ControlSource, data type, InputMask (with common ones predefined to select from), the special effects, and other common settings.
XML Web Service	Allows you to bind an XML Web service to a control on a Visual FoxPro form or to an object. One example of how this can be used is a cursor adapter in the data environment. (new in VFP 8)

All the source code to the builders (as well as wizards and Xbase tools) is included in the VFP Tools\XSource\ directory in a file named XSource.zip. If you do not find the builders to your liking you can always change them or customize them to your needs.

What happens if there is no builder for selected object?

In the previous section we listed the 44 baseclasses in Visual FoxPro and noted that there are only 15 native builders. So odds are pretty good that you will try to bring up a builder for an object that does not have a registered builder. Visual FoxPro does an admirable job in this situation by displaying a message box indicating this very situation.

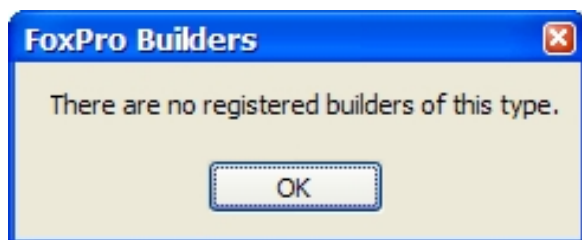


Figure 2. Visual FoxPro will kindly tell you when there are no registered builders available for the control you selected (either for the specific control/object, or the ALL type builders).

The no builder message is not display if you have the Builder Lock option set (see “Form Control toolbar builder lock” section later in this whitepaper).

How do I access Builders?

There are several ways to access builders. The most familiar way is to right-click on an object in the designer and select the builder option from the context menu. Microsoft has included several ways to run a builder in the development environment.

Control Context Menu

The most common way that a builder is accessed is through the control’s context sensitive menu. Right-click any control in the Form or Class Designer and select the Builder... menu item. The appropriate builder will be executed.

Form Control toolbar builder lock

The Form Control toolbar is available when the Form or Class Designer is active. If the Builder Lock is set, each time you drop a control on a form using the Form Control toolbar, the builder for that control will be run if one exists. If one does not exist there is no action taken.

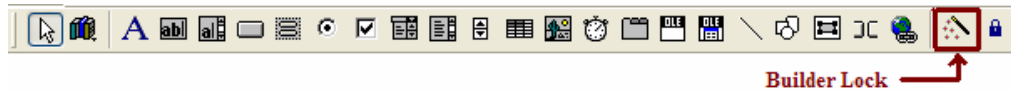


Figure 3. The Form Control toolbar Builder Lock is the second from last toolbar button.

The selection of the Builder Lock option is not persistent between designer sessions. If the designers are closed or the toolbar is manually closed, the Builder Lock selection is turned off. You will need to re-select it the next time the toolbar is activated.

Form Designer Toolbar

The Form Designer toolbar is optionally displayed when a form or form class is opened. The last two buttons on this toolbar are the Form Builder and AutoFormat Builder respectively.

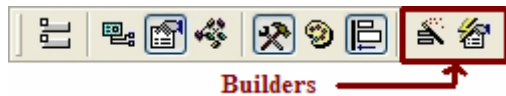


Figure 4. The Form Designer toolbar is available when the Form Designer is opened.

Project Builder from Project Manager

The Project Manager might initially sound like an odd place to find hooks for a builder, but there has been an Application Builder available since VFP 6. The hook to the builder is included in the right-click context menu.

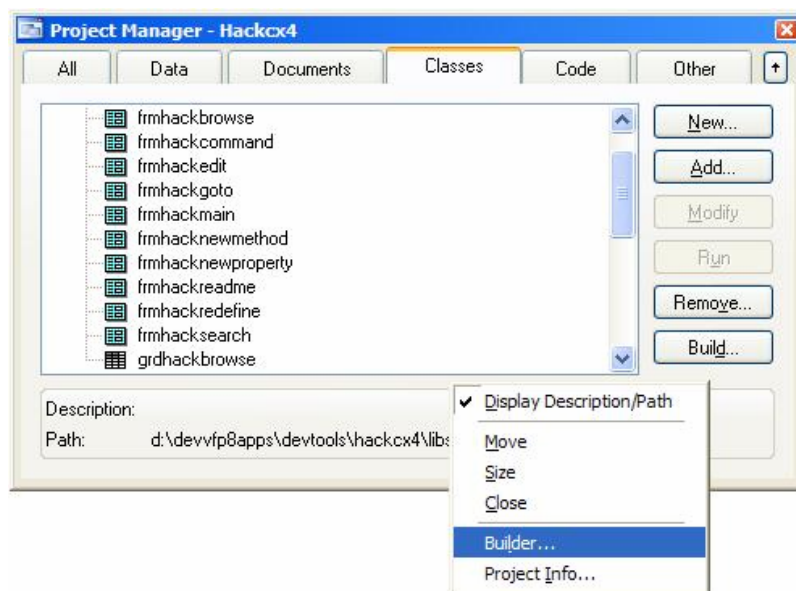


Figure 5. The context menu for the Project Manager has a Builder... option available to call up the Application Builder, Web Services Builder or one of your own custom project based builders.

Toolbox

The Toolbox introduced in Visual FoxPro 8 works very similarly to the Form Controls toolbar. You can drop controls on forms and classes in the native designers. If you want the builder to run when the control is dropped, you need to make a setting for the Toolbox that indicates this.

There are two ways to accomplish this. The first is to right-click on the Toolbox and select the Builder Lock context menu item. The second way is to get to the Toolbox's Customize Toolbox dialog and select the "Run associated builder" option found on the General Options page.

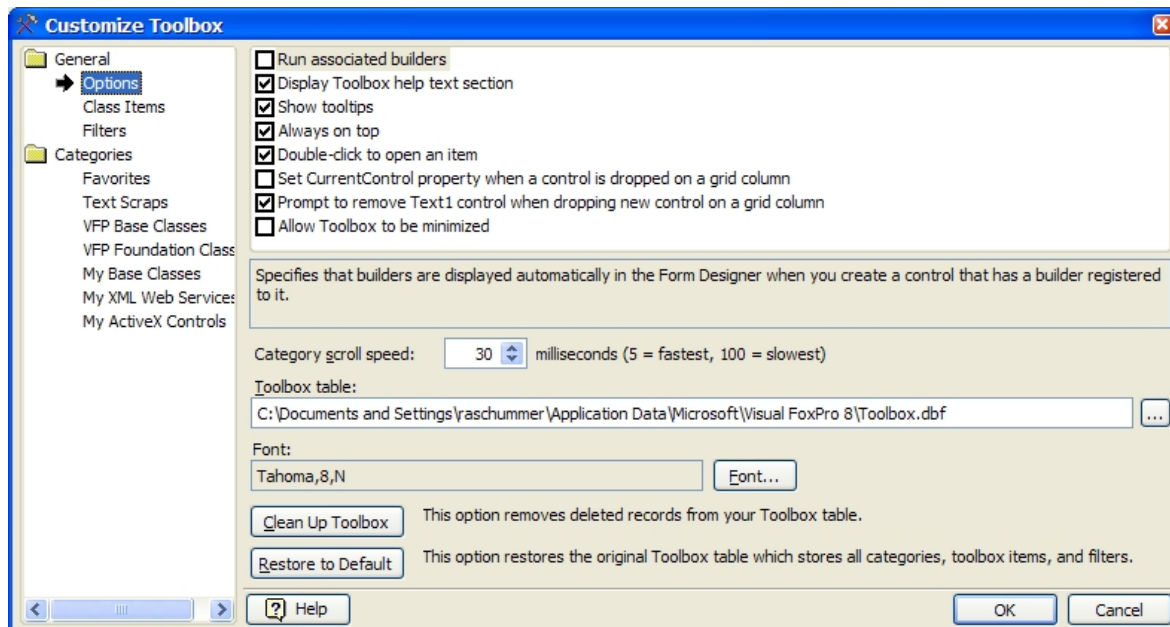


Figure 6. The Customize Toolbox options dialog has the option of "Run associated builders" on the General Options page.

Just like the Form Controls toolbar, if there is more than one register builder for the selected object, you will be presented with a dialog to pick which builder you want to run. The selection of the Builder Lock option is persistent between Toolbox sessions. So even if you close the Toolbox, the next time you open the Toolbox, the Builder Lock setting will be the same as what it was set to the last time you used it.

VFP Options Dialog

The Options dialog (found using the Tools | Options menu item) has a Builder Lock setting on the Forms page. This is the default behavior for the Form Control toolbar builder lock. This behavior is stored in the registry if you choose to "Set As Default".

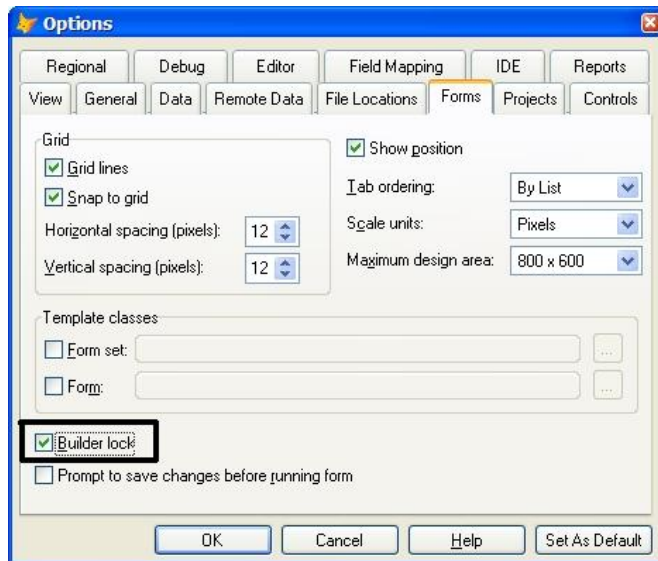


Figure 7. The Builder Lock is available on the Form page of the Tools | Options dialog.

If this is set on, each time the Form Control toolbar is loaded (when you open the Form or Class Designer or manually open it via the View menu) the Builder Lock checkbox will default on. You can override this temporarily, but the next time the Form Control toolbar opens up it will be back on.

Run it direct from Command Window or menu

There is nothing to stop you from running a custom builder from the Command Window or a menu. It is not how the native builders work, but this is not a limitation, rather it is just how Microsoft implemented the native builders. You do not have to register the builder in the builder registration table (discussed in the section “How does Visual FoxPro implement builder technology?”). You can run your program, form, app, etc. directly from the Command Window or add it to a developer menu.

How does Visual FoxPro implement builder technology?

Microsoft implemented a very elegant and highly flexible technology for builders. Simply stated, each time you select a builder to be run (see section “How do I access Builders?” for all the various techniques) the builder manager program runs and determines which builder to execute, or if there is more than one builder to run, present a list to select from. This is implemented in a fashion that is extendible and if not to your liking, completely replaceable.

BUILDER and Builder.app

Microsoft provides hooks into all of what it refers to collectively as the “Xbase tools” (called this because they are written in the VFP language, not C++ like the core product). These hooks are available with the system memory variables that start with the underscore. In the case of the builders, the name of the builder manager program is stored in BUILDER. The default builder manager is called Builder.app.

What does the builder manager do exactly? Microsoft has made the code to the builder manager available for us to look at and modify if we choose. I have not seen a reason to write my own builder manager program since the builder technology Microsoft implemented is quite

flexible. I will not go into great detail about what the builder manager does (since the code has a ton of error checking and validation), but here is a basic outline on how it works:

- Accepts parameters (possible object reference, origin of the call to the builder [property sheet, right-click of mouse, or the toolbox], and nine optional parameters to pass along to the builder)
- Parameter checking, validation, and determination of target object for builder
- Run the builder if directly specified with one of three conditions
 - If the name of the builder is passed to the builder manager (third parameter), run this builder.
 - If the object has a *BuilderX* property, run the builder specified in that property.
 - If the object has a *Builder* property, run the builder specified in that property.
 - Return value returned from the builder.
- -OR- run the builder from the registration tables if not directly specified
 - Hide the Property Sheet if not docked
 - Determine if MULTISELECT or AUTOFORMAT
 - Look up list of registered builders
 - Display list if more than one to select
 - Run the builder selected or run only builder for that object
 - Activate the Property Sheet (always done whether builder was run or not)
 - Not sure why, but the builder return value is never returned, unlike the specified builders

There are some interesting concepts that are not apparent unless you read the builder manager code. If the *Builder* or *BuilderX* properties have a value of a question mark (?), the builder manager will prompt you for the program to run (prg, scx, app). If the *Builder* or *BuilderX* properties have a value of an asterisk (*), the builder manager will set up an object reference to the selected object in a public memvar called "o" and activate the Command Window. This allows you to interactively work with the object. We are not sure why all this effort was given when we can do the same thing with `sys(1270)`, but it is there for your use.

NOTE:

You can switch between builder managers if you have your own customized version. All you have to do is change `_BUILDER` from the current setting to your program. This can be helpful if you have a customized builder that does not fit the model developed by Microsoft. You can even toggle between multiple managers if this fits your needs.

I hope this outline provides you with an understanding of how the builder technology is implemented internally. I also encourage you to review this code in detail if you have an interest.

The source code can be found in the VFP\Tools\Xsource\VFPSrc\Builders directory. It will also provide you with some key specifications if you decide to build your own builder manager.

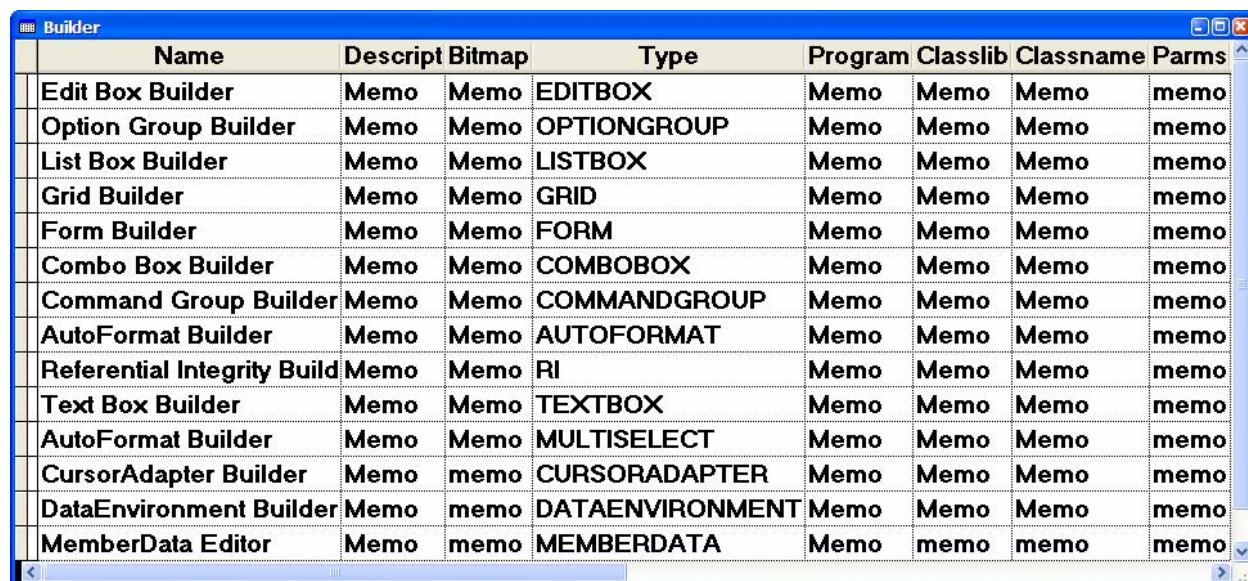
The Registration Tables

The list of builders available to the builder manager is stored in two registration tables. If these tables do not exist, Visual FoxPro will prompt you to locate the table or offer to build a new one with the default native builders. The tables are in the VFP\Wizards directory.

The first table is the Builder.dbf/fpt. This table holds all the builders except for Project builders (explained later in this section). You can add records which register new builders. The structure for this table is specified in **Table 3**.

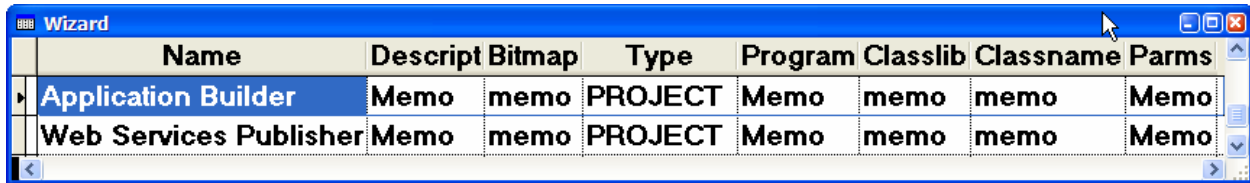
Table 3. File layout of the Builder.dbf and Wizard.dbf.

Name	Type	Description
Name	Character (45)	This is the name of the builder. This is displayed in the selection dialog if more than one builder is available for the object selected.
Descript	Memo (4)	This is the description of the builder. It is text that is displayed at the bottom of the builder selection dialog that provides the developer with a bit more information on what the builder is or can do.
Bitmap	Memo (4)	Not used
Type	Character (20)	This is the uppercased object baseclass, or identifier of the type of builder it is. This can be the baseclass for an object, "RI" for the referential integrity, "ALL" will be available for all objects, "MULTISELECT" for builders that act on multiple selected objects in a designer, and "AUTOFORMAT" for builders that work on formatting multiple objects.
Program	Memo (4)	This is the program name (fully pathed, relatively pathed, or available on the VFP path. This is used exclusive of the ClassLib/ClassName columns.
ClassLib	Memo (4)	This is the class library that the ClassName resides in. The builder must reside in this class library to be run.
ClassName	Memo (4)	This is the class name of the class that is the builder.
Parms	Memo (4)	Additional parameter(s) to pass to the builder.



	Name	Descript	Bitmap	Type	Program	Classlib	Classname	Parms
	Edit Box Builder	Memo	Memo	EDITBOX	Memo	Memo	Memo	memo
	Option Group Builder	Memo	Memo	OPTIONGROUP	Memo	Memo	Memo	memo
	List Box Builder	Memo	Memo	LISTBOX	Memo	Memo	Memo	memo
	Grid Builder	Memo	Memo	GRID	Memo	Memo	Memo	memo
	Form Builder	Memo	Memo	FORM	Memo	Memo	Memo	memo
	Combo Box Builder	Memo	Memo	COMBOBOX	Memo	Memo	Memo	memo
	Command Group Builder	Memo	Memo	COMMANDGROUP	Memo	Memo	Memo	memo
	AutoFormat Builder	Memo	Memo	AUTOFORMAT	Memo	Memo	Memo	memo
	Referential Integrity Build	Memo	Memo	RI	Memo	Memo	Memo	memo
	Text Box Builder	Memo	Memo	TEXTBOX	Memo	Memo	Memo	memo
	AutoFormat Builder	Memo	Memo	MULTISELECT	Memo	Memo	Memo	memo
	CursorAdapter Builder	Memo	memo	CURSORADAPTER	Memo	Memo	Memo	memo
	DataEnvironment Builder	Memo	memo	DATAENVIRONMENT	Memo	Memo	Memo	memo
	MemberData Editor	Memo	memo	MEMBERDATA	Memo	memo	memo	memo

Figure 7. This is the entire list of native builders in the Builder.dbf registration table.



	Name	Descript	Bitmap	Type	Program	Classlib	Classname	Params
▶	Application Builder	Memo	memo	PROJECT	Memo	memo	memo	Memo
	Web Services Publisher	Memo	memo	PROJECT	Memo	memo	memo	Memo

Figure 8. This is the entire list of native builders in the Wizard.dbf registration table.

The Project builders reside in the Wizard.dbf/ftp. This is not real obvious, but it is how it works. If you want to create your own project builder you need to add a record to this table.

How does Visual FoxPro know which Builder.dbf to use? (Example:

CHECKBUILDERREGINFOXUSER.PRG)

Visual FoxPro stores the location of the Builder Registration Table in the FoxUser.dbf resource file. This information is stored in a record identified with ID = "BUILDERS". To locate the record, execute the following code:

```
USE SYS(2005) AGAIN ALIAS curFoxUser
SET FILTER TO id = "BUILDER"
BROWSE
SET FILTER TO
USE IN curFoxUser
```

The location details are found in the Data column. It is my experience that this information is correct most of the time. During product betas the information occasionally points to another Builder Registration table, or if you share a resource file across different versions of VFP (not recommended). Altering the data in the Data column is fine. There is no checksum problem by correcting the pointer to the correct table.

How do I register a custom builder?

Adding your own builder to this table is as simple as **APPEND BLANK** and filling in the appropriate columns. I previously documented the tables columns in **Table 3**. Minimally you need the Name, Type, and either the Program or the ClassLib/ClassName columns. I also recommend filling in the Descript column for documentation purposes, especially if you make the builder available to other Visual FoxPro developers.

Here is a simple program code that registers a builder that resides in a class:

```
LOCAL lnOldSelect, ;
      lcBuilderName

lnOldSelect  = SELECT()
lcBuilderName = "WLC Label Builder"

USE (HOME()+"Wizards\Builder.dbf") AGAIN SHARED ALIAS curRegBuilder IN 0
SELECT curRegBuilder

LOCATE FOR Name = lcBuilderName

IF FOUND()
  * Already registered
  MESSAGEBOX(lcBuilderName + ;
    " was already registered with Visual FoxPro v" + VERSION(4), ;
    0+64, lcBuilderName)
ELSE
  SCATTER MEMVAR MEMO BLANK
```

```

m.Name      = lcBuilderName
m.Descript  = "Applies a properties to a label and allows " + ;
             "alignment with another object."
m.Type      = "MULTISELECT"
m.ClassName = this.Class
m.ClassLib  = this.ClassLibrary

INSERT INTO curRegBuilder FROM MEMVAR

MESSAGEBOX(lcBuilderName + ;
           " has been registered with Visual FoxPro v" + VERSION(4), ;
           0+64, lcBuilderName)
ENDIF

USE IN (SELECT("curRegBuilder"))
RETURN

```

If the builder is program or application based I would change the code to put the `sys(16, 0)` in the Program column instead of the class and class library in those columns.

There is nothing stopping you to register a builder for multiple objects. For instance, you might create a builder that will add code to the dataenvironment's *BeforeOpenTable* method. It might make sense to register the same builder as a form builder and a dataenvironment builder.

I create a method for class based builders and a procedure for program based builders called *SelfRegister*. In this method I add the code necessary to register the builder in the current version of Visual FoxPro. Since I have versions as far back as VFP 5, it is nice to be able to have the builder self register themselves in the appropriate table. It is also a nice feature for builders that I distribute to other developers in the Fox Community. It might be a good idea to check the version of Visual FoxPro you are registering your builder in since the builder may require a specific version to run without errors.

Can I have more than one builder for an object?

You absolutely can have more than one builder for a specific baseclass or set of classes. It is part of the extensibility of the builder technology included in the product. Visual FoxPro ships with only one per object (except the Project), but you can add your own builders. When you have more than one builder registered a dialog is presented for you to select which one you want to run when you initiate the builder.

This might sound strange, but you will be presented with one of two dialogs when multiple builders are registered. Most of the time you will be presented with the Builder Selection dialog except when you run a project builder. You will be presented with the Wizard Selection. The reason for this is that Microsoft implemented the project based builders in the Wizard.dbf. This is a bug in my opinion, but by design according to Microsoft. No big deal, just confusing to those of us who have tried to implement a project builder (see third-party tools section later in this whitepaper).

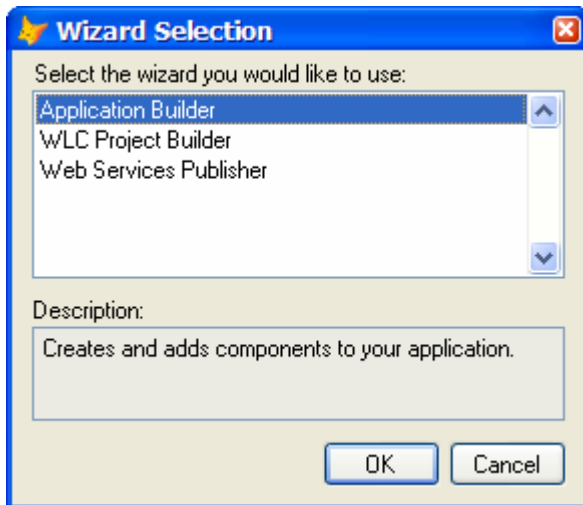


Figure 9. The Wizard Selection dialog might not sound intuitive as a Builder Selection dialog, but this is what is displayed when multiple project based builders are registered.

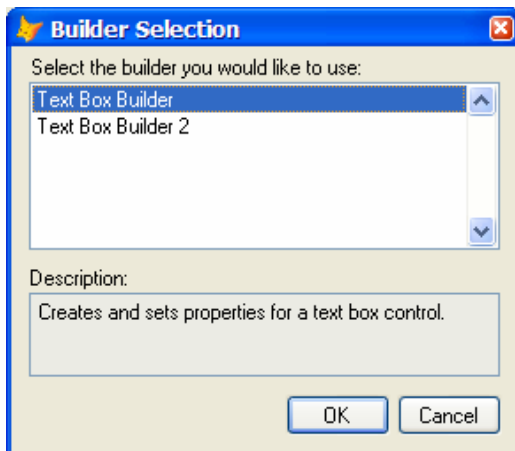


Figure 10. The Builder Selection dialog is displayed for all other builders that have more than one builder registered.

NOTE:

One word of caution with respect to naming the builders: it appears that when Visual FoxPro internally queries the Builder.dbf and Wizard.dbf it applies the distinct clause on the SQL. If you register two different builders with the same name (builder.name) the list will only show one.

NOTE:

Another word of caution with respect to deleting the registration of a builder: prior to Visual FoxPro 9, it appears that when Visual FoxPro internally queries the Builder.dbf and Wizard.dbf it does so with SET DELETED OFF. If you delete a builder and do not pack it, it will still show up in the selection dialog. This behavior was corrected in Visual FoxPro 9.

Which native builders are worthy of regular use?

So what native builders are worthwhile and assist in increasing your productivity? Determining the worthiness of anything is arbitrary. The criteria I use to make the list of builders that get regular use is really simple: increased productivity. If I can use a builder to do something faster than using the Property Sheet or some other manner, then the builder made this list.

Referential Integrity (RI)

The Referential Integrity Builder is the fastest way to build the RI code for a Visual FoxPro database. It generates code that is not exactly optimized and can write code that will exceed the 64K compiled code barrier (prior to VFP 9), but considering the alternative of writing your own code it is a good starting point.

I can say without reservation that one of the best developers in the FoxPro Community is Doug Hennig. He has written a replacement for the standard Visual FoxPro Referential Integrity Builder. His builder is based on the VFP RI Builder, but fixes a couple of bugs, and integrates Steve Sawyer's superior Referential Integrity code (called NEWRI.PRG, available in the Effective Techniques for Application Development in Visual FoxPro 6.0 book from Hentzenwerke Publishing and on his website at www.StephenSawyer.com).

NOTE:

More details on the Stonefield RI Builder and the implementation are in a white paper from Stonefield Group that can be downloaded from <http://stonefield.com/pub/devtools.zip>. It is also discussed in more detail later in this whitepaper in the "What third-party builders are available?" section.

Both the native and Stonefield RI builders present a developer with a list of related tables. The list is based on the persistent relations set up in the database. You then select how you want the rules enforced for updating, deleting, and inserting records into the tables.

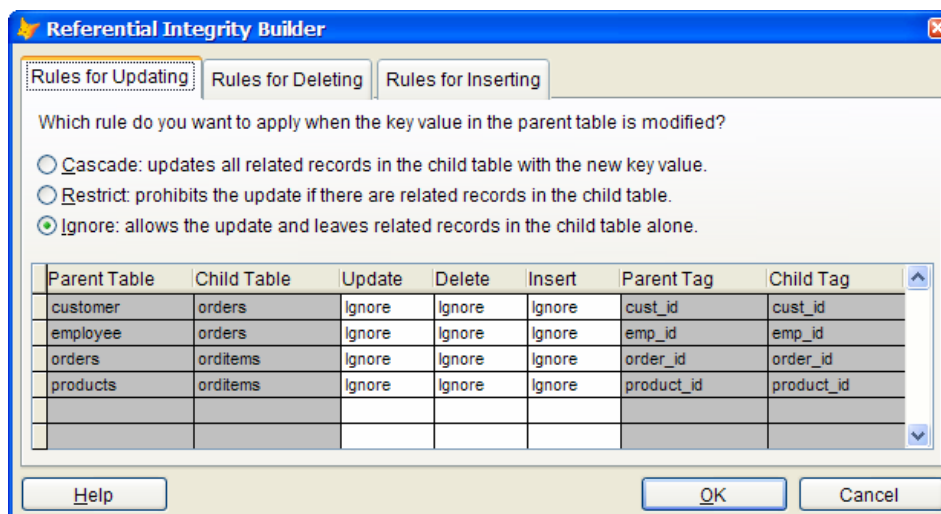


Figure 11. The Referential Integrity Builder simplifies the creation of trigger code to enforce rules for updating, deleting, and inserting records into tables.

Option Group

The Option Group object has a number of productivity killers built into the object. First, we never seem to have just two options so we have to add more options. Once we add them we need to traverse the Property Sheet to change the Caption properties, and then there is the tedious task of lining up all the options and make sure they are all evenly spaced.

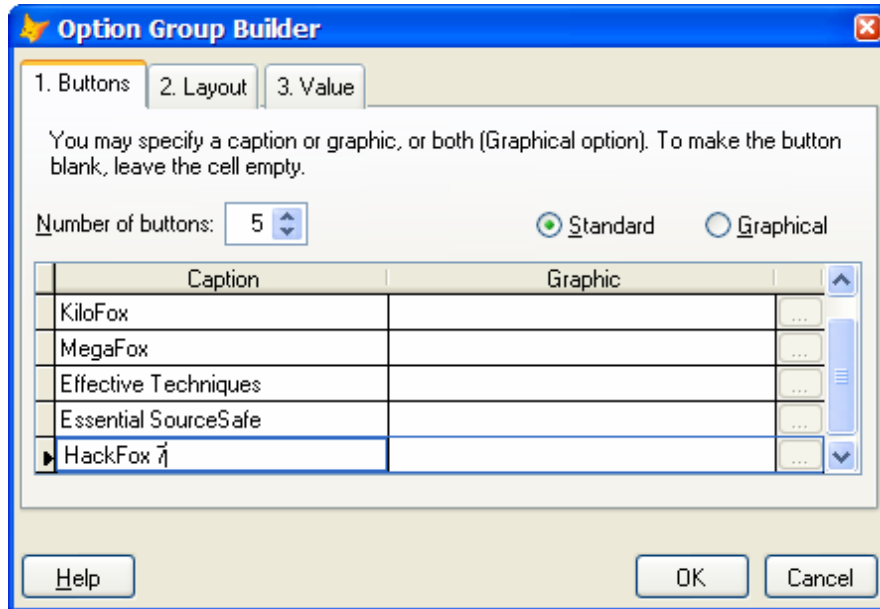


Figure 12. The Option Group Builder simplifies adding options and determining if the options are horizontal or vertical.

The Option Group builder actually assists in quickly adding options, and changing the Captions on the Buttons page. The Layout page allows developers to determine if they want the options aligned vertically or horizontally, and how many pixels to evenly distribute them.

If you are using Visual FoxPro 8/9, the Option Group Builder respects the *MemberClass* and *MemberClassLibrary* properties used for the option group when adding new option buttons.

DataEnvironment

The DataEnvironment builder productivity is in selecting the datasource. Use the interface to select the datasource, and the builder makes the corresponding property settings and writes code in the *BeforeOpenTables* method

```
*** Select code: DO NOT REMOVE
set multilocks on
***<DataSource>
This.DataSource = sqlstringconnect([dsn=NorthwindSQL;])
***</DataSource>
*** End of Select code: DO NOT REMOVE
```

The second page of the builder provides a simplified selection of existing cursor adapters and the ability to add new cursor adapters. You can add the cursor adapters via the data environment, but the builder allows you to add cursor adapters that are already created and stored in a visual class library. If you add them directly from the data environment it will add a baseclass cursor adapter.

CursorAdapters

The cursor adapter class is a class that offers support for handling a wide range of local or remote data source as a native VFP cursor. Datasources supported include the following:

1. VFP tables
2. Open Database Connectivity (ODBC)
3. ActiveX Data Object (ADO)
4. Extensible Markup Language (XML)

The builder has nearly all the features that you find when creating a view, but without the overhead of a database container. This is one builder that is almost indispensable. Yes, you can create a cursor adapter class and set all the corresponding properties, but it is far more productive to use the builder.

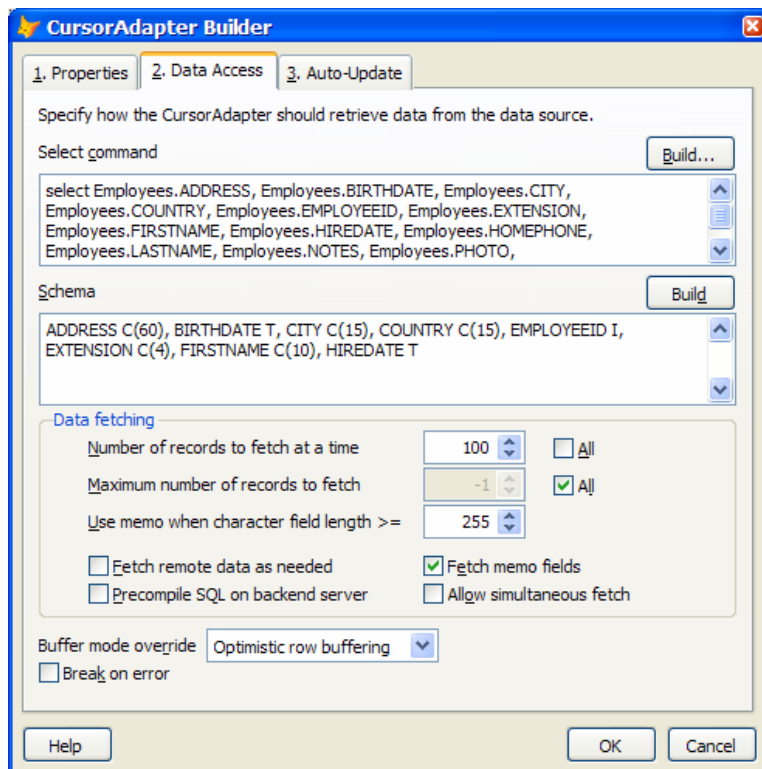


Figure 13. The CursorAdapter Builder can be called from a cursor adapter in the dataenvironment or from the Class Designer when editing a cursor adapter.

Member Data Editor

Doug Hennig wrote a little tool to help get you started on the possibilities with `_memberdata` and has an article on this topic in the June 2004 issue of FoxTalk 2.0 which will give you all the details of how this is implemented using the IntelliSense metadata. His discussion includes a demonstration on how you can extend this architecture to create “property builders”. This tool is now a builder included in the base product and has been improved significantly since the FoxTalk article.

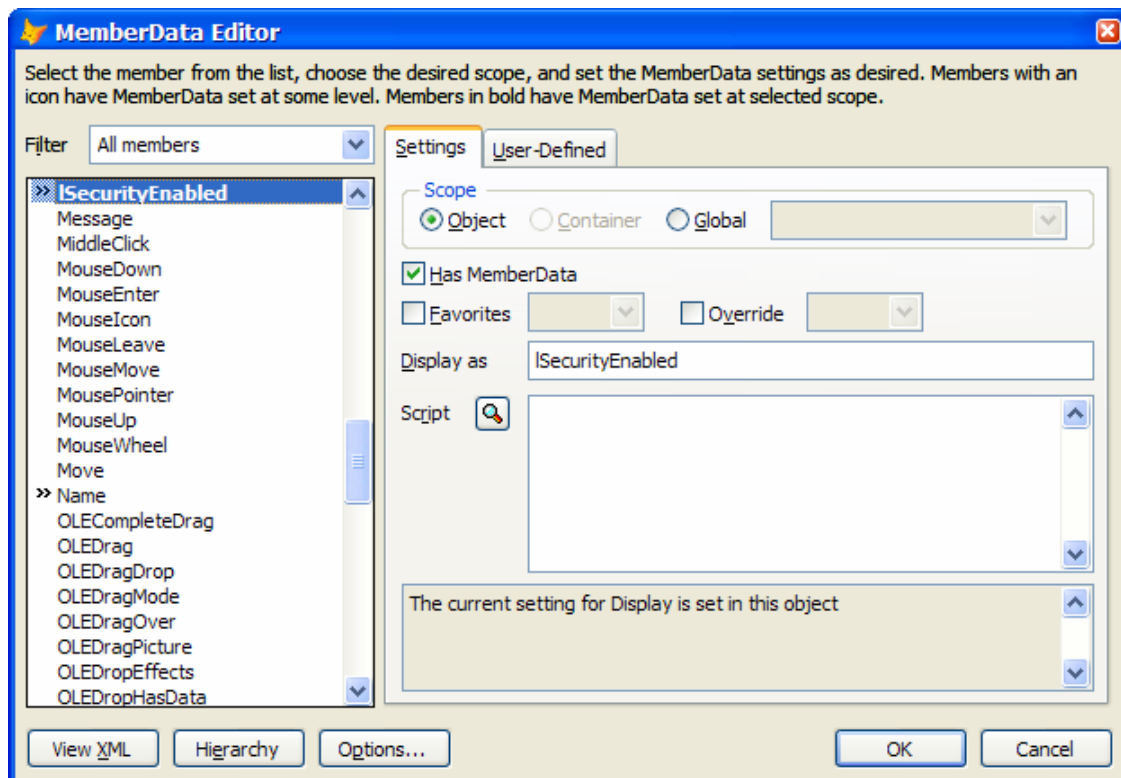


Figure 14. The Visual FoxPro Member Data Editor builder is a sure-fire way to simplify the editing of `_memberdata`.

The same mixed case is displayed in the editor through IntelliSense.

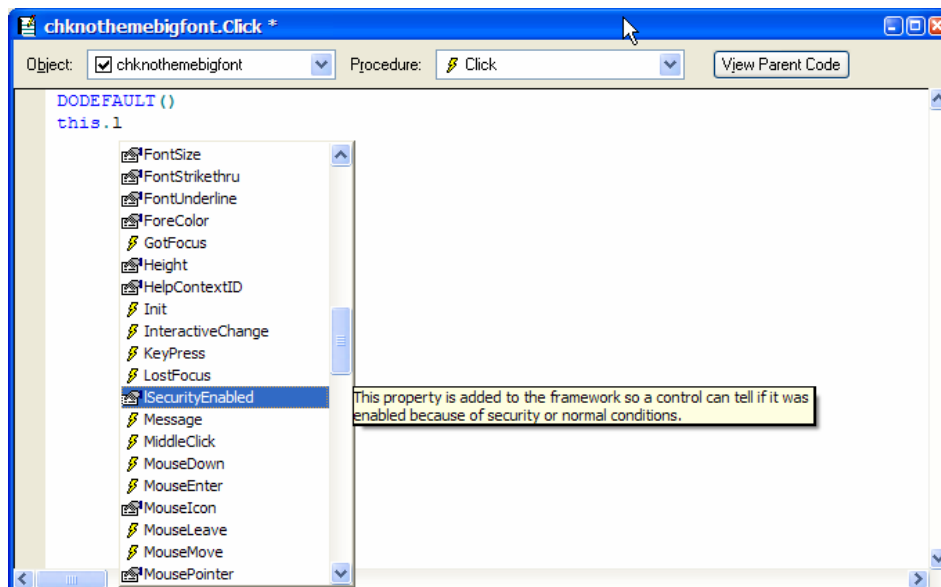


Figure 15. IntelliSense is even more productive with the inclusion of the mixed case members.

The builder does all the heavy lifting with respect to generating the XML metadata stored in the `_memberdata` property.

NOTE:

A word of caution with respect to `_memberdata` and backwards compatibility with VFP 8 and earlier: if you create member data that exceeds 255 characters, the Class Designer and the Form Designer will not be able to edit the class or form with the extensive member data. VFP 9 can deal with the property values greater than 255 characters, the earlier version have a limitation of 255 characters.

What are the important commands and functions for implementing a builder?

Visual FoxPro developers often read through the help file to see what commands are available as they develop custom software for their clients. I am not sure about you, but when I do this I often come across commands that I wonder about. “How can this be used in a production application?” Often I realize at some point that these commands are not designed to be used in a custom application, rather the command is constructed to make developer tools. I have listed several commands that I often use in builders in **Table 4** and why they are useful.

Table 4. A list of Visual FoxPro commands commonly used in builders.

Command	Use
ASELOBJ()	We see several code examples in this whitepaper that demonstrate how this function is a centerpiece of almost every builder I have written. It gets the object reference to the currently selected object(s) in the designer. If there is more than one object reference, then the collection with the object references has more than one row. You have to write code specific to the condition of one object or more than one object. Using ASELOBJ() is the easiest way to guarantee a reference to the object(s) that the builder processes.
SYS(1270)	This function will get a reference to the object under the mouse. This object reference could be used by the builder or passed into the builder as the first parameter. This is a different approach than to use ASELOBJ().
AMOUSEOBJ()	See SYS(1270)
WriteExpression / ReadExpression	These two methods allow a developer to read from and write expressions to properties that are evaluated at runtime. You can also use the approach to save and set property settings using the normal assignment code.
WriteMethod / ReadMethod	These two methods allow a developer to read from and write code to methods of the object. If the method does not exist and one performs a WriteMethod, the method is optionally created.
ResetToDefault	Just like the interaction with the VFP Property Sheet, a developer can reset the property or method code so inheritance is no longer broken.

How do I create my own builder?

Implementing builders is fairly easy once you know all the techniques involved. This section will discuss techniques for the manual creation of a builder using a program and a class.

Writing a builder from scratch has some definite advantages:

- You have more control
- It is easy to integrate into existing VFP builder implementation
- You can write them as a program, class, form, or application
- Customizable without fitting it to predefined architecture
- It can be educational

The disadvantages to creating builders from scratch include:

- More grunt work
- Individual builders for each type of class

Create a class *(Example: WLCBUILDERS.VCX:: FRMCHECKBOXBUILDER)*

Creating a class is no harder to create than a program, but it provides new challenges when binding controls to the properties that they will change. The example I have created for this is the simplest builder I could think up. The checkbox object does not have a native builder so I thought that might be a good example to tackle. The checkbox builder exposes the *AutoCenter*, the *Alignment*, and the *BackStyle* properties. Interacting with these objects directly changes the properties of the target object. These changes are reflected in the designer and the Property Sheet.

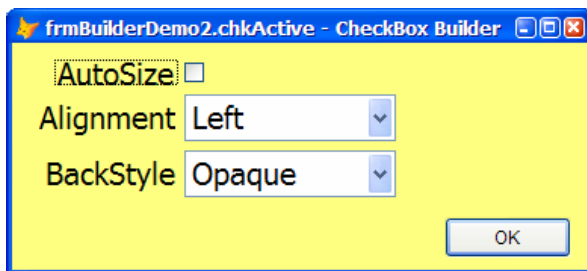


Figure 16. The Checkbox builder example demonstrates how developers can bind controls to the properties of the object that the builder is working with.

The base code in the builder form (frmCheckboxBuilder) is in the Load() event method:

```
ASELOBJ(thisform.aChange)

IF VARTYPE(this.aChange[1]) # "L"
    IF LOWER(this.aChange[1].BaseClass) = "checkbox"
        this.Caption= this.aChange[1].parent.Name + "." + ;
                      this.aChange[1].Name + " - " + this.Caption
    ELSE
        MESSAGEBOX("Object selected is not a checkbox.", ;
                    0 + 16, ;
                    "Checkbox Builder")
        RETURN .F.
    ENDIF
ELSE
    MESSAGEBOX("No object selected for this builder.", ;
                0 + 16, ;
                "Checkbox Builder")
    RETURN .F.
ENDIF

RETURN .T.
```

This code gains a reference to the currently selected object in the Form or Class Designer. The first check is to see if a reference to an object was obtained and then checks to see if the object is a checkbox. If it is, the builder form caption property is changed to reflect the name of the target object that the builder is run against. This is all the necessary code in the builder. The code in the Init method is only for demonstration purposes.

The user interface controls have the *ControlSource* property bound to the object reference stored in the form array property called *aChange[]*. The three controls are bound like this:

```
thisform.chkAutoSize.ControlSource = "thisform.aChange[1].AutoSize"
thisform.cboAlignment.ControlSource = "thisform.aChange[1].Alignment"
thisform.cboBackStyle.ControlSource = "thisform.aChange[1].BackStyle"
```

That is all there is to setting up a simple builder. First gain the reference to the object, establish this reference in the Load method, and bind the controls to the properties of the object.

Write a program (Example: *GENPROJECTHOOK.PRG* and *BEFOREOPENTABLESCODEBUILDER.PRG*)

Most developers think of builders as a form with objects to interact and set properties. This does not have to be the case. The `ASELOBJ()` function gains the object references in the designer for a form interface, but it gains the object references for any selected objects in the foremost designer.

One example is to programmatically subclass an object and set properties.

Listing 1. This is a partial listing of the *GenProjectHook* program.

```
#DEFINE ccPROJECTHOOKSLIB "D:\pres\FishingWithProjectHook\ cPhkDevelopment"
#DEFINE ccPROJECTHOOKSBASE "phkCompanyStandard"

CREATE CLASS (tcProjectHook) OF ccPROJECTHOOKSLIB ;
    AS ccPROJECTHOOKSBASE FROM ccPROJECTHOOKSLIB NOWAIT

* Set the class property for the projecthook cFieldMappingCategory property
ASELOBJ(laProjectHookRef, 1)

IF TYPE("laProjectHookRef[1]") = "O"
    laProjectHookRef[1].cFieldMappingCategory = tcProjectFieldMappingConfig
ENDIF

* Make sure the reference to the projecthook is released
RELEASE laProjectHookRef

* Handle the VFP Windows that open with no Resource File
IF WEXIST("PROPERTIES")
    RELEASE WINDOW "Properties"
ENDIF

IF WEXIST("FORM CONTROLS")
    RELEASE WINDOW "FORM CONTROLS"
ENDIF

* Close the newly created class opened in Class Designer
* The keystrokes are "buffered" until all classes are created
KEYBOARD '{CTRL+W}'

* Added to close the class designers down
DOEVENTS()
```

Creating the class opened the Class Designer. Therefore `ASELOBJ()` got a reference to the projecthook that was created. Using this reference I can manipulate a property and even write code to a method if that was desired.

The next example writes code in the *BeforeOpenTables* method of a dataenvironment that calls a form method. The form method that is called (*SetDbc*) changes the database property of each cursor in the dataenvironment. The code is written to the *BeforeOpenTables* method only if

the form has the *SetDbc* method and only if the call is not already in the *BeforeOpenTables* method.

This particular builder runs in several modes. The code is lengthy, so only parts of the program are include here. This builder can be run from the dataenvironment, it can be run from a form, or it can be run from the Command Window. The builder code uses all three modes of `ASELOBJ()`.

If the builder recognizes it is called from the form it will get a reference to the dataenvironment. It needs this reference so it can write the code to the method. If the builder recognizes it is called from the dataenvironment it will get a reference to the form so it can check that the *SetDbc* method exists before we write code in the *BeforeOpenTables* method. The builder gets the reference to the form and dataenvironment in the following code:

Listing 2. This is a partial listing of the *BeforeOpenTablesCodeBuilder* program. This section of code grabs the needed references to the form and dataenvironment.

```
* Get possible reference to current select object
lnResult = ASELOBJ(laObjects, 1)

IF lnResult = 1
  DO CASE
    CASE LOWER(laObjects[1].BaseClass) = "form"
      * Have a form, get reference to DE
      lnResult          = ASELOBJ(laDEObject, 2)
      llMethodUpdated = UpdateBeforeOpenTablesMethod(laObjects[1], ;
                                                    laDEObject[1])
    CASE LOWER(laObjects[1].BaseClass) = "dataenvironment"
      * Have DE, get reference to the form
      lnResult          = ASELOBJ(laFormObject, 3)
      llMethodUpdated = UpdateBeforeOpenTablesMethod(laFormObject[1], ;
                                                    laObjects[1])
    OTHERWISE
      * Incorrect object
      MESSAGEBOX("The object selected was not a form or dataenvironment, "+
                "which this builder is designed to process.", ;
                0 + 16, _screen.Caption)
  ENDCASE
ELSE
  * Check to see if the builder was called from Command Window
  * with a project open to process.
...
ENDIF
```

If the code is called from the Command Window the first `ASELOBJ()` will return a zero since no object is selected for the builder. This forces the builder to check to see if a current project is opened. If one is opened the builder prompts the developer to see if they want all the forms in the project to be processed for this additional code.

Listing 3. This is a partial listing of the *BeforeOpenTablesCodeBuilder* program. This section of code grabs the needed references to the form and dataenvironment.

```
FOR EACH loFile IN _vfp.ActiveProject.Files
  IF loFile.Type = "K"
    MODIFY FORM (loFile.Name) NOWAIT
    lnResult          = ASELOBJ(laFormObject, 1)
    lnResult          = ASELOBJ(laDEObject, 2)
    llMethodUpdated = UpdateBeforeOpenTablesMethod(laFormObject[1], ;
```

```

laDEObject[1])

IF llMethodUpdated
    lnUpdated = lnUpdated + 1
ELSE
    lnNotUpdated = lnNotUpdated + 1
    STRTOFILE(loFile.Name, ccLOGFILE, .T.)
ENDIF

* Handle the VFP Windows that open with no Resource File
IF WEXIST("PROPERTIES")
    RELEASE WINDOW "Properties"
ENDIF

IF WEXIST("FORM CONTROLS")
    RELEASE WINDOW "FORM CONTROLS"
ENDIF

* Close the form opened in Form Designer. The keystrokes are
* "buffered" windows events until all forms are modified
KEYBOARD '{CTRL+W}'

DOEVENTS
ENDIF
ENDFOR

```

Each form is opened in the designer via the `MODIFY CLASS` command. This allows the builder to get a reference to the form and dataenvironment via `ASELOBJ()` function. All three of these methods call the *UpdateBeforeOpenTablesMethod* and pass along the reference to the form and dataenvironment. This function performs all the validation to insure the method exists on the form, and that the code is not already in the *BeforeOpenTables* method. If all the checks pass, the builder writes the code via the *WriteMethod* method.

Listing 4. This is a partial listing of the *BeforeOpenTablesCodeBuilder* program. It updates code in the *BeforeOpenTable* method for each form it process.

```

FUNCTION UpdateBeforeOpenTablesMethod(toForm, toDE)

LOCAL lcDocMethod, ;
    lcExistingMethodCode, ;
    lcDeveloperLogin, ;
    loForm

lcDocMethod = "BeforeOpenTables"
lcExistingMethodCode = SPACE(0)
lcDeveloperLogin = LOWER(SUBSTRC(SYS(0), ATC("#", SYS(0)) + 2))

IF PEMSTATUS(toForm, "SetDBC", 5)
    IF LOWER(PEMSTATUS(toForm, "SetDBC", 3)) # "method"
        * No need to call a method that does not exist
        RETURN .F.
    ENDIF
ELSE
    * No need to call a method that does not exist
    RETURN .F.
ENDIF

IF PEMSTATUS(toDE, lcDocMethod, 5)
    lcExistingMethodCode = toDE.ReadMethod(lcDocMethod)
    lcExistingMethodCode = IIF(EMPTY(lcExistingMethodCode), SPACE(0), ;
        ccCRLF + ccCRLF) + ;

```

```

                                lcExistingMethodCode
ELSE
    * Not a native DataEnvironment or does not have a BeforeOpenTables
    RETURN .F.
ENDIF

SET TEXTMERGE OFF
TEXT TO lcNewMethodCode NOSHOW
* <<lcDeveloperLogin>> <<DATE()>>, added via the G2 BeforeOpenTable Code Bdr
thisform.SetDBC()
ENDTEXT

IF "thisform.setdbc()" $ LOWER(lcExistingMethodCode)
    * Call already exists
    RETURN .F.
ELSE
    * Add the code to the existing code since it is not in the method
    * Add before the existing code to avoid a trailing RETURN statement
    toDE.WriteMethod(lcDocMethod, ;
                    TEXTMERGE(lcNewMethodCode) + ;
                    lcExistingMethodCode, .F.)

    RETURN .T.
ENDIF

ENDFUNC

```

The following code demonstrates how a developer can register a builder using the registration table and how a builder can be registered for multiple object types. In this case the builder makes sense to be run from the form or for the form dataenvironment, so it is registered for both. Another key aspect to recognize in this code is that it stores `sys(16, 0)` into the Program column with the full path so no matter what the default directory the developer is in, the builder will run.

Listing 5. This is the *SelfRegister* function of the *BeforeOpenTablesCodeBuilder* program. It registers this builder in the *Builder.dbf* as a form builder and as a dataenvironment builder.

```

FUNCTION SelfRegister()

LOCAL lnOldSelect, ;
    lcBuilderName, ;
    llAlreadyRegistered, ;
    llRegistered

lnOldSelect      = SELECT()
lcBuilderName    = "G2 BeforeOpenTables Builder (thisform.SetDBC())"
llAlreadyRegistered = .F.
llRegistered     = .F.

USE (HOME()+"Wizards\Builder.dbf") AGAIN SHARED ALIAS curRegBuilder IN 0
SELECT curRegBuilder

* Register as a Form Builder
LOCATE FOR Name = lcBuilderName AND Type = "FORM"

IF FOUND()
    llAlreadyRegistered = .T.
ELSE
    SCATTER MEMVAR MEMO BLANK

    m.Name      = lcBuilderName
    m.Descript  = "Adds a call in the BeforeOpenTables() method to " + ;

```



```

                thisform.SetDBC() if it exists."
m.Type          = "FORM"
m.Program       = SYS(16, 0)

INSERT INTO curRegBuilder FROM MEMVAR
llRegistered = .T.
ENDIF

* Register as a DataEnvironment Builder
LOCATE FOR Name = lcBuilderName AND Type = "FORM"

IF FOUND()
    llAlreadyRegistered = .T.
ELSE
    SCATTER MEMVAR MEMO BLANK

    m.Name          = lcBuilderName
    m.Descript      = "Adds a call in the BeforeOpenTables() method to " + ;
                    "thisform.SetDBC() if it exists."
    m.Type          = "DATAENVIRONMENT"
    m.Program       = SYS(16, 0)

    INSERT INTO curRegBuilder FROM MEMVAR
    llRegistered = .T.
ENDIF

IF llRegistered
    MESSAGEBOX(lcBuilderName + " has been registered with Visual FoxPro v" + ;
                VERSION(4), ;
                0+64, lcBuilderName)
ELSE
    IF llAlreadyRegistered
        MESSAGEBOX(lcBuilderName+" was already registered with Visual FoxPro v";
                    + VERSION(4), ;
                    0+64, lcBuilderName)
    ENDIF
ENDIF

USE IN (SELECT("curRegBuilder"))
SELECT (lnOldSelect)
RETURN

ENDFUNC

```

This particular builder was used on three projects with nearly 300 forms. If we changed each manually it would have taken days. I wrote the builder in a couple of hours and it takes very little time to run the builder for each of the projects. This was a huge time saver for both us and the customers. It is a fairly simple builder conceptually, but the code is fairly sophisticated because of the various modes that it can be run.

What are the Builder and BuilderX properties? (Example: BUILDERDEMOVFP8.SCX, DEMO.VCX::FRMCHECKBOXBUILDER)

You can add a custom property called *Builder* and *BuilderX* to all classes (hopefully at the highest level of your class hierarchy). The native builder technology will use the setting in this property to run the builder designated. The property can be set to a program or a class library/class name combination separated by a comma. VFP will use this as the builder and ignore all the builders registered in the BUILDER.DBF table for that object.

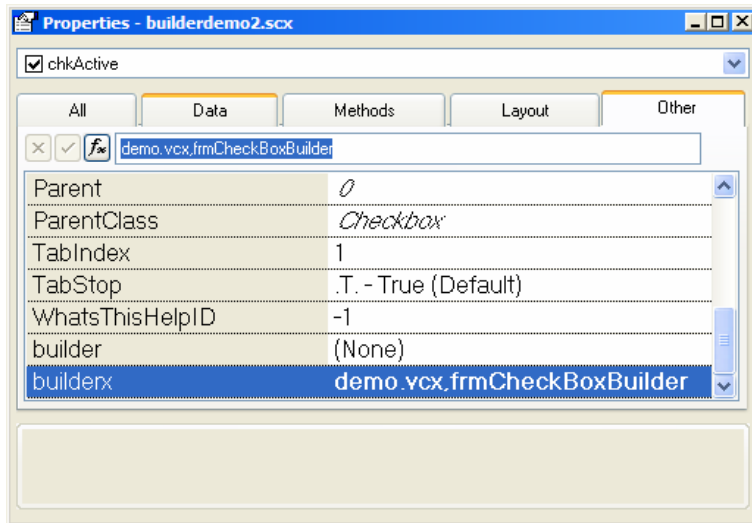


Figure 17. This example shows the class with the custom BuilderX property.

Doug Hennig explains this as well as I have heard anyone explain these two properties – “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.”

You will also see in the BuilderB and BuilderD sections that the *Builder* and *BuilderX* properties are important for the implementation these builder frameworks.

How do I implement a builder using BuilderB? *(Example:*

BUILDERBEXAMPLE::FRMEDITBOXBUILDERB, FRMSPINNERBUILDERB, FRMSHAPEBUILDERB, FRMLABELBUILDERB)

BuilderB is a technology originally developed by Ken Levy back in 1995 while he worked at Flash Creative Management and before he went to work at Microsoft. It is called BuilderB because it is a “Builder-Builder”. It is a framework set of classes that allow developers to rapidly develop builders. Each builder starts with a form that is a subclass of the base BuilderB form. You add controls to the subclassed form for each property you want to maintain. This is faster than developing a completely new builder form from scratch each time because the base form and controls have builder intelligence already coded. As we will see stepping through some examples, it still can be tedious and time consuming, but faster in most cases than building something from scratch.

The advantages of writing a builder using BuilderB include:

- Framework removes some of the grunt work
- Consistent user interface
- Extendible
- Easy to learn
- Easy to integrate into VFP

There are definitely some disadvantages to using BuilderB:

- Need to create a builder for each class or set of classes (which is more code exposure and maintenance than a framework like BuilderD)
- Need to learn framework

NOTE:

*All of the classes necessary to build a BuilderB builder are included in the BUILDERB.VCX class library (see **Figure 18**). This class library is available on several websites on the web including the UniversalThread. I have included it in the whitepaper downloads for your convenience.*

NOTE:

All the BuilderB examples can be demonstrated by running the BuilderBDemo form available in the whitepaper downloads.

The first thing you will want to do is become familiar with the classes available in the BuilderB framework (see Figure 18). There are three groups of classes to be concerned with. The first is the BuilderForm and the associated subclasses. The BuilderForm is the class you will start with to create your builders. The subclasses of this form are pre-constructed builders for forms, properties (based on the checkbox, textbox, and editboxes), and captions. The second group is the BuilderCommandbutton and the associated subclasses. The buttons are used on the BuilderForm to provide standard functionality for all BuilderB builders. The third group of classes is a set of controls that you can use to expose properties of the object. The user interface controls include a checkbox (BuilderCheckbox), editbox (BuilderEditBox), label (BuilderLabel), pageframe (BuilderPageframe), and textbox (BuilderTextbox). These have the intelligence to bind to properties of the selected control. Each of these base controls has a BuilderB builder. You can also add other controls if the need arises (an example of this is in the frmEditboxBuilderB example class).

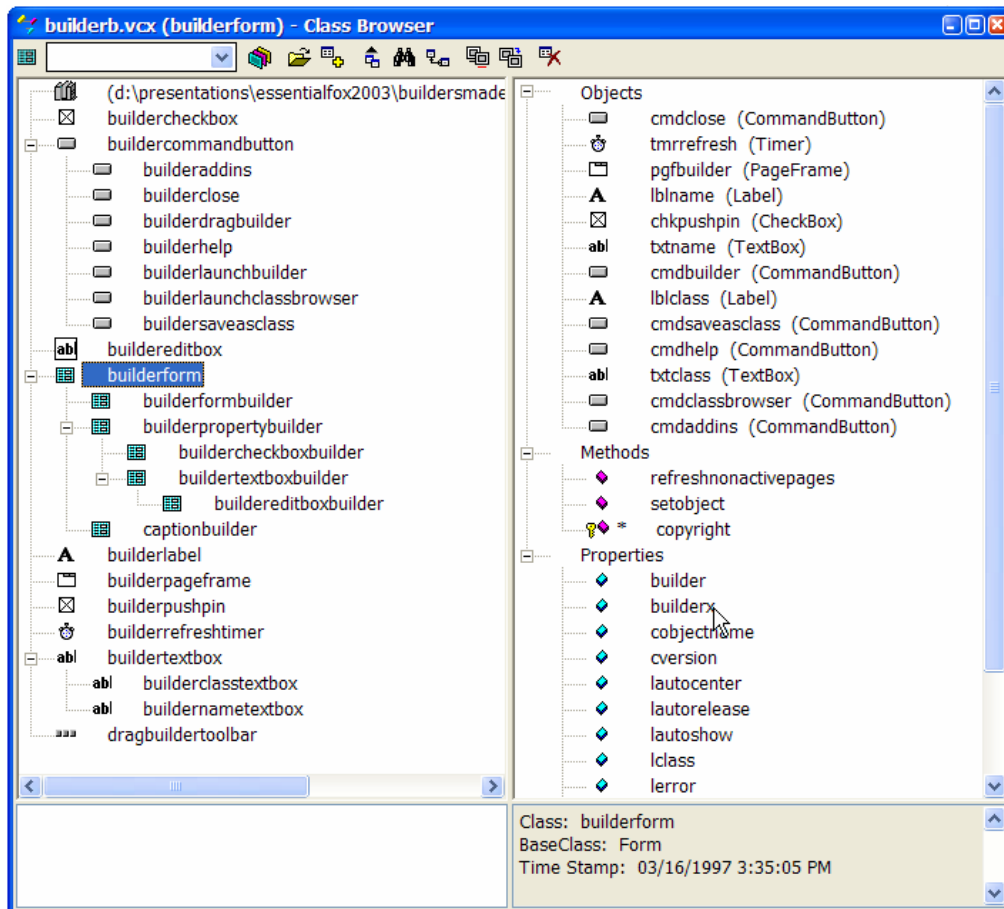


Figure 18. The BuilderB.vcx contains all the classes necessary to start creating BuilderB builders.

Once you determine that you have a need for a builder for a particular object, creating the builder follows a few straightforward steps. The recipe to create a BuilderB builder is this:

Determine the properties you want exposed on the builder and the type of user interface controls that you want to use to expose these properties.

Create a class based on the BuilderForm via the **CREATE CLASS** command, the File | New dialog, or the new class button in the Class Browser or Project Manager. (see **Figure 19**)

Add the user interface control for a property. The easiest way is to open the BuilderB class library in the Class Browser or the Project Manager. You can edit the page of the pageframe on the builder form and then drag and drop controls from the Project Manager or Class Browser. If you are using VFP 8 you can use the new Toolbox to do the same thing. You can add base VFP controls as well if they are ones not included in the BuilderB class library.

If the control you dropped on the builder is one from the BuilderB class library, you can run the BuilderB builder for that control (see **Figure 20**). If you used a control that is not from the BuilderB class library, but it has a builder, you can do the same. If you prefer, you can make changes to properties and methods using the Property Sheet as well.

Repeat steps 3 and 4 until you have a user interface control for each property you want exposed and have implemented the behavior for each.

Specify the builder for the objects that the builder is implemented.

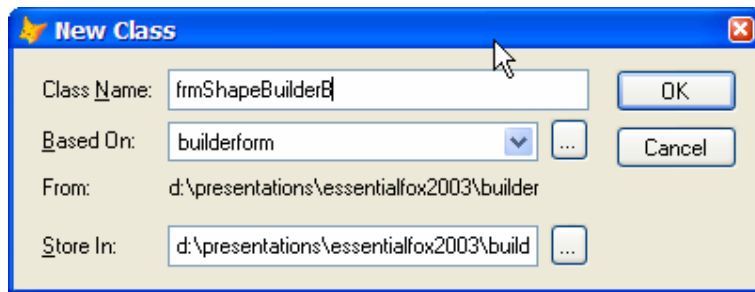


Figure 19. The first thing you need to do when creating a BuilderB builder is to subclass the BuilderForm class.

The BuilderB controls are bound to the object properties via the *cProperty* property. This can be set in the Property Sheet or via the builder for the control. You cannot set the ControlSource because the target object reference for the builder does not get set until after the form's Init method runs. Each of the controls on the builder are bound after the target object reference gets set.

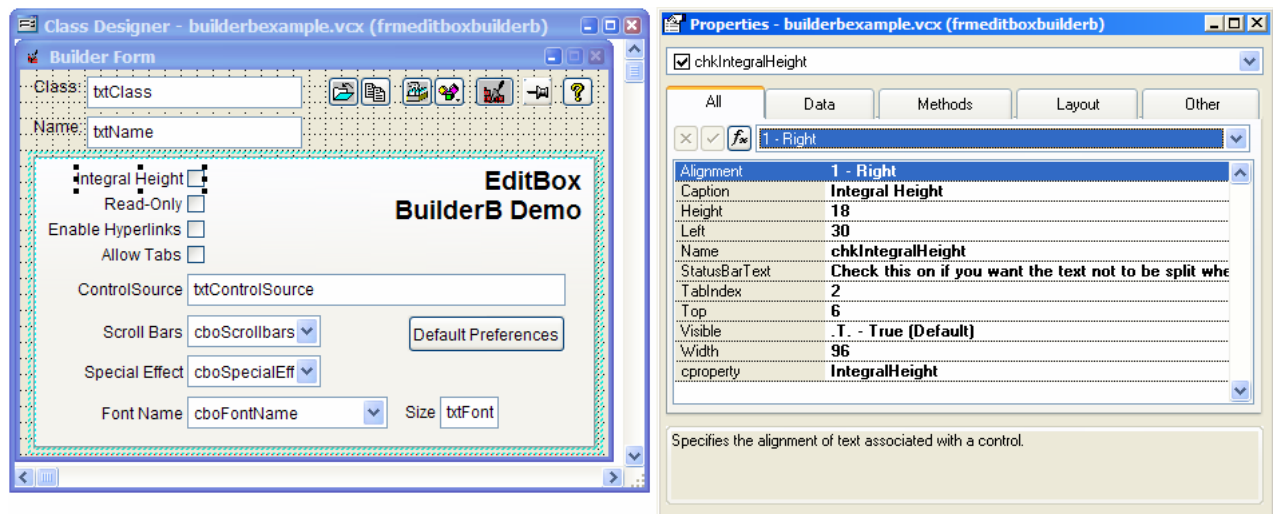


Figure 20. The BuilderB controls are bound to the property by setting *cProperty*, not through the ControlSource.

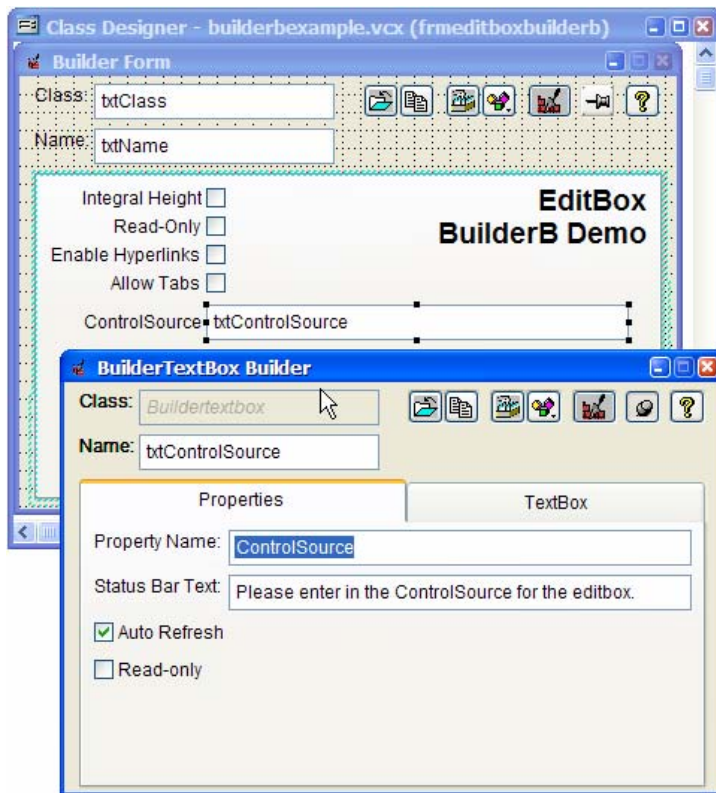


Figure 21. The sometimes mind-boggling concept of running a BuilderB builder when building an object on another BuilderB builder is something that will become quite common.

The BuilderB builders are normally called via the *Builder* or *BuilderX* property setting of the object. In the case of the editbox builder example, you can see that the *Builder* property is set to "BuilderBExample.vcx, frmEditBoxBuilderB".

The frmSpinnerBuilderB and frmLabelBuilderB builders demonstrate basic property binding to textbox and checkbox objects. They are very simple builders that show how a developer can expose specific properties that are commonly changed for a control. In this case the spinner and keyboard high and low values are available so the developer does not have to hunt them down in the Property Sheet.

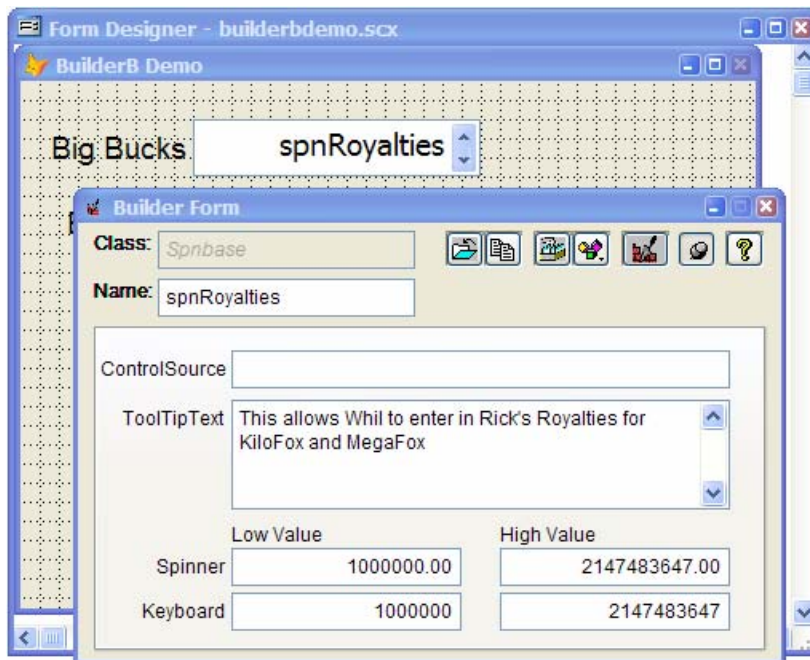


Figure 22. The spinner control builder demonstrates exposing common properties so developers do not have to find them in the Property Sheet.

The frmEditboxBuilderB builder demonstrates how you can extend the base functionality provided by the base BuilderForm and the controls found in the BuilderB class library. The form demonstrates how you can add comboboxes to the builder. This is no small task if you have not tried this before. You need to understand how to bind to the target object reference. It also demonstrates that you can add functionality to the builder without binding the user interface to a specific property. The Default Preference button will set several properties all at once. The first attempt I made at this was to change the *Value* properties of the controls on the page. This did not work because the form has a timer that does an automatic *Refresh* which gets in the way. Therefore I changed my approach to change the object properties directly. There are more details in the “What is the object reference to bind if I am not using a BuilderB control?” section later in this whitepaper.

What do the buttons at the top of the BuilderB builder do?

One of the nice things about working with a builder framework like BuilderB is that someone else added base functionality to the builder form. In the case of the BuilderB framework, we have seven buttons along the top of the form that extend the builder (see **Figure 23**). It is on all BuilderB builders since the button set is on the BuilderForm class, and this is the class that is used to create a new builder.

The first button closes the builder despite the fact that the icon looks very similar to the open icon used in Visual FoxPro and other programs. You can also use the close button in the upper right corner of the form without losing any changes you make to the target object properties. The second button is the Save as Class. You can save the current target object as a class in a class library.

The third button opens up the Class Browser with the class library of the object that the target object is based on. Naturally you cannot open up the class that the target object is based on

since the target object is instantiated in the designer. Once the Class Browser is opened you can close the builder and the designer and then edit the class.



Figure 23. All the BuilderB builders will have this toolbar with base functionality since it is defined on the BuilderForm class.

The fourth button (three shapes) opens up the add-in dialog if there is registered add-ins for the builder. The fifth button (trowel and bricks) brings up optional builders registered in the BUILDER.DBF file. This allows developers to use both their custom builder specified for the object, and then have access to any other builder registered, including the native Visual FoxPro builders. The sixth button is a graphical checkbox and is the push pin that toggles the form's always on top state. The last button is basic help (question mark) and brings up a text file with some fundamental help for BuilderB.

What is the object reference to bind if I am not using a BuilderB control?

The BuilderB class library contains a checkbox (BuilderCheckbox), editbox (BuilderEditBox), label (BuilderLabel), pageframe (BuilderPageframe), and textbox (BuilderTextbox). Each of these controls has a *cProperty* property that the control gets bound after the builder determines the target object. What happens if you want to add a combobox? What is the object reference and property bound to?

The BuilderForm has property called *oObject*. If you want to bind a control you need to set the *ControlSource* to "thisform.oObject." plus the property name that you want the control to be bound. You need to set this in the form's Init method after a call to `DODEFAULT()`. Here is a code example from the frmEditBoxBuilderB form:

```
LPARAMETERS toObject,tuSource,tlSkipSearch

DODEFAULT(toObject,tuSource,tlSkipSearch)

WITH this.pgfbuilder.fpgPage1
    .cboScrollbars.ControlSource = "thisform.oObject.Scrollbars"
    .cboSpecialEffect.ControlSource = "thisform.oObject.SpecialEffect"
    .cboFontName.ControlSource = "thisform.oObject.FontName"
ENDWITH

RETURN
```

What do you have to do to run a BuilderB builder via Builder.app?

The BuilderB builders normally are run via the *Builder* or *BuilderX* property. You can modify your builder if you want to register them as classes in the Builder.dbf registration table. The frmShapeBuilderB example form found in the BuilderBExample class library is set up to do this. The advantage of this approach is so you can have the builder available to all objects, not just the objects that specify the builder in the *Builder* or *BuilderX* property.

The key to solving this problem is to recognize that the default parameters sent from the builder manager program (defaults to Builder.app, stored in `_BUILDER`) do not match the parameters statement of the builder form. A builder that processes through the builder manager receives three parameters. They are: "wbReturnValue", a null string, and another null string. A

builder called via the *Builder* and *BuilderX* properties has the parameters: a reference to the target object (toObject), the method it was called (tuSource), and a logical to determine if the object reference needs to be determined by the builder (tlSkipSearch). So you can see the mismatch in data types and complete meaning. Adding code to the builder's Init method to "translate" the parameters will allow the builder to run both ways.

```
LPARAMETERS toObject,tuSource,tlSkipSearch

* RAS 30-Mar-2003, Discovery!
* Handle issues with calling this through the normal
* Builder.APP call (via the Builder registration table).
IF VARTYPE(toObject) == "C"
    toObject = .NULL.

    * Need to run this form modal since the builder manager
    * does not handle this automatically, and BuilderB
    * is a modeless builder tool
    this.WindowType = 1
ENDIF

IF VARTYPE(tlSkipSearch) # "L"
    * This setting allows the builder to gain reference to
    * the target object
    tlSkipSearch = .F.
ENDIF

* Now call the superclass code with possibly corrected parameters
DODEFAULT(toObject,tuSource,tlSkipSearch)

* Bind the non-BuilderB comboboxes
WITH this.pgfbuilder.fpgPage1
    .cboBackStyle.ControlSource      = "thisform.oObject.BackStyle"
    .cboSpecialEffect.ControlSource = "thisform.oObject.SpecialEffect"
ENDWITH

RETURN
```

The builder is registered in the BUILDER.DBF in the same fashion as a regular builder. For more details on how this is done, see the section "How do I register a custom builder?" earlier in this whitepaper.

How do I implement a builder using BuilderD?

As fast as BuilderB made development of builders, it was replaced by a newer builder technology, the third generation of builder technology called BuilderD (for "Dynamic", but I like to think of it as "Data"). It has been included in Visual FoxPro since VFP 6. This builder technology is data driven using some metadata stored in the BuilderD table (VFP\WIZARDS\BUILDERD.DBF). The builder form looks very familiar if you have spent any time with the BuilderB framework. Coincidentally, this framework was written by Ken Levy as well.

The advantages of using BuilderD over other builder technologies include:

- Framework removes the grunt work
- Consistent user interface
- Least code exposure of all techniques since it is data driven
- Extendible

- Built into VFP

The disadvantages:

- Need to learn framework (metadata is complex)
- Least Flexible
- Additional implementation headaches (distribution of metadata)

First we need to understand a little about the metadata table that contains the records that drive the BuilderD builders. The metadata is fairly complex, but it is only data. Once you understand the implementation you will see the flexibility and the drawbacks involved in writing a BuilderD builder.

So the question begs: How do I start to learn how the BuilderD data is implemented to create my own builders? I think you will find the easiest way is to see one in action. It just so happens that many of the Fox Foundation Classes (FFCs) included with Visual FoxPro have BuilderD builders. One way to get at the FFCs is to use the Component Gallery and drill down to the Visual FoxPro Catalog and one more level to the Foundation Classes.

The example I am going to use here is the simplest example, the `_HyperlinkLabel` FFC (found in the Internet folder). Open up the class and run the builder. This will bring up the BuilderD builder (see **Figure 24**).

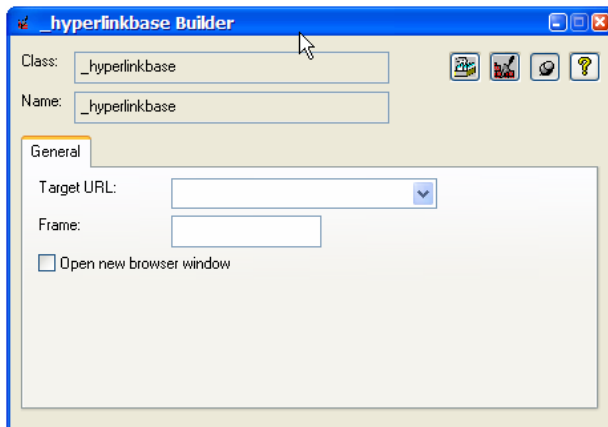


Figure 24. The BuilderD builder for the Fox Foundation Class `_HyperlinkBase` shows three properties exposed via the metadata.

There are four metadata records associated with this builder.

Table 5. Records in `BUILDERD.DBF` for the `_HyperlinkBase` class.

Type	Id	Links	Text	Member
CLASS	<code>_HyperLinkBase</code>	<code>cTarget</code> <code>cFrame</code> <code>INewWindow</code>		
PROPERTY	<code>cFrame</code>		Frame:	<code>cFrame</code>
PROPERTY	<code>cTarget</code>		Target URL:	<code>cTarget</code>
PROPERTY	<code>INewWindow</code>		Open new browser window	<code>INewWindow</code>

The builder is started because the *BuilderX* property of the class is set to `"=HOME()+"Wizards\BuilderD,BuilderDForm"`. The BuilderDForm resides in the BuilderD class library. This form has all the intelligence in it to read the BuilderD table and populate the builder user interface.

I have a couple of observations after reviewing the metadata. The first is that there are two types of records in the metadata, classes and properties. The Id field uniquely identifies the record in the table. This identifier is used to hook up with “records” identified in the Links column. In the *_HyperlinkBase* class, we have three links. If the link has a corresponding “property” record (based on Type column), the property record defines a user interface element on the builder. If there is no corresponding record, it is assumed that the link corresponds to a property on the target object and a textbox is instantiated for the property. The Member column defines the property that the control will be bound. Confused yet? Like I noted earlier, this is a bit complex and could take some time to get your head around it.

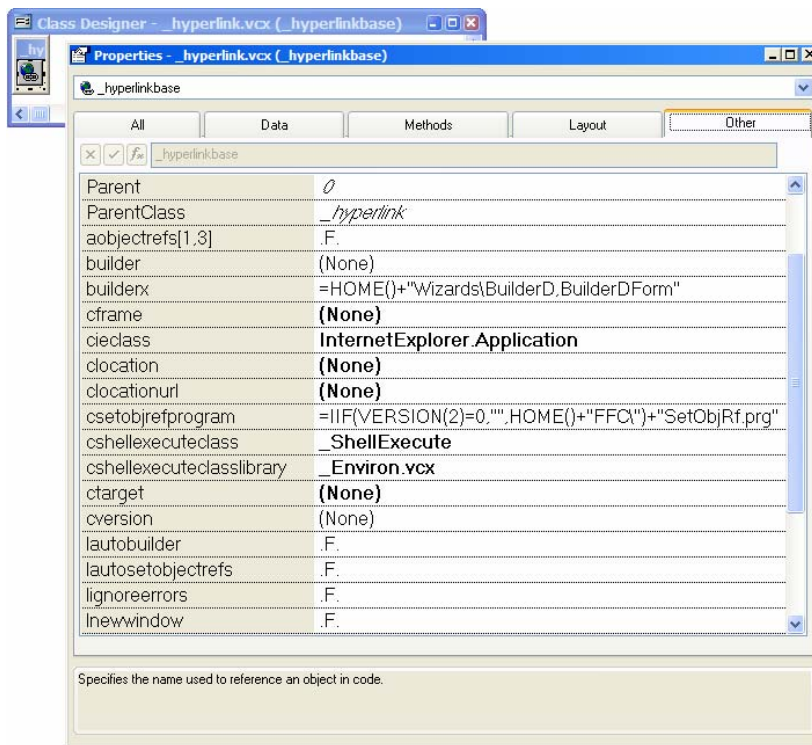


Figure 25. The *_HyperlinkBase* shows several properties exposed in the Property Sheet including the *cFrame*, *cTarget*, and *INewWindow*.

So how does the Class record get looked up by the BuilderDForm? When the builder is called, the BuilderD logic looks at the target object, determines the class name and library and looks up the record based on this combination in the ClassName and ClassLib columns. This record will define the Links. Each of the links is looked up in the BuilderD table. If found the property is bound to the control as defined in the property record. All the columns in the BuilderD metadata table are described in **Table 6**.

Table 6. File layout of the *BUILDERD.DBF*.

Field	Type	Description
Type	C(12)	Record type. Set to "CLASS" if the record is for a class that the builder will be instantiated. "PROPERTY" if the record indicates a user interface object that will be instantiated on the builder for the specific property.
Id	C(24)	Record identifier. Microsoft typically will set this to the class name of the class that the builder is designed for, which is self documenting. It can be set to anything as long as it is unique (not enforced at the table level since it is a free table).
Links	M	This is literally a set of links to other records (Type = "PROPERTY") defined by the Id column, separated by a carriage return. Typically this column is a list of properties that are exposed on the builder user interface. If there is no specific property record the property is exposed in a text box unless it is a logical property.
Text	M	This is the caption displayed for the property on the builder user interface when Type = "PROPERTY", and the builder form caption when Type = "CLASS"
Desc	M	Description used on the status bar if running with SET STATUS BAR ON.
Classname	M	<p>If Type = "CLASS", the column indicates the class that the builder is executed to expose properties. This cannot be blank if the Type = "CLASS".</p> <p>If Type = "PROPERTY", this is the class used to present the property on the builder form. This can be left blank and a default object will be instantiated from the BuilderD.vcx. If you decide to customize your own classes, make sure they are subclassed from the classes in the BuilderD.vcx.</p>
Classlib	M	<p>If Type = "CLASS", the column indicates the class library that the builder is executed to expose properties. If it is blank, the specified class in the ClassName column can reside in any class library.</p> <p>If Type = "PROPERTY", and this field is blank and ClassName is specified, BUILDERD.VCX is assumed, otherwise this is the class library that the class specified in the ClassName field is located.</p> <p>NOTE: You can specify the class library as an expression if you include the entire value with parenthesis. This is a common practice for Fox Foundation Classes that have BuilderD based builders. An example of this is: (HOME() + "WIZARDS\BUILDERD.VCX")</p>
Member	M	<p>If the Type = "PROPERTY" this is the member (property) that is maintained via the builder. If it is blank the member is determined by the Id column.</p> <p>It should be empty if the Type = "CLASS".</p>
Helpfile	M	Name of the HTML help file (CHM). If it is blank it will use the current help helpfile defined by SET HELP
HelpId	M	This is the help id that indexes into the help file.
Top	N(6,0)	This is the Top property of the object used to expose the property. If zero, the object is placed just below the previously instantiated property object.
Left	N(6,0)	This is the Left property of the object used to expose the property. If zero, the object is placed on the left side on the page that the object is instantiated on.
Height	N(6,0)	This is the Height property of the object used to expose the property. If zero, the object default Height is used.
Width	N(6,0)	This is the Width property of the object used to expose the property. If zero, the object default Width is used.
RowSrcType	N(1,0)	If the object used to expose the property is a combo box, then this is the RowSourceType property for that object.
Rowsource	M	If the object used to expose the property is a combo box, then this is the value of the RowSource property for that object.
Style	N(1,0)	If the object used to expose the property is a combo box, then this is the value of the Style property for that object.
Validexpr	M	This is the expression that is evaluated when the property is validated.
ReadOnly	L	Make this .T. if you want the object that exposes the property made ReadOnly on the builder.

Field	Type	Description
Updonchg	L	Make this .T. if you want the object value written to the object's property as it's changed from the InteractiveChange method.
Updated	T(8)	This is the date/time value the record was updated. Must be set manually since this is a free table and is not used by builder technology (BuilderD).
Comment	M	This is free form comments that developers can use for their own needs.
User	M	Standard metadata User field that developers can leverage for their own needs.

I have added a couple of class and three property records to my BuilderD table for the whitepaper examples. These records are specified in **Table 7**.

Table 7. Details “class” records added to the BuilderD.dbf for an editbox class called edtBase and a commandbutton called cmdBase.

Id	Links	ClassName	ClassLib
edtBase	BackStyle AllowTabs IntegralHeight BuilderX Mouselcon	edtBase	D:\Presentations\SouthwestFox2004\BuildersMadeEasy\Examples\demo.vcx
cmdBase	Default Cancel WordWrap cCaptionWide cPictureProperty nPicturePosition	cmdBase	D:\Presentations\SouthwestFox2004\BuildersMadeEasy\Examples\demo.vcx

Table 8. Details “property” records added to the BuilderD.dbf for cmdBase links.

Id	Member	Text	Other Columns
cCaptionWide	Caption	Caption:	Width = 200 UpdOnChg = .T.
cPictureProperty	Picture	Picture:	Width = 300
nPicturePosition	PicturePosition	Picture Position:	Style = 2 RowSrcType = 1 RowSource = 0,1,2,3,4,5,6,7,8,9,10,11,12,13

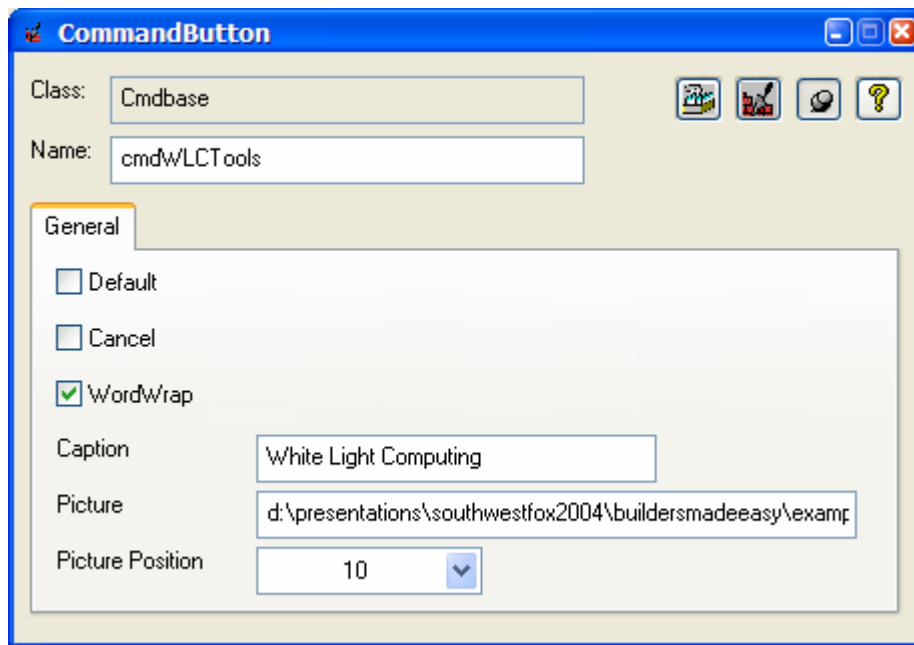


Figure 26. The results of the BuilderD metadata changes on the builder user interface for the cmdBase classes.

All of the links in the edtBase are properties of the object, not pointers to the property type records in the metadata. In literally minutes I can create a builder for any custom class I develop without writing a line of code if this is the case. Talk about enhancing productivity.

Unfortunately the likelihood of all properties being presented in a textbox is low. So we have to create property records for all the properties that do not fit the mold of the default textbox. One limitation we have noticed is that comboboxes created with the Style, RowSrcType, and RowSource columns do not create multiple column combos. Even more complicated is that the BoundTo property cannot be set. This limits combos to be bound to character based data. This is a severe limitation.

The idea is to create custom property records that provide the user interface control that appears on the builder. Each of the property records define the member (property) that is maintained through the control. The class record defines which of the properties for the class are exposed on the interface by selecting the property records to link to or just list the property if a textbox is an acceptable user interface element.

What do the buttons at the top of the BuilderD builder do?

One of the nice things about working with a builder framework like BuilderD is that someone else added base functionality to the builder form. In the case of the BuilderD framework, we have four buttons along the top of the form that extend the builder (see **Figure 27**). The reason it shows on all the BuilderD builders is that only one form is used to display the builders, only the properties exposed changed based on the metadata.

The first button opens up the Class Browser with the class library of the class that the target object is based on. Naturally you cannot open up the class that the target object is based on since the target object is instantiated in the designer. Once the Class Browser is opened you can close the builder and the designer and then edit the class.



Figure 27. All the BuilderD builders will have this toolbar with base functionality since it is defined on the BuilderDForm class.

The second button (trowel and bricks) brings up optional builders registered in the BUILDER.DBF file. This allows developers to use both their custom builder specified for the object, and then have access to any other builder registered, including the native Visual FoxPro builders. If there are no specific builders, but one “ALL” builder is registered, it will run without the selection dialog. The fourth button is help (question mark) and brings up the help file specified in the HelpFile column of table BuilderD and positions it to the topic specified in the HelpId column. If the class is a Fox Foundation Class (FFC), the help topic in the Visual FoxPro help file is displayed.

What tools are available to edit Builder metadata? *(Example:*

WlcBUILDERMETADATAEDITOR.EXE)

The nice thing about the builder technologies built into Visual FoxPro is that they are data driven. Unfortunately Microsoft did not provide tools to interface with these metadata tables. I am not sure about you, but while I think the **BROWSE** window is a nice to look at data, it is not fun creating and editing data.

The biggest drawback to this technique of opening a table (after hunting it down in the directory structure), and then browsing it is that it, is typically in an unbuffered state. I am sure I am the only developer on the planet that makes mistakes as I enter in changes. If it is unbuffered I have to remember what the original values are for the columns and rows that I entered invalid or possibly corrupt information. The other big drawbacks are the lack of validation of the information I am entering in and lack of shortcuts to entering in the details of each column.

So with these drawbacks in mind I set out to simplify my life with builders and created a tool known as the WLC Builder/Wizard/BuilderD Registration Editor. This tool buffers all changes to the Builder, Wizard, and BuilderD tables. You can add new records, delete obsolete records, undo changes to the current registration record, or all the changes you have made, but not yet saved. There is a checkbox at the top of the form to show or eliminate the deleted registration records. The About page has version information as well as websites where this tool will be available when a new version is released.

The Builder table is found on the first page. All the columns for the table are exposed (even Bitmap that is not used). The grid can be used for navigation. The rest of the columns are free-form entry. There is a program picker button to use the GetFile dialog and a class picker button to select a class from a class library to speed up the selection process.

The Wizard table information is found on the second page and is identical in functionality to the Builder page.

The BuilderD has a class picker like the Builder table. The hardest part of the BuilderD is noting all the Links (properties and property records that already exist. Noting this pain while learning the metadata, I came up with a picker dialog (see **Figure 28**) to ease the selection of existing property records as well as the actual properties on the class specified. This greatly simplifies the linking process.

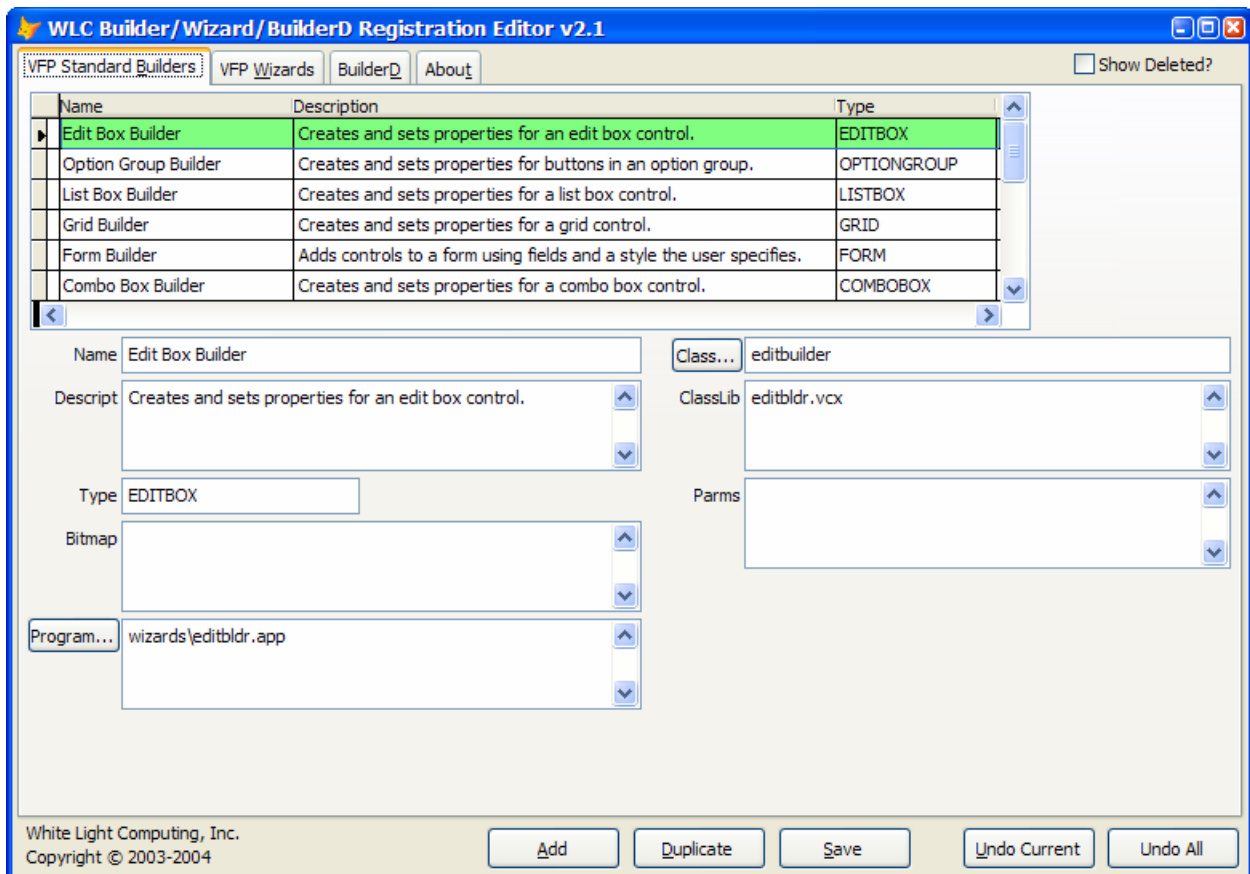


Figure 28. The WLC Builder/Wizard/BuilderD Registration Editor tool simplifies the builder registration process.

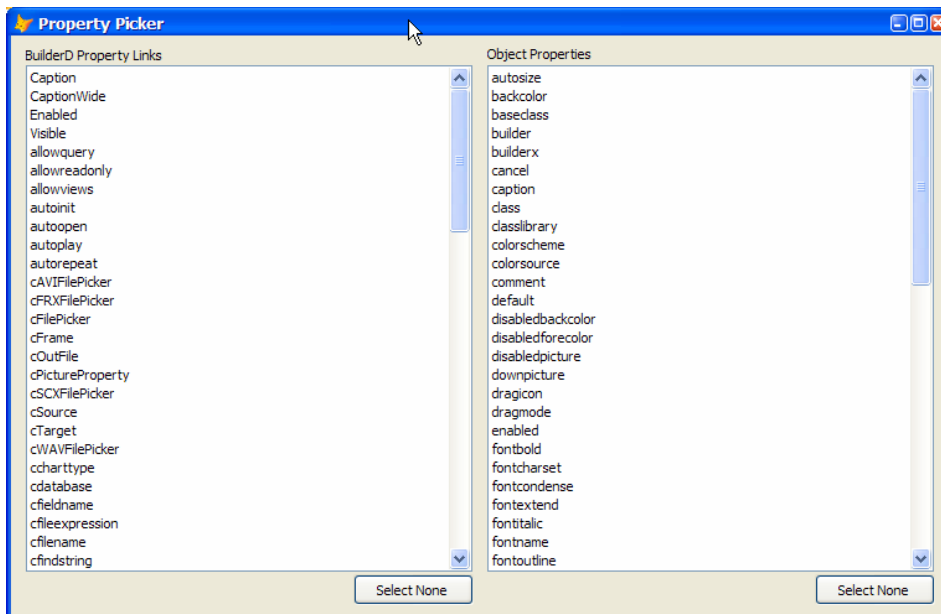


Figure 29. Setting up the Links column of the BuilderD metadata is much easier when you can pick from the available property records and the actual properties from the target object.

I have an enhancement request list that includes adding search and filter capabilities in a future version.

What about “Property Editors”

Property editors are brand new in Visual FoxPro 9. They are not directly related to the builder technology, but they act and respond to properties in a similar fashion as a builder. They are often implemented using the same techniques as a builder, but are not registered in the Builder Registration tables (BUILDER.DBF and BUILDERD.DBF). The Property Editors are implemented via the IntelliSense table (FOXCODE.DBF). A Property Editor typically works with one property for a single object at a time. A regular builder can work with a single property or multiple properties, for one object or multiple objects.

First I want to show you how a Property Editor works, then explain how it is implemented inside of the IDE. The Property Editor is accessed on the Property Sheet via the ellipses button (see highlighted area in **Figure 30**) next to the property textbox. This button is only available if the property has a Property Editor registered in FoxCode. Pressing this button will start the Property Editor per the code implemented in FoxCode. In Figure 30 and Figure 31 I show you the new Visual FoxPro 9 *Anchor* property and the related Anchor Editor.

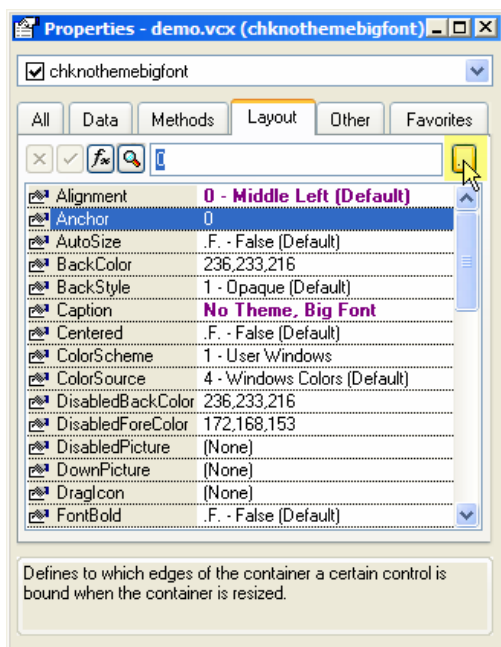


Figure 30. Only properties with a Property Editor have the property ellipses button available to the right of the property textbox.

There are two Property Editors shipping in the current beta version of Visual FoxPro 9. The *Anchor* and *Caption* properties have editors that the Fox Team developed or contracted. The *Caption* property is simply an `INPUTBOX()`. The *Anchor* Editor is a sophisticated user interface to a complex property. This is where Property Editors shine. Like builders, they can provide a simple user interface or run pure code to set the property to a specific value. The *Anchor* Editor has a user interface because the *Anchor* property has many settings and choices. The *Caption* Property Editor is less sophisticated because you are entering in plain text.

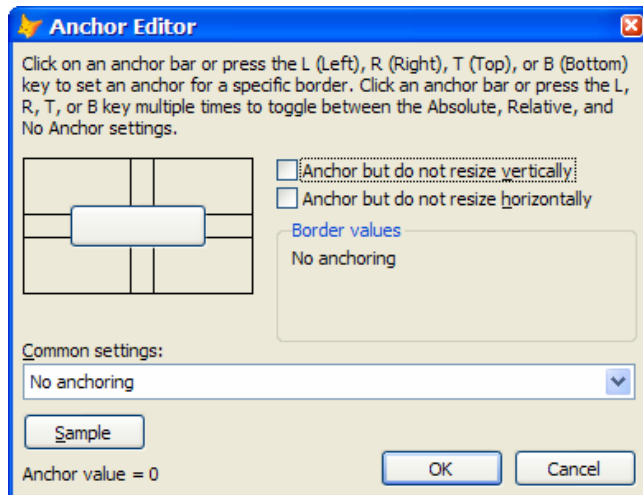


Figure 31. The Anchor Editor is an example of a Property Editor and it ships with Visual FoxPro 9.

The implementation is handled by adding “E” records in the FoxCode table. **Table 9** shows the new “E” records added to FoxCode in VFP 9. The trick to get Visual FoxPro to execute a Property Editor is adding Member Data to the TIP column.

NOTE:

Member Data is also new to Visual FoxPro 9 and works to configure the Property Sheet for the property. See the “MemberData Extensibility” topic in the Visual FoxPro 9 Help file for more details on how to set up Member Data.

In this case the “E” record is defining global Member Data for the four properties. Only three of the records (Anchor, Caption and _memberdata) have script executed (see the blue highlighted script code in Table 9). The Script attribute for the property defines what code is executed. It is up to you as the developer to get a reference to the current object, determine the property if necessary, and set the property via the object reference. Just like regular builders, you most likely will be using the `ASELOBJ()` function to get a reference to the object. If your Property Editor is specific to one property, then you can set the property directly. If the Property Editor is generic, you might need to pass in a parameter indicating the property you want to change or determine this programmatically.

NOTE!

The Member Data Editor “E” record is not part of the default VFP 9 install. You need to run the MemberDataEditor.app without any parameters to have it register itself as a Property Editor. The Member Data Editor is registered as a builder when you first install Visual FoxPro 9.

Table 9. The records in VFP's FoxCode table define the properties with global member data and how these properties have Property Editors implemented.

Type	Abbrev	Cmd	Tip	Case	Save	Source
E	Caption	{CaptionScript}	<VFPData><memberdata name="caption" type="property" favorites="True" script="DO (_CODESENSE) WITH 'RunPropertyEditor','', 'caption'"/></VFPData>	U	T	RESERVED
E	Name	{}	<VFPData><memberdata name="name" type="property" favorites="True"/></VFPData>	U	T	RESERVED
E	Value		<VFPData><memberdata name="value" type="property" favorites="True"/></VFPData>	U	T	RESERVED
E	ANCHOR		<VFPData><memberdata name="anchor" type="property" favorites="True" script="do HOME()+ 'WIZARDS\AnchorEditor.app'"/></VFPData>	U	T	RESERVED
E	_memberdata		<VFPData><memberdata name="_memberdata" type="property" display="_MemberData" script="do [C:\PROGRAM FILES\MICROSOFT VISUAL FOXPRO 9\MemberDataEditor.app]"/></VFPData>			

(* The Expanded, Data, Timestamp, and UniqueID columns were removed since they do not have an impact on the implementation)

The Caption Property Editor is an example of how you might build a generic Property Editor. You can see in Table 9 that the script is:

```
script="DO (_CODESENSE) WITH 'RunPropertyEditor','', 'caption'"
```

You can observe that the FOXCODE.APP file is run and several parameters are passed in. These parameters are telling FOXCODE.APP to run the generic Property Editor and to display entry for the Caption property.

The Anchor “E” entry in FoxCode directly runs the ANCHOREDITOR.APP file with no parameters:

```
script="do HOME()+ 'WIZARDS\AnchorEditor.app'"
```

The Anchor Editor knows it is only working with the *Anchor* property and does not need any special parameter to tell it to work with something generic. This entry also shows another important aspect of the implementation of Property Editors, the script code is evaluated before it is executed. In this example, the location of the Anchor Editor is located in the HOME()+ 'Wizards\ ' folder. This is flexible so you can install your Property Editors any where and VFP will find them.

Standardize the Name property (Example: ApplyNamingStandardsPropertyEditor.prg)

One example of how you can implement a Property Editor is to have it fix your object *Name* property to conform to standards (APPLYNAMINGSTANDARDSPROPERTYEDITOR.PRG). I prefer to stick to industry standards with respect to object names. This means an object name starts with a three character prefix, followed by a meaningful name. This can be time consuming depending on the number of objects, and the style the form or class was created. If you drag and drop objects to the form from the DataEnvironment, then VFP does this for you. If you drag and drop from the ToolBox or the Forms Control toolbar, you get names like Text1, Text2, and Combo1. I can quickly set the *ControlSource* by picking a cursor column. Then I edit the *Name* property to match the *ControlSource*. This allows me to understand the data and what the objects are bound

to without needing to review all the *ControlSource* properties. As noted earlier this can be tedious. When the beta of Visual FoxPro 9 rolled out I decided I could take advantage of this technology to automatically alter the object names to conform to my standards.

The code that follows changes the name. Originally we thought it would be cool to select several objects and change all the names at once, but the Name property is not available when multiple objects are selected. If you do not use Hungarian notation for your column and property names, change the llHungarian variable to .F. and recompile the program.

```

LOCAL llHungarian, ;
      lnResult, ;
      lcPrefixes, ;
      lnObjects, ;
      lcBaseName, ;
      lcObjectName, ;
      lcBaseClass

DIMENSION laObject[1]

llHungarian = .T.
lnResult    = ASELOBJ(laObject)
lcPrefixes  = [chk|col|cbo|cmd|cmg|cnt|ctl|cur|cad|cus|dte|edt|frm|] + ;
               [frs|grd|grc|grh|hpl|img|lbl|lin|lst|olb|ole|opt|opg|] + ;
               [pag|pgf|phk|rel|sep|shp|spn|txt|tmr|tbr|xad|xfd|xtb|]

IF lnResult > 0
  lnObjects = ALEN(laObject, 1)

  * The Name property that this Property Editor (PE) works with
  * cannot be selected when multiple objects are selected
  * in the designer. Therefore it might be unnecessary in this
  * PE. We left this infrastructure here since it was created
  * as an example of how you can write these tools.
  FOR lnI = 1 TO lnObjects
    IF LOWER(SUBSTR(laObject[lnI].Name, 1, 3)) $ lcPrefixes
      * Nothing to do
    ELSE
      IF NOT PEMSTATUS(laObject[lnI], "ControlSource", 5) OR ;
        EMPTY(laObject[lnI].ControlSource)
        lcBaseName = laObject[lnI].Name
      ELSE
        * Need to look for the period delimiter between cursor and fieldname
        * if it exists, otherwise keep on moving with first position of
        * bound object property or variable.
        lnPeriodPosition = RATC(".", laObject[lnI].ControlSource, 1) + 1

        IF llHungarian
          lcBaseName = SUBSTR(laObject[lnI].ControlSource, lnPeriodPosition + 1)
        ELSE
          lcBaseName = laObject[lnI].ControlSource
        ENDIF
      ENDIF

      lcBaseClass = LOWER(laObject[lnI].BaseClass)

      DO CASE
        *=====
        CASE lcBaseClass = [checkbox]
          lcObjectName = [chk] + lcBaseName

        *=====

```

```

CASE lcBaseClass = [collection]
    lcObjectName = [col] + lcBaseName

*=====
CASE lcBaseClass = [combobox]
    lcObjectName = [cbo] + lcBaseName

*=====
CASE lcBaseClass = [commandbutton]
    lcObjectName = [cmd] + lcBaseName

* Lots of code cut out here for this whitepaper, see program
* to see all the objects handled.

*=====
CASE lcBaseClass = [textbox]
    lcObjectName = [txt] + lcBaseName

*=====
CASE lcBaseClass = [timer]
    lcObjectName = [tmr] + lcBaseName

*=====
CASE lcBaseClass = [toolbar]
    lcObjectName = [tbr] + lcBaseName

*=====
CASE lcBaseClass = [xmladapter]
    lcObjectName = [xad] + lcBaseName

*=====
CASE lcBaseClass = [xmlfield]
    lcObjectName = [xfd] + lcBaseName

*=====
CASE lcBaseClass = [xmltable]
    lcObjectName = [xtb] + lcBaseName

*=====
OTHERWISE
    lcObjectName = [NotHandled] + lcBaseName
ENDCASE

    laObject[lInI].Name = lcObjectName
ENDIF
ENDFOR
ELSE
    * Nothing to do
ENDIF

RETURN

```

The program grabs a reference to all objects selected. In the case of the *Name* property, this will only be one object. If the Name property already has a valid prefix, nothing changes. If the object has a *ControlSource*, the object name is based on the *ControlSource* and Hungarian rules. If no *ControlSource* exists, the object name is the current name with a new prefix. The name is changed when the `laObject[lInI].Name = lcObjectName` code executes. You must programmatically change the property.

The implementation is fairly straightforward. First install the program in your favorite tools folder. You need to edit the “E” record in the FoxCode table mentioned earlier in this section

where the Abbrev column is “Name”. The TIP column metadata needs a script attribute added to the existing Member Data XML. Here is an example:

```
<VFPData><memberdata name="name" type="property" favorites="True" script="DO <your folder here>\ApplyNamingStandardsPropertyEditor.prg"/></VFPData>
```

Replace the <your folder here> with the folder you install the program. I use an absolute folder since it is the easiest way to guarantee Visual FoxPro can find it. Relying on relative pathing or `SET PATH` could be a problem depending on your environment. When you select the *Name* property and press the Property Editor button, the `ApplyNamingStandardsPropertyEditor` program will execute and change the Name property accordingly. If Visual FoxPro cannot find the program an error is thrown.

The biggest advantage of a Property Editor is working with your own custom properties. The Property Sheet typically has native editors for intrinsic properties and a user interface making the selection of a property value very easy. These custom properties might be part of your custom framework, or they could be just for a generic class you created and made available to your team or the Fox Community.

It might be obvious, but we want to point out that Property Editors are only available at development time. Visual FoxPro 9 has extended the implementation of IntelliSense to the run-time, but Property Editor Member Data is specific to defining how the Property Sheet works. The Property Sheet is not available at run-time, so the Property Editors are not available at run-time.

BuilderX property editor (Examples: *BuilderXPropertyEditor.scx/sct*)

Earlier in this whitepaper we address the BuilderX property and how it directs the builder manager in VFP to run a specific builder for the object. This property requires some settings to run a specific class. While this is not a problem to set, we created a property editor to ease this setting.

This property editor demonstrates how you can directly call a form. You only need one row in the IntelliSense table if you want the property editor globally available for all objects. You add an Editor record per the specifications found in **Table 10**. This form will assemble the class library and class included in BuilderX when you have a class as the builder. This builder can be extended to handle standard forms and programs.

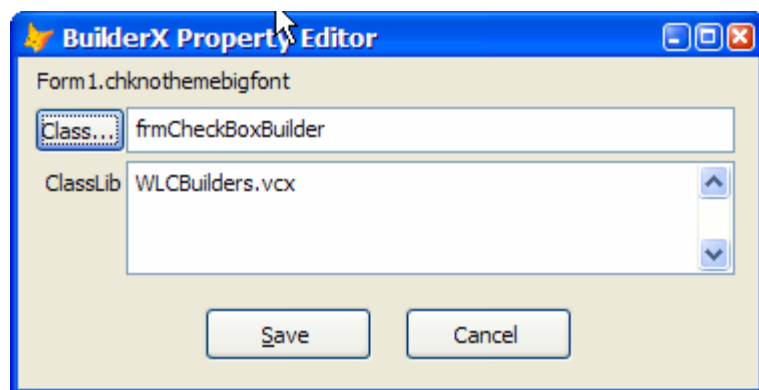


Figure 32. The BuilderX Property Editor is an example of a Property Editor.

Table 10. The record details in VFP's FoxCode table to define how the BuilderX property editor is called globally (you need to substitute the folder in the script with the folder you install the form.

Type	Abbrev	Cmd	Tip	Data
E	Builderx	{}	<VFPData><memberdata name="builderx" type="property" display="builderx" script="DO FORM D:\Presentations\SouthwestFox2004\BuildersMade Easy\Examples\BuilderXPropertyEditor.scx"/></VFPData>	

Logical toggler property editor (Examples: BuilderXPropertyEditor.scx/sct)

One of the features of the VFP Property Sheet I like is the ability to double click on a logical property and have it toggle the setting from true to false or vise-versa. I have always wanted to have this available for my custom properties that are logical. With the addition of property editors in VFP we can easily implement this.

This example demonstrates how you can implement a property editor completely in code in the IntelliSense table. There is no user interface since all this does is toggle the property setting. The code that toggles the setting is placed in a script record (S), in the DATA column (see **Table 11**). Once this record is established you establish an editor record (E) for each logical property you want defined. In the example defined in Table 11, you specify the logical property in the ABBREVS column, define which script record is called in the CMD column ({ WLCLogicalSpt}), and add the member data to call the generic RunPropertEditor feature of IntelliSense, passing it the custom property you define in the ABBREVS column (replace the <YOUR PROPERTY> text with the property name in lower case). Now the property editor button is enabled for you custom logical property and you can double-click on you property in the VFP Property Sheet and have the same behavior as an intrinsic logical property.

Table 11. The record necessary in VFP's FoxCode table to define how the logical toggler property editor is called.

Type	Abbrev	Cmd	Tip	Data
S	WLCLogicalSpt	{}		<pre>#DEFINE IBX_CAPTION "Logical Property Editor" #DEFINE USER_CANCEL "__usercancelled__" LPARAMETERS tcProp LOCAL ARRAY laObjs[1] LOCAL lcRetVal, lnCnt, loCtl, llDefValue IF ASELOBJ(laObjs)=0 IF ASELOBJ(laObjs,1)=0 RETURN ENDIF ENDIF ENDIF llDefValue = IIF(LEN(laObjs,1)=1,laObjs[1].&tcProp,"") FOR lnCnt = 1 TO LEN(laObjs,1) loCtl = laObjs[lnCnt] IF PEMSTATUS(loCtl, tcProp, 5) loCtl.&tcProp = NOT llDefValue ENDIF ENDFOR</pre>
E	<YOUR PROPERTY>	{WLCLogicalSpt}	<VFPData><memberdata name="lsecurityenabled" type="property" script="DO	

Type	Abbrev	Cmd	Tip	Data
			(_CODESENSE) WITH 'RunPropertyEditor','<Y OUR PROPERTY>'></VFPDa ta>	

What third-party builders are available?

The Fox Community is well known for the contributions each individual developer makes when they share something they have created for themselves and then make it available for other developers to use and learn from. I am not endorsing any of these builders, just passing along information on some builders available for free.

WLC ProjectBuilder (Example: *CPROJECTHOOK5::FRMPROJECTBUILDER*)

The WLC Project Builder from White Light Computing (www.whitelightcomputing.com) is a combination of the VFP Project Build dialog, the Build Version dialog and the Project Information dialog. How many times have you made that last gold production build and find out that you forgot to set Debug Code off in the Project Information dialog resulting in a 50 megabyte executable on the 500 CDs that were just cut? This dialog brings all the compiler settings together so you can build the executable with all the information in front of you at one time.

It is important to note that there are several features in the WLC Project Builder (see **Figure 33**) that work in conjunction with the WLC ProjectHook (included in the same class library), but it is not required. In fact, there is no requirement for any projecthook at all. The only requirement is that one project (or more) must be open.

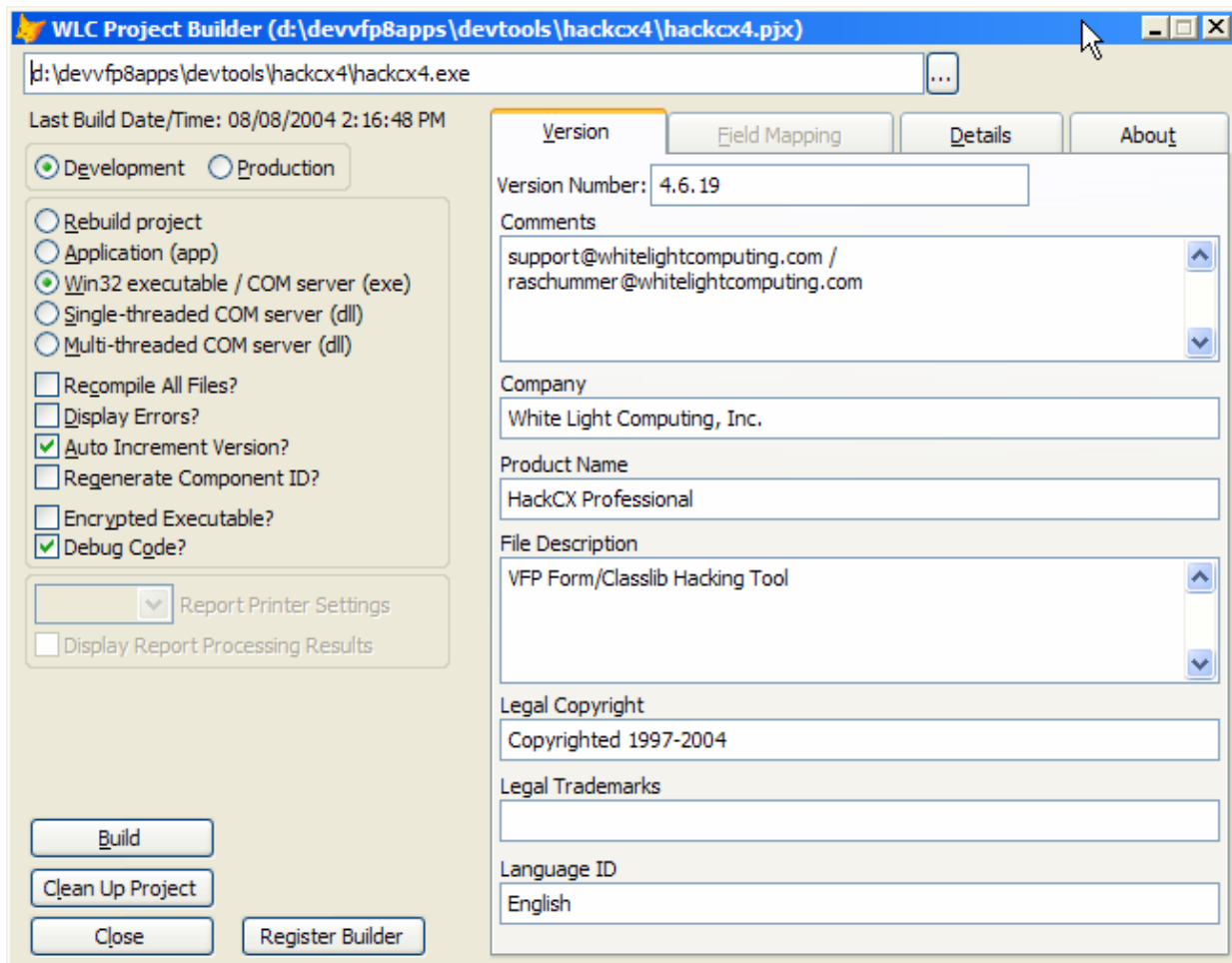


Figure 33. The WLC Project Builder in action for a project using the WLC ProjectHook.

WLC Label Builder (Example: `WLCBUILDERS.VCX::FRMLABELBUILDER`)

The WLC Label Builder was developed out of pure frustration to line up labels and have a label named almost identical to the entry object that it is associated with. This simple builder lines up two objects selected together (of which one is a label). If there are two objects it will name the label object “lbl” followed by any text in the name of the other object after the first three characters. This means the builder will rename a label named “Lblbase2” selected with a textbox named “txtLastName” will rename the label “lblLastName”. Other common properties are exposed and there is a developer tool font enforcer (our standard is Tahoma 8).

The labels can be lined up next to the object or on top of the object. If you line up a label and a shape object, the label object is place on top of the shape and is given the *BackStyle* of Opaque.

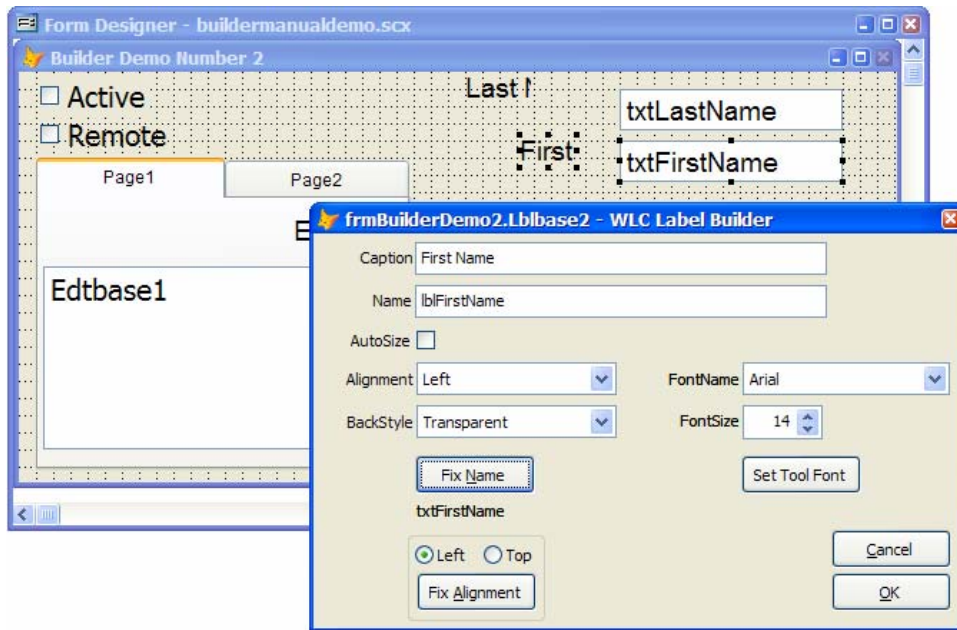


Figure 34. The WLC Label Builder eliminates frustration lining up labels with data entry objects and enforces development naming conventions..

Stonefield Grid Builder (Example: DEVTOOLS.ZIP)

Have I mentioned that Doug Hennig creates some of the best tools around? Well if I have not, let me mention it one more time. Doug takes the VFP native grid builder and makes it do one simple little thing better, sizing columns to the size of the data. Doug explains in detail all that it took to do this in his whitepaper that accompanies the source code for this upgrade. Simple and very useful.

Stonefield RI Builder (Example: DEVTOOLS.ZIP)

Doug Hennig's Stonefield RI builder was the first builder I downloaded and the one I used the most often until recently. Doug has taken the VFP native RI Builder, fixed some bugs, added some capabilities to integrate Steve Sawyer's lean and mean NEWRI.PRG code, and made a respectable RI Builder that Microsoft should have shipped two versions ago. If you are using the VFP RI Builder you will be very familiar with this builder since they look identical. The resulting code can look very different (in a good way). This one is a must have for developers writing applications that use VFP tables and database containers, second in importance only to the Stonefield Database Toolkit.

Tax RIBuilder (Example: TAXRIBUILDER.ZIP)

This shareware builder was written by Walter Meetzer is available on the www.UniversalThread.com. The claim to fame is that it creates faster and more compact referential integrity code. It also claims to provide more flexibility maintaining tables doing transactions that normally violate the referential integrity. Rules that I have not seen in other referential integrity builders that this one does handle is the Force Blank when deleting, and the Allow Blank when inserting. Details are provided in a Readme.txt file.

Mark McCasland's CursorAdapter Builder (Example: CABUILDER.ZIP)

This is a brand new builder and a work in progress as of the writing of this whitepaper. Mark has jumped on the new CursorAdapter class and created a builder that will connect to a VFP database container, SQL Server, or an Oracle database and create a Cursor Adapter for each table. Not only that, it will generate code for Insert, Update and Deletes. All the generated CursorAdapters are stored in a class library, subclassed from your specific CursorAdapter baseclass. If you are moving to VFP 8 and are looking at the CursorAdapter technology, this builder is worth a look for sure. This builder is available on www.UniversalThread.com.

ProjBuilder (Example: PROJUILDER.ZIP)

This builder was developed by Erik Moore (well known for his numerous tool contributions to the Fox Community) and is available on www.UniversalThread.com. It is a replacement for the Visual FoxPro Build Dialog with additional functionality that includes saving the path that the executable was compiled to for each project, which saves a time locating the directory each time. This builder handles some of the mundane tasks we need to perform before that gold build like packing class libraries and cleaning the printer driver information from reports. Erik has numerous features for development of COM servers including bouncing IIS and COM+ applications so builds can complete successfully. All settings are saved to the Windows registry so they do not have to be established each time you build.

Stored Procedure Builder (Example: SPBUILDER.ZIP)

This builder was developed by Erik Moore and is available on www.UniversalThread.com. This builder generates stored procedures in a SQL Server database to insert new records, update and deleting existing ones, retrieving primary keys, getting all records from a table, and getting a zero record set from a table. This builder generates a script file that can be saved and run later, or can generate the stored procedures directly in the database. The builder requires SQL Server 7.0 or later and has to be run on a computer that has SQL Server loaded because it requires SQL DMO which is used to manipulate the database.

What third-party builder resources are available to developers?

I did find it interesting when I started to research builder technology that there were very few resources available on this topic, but it matches the response I typically get during discussions with Fox developers. There are several excellent resources available that I can recommend without hesitation.

The first is Doug Hennig's whitepaper on developer tools. Doug wrote this many years ago, yet is still is one of the leading sources of builder information. This whitepaper is available from www.Stonefield.com.

If you do not own a copy of the *Hacker's Guide to Visual FoxPro 7.0* available from www.Hentzenwerke.com, make sure you put this paper down and head over and buy a copy. This is the bible of Visual FoxPro in my opinion and it has a terrific section on builders that Steven Black contributed. This is top gun information from a leading authority.

Speaking of Steve Black, the Fox Wiki (www.fox.wikis.com) has several topics on builders that I found useful over the years. The www.UniversalThread.com has one of the single largest download collections around for Fox developers. There are several builders that were mentioned earlier in this whitepaper that are available from this site.

There are numerous articles in FoxTalk (mostly written by Doug Hennig) and FoxPro Advisor. You can order back copies and CD-ROMs from both publications.

What issues might I face with builder technology?

There is one specific gotcha that I want to mention before closing this whitepaper. If no specific builder registered in the builder table (Builder.dbf) and there is only one “ALL” builder, the “ALL” builder is run without any prompting. This can be a serious problem if the builder is not written to make some validation checks you could find the builder behavior undesirable. I recommend that you write at least two “All” builders so you are forced to pick one builder when selecting an object that does not have a builder registered for its type.

Conclusion

Visual FoxPro 3.0 was introduced back in 1995 in San Diego to a couple thousand developers at DevCon. At this conference I saw a session on builders tag teamed by John Alden and David Anderson. It was the best session of the conference and really inspired me at the time. I saw a lot of possibilities with this technology. Unfortunately I did not act on the inspiration. At another conference Doug Hennig did a spectacular session on developer tools which included the builder technology. Again I was inspired, and again I did not act upon the inspiration. I later saw an opportunity to implement a bunch of builders for Visual MaxFrame v4.0 for my team, but was too busy with my regular job duties to do anything about it. Finally I saw the light yet one more time when I started writing applications using Visual FoxExpress which is littered with useful builders. I wanted to write a chapter in MegaFox on builders, but shipping was a feature.

The bug has bitten me and I have finally done something with my inspiration. I hope you can learn from my mistakes and react to any inspiration this whitepaper might have provided to you. I also look forward to feedback on the types of builders you create and hope that you will make them available to the rest of the Fox Community.

Special Thanks

I want to thank the user groups that put up with the rehearsals to insure that this presentation was refined for primetime initially at the Essential Fox conference. The Grand Rapids Area Fox User Group and the Detroit Area Fox User Group members provided excellent feedback to me and I really appreciate the frank and honest evaluations that were provided.

Copyright

Copyright © 2002-2004 Richard A. Schummer. All Worldwide Rights Reserved

Author Profile

Rick Schummer is the President and lead geek at his company White Light Computing, Inc., headquartered in southeast Michigan, USA. He prides himself in guiding his customer's Information Technology investment toward success. After hours you might find him creating developer tools that improve developer productivity, or writing articles for his favorite Fox periodicals and user group newsletters. Rick is a co-author of Deploying Visual FoxPro Solutions, MegaFox: 1002 Things You Wanted To Know About Extending Visual FoxPro, and 1001 Things You Always Wanted to Know About Visual FoxPro. He is a founding member and Secretary of the Detroit Area Fox User Group (DAFUG) and a regular presenter at user groups in North America. Rick has enjoyed presenting at GLGDW 2000-2003, Essential Fox 2002-2004, VFE DevCon2K2, and the Southwest Fox 2004 conference.

*raschummer@whitelightcomputing.com, rick@rickschummer.com,
<http://www.whitelightcomputing.com> and <http://www.rickschummer.com>*