Tech Note

embarcadero

Understanding RAD Studio's LiveBindings

Cary Jensen

February 2012

Americas Headquarters
100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

ABSTRACT

LiveBindings, first introduced in RAD Studio XE2, is a mechanism for updating the properties of an object based on an expression. LiveBindings represents a significant new mechanism for creating associations between objects, adding capabilities not previously found in RAD Studio.

LiveBindings is available to every type of application supported by RAD Studio XE2, both Delphi and C++ languages, both VCL and FireMonkey component libraries, and every deployment platform, from Windows to Mac OS to iOS.

This white paper is designed to familiarize you with LiveBindings. Here you will learn how to link controls in both VCL (visual component library) and FireMonkey applications using LiveBindings. I also examine the role of the expression engine, which will include a look at expressions and scopes.

LiveBindings

LiveBindings is a mechanism in RAD Studio for associating the property of an object with an expression. The expression can be another property of the same object, a property of a second object, or a complex expression that includes properties of objects, literals, operators, and methods.

What LiveBindings do is not new; RAD Studio has always had the ability to assign a value to an object's property. However, how LiveBindings work and how they can be applied is far different than the traditional VCL associations.

Let's consider a classic example of traditional VCL interaction: data awareness in VCL controls. Let's begin with a DBEdit. This control can be associated with a DataSource, which in turn can be associated with a DataSet, such as a ClientDataSet.

Depending on the properties you set on these three controls (the DBEdit, the DataSource, and the ClientDataSet), changes to the data in the DataSet are reflected in the text that appears in the DBEdit. Furthermore, if changes are made to the DBEdit, these changes will automatically appear in the associated field of the current record of the ClientDataSet (assuming that the ClientDataSet is Active, the associated field is writable, and the DataSource is using its default properties).

While the VCL has supported dynamic component interaction, it has always been limited to certain combinations of components, as in the data awareness example, or for specific parts of the VCL framework.

An example of framework-level interaction can be found in the garbage collection scheme introduced in the TComponent class. All TComponents can have an owner. When they do, they inform their owner of their existence upon their creation, and inform their owner once again upon their destruction. During an owned component's lifecycle, the owner holds a reference to the owned component. This reference is used to free the owned component at the time of the owner's own destruction. By this mechanism, any component that is placed on a form at design time is guaranteed to be freed when the form on which it is placed is freed.

Both data awareness and garbage collection rely on features of specific classes, and affect specific properties, as part of the design of the components that interact using these mechanisms. LiveBindings, by comparison, works by a different mechanism, and can be applied to a much wider set of classes and to almost any kind of property.

For example, LiveBindings can be used to associate the Position property of a ProgressBar with the Text property of a spinner control (like the SpinEdit component found on the Samples page of the VCL Tool Palette). The ProgressBar and the SpinEdit have not been

designed to be aware of each other. Instead, a LiveBinding can perform this association by the fact that these two objects have compatible properties (TProgressBar.Position and TSpinEdit.Value).

Actually, this example is a little too simple since the two properties mentioned are Integer properties. LiveBindings can also keep the Text property of an Edit and the Position property of a ProgressBar synchronized, even though one is a String and the other an Integer (though entering a value into the Edit that cannot be converted to an Integer would cause problems).

So far, I've only discussed the VCL when speaking about LiveBindings. As far as RAD Studio XE2 is concerned, it is the FireMonkey component library that is the biggest beneficiary of LiveBindings. This is because, unlike the VCL, FireMonkey components lack the data-awareness framework built into the VCL.

That FireMonkey does not supply classes like DBEdit, DBGrid, and DBImage is not a problem for FireMonkey. LiveBindings, and their flexibility, allows you to associate most FireMonkey components with bidirectional links to DataSets, as well as other FireMonkey controls.

In this paper I am going to provide you with a thorough introduction to LiveBindings. I am going to start slow, showing you how to use LiveBindings to create associations between controls, both in the world of VCL and FireMonkey.

Next, I examine the various concepts that make LiveBindings work. Here you will learn about expressions, scope, managed versus unmanaged LiveBindings, and more.

I continue with a look at the various design-time components provided by the LiveBindings framework, including BindingsList, BindScope, BindExpression, List, and Link. Here you will be introduced to a number of design-time component and property editors that assist you in configuring LiveBindings.

This paper concludes with a discussion of the current state of LiveBindings, as they appear in RAD Studio XE2, and considers what types of enhancements we might expect as this framework evolves over the next few versions of RAD Studio.

GETTING STARTED WITH LIVEBINDINGS

I've seen a number of demonstrations of LiveBindings that jump right into the deep end, beginning with a detailed discussion of the inner workings of the expression engine. While there is a place for that approach, I am going to start this paper by looking at how to configure a simple LiveBinding, and watch it in action. After that we can talk about some of the underlying principles.

Author's note: I am a Delphi developer, and all sample project associated with this paper are Delphi projects. Note, however, that C++Builder supports LiveBindings.

Let's begin by creating a simple LiveBinding in a VCL application.

Code: The code for this project is available in the SimpleVCLLiveBindingsExample project, which you can download along with this white paper.

A SIMPLE VCL LIVEBINDINGS EXAMPLE

Use the following steps to create a simple LiveBindings for a VCL application:

1. From RAD Studio's XE2's main menu, select File | New | VCL Forms Application - Delphi.

2. Add to the main form a StatusBar control from the Win32 page of the Tool Palette.

3. Using the Object Inspector, set the StatusBar's SimplePanel property to True. This will permit you to use the SimpleText property of the StatusBar.

4. Now you are ready to add a LiveBinding. Right-click the StatusBar and select New LiveBinding…. RAD Studio responds by displaying the New LiveBinding dialog box shown in Figure 1.

The New LiveBinding dialog box has quite a few LiveBinding types listed in it. I will explain what each of these do, and when you should use them, but that discussion should wait until we have covered a little more of the underlying concepts. For now, simply select the TBindExpression LiveBinding and click OK.

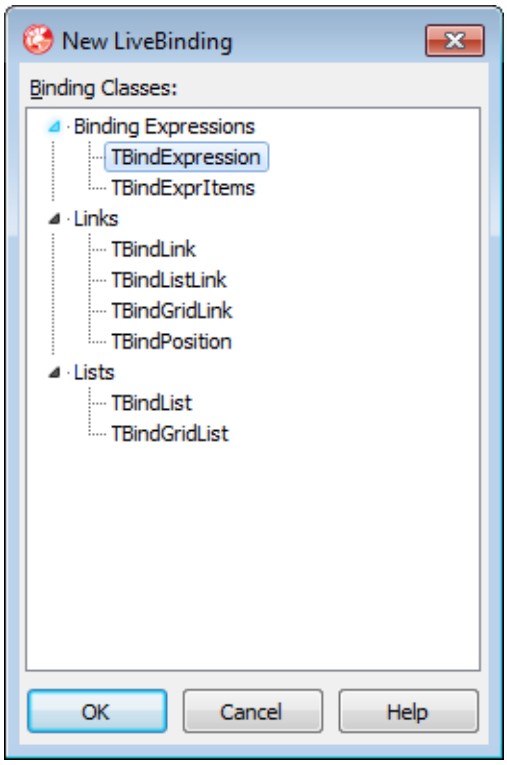

Figure 1. The New LiveBinding dialog box

Several things happen when you accepted this dialog box. First, a new component was added to your form: BindingsList1. This is a BindingsList, and by default there will be one on each form to manage the LiveBindings that you define on that form.

The second thing that happened is that a new LiveBinding, BindExpressionStatusBar11, was created, and it now appears in the Object Inspector, as shown in Figure 2.

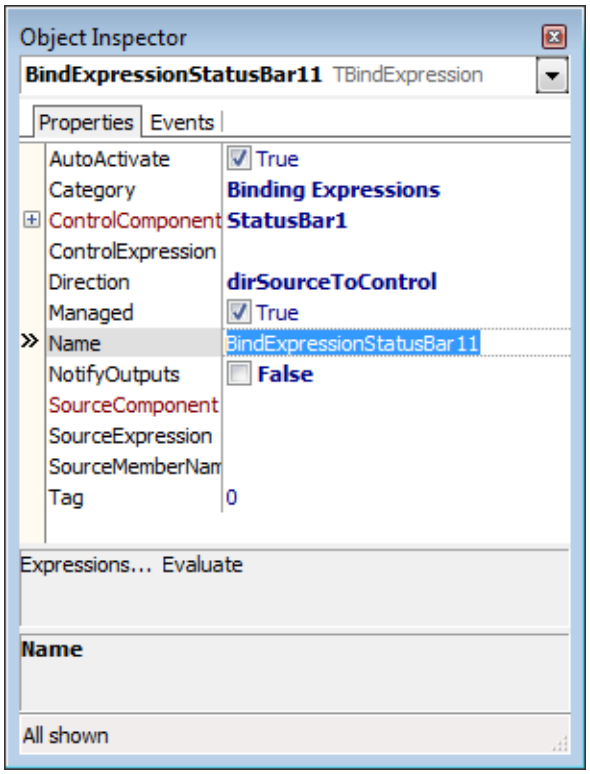

Figure 2. The newly created LiveBinding appears in the Object Inspector

5. One of the more important properties of this LiveBinding, ControlComponent, was set for you. You should now set the ControlExpression property to SimpleText (referring to the SimpleText property of the StatusBar), set the SourceComponent to Form1, and set the SourceExpression property to the following string:

```
ClassName() + ', Width: ' + ToStr(Width) + ', Height: ' + ToStr(Height)
```

When you are done, your Object Inspector should look something like that shown in Figure 3.

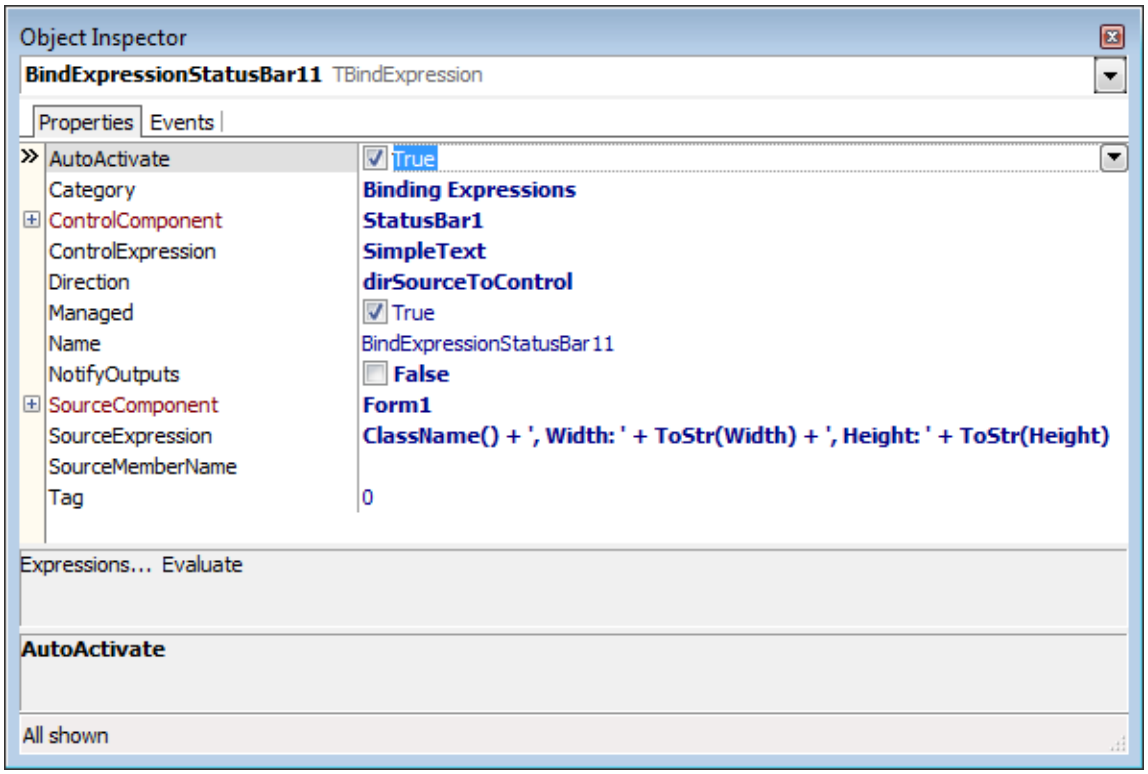

Figure 3. The new LiveBinding has been configured in the Object Inspector

Without going into a lot of detail, what this LiveBinding says is that the expression engine should assign a string to the SimpleText property of the StatusBar. Furthermore, this string should call the form's ClassName method, read the Width and Height properties of the form and convert these Integer values to strings, and concatenate those strings with the form's class name and the string literals you included in the SourceExpression property.

All you need to do now is to inform the expression engine to apply the LiveBinding. Since the SourceExpression includes information about the Width and the Height of the form, this notification should probably be performed each time the form changes its size. Fortunately, there is an event handler for that: OnResize. Use the following steps to perform this notification.

6. Select the Form in the Object Inspector, select the Events tab, and then double-click on the OnResize event handler. Delphi will generate a stub for this event handler.

7. Enter the following code in this event handler:

```
BindingsList1.Notify(Sender, '');
```

When you are done, your event handler will look something like this:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  BindingsList1.Notify(Sender,'')
end;
```

8. You are ready to watch this LiveBinding in action. Press F9, or click the Run button on RAD Studio's Toolbar, to run the form. Next, use your mouse to drag the lower-right corner of the form in order to change the shape of the form. As you do, the string that appears in the StatusBar will update, similar to what you see in Figure 4.

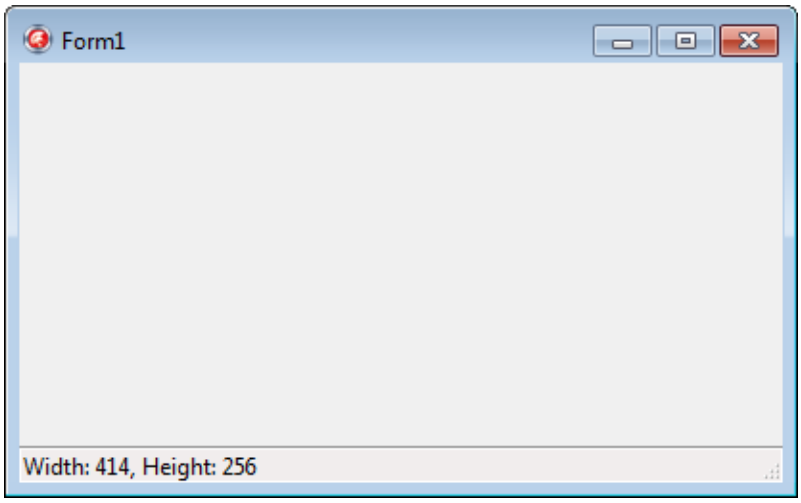

Figure 4. The LiveBinding is updating the StatusBar each time the form changes shape

Ok, I know what you are probably thinking. You're probably saying to yourself, "So what? I could have written a simple event handler that does the same thing, without having to use the LiveBinding."

You are correct, and here is an example of what that event handler might have looked like:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  StatusBar1.SimpleText := Self.ClassName + ', Width: ' + IntToStr(Self.Width) +
    ', Height: ' + IntToStr(Self.Height);
end;
```

My response to that comment has three parts. First, the second event handler, the one that explicitly assigns a value to the StatusBar's SimpleText property, needs to have intimate knowledge of what operation has to be performed. By comparison, the first event handler simply notifies the expression engine that something about the Sender has changed. The first event handler has no details about the change, nor does it specify what should happen in response. The expression engine does the actual assignment based on the LiveBinding, and the LiveBinding, not code, defines what happens.

Second, not all LiveBindings require that you notify the expression engine. Many of the LiveBindings, including Lists and Links, require no event handlers at all. If we used those LiveBindings (and we will get around to that) the forms that employ LiveBindings have far less code than those that use traditional event handlers to perform the actions.

Third, for those LiveBindings that do require an event handler, all of them could potentially refer to this one, simple event handler, the one that calls the Notify method of the BindingsList. As a result, a form that uses LiveBindings to perform its various tasks may have none or just one event handler. By comparison, if you performed those tasks using code, as in the second OnResize event handler, there would have to be many different event handlers, each one invoking its specific task.

Let us now take a look at using LiveBindings in a FireMonkey application.

Code: The code for this project is available in the SimpleFireMonkeyLiveBindingsExample project, which you can download along with this white paper.

A Simple FireMonkey LiveBindings Example

Use the following steps to create a FireMonkey application that uses LiveBindings. Before you start, make sure to close any applications you have open in RAD Studio.

1. Select File | New | FireMonkey HD Application - Delphi.

2. Add to your form two components, a Label and a TrackBar, both of which can be found on the Standard page of the Tool Palette. When placing these components, ensure that you place them both on the form. Many FireMonkey components can be parents, and it would be easy to place the TrackBar within the Label (or visa versa). You can use the Structure pane in RAD Studio's IDE to confirm that both the Label and TrackBar appear directly on the form.

3. Select the Label. Using the Object Inspector set the Text property to FireMonkey, and the TextAlign property to taCenter.

3. With the Label still selected, double-click the Font property to display the Font dialog box.

4. Using the Font dialog box, set Font to Comic Sans MS and Size to 48. Close the Font dialog box.

5. You will now need to adjust the size of the Label, using its grab handles, to make the text of the label visible. While you are doing this, position the Label in about the center of the form.

6. Now select the TrackBar. Using the Object Inspector, set the Max property to 180 and the Min property to -180.

7. You should now position the TrackBar beneath, and centered below, the Label. You might also want to stretch the TrackBar. When you are done, your form should look something like the form shown in Figure 5.

Figure 5. The main form of a new FireMonkey application

We are ready to create the LiveBinding. Use the following steps:

8. Right-click the Label and select New LiveBinding. RAD Studio responds by displaying the New LiveBinding dialog box shown in Figure 6.

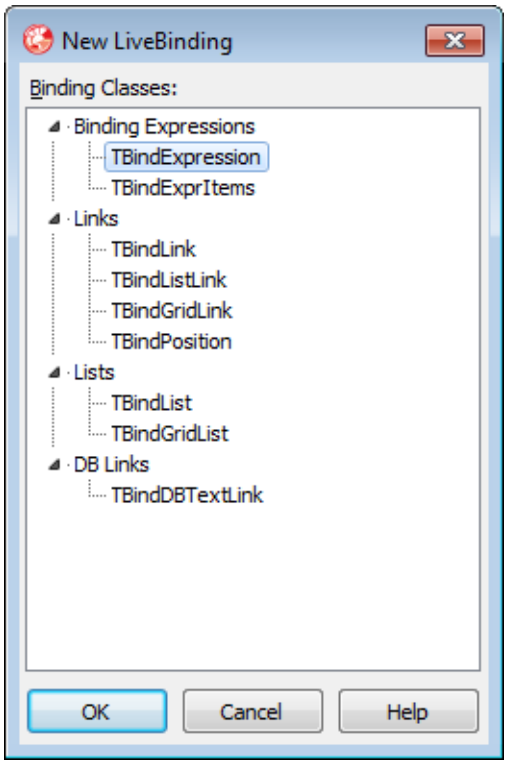

Figure 6. The New LiveBinding dialog box for a FireMonkey component

If you compare this New LiveBinding dialog box to the one in Figure 1, you might notice that this dialog box lists one extra LiveBinding, the one that appears under the DB Links node. In this release of RAD Studio FireMonkey supports a handful (seven) LiveBinding classes not found in the VCL library. These classes, however, are very closely associated with classes that are available in the VCL. I'll talk about these FireMonkey classes in more detail towards the end of the paper.

9. Once again select the TBindExpression LiveBinding and click OK.

As before, RAD Studio responds by creating a BindingsList and a TBindExpression. The TBindExpression, named BindExpressionLabel11 in this case, is selected in the Object Inspector.

10. With BindExpressionLabel11 selected in the Object Inspector, set ControlExpression to RotationAngle, SourceComponent to TrackBar1, and SourceExpression to Value.

The LiveBinding is now fully configured. However, with this type of LiveBinding we still need to inform the expression engine that something has changed. One way to do this is to create an OnChange event handler for the TrackBar. To do this, double-click the

TrackBar and add the Notify call to the generated OnChange stub. When you are done, this event handler will look something like the following:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  BindingsList1.Notify(Sender, '');
end;
```

11. You are done. Now run the form (press F9). Next, change the position of the TrackBar. In response, the Label will rotate like that shown in Figure 7.

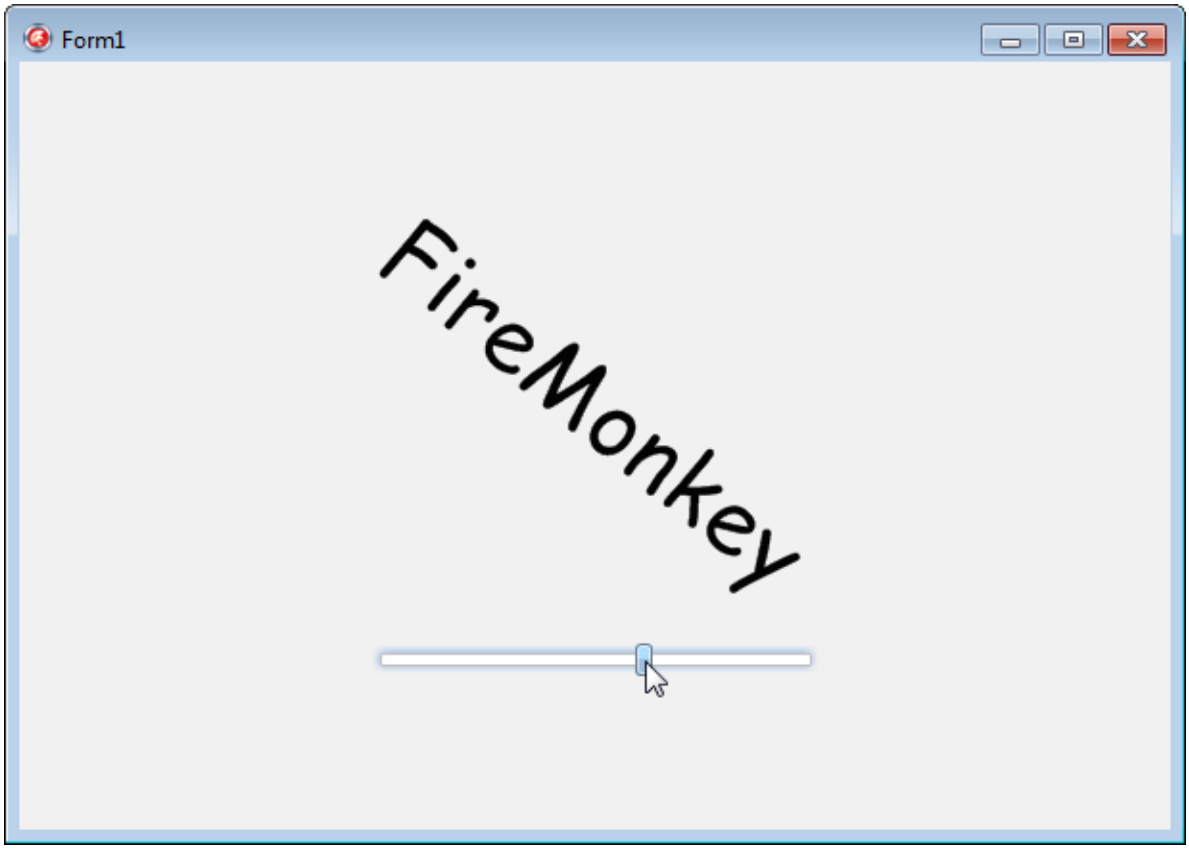

Figure 7. Changing the position of the TrackBar causes the Label to rotate

How it Works

Now that you've seen LiveBindings in action, it's time to take a closer look at how they work. At its core, LiveBindings is a mechanism for creating associations between objects and expressions. Expressions are strings that are evaluated to a value by the expression engine. LiveBindings use the output of the expression engine to assign the results of the evaluation to a property of an object.

Let's begin this discussion by looking at the expressions themselves. As part of this discussion I'll talk a bit more about scope. After that I'll take an in-depth look at each of the LiveBindings components.

EXPRESSIONS

Expressions are strings. These strings can consist of literals, operators, the properties of objects, and methods.

Consider the SourceExpression that we used in the VCL LiveBindings example, which looked like this:

```
ClassName() + ', Width: ' + ToStr(Width) + ', Height: ' + ToStr(Height)
```

This string contains all four elements.

LITERALS

The literals in this case are the string literals:

```
', Width: '
```

and:

```
', Height: '
```

These look like the same kinds of strings that you would use in your Delphi code, but there are differences. For one, these strings are evaluated by the expression engine, and unlike Delphi's compiler, the expression engine recognizes string literals enclosed in both single quotes, as in the preceding expression, as well as with double quotes. In fact, it's perfectly acceptable to mix and match string literals based on these different quote characters within the same expression. For example, the following expression would evaluated to the same result as the one used in the VCL example:

```
ClassName() + ", Width: " + ToStr(Width) + ', Height: ' + ToStr(Height)
```

There is a limitation however. If you begin a string literal with a single quote, it must close with a single quote, and the same is true about double quotes. If you are familiar with JavaScript, you are already accustomed to this approach to strings, and appreciate the value of it. For example, if your expression needs to include a quotation, you use the single quotes to define the literal, like this:

```
'He said "Yahoo!"'
```

Alternatively, if you want to use a contraction or a possessive, you can use the double quotes to delimit your string literal, like this:

```
"I'd say it's the cat's meow"
```

OPERATORS

The operator used in the expression from the VCL example is the concatenation operator, the + sign. Other operators can be used, where appropriate. Specifically, the literals used in the preceding expression were string literals, but that was coincidental. You could have used numeric literals, in which case numeric operators might be useful. For example, the following are both valid expressions, as far as the expression engine is concerned:

```
2 + 5
1.2 > 4.23
```

The expression engine evaluates the first expression to 7, and the second expression to a Boolean false. The expression engine also recognizes parentheses for prioritizing expression evaluation in complex expressions. For example, the following expression evaluates to 4, while the second one evaluates to 2:

```
2 * 3 - 2
2 * (3 - 2)
```

PROPERTIES

Returning to the expression from the VCL example, Width and Height are both properties of an object. Because this expression is the SourceExpression, these properties are necessarily those of the SourceComponent, Form1.

This is an issue of scope, and scope is a critical concept when it comes to expression evaluation. We will return to scope a just a moment, but for now it is sufficient to say that only properties of the SourceComponent can appear in the SourceExpression, and only properties of the ControlComponent can appear in the ControlExpression.

METHODS

The last of the expression building blocks are methods. There are two types of methods that you can call: custom methods and methods of object within scope of the expression. Let's consider each of these in turn.

Custom methods are special and are intended specifically for use in expressions that you want evaluated by the expression engine. In the VCL expression example, the ToStr method is used twice, and each time is it used to convert an Integer value to a string in order to create the resulting string expression that is assigned to the StatusBar's SimpleText property.

LiveBindings comes with a collection of utility custom methods that you can use with LiveBindings. We will see the list of the custom methods that you can use in your expressions when I talk about the BindingsList component. You can even write new custom methods, which you typically install by registering them using a design-time

package, but this is an advanced step that is normally not necessary since it is much easier to use methods in scope.

Methods of objects in scope are the public and published methods of objects visible to the expression engine. Consider the VCL example. The SourceControl is the form, which places the form in the scope of the expression engine for the purpose of evaluating the SourceExpression. ClassName is a class function that returns the name of the class on which the method is invoked. ClassName is introduced in TObject, making it available to all TObject descendants, of which TForm is one.

In the VCL expression example we used an unqualified reference to the ClassName method (just as we used an unqualified reference to its Width and Height properties). We could also have qualified the ClassName method with the variable Self, since the Form is the SourceControl. (We could have qualified the referenced property using Self as well).

There are a couple of restrictions associated with using methods in scope. First, you must always include open and close parens when referencing a method in scope, even if that method requires no parameters (as is the case with ClassName). Second, the method must be either public or published, as mentioned previously.

Methods can be either functions or procedures. Obviously, when it is a function it should return a type that is consistent with the expression, or which can be converted by one of the expression engines output converters, which I'll discuss next. When the method is a procedure it returns nil.

If you want to write a method that you call from an expression, you can add it to the class that you use as the ControlComponent or SourceComponent, depending on which expression (ControlExpression or SourceExpression) that you want to use the method in.

In some cases, such as when your control is not one that you wrote, you can add the method to the Form class itself. This is a valuable technique when your class is a Component that appears on the form at design time, in which case the Form is in scope through the Owner property of the Component. For example, if you added a method to your form that returned the current time as a string, and your SourceComponent is a Button that you placed on the form at design time, your expression may look something like the following:

```
"The time is " + Owner.GetCurrentTimeAsString()
```

That you can invoke methods that you write from your LiveBindings expressions is profound, and I am going to bring this point up again later in this paper. In short, method invocation permits LiveBindings to perform side effects. These side effects can be a tremendous source of power, power that is similar to the side effects that can be implemented through property accessor methods. How these side effects can be a game

changer, as far as how you implement features within your applications, is demonstrated later in this paper, in the section titled *The Future of LiveBindings*.

Author's Note: Realizing that LiveBindings could produce side-effects was a breakthrough for me. It meant that not only could a LiveBinding update an object in response to some change in data, it meant that a LiveBinding could also be a powerful source of change. Ironically, this realization came to me at 3:00am, making it nearly impossible for me to go back to sleep.

OUTPUT CONVERTERS

Before we move on to additional topics, I want to introduce another concept related to custom methods: output converters. Like custom methods, output converters are special routines that can perform transformations on the type of an evaluated expression. However, unlike custom methods, output converters are not something you include in your expressions. Instead, they are used by LiveBindings to transform the type of an expression result when the target property of the expression is of a different data type than the evaluated expression.

SCOPE

Scope defines what is visible to the expression engine by associating a name with an object, or in the case of an unnamed object, associating the variable Self with an object (this Self is distinct from the Self variable available from within a non-class method). At a minimum, the expression engine needs only one scope. However, when using LiveBindings, you generally provide both an input scope and an output scope, though it is entirely possible to use the same scope for both the input and the output, such as when you bind one property of an object to another property of that same object. And, in some cases, such as when the expression that is being evaluated is a constant, only an input scope is required.

The expression engine automatically adds the operators that you can use in expressions to its input scope. The LiveBindings components take responsibility for adding the registered custom methods and output converters to the input scope.

Scope is an abstract concept, but in practice, you really don't have to worry about it. LiveBindings components typically use a ControlComponent and a SourceComponent. In most cases, the LiveBinding adds the ControlComponent to the input scope and the SourceComponent to the output scope. (In some cases, it is the other way around depending on the direction of the expression evaluation. We'll get to that in a moment.) In most cases there's really nothing more that you need to do. When the same component is both the ControlComponent and SourceComponent, you are using the same scope for both the input and output. It's as simple as that.

However, if you scour the RAD Studio source code, as many developers do, you will run into references to scope repeatedly, with the IScope interface being one of the most common types. Scope means visibility as far as the expression engine is concerned. As a result, it is essential to the workings of LiveBindings.

LiveBindings Concepts

LiveBindings provide an interface between components and the expression engine, a lot like a DataSource plays the negotiator between DataSets and data-aware controls. In this section, I will introduce the principle properties that you will typically configure on any LiveBinding component. I will also delve a bit into expressions, binding direction, and managed versus unmanaged LiveBindings. This will prepare you for later when I cover the individual components of the LiveBindings framework.

ControlComponent and SourceComponent

Most LiveBindings require a control component (the object to which the expression will be applied), and in most cases, a source component as well. In the case of these LiveBindings components, the control and source components are both in a scope, and these scopes make the properties of their respective wrapped objects available to the expression engine.

Each LiveBinding has four principle properties when it comes to their configuring, and some have a fifth property. The four principle properties are the ControlComponent, the control expression or expressions, SourceComponent, and source expression or expressions. The fifth property, which is available when a BindScopeDB component is being used, is SourceMemberName.

Let's begin with the rather simple case of the TBindExpression, the simplest of the LiveBindings. ControlComponent is the component to which the LiveBinding is associated, and this is typically the component that has a property to which the result of the evaluated expression will be assigned. This LiveBinding has one ControlExpression, and it is the property that will receive the results of the evaluated expression.

In the SimpleFireMonkeyLiveBindingsExample project, the ControlComponent is the Label, and the ControlExpression is RotationAngle. Consequently, this configuration says "When the expression is evaluated, assign the resulting value to the RotationAngle of the Label."

The SourceControl is the component whose properties are available to the expression engine for the purpose of evaluating any expressions. In the case of the BindExpression LiveBinding, there is a single source expression, named SourceExpression.

Referring again to the SimpleFireMonkeyLiveBindingExample project, the SourceComponent is the TrackBar, and the SourceExpression is Value, a property that represents the position of the TrackBar. Taken together, the LiveBinding in this project says, "When notified, evaluate the Value property of the TrackBar and assign that value to the RotationAngle property of the Label."

SourceMemberName is a property of the SourceComponent. This property only makes sense in this version of RAD Studio if the SourceComponent is BindScopeDB. SourceMemberName permits a BindScopeDB to expose the individual fields of a DataSet. We will see BindScopeDB, and the SourceMemberName property, later in this paper.

LiveBindings Versus Expressions

Another distinction that is important in the world of LiveBindings is between LiveBindings and expressions. LiveBindings are components that support one or more expressions. Expressions are evaluated by the expression engine, and a LiveBinding uses the results of that evaluation to do something.

In the two simple examples presented earlier in this paper, a BindExpression LiveBinding was used. BindExpressions support one expression, and, as a result, produces the effect of assigning one value to one property. Most LiveBindings, however, support many expressions. As a result, a single LiveBinding may affect many different properties, and those properties might actually belong to more than one object.

In addition, if one or more of your expressions call methods that perform side effects, or read from or write to property accessor methods that exhibit side effects, the result can be remarkable. I hope that you find that exciting, because it is.

Control to Source and Back

So far, this discussion might have made it sound like the expression engine always assigns the value of the SourceExpression of the SourceComponent to the ControlExpression of the ControlComponent. While that is the default behavior for most expressions, it is not necessarily always the case. Depending on the LiveBinding component you are using, sometimes the assignment goes from source to control, sometimes it goes from control to source, and sometimes it goes both ways.

This is true concerning the expressions associated with the Bind Expression components (BindExpression and BindExprItems), the first type of LiveBindings that we will consider. Some of the more advanced LiveBinding types, such as List LiveBindings, have expressions that are distinctly unidirectional in nature. Again, I will discuss those details when we consider those particular components.

MANAGED VERSUS UNMANAGED LIVEBINDINGS

At the end of the section on the SimpleVCLLiveBindingExample project, I addressed the question about how LiveBindings are different from traditional event handlers, given that it was necessary to use an event handler to instruct the expression engine to evaluate the expression. In that discussion, I mentioned that some LiveBindings do not require event handlers. What this boils down to is the difference between managed and unmanaged LiveBindings.

Managed bindings require that the expression engine be notified that a change has occurred, and this does not happen automatically. With managed bindings, you must add code somewhere in your project that notifies the expression engine that it must re-evaluate the LiveBinding and perform the associated assignment. In most cases, you would add this code to an event handler, though it is also possible that you could execute this notification from within one of your custom classes, in response to a change that your class detects.

You have several options for notifying the expression engine. If you add the System.Bindings.Helper unit to your uses clause, you can use the TBindings class to call the class function Notify, to which you pass a reference to the source component, a string that contains the name of the property affected, and optionally, a BindingsManager reference. Such an invocation will look something like this:

```
TBindings.Notify(TrackBar1, 'Position');
```

This line of code instructs the expression engine to evaluate any expression associated with the SourceComponent TrackBar1 where the SourceExpression includes the Position property. Consider, for comparison, the following call:

```
TBindings.Notify(TrackBar1, '');
```

Here we are asking the expression engine to evaluate all expressions associated with the SourceComponent TrackBar1. Specifically, when you pass an empty string in the second parameter, all expressions linked to the specified SourceComponent are evaluated.

The Notify method accepts a third default parameter. If you omit this parameter, the default BindingsManager, named AppManager, is used. If you are employing a custom BindingsManager, you should pass it in this third parameter. For the record, few developers employ a custom BindingsManager, and therefore, you rarely see a third parameter in the Notify invocation.

While managed bindings are just that, bindings that you manage, unmanaged bindings take responsibility for notifying the expression engine for you. As a result, unmanaged bindings share some characteristics with RAD Studio's traditional component binding, such as that associated with the VCL's data-aware controls.

Unmanaged LiveBindings work by employing the Observer design pattern, which places some limits on which components you can use with unmanaged LiveBindings. Unlike RAD Studio's data-aware controls, however, LiveBindings represent a general framework, one that can be used to bind a wide variety of components or classes. RAD Studio's traditional binding mechanisms are much more limited, and often apply to a particular subset of components.

With respect to the LiveBindings components, the Bind Expression components support both managed and unmanaged LiveBindings, while the List and Link LiveBindings are strictly unmanaged LiveBindings.

There are additional classes that participate in LiveBindings, but most of these are under-the-cover type classes, including observers and editors. These are not components that you work with directly, but you will run into them if you poke around in the LiveBindings source code.

The Classes

Now that we've discussed the core concepts behind LiveBindings, it's time to take a closer look at the various classes that appear on the RAD Studio Tool Palette, as well as those available at design time from the various component and property editors. We'll begin by discussing the BindingsList and BindScope components. Then we'll move on to the LiveBindings components, specifically Bind Expressions, Lists, and Links.

BindingsList

The BindingsList component is a utility component for working with LiveBindings at design time. It provides you with a number of useful facilities. From an interactive perspective, the BindingsList component provides you with a convenient interface to the various LiveBindings that you have created on a form, which you can then use to access the individual expressions associated with those LiveBindings.

Code: The code for this project is available in the VCLLiveBindingsproject, which you can download along with this white paper.

You can place a BindingsList component onto your form from the Tool Palette, but that is rarely necessary. The first time you create a LiveBinding interactively on your form, RAD Studio will place a BindingsList on your form for you. That was probably obvious to you if you followed along with the hands-on steps provided at the beginning of this paper.

Once there is a BindingsList on your form, it will give you access to the list of all of your LiveBindings and their expressions. To access this list, you can double-click the BindingsList component or you can right-click the BindingsList component and select

Binding Components. The BindingsList responds by displaying the BindingsList component editor, like that shown in Figure 8.

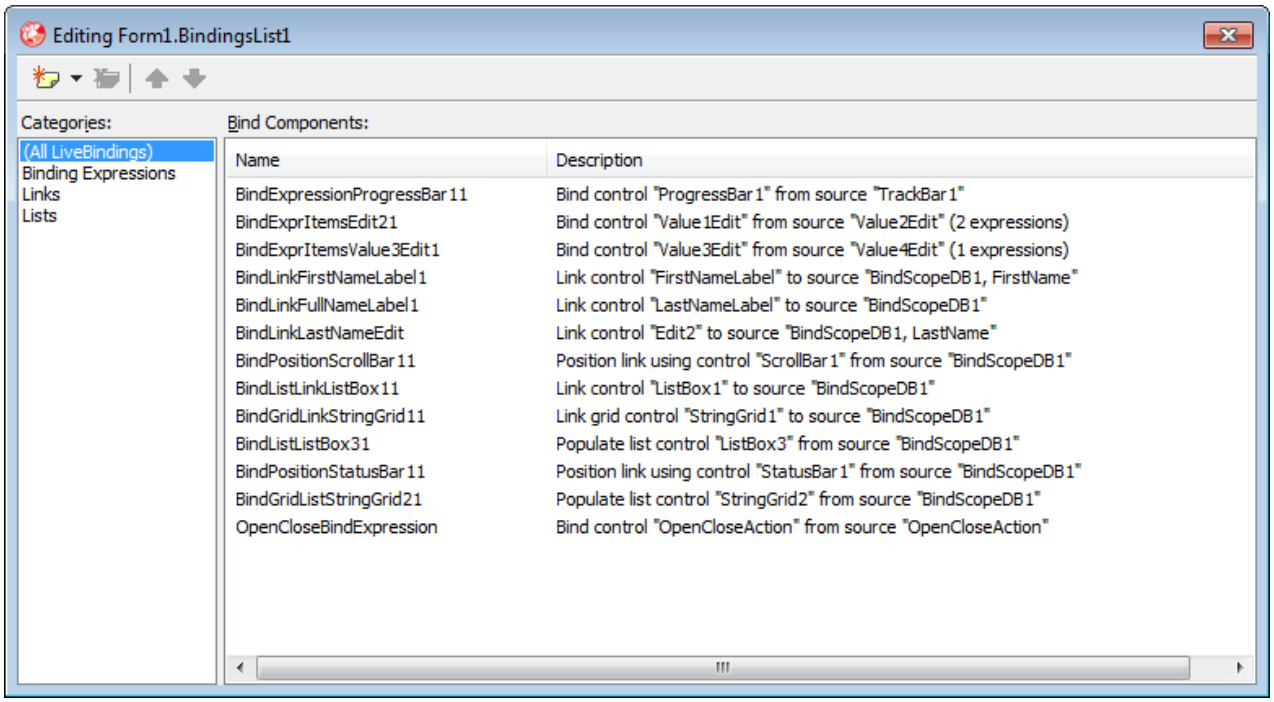

Figure 8. The BindingsList component editor

This is the BindingsList component editor from the BindingsList for the VCLLiveBindings project. As you can see, on the left-hand side you can select to view all LiveBindings that have been configured at design time, or you can select a particular class of LiveBindings. Depending on which category you select (or All LiveBindings), the BindingsList component editor will filter the LiveBindings that appear in the right-hand pane of the editor.

You can also use the BindingsList component editor to add or remove a LiveBinding from your form. To add a LiveBinding from the BindingsList component editor, right-click and select New LiveBinding, press the Ins key, or click the Add New Binding button from the BindingsList component editor tool bar. To delete a LiveBinding, select the LiveBinding you want to delete and press Del, right-click it and select Delete, or select it and click the Delete button from the BindingsList tool bar. However, most developers add LiveBindings directly to a control, like you did in the hands-on examples described earlier in this paper.

You don't actually see the individual expressions from this page, but you will if you double click one of the LiveBindings that appear in the left-hand pane of the BindingsList component editor. Figure 9 shows then Expressions editor that will appear if you double-click the BindGridListStringGrid21 LiveBinding from the BindingsList component editor

shown in Figure 8. This LiveBindings is a BindGridList LiveBinding, and it is one of the more sophisticated of the LiveBindings, at least as far as this version of RAD Studio is concerned.

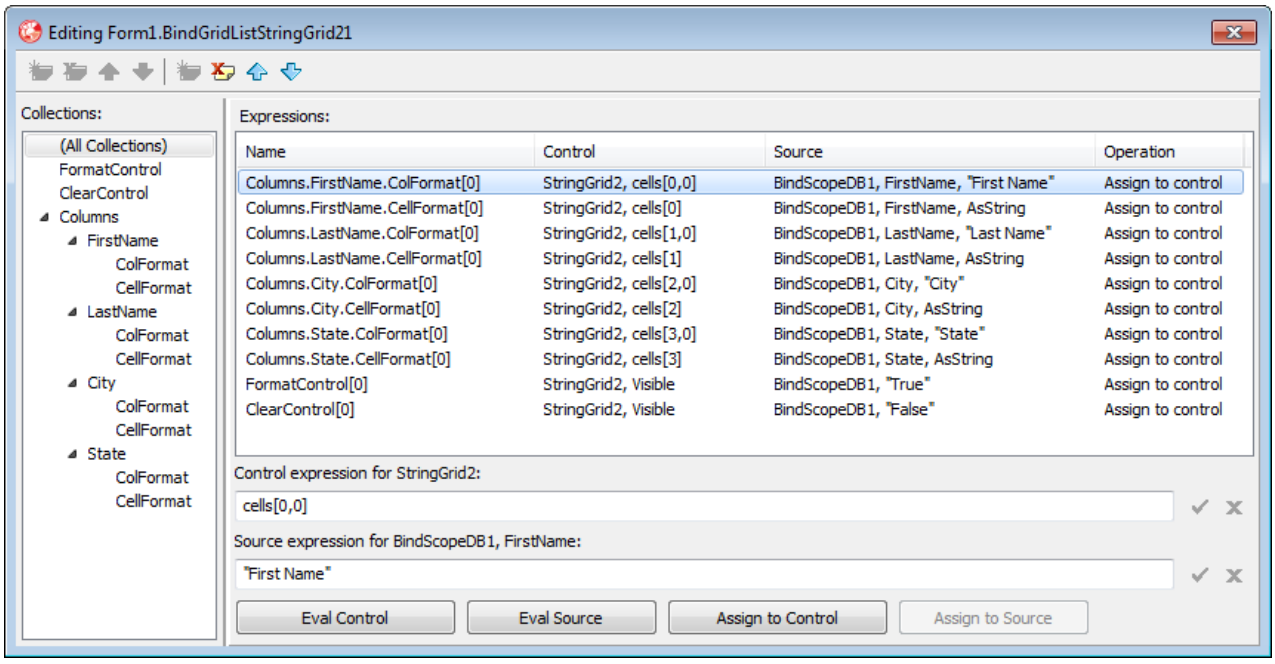

Figure 9. The Expression editor

The Expression editor is a dialog box that you will use often, and there are other ways to display it, other than using the BindingsList component editor. You can also select a component that serves as the ControlComponent of a LiveBinding, expand the LiveBindings property, and then select Expressions... from the dropdown property editor, as shown in Figure 10.

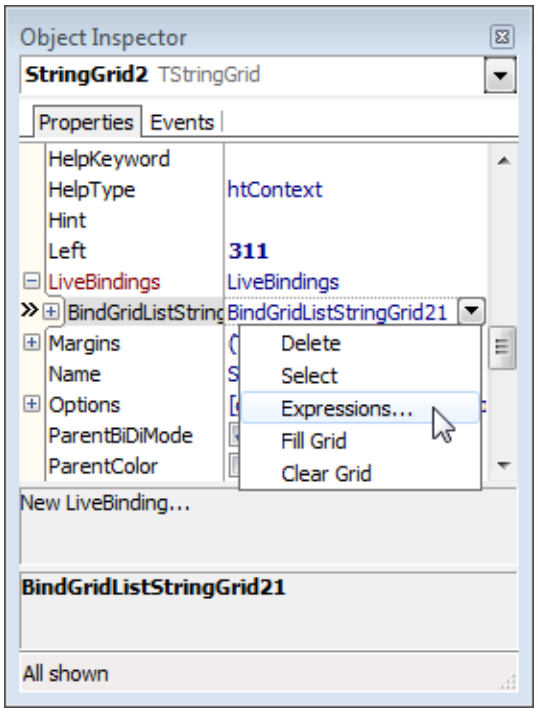

Figure 10. You can access the Expressions editor by selecting Expressions...

There are other ways to view LiveBindings and their expressions, and I will describe some of them as we continue looking at the LiveBindings components. Nonetheless, the BindingsList is certainly one of the most convenient, and the only way to easily view all of the LiveBindings that you have current configured on your form.

In addition to giving you access to your configured LiveBindings, the BindingsList plays another central role in most LiveBindings applications. It exposes a Notify method that you can invoke to notify the expression engine when you are working with managed LiveBindings. The BindingsList Notify method has the same signature as that of the TBindings class, but it is easier to use.

In order to use the TBindings Notify method, you must remember to add the System.Bindings.Helper unit to your uses clause. By comparison, as soon as you've added a LiveBinding to your form, the BindingsList class is added to your form automatically, and as soon as you save or compile your form, the Data.Bind.Components unit is added to your uses clause for you.

There is one more thing that a BindingsList provides you and that is the ability to view, and even disable, individual custom methods and output converters. To view the custom methods that are available to the expression engine, click the ellipsis associated with the

Methods property of a BindingsList in the Object Inspector. RAD Studio responds by displaying the property editor shown in Figure 11.

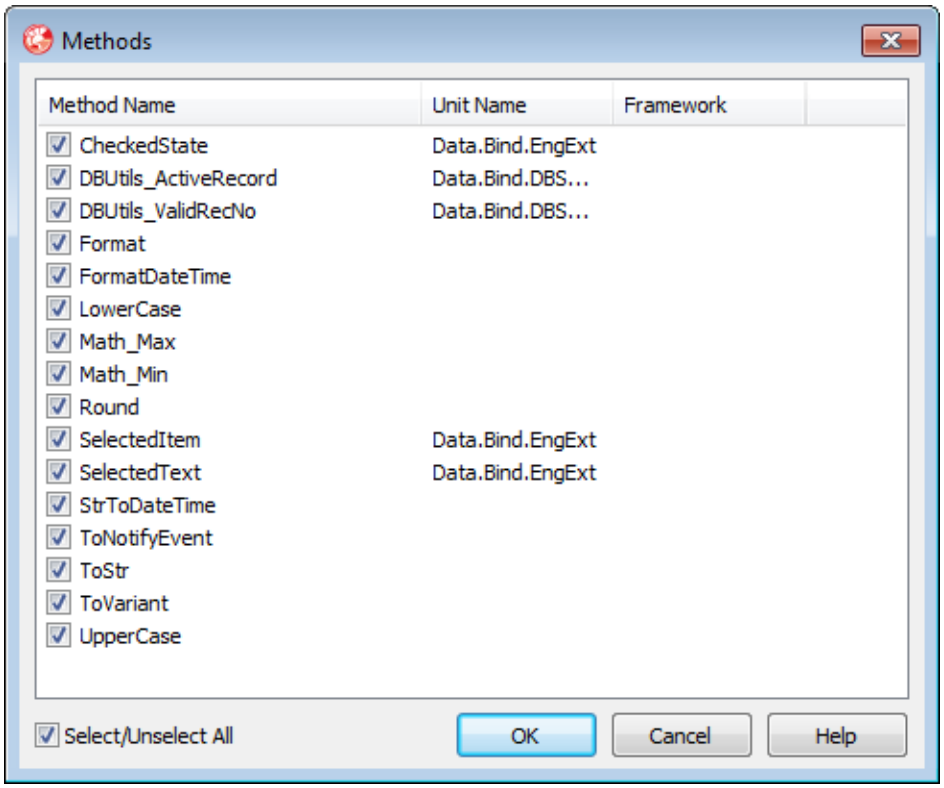

Figure 11. The Methods property editor displays the registered custom methods

Each method displayed in the Methods property editor is associated with a checkbox. If the method is checked, the method can be invoked by the expression engine. If the name of one of these registered custom methods conflicts with a property of an object that you are going to use in an input scope or output scope, you can disable the method by unchecking that checkbox.

To view the registered output converters, click the ellipsis associated with the BindingsList OutputConverters property. The Output Converters property editor is shown in Figure 12.

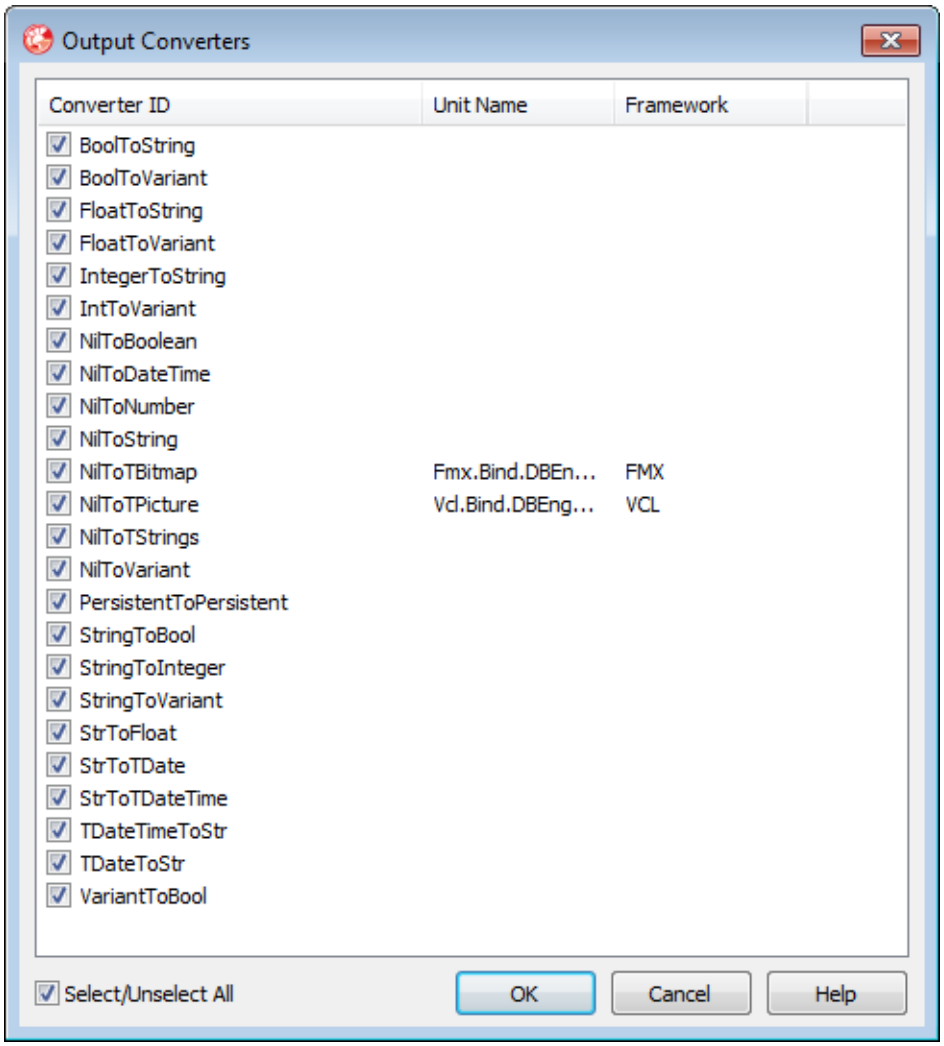

Figure 12. The Output Converters property editor displays the registered output converters.

BINDSCOPE

As I mentioned earlier, the expression engine requires at least one scope in order to work with something interesting. For example, without the input scope, the expression engine could only work with the literals you provide in the expression and the operators that it adds automatically. When you add a scope, you can introduce objects whose properties and methods can be used in expressions.

Two of the components on the Tool Palette are specifically designed to provide a rich scope for the expression engine. One of these, the BindScope component, can be used to wrap objects in a scope that provides extensions beyond what you would get if you added

a wrapped object directly to the scope. For example, BindScope adds support for the IScopeEnumerator interface, permitting an expression to retrieve an IEnumerable member of the wrapped component.

Another feature of the BindScope components (BindScope and BindScopeDB) is that they support LiveBinding activation using the AutoActive and Active properties of the LiveBinding components that link to them. When a BindScope becomes active, it activates all LiveBindings that have used that BindScope as its SourceComponent and which have their AutoActivate property set to True, and deactivates these components when the BindScope is deactivated.

While the BindScope component is limited in capability, the same cannot be said about the BindScopeDB component. BindScopeDB is a specialized type of BindScope component that understands RAD Studio's DataSet interface. Specifically, it knows how to link to a DataSource and it understands the fundamental operations of the DataLink interfaces that make RAD Studio's VCL data-aware components work.

This is an important point, but not necessarily a point to dwell on. Specifically, what BindScopeDB does is permit components in general to interact with RAD Studio's data access mechanism, even when the VCL itself is not present. For instance, the ClientDataSet class, as well as dbExpress, is available to developers who want to target their compilation to the Mac OS (operating system). BindScopeDB permits you to bind your FireMonkey controls to a DataSet, providing you with data awareness in the absence of the VCL data-aware controls.

This might sound trivial, but it is not. LiveBindings extend the already powerful capabilities of RAD Studio's ClientDataSet component and the dbExpress framework to all components. It is essential for those applications that need to target non-Windows platforms. On the other hand, BindScopeDB permits you to add data awareness to VCL components that are not other wise data aware, such as a regular ListBox or ListView.

Bind Expressions

The components that I've talked about so far are components of the LiveBindings framework, but they are not specifically LiveBindings components. In other words, they support LiveBindings but they do not implement LiveBindings. The Bind Expression components, as well as the List and Link components that I discuss in the following sections, are LiveBindings components in that you use them to create LiveBindings at design time.

There are two components that fall under the guise of Bind Expression components; these are BindExpression and BindExprItems. These LiveBindings components, and only these, can be configured as either managed or unmanaged LiveBindings, but are nearly always

used in RAD Studio applications as managed components. As you may recall, this means that you not only configure these LiveBindings, but you also add additional code to instruct the expression engine to evaluate the LiveBinding expressions and assign the resulting value to its target.

BindExpression and BindExprItems are different from each other in only one way. BindExpression components support only one expression while BindExprItems support two or more expressions. Each expression is capable of evaluating one expression (a string), and assigning the value of that expression to a target. This target may be the ComponentExpression (typical), the SourceExpression, or both.

Let's begin by taking a look at the BindExpression defined in the VCLLiveBindings project.

BINDEXPRESSION

Because a BindExpression supports a single expression (assignment), you typically use a BindExpression when linking a single expression to the property of another control. In the VCLLiveBindings project, a BindExpression is used to link the Position property of a TrackBar to the Position property of a ProgressBar.

If you were going to create this LiveBinding from the scratch, assuming that you already have a TrackBar and a ProgressBar on a form, you would use the following steps:

1. Your first step is to determine which component is the ControlComponent, and which is the SourceComponent. Recall that most LiveBindings assign a value to the ControlComponent. In this case, that would make the ProgressBar the ControlComponent (since a user cannot interactively control a ProgressBar from the user interface).

2. Right-click the ProgressBar and select New LiveBindings. The New LiveBinding dialog box is displayed like the one you saw back in Figure 1. Select BindExpression.

3. The new BindExpression is selected in the Object Inspector. Set SourceComponent to TrackBar1, SourceExpression to Position, and ControlExpression to Position (which is the ProgressBar's Position). You will also want to set the Max property of the TrackBar1, which will make it consistent with the Max property of the ProgressBar. This has nothing to do with LiveBindings, but it will make the resulting effect more impressive.

4. The LiveBinding has been configured, but this being a managed LiveBinding, it is necessary to instruct the expression engine to evaluate the expression. To do this, double-click the TrackBar to create a OnChange event handler. Change the generate event stub to look like the following:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
```

```
  BindingsList1.Notify(TrackBar1, 'Position');
end;
```

5. The LiveBinding is complete. Run your project. When you change the position of the TrackBar, the ProgressBar adjusts to match the TrackBar, as shown in Figure 13.

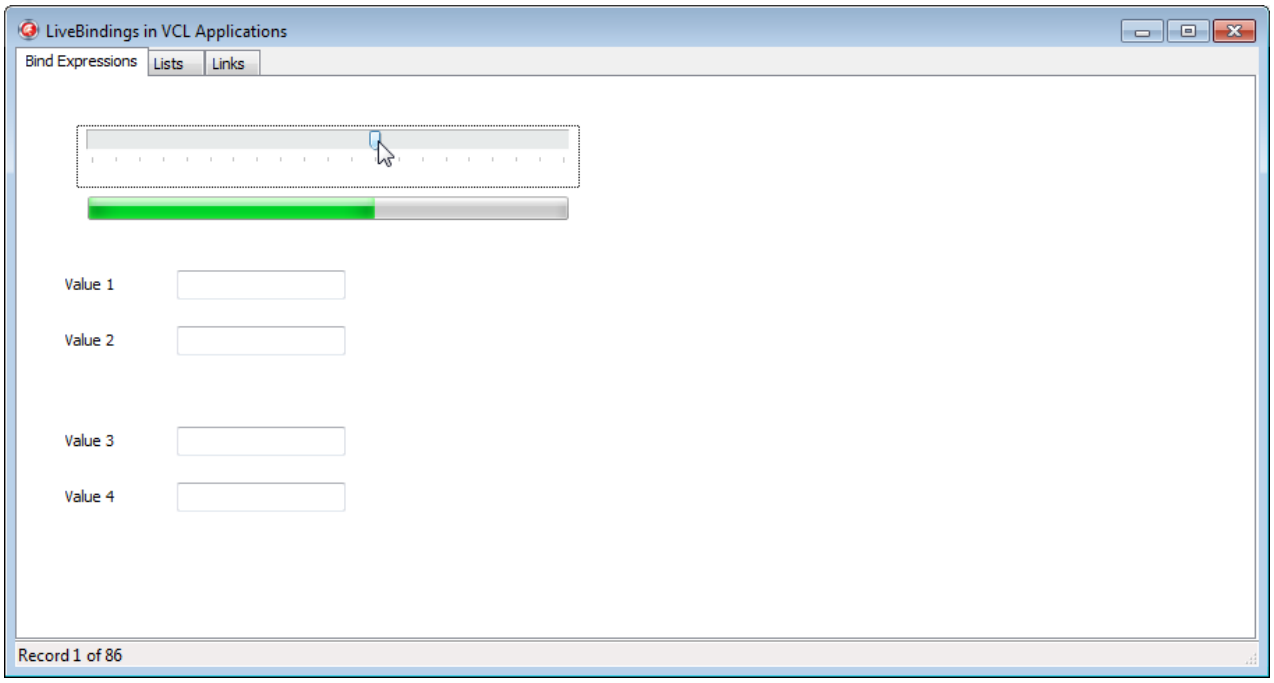

Figure 13. A BindExpression keeps a ProgressBar synchronized with a TrackBar

To see that the BindExpression LiveBinding supports only a single expression, all you need to do is the view the Expression editor for the LiveBinding. As mentioned earlier, you can do this from the BindingsList component that appears on your form after you have created your first LiveBinding at design time.

There are several additional ways to display this editor. For example, after creating LiveBindings, you can use the Object Selector of the Object Inspector (that's the combobox that appears at the top of the Object Inspector) to select any of your defined LiveBindings. When a LiveBinding is selected, links appear at the bottom of the Object Inspector, including one labeled Expressions..., as shown in Figure 14.

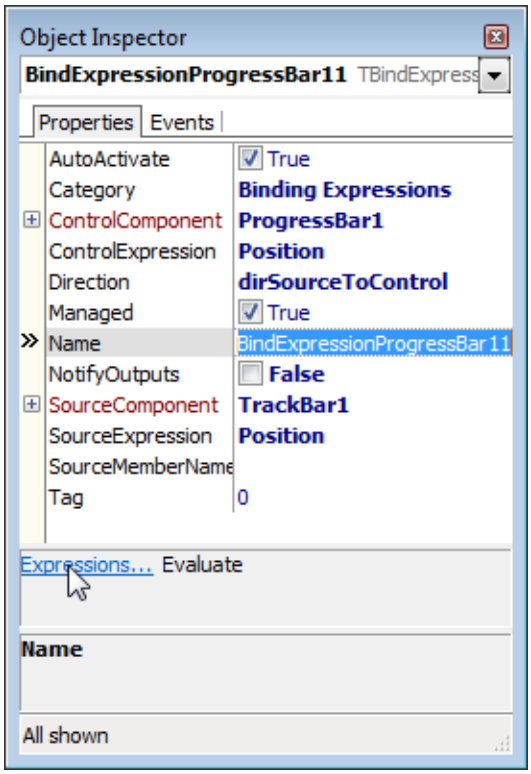

Figure 14. Use the Expressions... link that appears in the Object Inspector when a LiveBinding is selected to display the Expression editor

Click that link to display the Expression editor. The Expression editor shown in Figure 15 is the one that appears when you have a BindExpression selected in the Object Inspector. Notice that the Expression editor Tool bar is grayed out, it having been disabled. Furthermore, there is no Collections pane like the one that appears on the left in Figure 15. Again, this is because BindExpressions support one, and only one, expression.

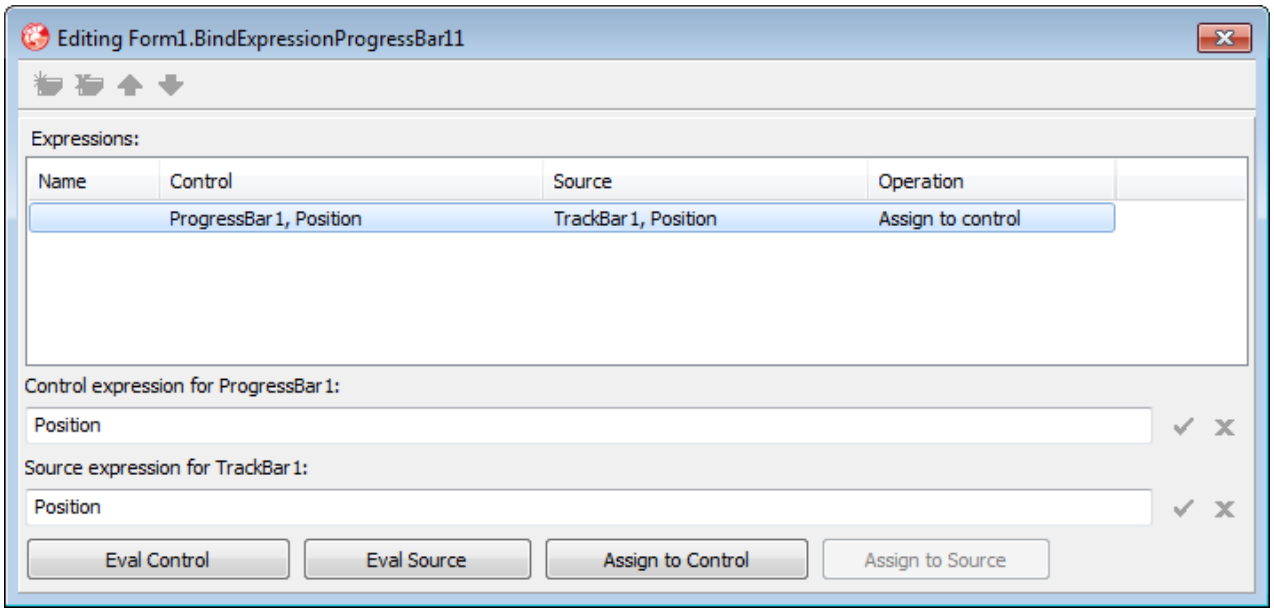

Figure 15. The Expression editor for a BindExpression can contain only one expression

The VCLLiveBindings project includes an additional BindExpression LiveBinding. This LiveBinding is special in that it represent a new way to think about implementing behavior in your applications, and highlights how LiveBindings are different from other mechanisms available in RAD Studio. As a result, I will save the discussion for this BindExpression LiveBinding for later in this paper, in the section titled *The Future of LiveBindings*.

When you need a managed LiveBinding to evaluate two or more expressions, you want to use the BindExprItems LiveBinding discussed next.

BINDEXPRITEMS

The VCLLiveBindings project includes a BindExprItems that includes two expressions. In this particular LiveBinding, one expression is responsible for keeping the Edit named Value1Edit synchronized with the contents of the Edit named Value2Edit. The second expression keeps the Edit labeled Value 2 synchronized with the Edit labeled Value 1. Figure 16 depicts the Expression editor when this LiveBinding, named BindExprItemsEdit21, is selected.

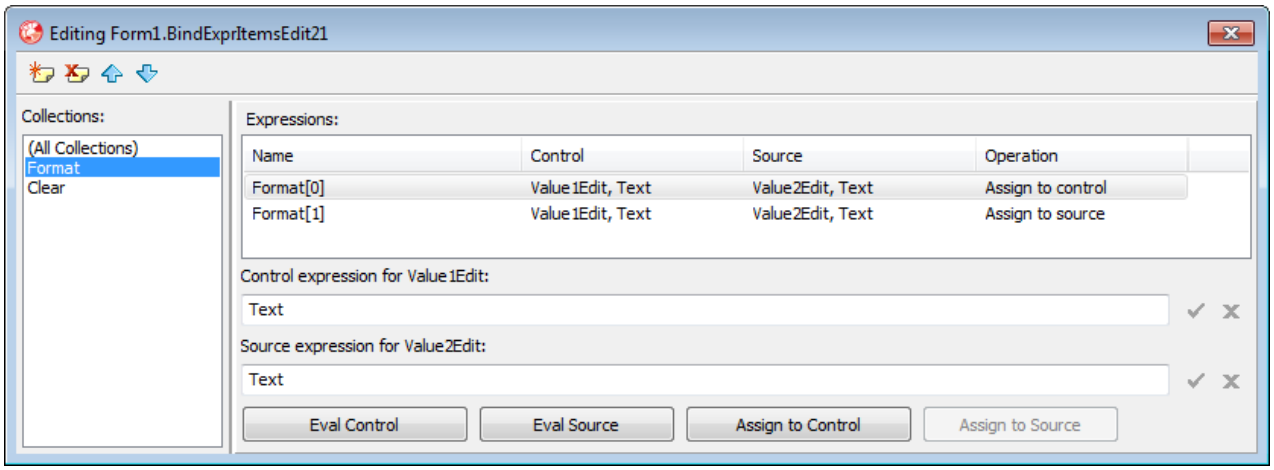

Figure 16. With the exception of BindExpression LiveBindings, all other LiveBindings support more than one expression

Figure 16 begins to reveal the richness of LiveBindings. In the Collections pane, notice that there are two types of collections: Format and Clear. Format collections are expressions that assign a value, based on the Control and Source components and expressions, as well as the direction of the assignment when the LiveBinding is active. By comparison, Clear expressions are used to perform assignments based on the expressions when the LiveBinding is being deactivated.

Format expressions tend to do most of the heavy lifting for LiveBindings as most of the LiveBindings are configured with AutoActivate set to True, meaning that they become active upon creation. But for those LiveBindings where activation comes and goes, such as those whose SourceComponent is a BindScope, the Clear expressions can reset the associated components based on expressions being evaluated by the expression engine. Clear expressions tend to be more useful when you are using LiveBindings that are associated with DataSets that might change from active to inactive and back. Nonetheless, if you need this capability, you will be grateful that BindExprItem LiveBindings supports these collections.

Turning our attention back to the BindExprItemsEdit21 BindExprItems LiveBinding, I do have a confession about that example. Two expressions were not necessary in this case; one expression would have sufficed. This is demonstrated in the BindExprItemsValue3Edit1 BindExprItem LiveBinding, which also appears in the VCL LiveBindings Project.

This LiveBinding binds the Edits associated with labels Value 3 and Value 4. Here, a single expression performs the task performed by the two expressions associated with the BindExprItemsEdit21 BindExprItem LiveBinding. This expression is a bidirectional

expression. When the expression engine is notified, changes to the Text property of Value3Edit are assigned to the Text property of Edit Value4Edit, and visa versa. The single expression associated with this LiveBinding is shown in the Expression editor in Figure 17.

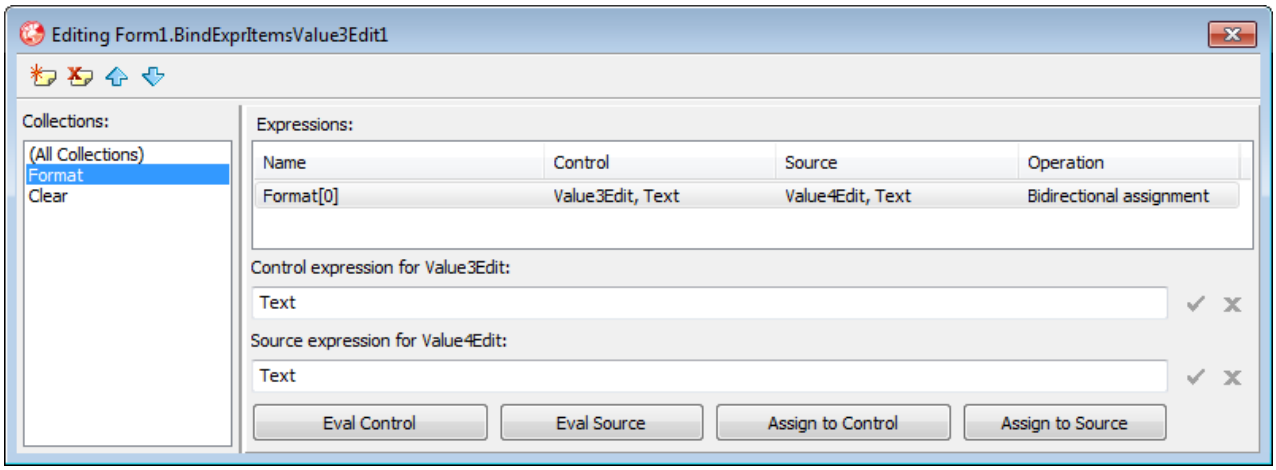

Figure 17. A bidirectional expression is shown in the Expression editor

Not all BindExprItems expressions can be bidirectional. For example, both the ControlExpression and the SourceExpression must be read/write properties of their respective objects. If your SourceExpression includes literals and operators, there is no conceivable way for the expression engine to assign a value bidirectionally (dirBidirectional), or even control to source (dirControlToSource). Likewise, if one of the expressions includes a readonly property, there can be no assignment to that expression.

LIST LIVEBINDINGS

There is a lot you can do with Bind Expressions, but things start to get more interesting when you move to the more sophisticated LiveBindings. In this section, I will consider the List LiveBindings. There are two List LiveBindings, BindList and BindGridList.

As their names suggest, Lists LiveBindings are used for assigning the values of expressions to lists and grids. These bindings are specifically unidirectional, in that changes to the SourceExpression of the SourceComponent can produce changes in the ControlExpression of the ControlComponent, but the LiveBindings will not affect the SourceExpression of the SourceComponent.

List LiveBindings share some commonalities with BindExprItems. Specifically, they support expression collections, permitting you to define many different expressions that can be used to assign values to a number of different properties of the ControlComponent. For example, both BindExprItems and List LiveBindings support both the Format and Clear expression collections.

List LiveBindings go a step further in that they are designed to work with the list classes that form the foundation of list and grid controls, such as ListBox, ComboBox, ListView, and StringGrid classes. They do this by exposing expression collections that can be configured to assign the values of expressions to the inner members of their classes.

Another aspect of List LiveBindings that is different from the Bind Expression LiveBindings is that they often employ a BindScope, and in most applications, a BindScopeDB. BindScope components can be used with List LiveBindings to bind data to an Object that supports lists or grid-list structures. While this is possible, at least in this release of RAD Studio, you will almost always be using the List LiveBindings with a BindScopeDB, permitting your list and grid objects to display data from some underlying DataSet. On the other hand, when you want to fill a list with an enumerable object like a generic TList<T>, you would use a TBindScope and set the DataObject property at runtime.

The VCLLiveBindings project includes examples of both a BindList and a BindGridList LiveBindings. Let's begin by considering the BindList component.

BINDLIST

As I already mentioned, most uses of List LiveBindings employ a BindScopeDB and a DataSet, and the VCLLiveBindings project is no exception. The BindScopeDB and DataSet used in these examples are also employed in the demonstration of BindLink, which I will discuss in detail in the following section. Consequently, I will take a moment to discuss how these data-related components are configured.

A BindScopeDB plays a role not dissimilar to that of a DataSource, though it is not a DataSource replacement. In fact, a BindScopeDB does not connect directly to a DataSet. Instead, it connects to a DataSource, which in turn connects to a DataSet.

The VCLLiveBindings project uses a ClientDataSet as its DataSet, and a sample ClientDataSet *.cds file as the source of its data. The sample cds file used in this project is called sampledata.cds, and it is located in the same directory in which the executable will be compiled. The following code, found on the OnCreate event handler of the main form of this project, sets the FileName property of the ClientDataSet (the ClientDataSet is not opened here. It is opened by a LiveBinding):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ClientDataSet1.Close;
  LogChanges := False;
  ClientDataSet1.FileName :=
    ExtractFilePath(Application.ExeName) + 'sampledata.cds';
end;
```

Authors note: If we set the FileName property of the ClientDataSet, we could make the ClientDataSet active at design time. This would let us also use a feature of LiveBindings

that permits us to fill lists and grids at design time. While this technique is useful, it causes the fully qualified name of the cds file to be stored in the FileName property. I left this property empty, and set it at runtime, which permitted me to calculate the directory in which the cds file is stored, based on the location of the executable. Doing this helps ensures that you will have minimal problems trying to run the example applications.

Let's now turn our attending to the BindList that is used on this form. This BindList populates a ListBox based on the contents of the ClientDataSet. The results of this binding are shown in Figure 18, which displays the results of an expression evaluation in each element of the ListBox.

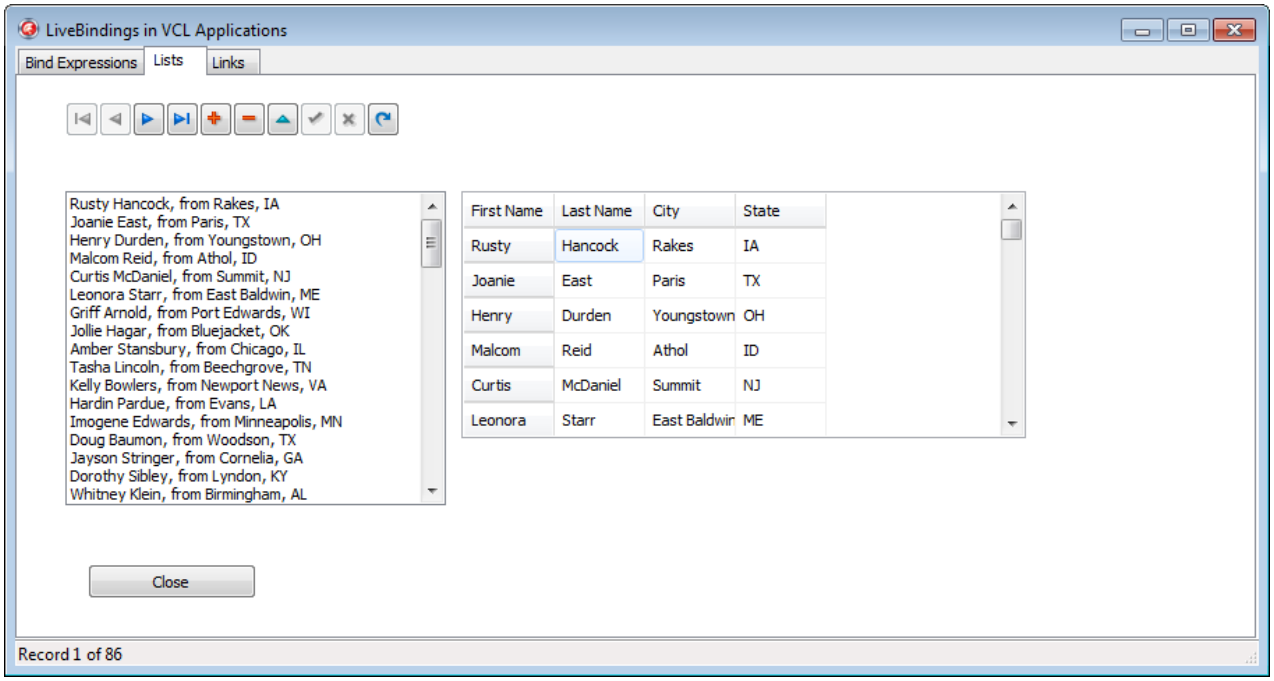

Figure 18. A ListBox and a StringGrid are populated by a BindList and a BindGridList, respectively

How the BindList is configured can be seen in the Expression editor shown in Figure 19. Here you can see three expression collections. Like the BindExprItems, there is a Format collection. Unlike the BindExprItems, however, this binding applied to a property of the ControlComponent, the Strings property, specifically.

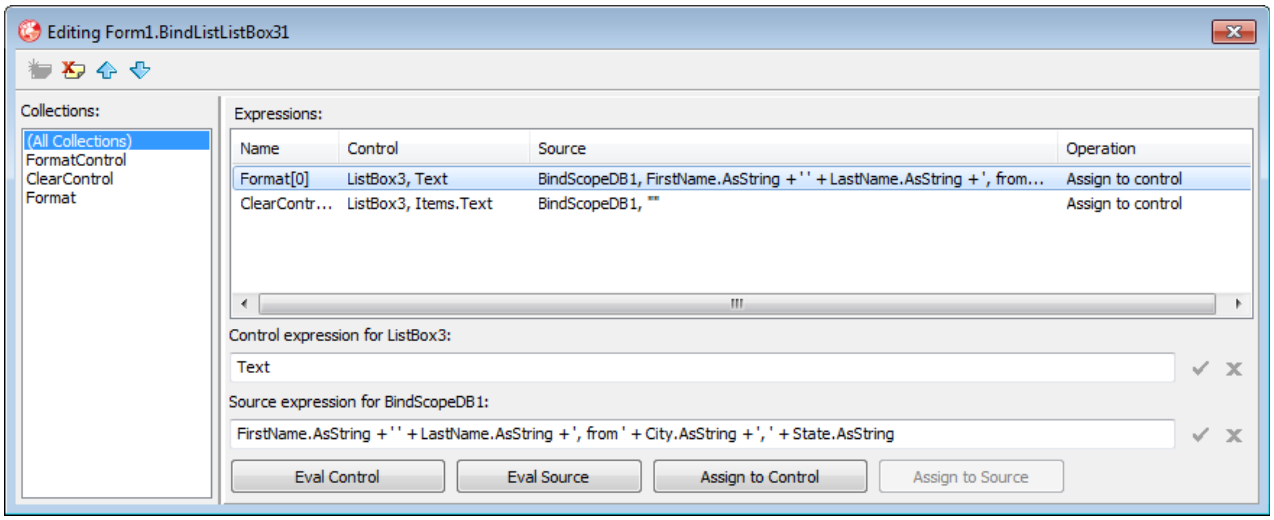

Figure 19. A BindList supports three expression collections

The other two collections are FormatControl and ClearControl, and they apply to the ListBox itself. FormatControl is used to apply its expressions when the LiveBinding is active, and the ClearControl expressions are applied when the LiveBinding deactivates. In the case of the BindListListBox31 LiveBinding, there is a single Clear expression, which serves to empty the ListBox when the LiveBinding is being deactivated. The SourceExpression is an empty string ('' '') and the ControlExpression is Items.Text. In other words, when the DataSet is closed, the ListBox will be cleared of all items.

There is a third collection here, FormatControl. FormatControl can hold a collection of expressions that are applied to the ControlComponent itself rather than the ControlComponent's List. This particular ListBox does not use FormatControl expressions, but examples of FormatControl expressions are covered a little later in this paper.

Most of the work of the BindListListBox31 LiveBinding is performed by a single Format expression. As shown in Figure 19, the SourceExpression concatenates the FirstName, LastName, City, and State fields with literal strings to form the descriptive text that appears in the ListBox in Figure 18.

This expression reveals some of the power of the BindScopeDB. The BindScopeDB makes the individual fields of the DataSet to which it points (albeit through a DataSource) visible in the scope associated with the SourceComponent. As a result, the DataSet's fields can be used in the expressions almost effortlessly. This one Format expression is shown in Figure 19.

This is a simple BindList, in that it contains a total of two expressions. It is conceivable that a BindList could have many expressions, each applying the results of the expression

evaluation to various properties of the ListBox. But practically speaking, most BindLists are really concerned with populating and clearing the contents of a List-oriented control. As a result, this LiveBinding is typical.

BindGridList

When it comes to a BindGridList, however, things are different. StringGrid, and other grid-related controls, are much more complex, and as a result, the LiveBindings that you configure for them typically have many expressions. The StringGrid shown in Figure 18, and its associated LiveBinding, is a good example of this.

The BindGridListStringGrid21 has a total of 10 expressions associated with it as shown in Figure 20. Like the BindList LiveBinding, a FormatControl collection supports expressions that apply to the grid when the LiveBinding is active, and a ClearControl collection that is used to clear the contents of the grid.

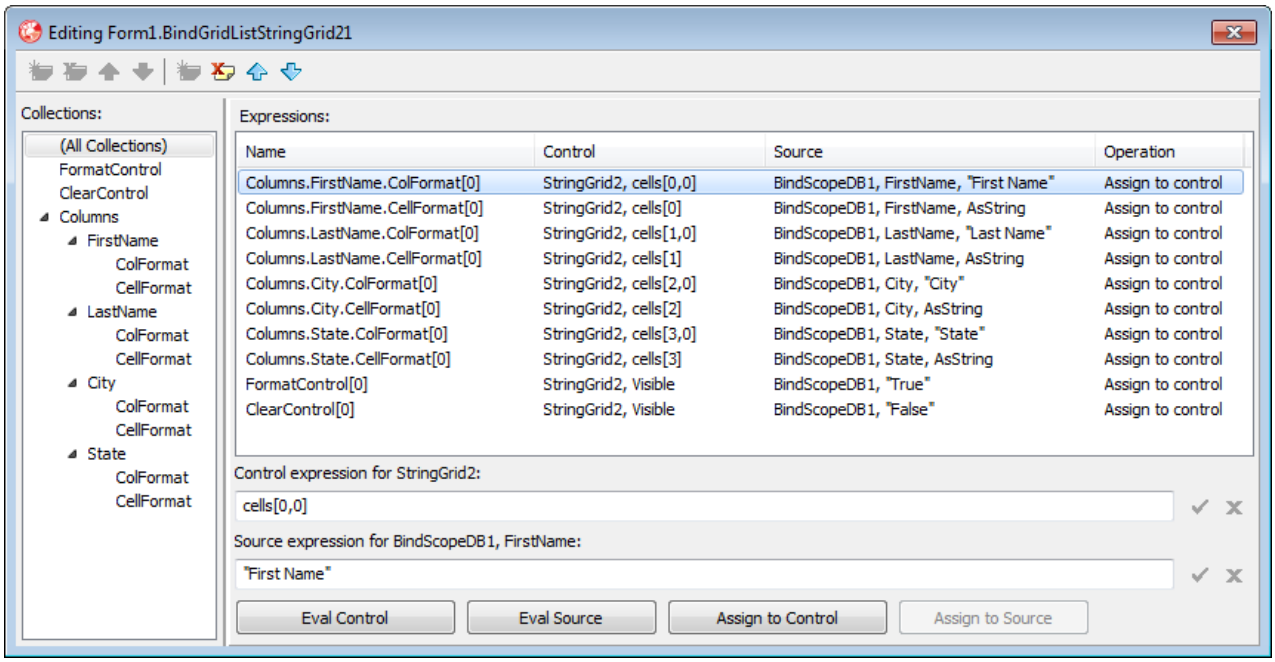

Figure 20. The expressions of a BindGridList LiveBinding in the Expression editor

The third collection, Columns, is actually a collection of expression collections. Well, to be accurate, it is a collection of collections of collections. You typically will add one Column collection for each column of the grid you want to work with. Each column has both a ColFormat and a CellFormat collection, each of which can support zero or more expressions. ColFormat expressions are normally used to apply expressions to individual columns, as a whole. CellFormat expressions, by comparison, are used to apply expressions to the individual cells of the associated column.

Turning our attention back to the Column collection, here is where we finally meet the SourceMemberName property. With a BindGridList, the SourceMemberName associates a specific column of the grid with a field name in the DataSet being referenced by the BindScopeDB. For the BindGridListStringList21 LiveBinding, the three Columns are associated with the FirstName, LastName, and City fields of the ClientDataSet, and these values appear in the SourceMemberName property for each of the Column collections, respectively.

Now that you have seen a number of LiveBinding expressions, you are probably getting pretty good at reading the contents of the Expression editor. Taking a look at the expressions in Figure 20, you can see that there are two FormatControl expressions, one that makes the grid visible when the DataSet is activated, and another that set the number of columns. There is also a ClearControl expression. This one makes the grid invisible when the DataSet is inactive.

There are four Columns in the Column collection, and each one has at least one expression each in its ColFormat and CellFormat collections. The ColFormat expression sets the text of the column header, and the CellFormat assigns the current cell with the data associated with the SourceMemberName field in the DataSet. As you can see back in Figure 18, the effect is similar to what you might find with a readonly DBGrid.

The size of each column in this StringGrid is the default size of a column in a string grid, 64. Let's change that, which will give you the opportunity to use the Expression editor. Use the following steps to perform this task.

1. Begin by selecting the ColFormat collection associated with the State column in the Collections pane of the Expression editor. Next, use the Add button in the Expression editor's toolbar (or press Ins) to insert a new ColFormat expression.

2. In the Control Expression field, which is located in the lower portion of the right pane, enter ColWidths[3], which refers to the width of the fourth (zero-based) column of the StringGrid. Apply this expression by click the checkmark to the right of the Control Expression field.

3. Next, enter Size * 2 in the Source Expression field. Since this fourth column is associated with the SourceMemberName value of State, this expression is referencing the Size property of the TField associated with the State field of the ClientDataSet. In short, we are instructing the LiveBinding to set the width of the State column based on the maximum number of characters that can appear in the State field. Click the checkmark to the right of the Source Expression field to apply this change. Your Expression editor should now look something like that shown in Figure 21.

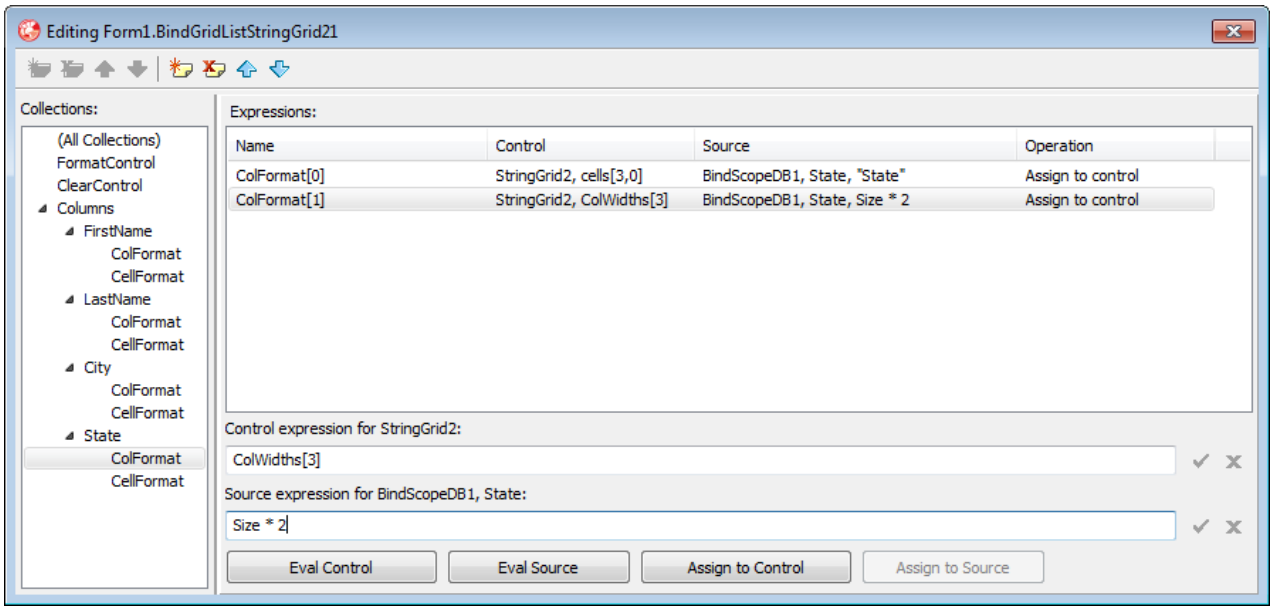

Figure 21. A new ColFormat expression sets the width of the State column of the StringGrid based on the size of the State field in the ClientDataSet

4. Run this project and view the TBindList and TBindGrid tab, and you should see something like that shown in Figure 22. Notice that the State field is now narrower than the other three fields.

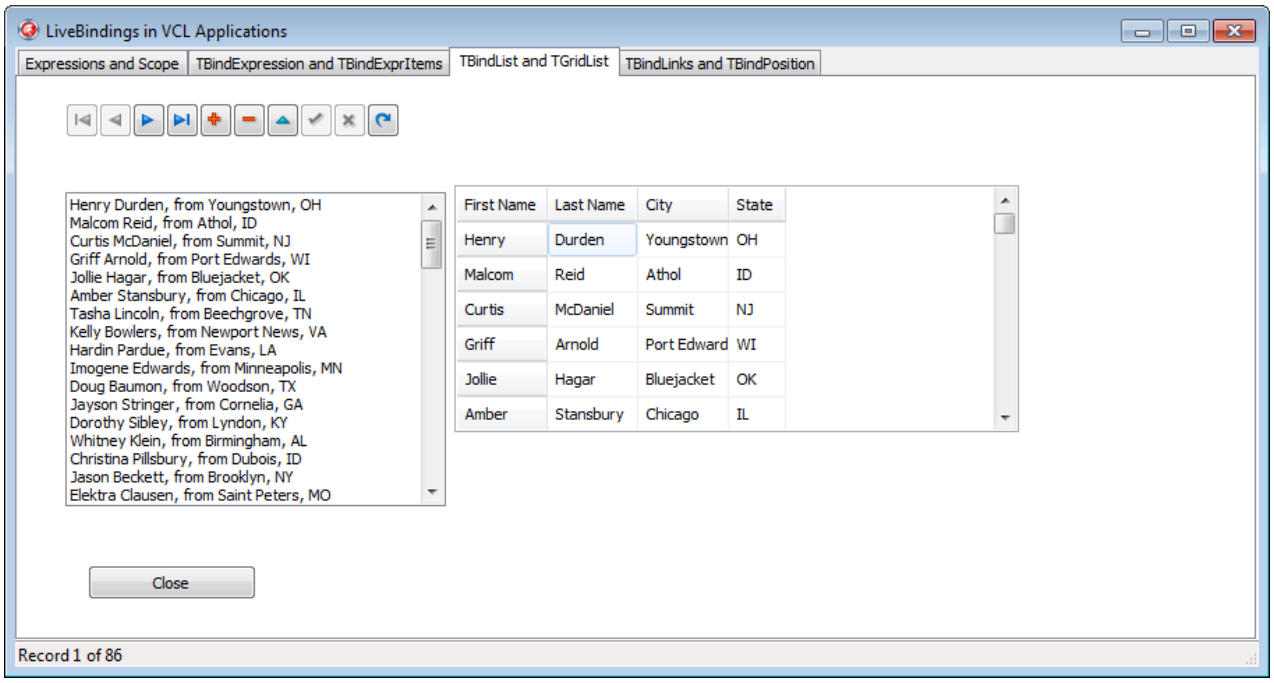

Figure 22. The width of the State column is now based on the size, in characters, of State field

We could go on and on, creating additional expressions, each which can control an elements of the grid. However, as I will point out towards the end of this paper, LiveBindings, at least in this version, are somewhat slow. As a result, we want to use LiveBindings where their dynamic nature gives us an advantage, and not use them for properties that could just as easily be configured at design time.

Since we've digressed for the moment to talk about using the Expression editor, let me make a couple of additional points. If you want to add more columns collection to the LiveBinding, you can select the Columns Node in the Collections pane and click the Insert button (or press Ins). To delete an expression or a column, select the expression or column and click the Delete button in the tool bar (or press Del).

You can also use the buttons below the Source Expression field to evaluate your expressions, or to force the evaluation of the expressions and assign the result to the target of the expression (which as I mentioned earlier, is always Source to Control in a List LiveBinding). However, you can only do this successfully for expressions that make sense at design time. In this project, the ClientDataSet is inactive at design time so it doesn't make sense to evaluate the new Source expression that you just created for the State field. However, the StringGrid does have four columns, so you can evaluate the ColWidths[3] expression.

To do this, click the Eval Control button. The Expression editor will display the dialog shown in Figure 23. This dialog is telling us that it is evaluating the ColWidths[3] expression of the StringGrid2. If String is selected in the View As list, you will see that it displays 64, which is the default width of a column in a StringGrid. If you set View As to Type, it will report that the expression evaluates to an integer.

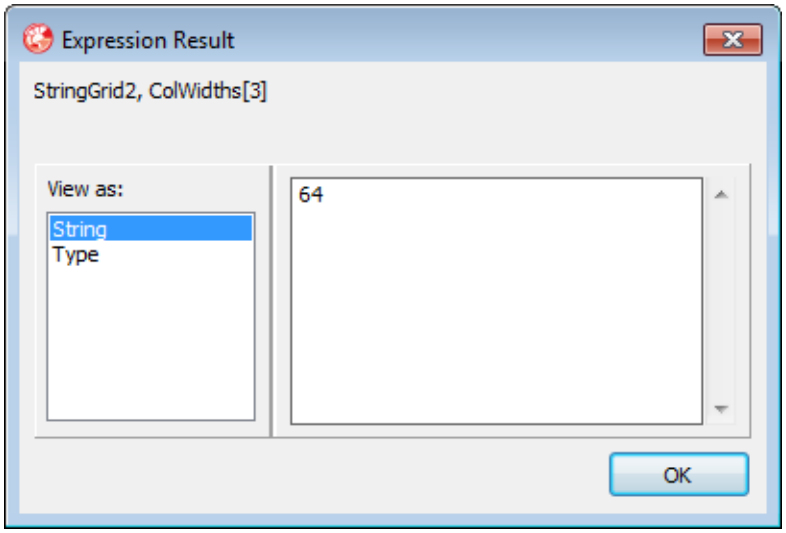

Figure 23. Depending on the state of your ControlComponent and SourceComponent, you may be able to evaluate their associated expressions at design time

Let's now turn our attention to the most capable of unmanaged LiveBindings, Link LiveBindings.

LINK LIVEBINDINGS

There are four Link LiveBindings, and they provide read/write capability. Like the List LiveBindings, Link LiveBindings are typically used with BindScopeDB components implementing data awareness in controls that otherwise would not have that capability.

Two of the Link LiveBindings share a great deal in common with the two List LiveBindings. Consequently, I will begin by discussing the two Link LiveBindings that are not shared between the two types. These are the BindLink and BindPosition LiveBindings.

BINDLINK

BindLink LiveBindings are used to provide read/write binding between a control that supports a single field and another component, typically a BindScopeDB. More specifically BindLinks can bind a control that implements either the IEditLinkObserver or

IEditGridLinkObserver interface to a BindScopeDB. The VCLLiveBindings project includes two BindLinks.

The first BindLink is associated with a control that cannot be edited by the end user; it is associated with a Label component. This LiveBinding displays the full name of the person associated with the current record. Because the expression associated with this LiveBinding needs to be able to refer to more than one property of the BindScopeDB (two of the fields of the ClientDataSet), this LiveBinding does not employ the SourceMemberName property.

If you view this BindLink in the Expression editor, you can see that this LiveBinding supports three collections: Format, Parse, and Clear. The Format and Clear collections are used for the same purpose as those collections found in the BindExprItems LiveBindings. You add one or more Format collections to assign expressions to the ControlComponent when the LiveBinding is active, and Clear to assign one or more expressions when the LiveBinding is being deactivated. I have added one Format expression and one Clear expression. The Format expression combines the FirstName and LastName fields of the ClientDataSet and assigns the resulting full name to the Text property of the Edit. The clear expression assigns an empty string to the Text property.

As you might be able to predict by now, a Format expression of this LiveBinding assigns an expression that concatenates the FirstName and LastName fields of the ClientDataSet with a string literal (a space) to form the full name, which is assigned to the Caption property of the Label. The single Clear expression assigns an empty string to the Label's caption.

Let's now turn our attention to the BindLink LiveBindings that can write back to the ClientDataSet (through the BindScopeDB). The first is a classic ControlComponent for a BindLink LiveBinding, a single field editable control, which is an Edit in this example.

Like the LiveBinding associated with the StatusBar and the Label, a Format expression retrieves the value of a field exposed by the BindScopeDB, the LastName field in this case, and the Clear expression sets the Edit's Text property to an empty string.

Writing back to the ClientDataSet is performed by the expressions in the Parse collection, and these expressions use a control to source direction. Format and Clear expressions are exclusively source to control in direction. Interestingly, the Parse expression used by this LiveBinding is pretty straightforward. In fact, it's exactly the same as the Format expression with the exception that the direction of the expression is reversed, as you can see in Figure 24.

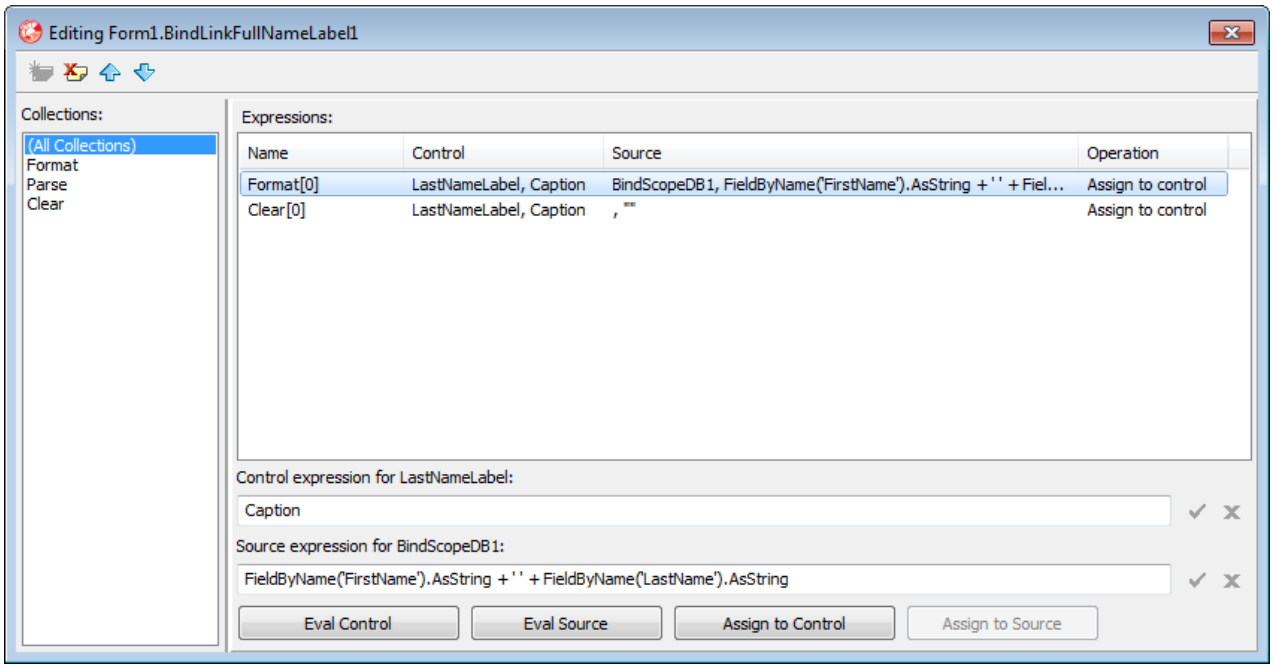

Figure 24. The expressions associated with the BindLinkLastNameEdit LiveBinding

BINDPOSITION

The next LiveBinding we are going to look at is the BindPosition LiveBinding. This LiveBinding is the most limited of the unmanaged LiveBindings in the sense that it is designed to work with ControlComponents that reflect and affect relative position. In fact, the BindPosition LiveBinding can be used with any component that implements the IPositionObserver interface. The ScrollBar component is an example of this kind a ControlComponent.

The BindPosition LiveBinding has three expression collections: PosControl, PosClear, and PosSource. You create PosControl expressions to affect the ControlComponent when the LiveBinding is active. An obvious task that you might use a PosControl expression for is to set the relative position of the ControlComponent based on some ordinal property of the SourceComponent. Similarly, you use the PosClear expressions to set the ControlComponent to a neutral condition when the LiveBinding is deactivated.

PosSource, like PosControl, is used to change the position of the target component. Like the Parse expression collection found in the BindLink LiveBindings, the direction of these expressions are control to source, giving your user a chance to affect the underlying source control by interacting with the ControlComponent.

The VCLLiveBindings project includes two BindPosition LiveBindings. Let's start by considering the BindPosition as it is used with the ScrollBar. Figure 25 shows the Expression editor for this LiveBinding.

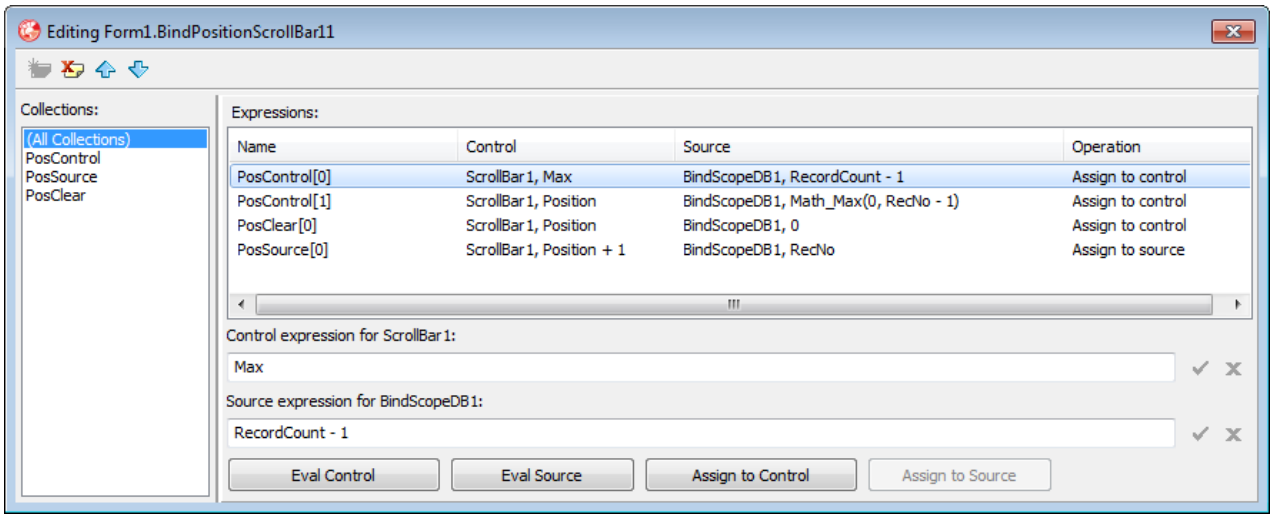

Figure 25. The expressions associated with a BindPosition LiveBinding in the Expression editor

As you can see, there are two PosControl expressions. One of these, the first expression, sets the Max property of the ScrollBar, identifying what value, base on the number of records in the ClientDataSet, represents the extreme right position of the ScrollBar. The second expression, also a PosControl expression, sets the Position property to either 0 (which is the default value for the ScrollBar's Min property) or the current record number (minus 1). These two expressions have the effect of keeping the position of the ScrollBar's thumb tab synchronized with the current record position in the ClientDataSet.

The PosSource expressions are used to permit the BindPosition LiveBinding to change the SourceComponent. As you can see in Figure 25, changes made to the Position property will attempt to change the RecNo property of the DataSet, assuming that there is nothing preventing the ClientDataSet from navigating to a new record, such as a primary key violation, a BeforePost event handler, or some other restriction.

Figure 26 shows the Link tab of the VCLLiveBindings project. Notice that the relative position of the ScrollBar matches the text that appears in the StatusBar. It also matches the relative position of records in the ListBox and the StringGrid, which provides us with a perfect opportunity to consider the BinkListLink and BindGridLink LiveBindings.

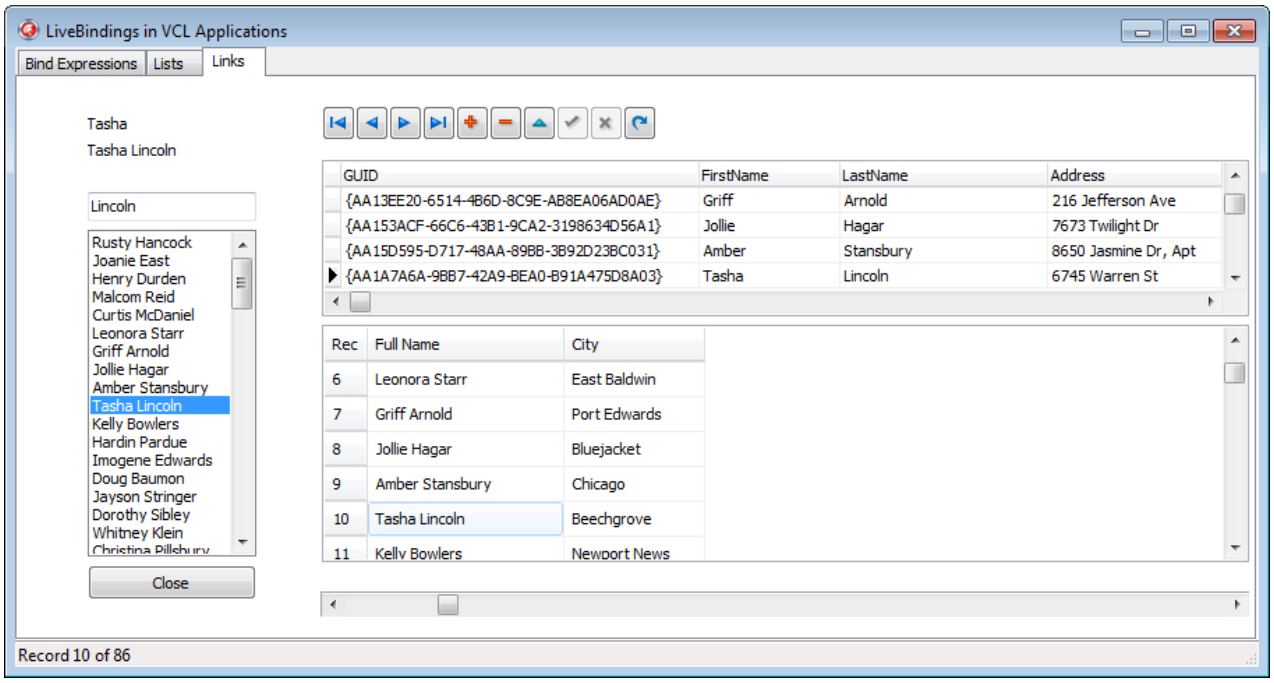

Figure 26. BindPosition LiveBindings keep the ScrollBar and StatusBar synchronized with the ClientDataSet

The second control that uses a BindPosition LiveBinding is the StatusBar. This project uses the StatusBar to identify to which record the ClientDataSet is pointing. A BindPosition LiveBinding is ideal for this task, though we only need the PosControl and PosClear expressions since the user cannot interact with the StatusBar.

The PosControl expression uses SimpleText as the ControlExpression. (This requires that we set the StatusBar's SimplePanel property to True.) The SourceExpression is set to the following string:

```
'Record ' + ToStr(RecNo) + ' of ' + ToStr(RecordCount)
```

The single Clear expression also uses SimpleText as the ControlExpression, and an empty string as the SourceExpression. The effect of these two expressions is to display the ClientDataSet's record position, relative to its total number of records when the ClientDataSet is active, and to display no text when the ClientDataSet is closed.

As mentioned earlier, the BindListLink and BindGridLink LiveBindings share a lot in common with the BindList and BindGrid LiveBindings. The major difference is that these LiveBindings, like the other LiveBindings in the Link category, support control to source expression collections. And because both the BindListLink and BindGridLink LiveBindings refer to multiple records, these collections can affect both fields of the SourceComponent as well as the position of the current record within the Source component.

BindListLink

Figure 27 shows the LiveBinding associated with the ListBox on the Links tab of the VCLLiveBindings project. Like the List LiveBinding associated with the ListBox on the Lists tab of the VCLLiveBindings main form, this BindListLink includes Format expressions that set the values of the ListBox items based on the SourceComponent, as well as a Clear expression that removes items from the ListBox when the ClientDataSet is closed. But also note that it also includes a PosControl expression that keeps the current item in the ListBox synchronized with the current record position in the ClientDataSet, as well as a PosSource expression that has the effect of changing the current record in the ClientDataSet, if possible, in response to the user's interaction within the items in the ListBox.

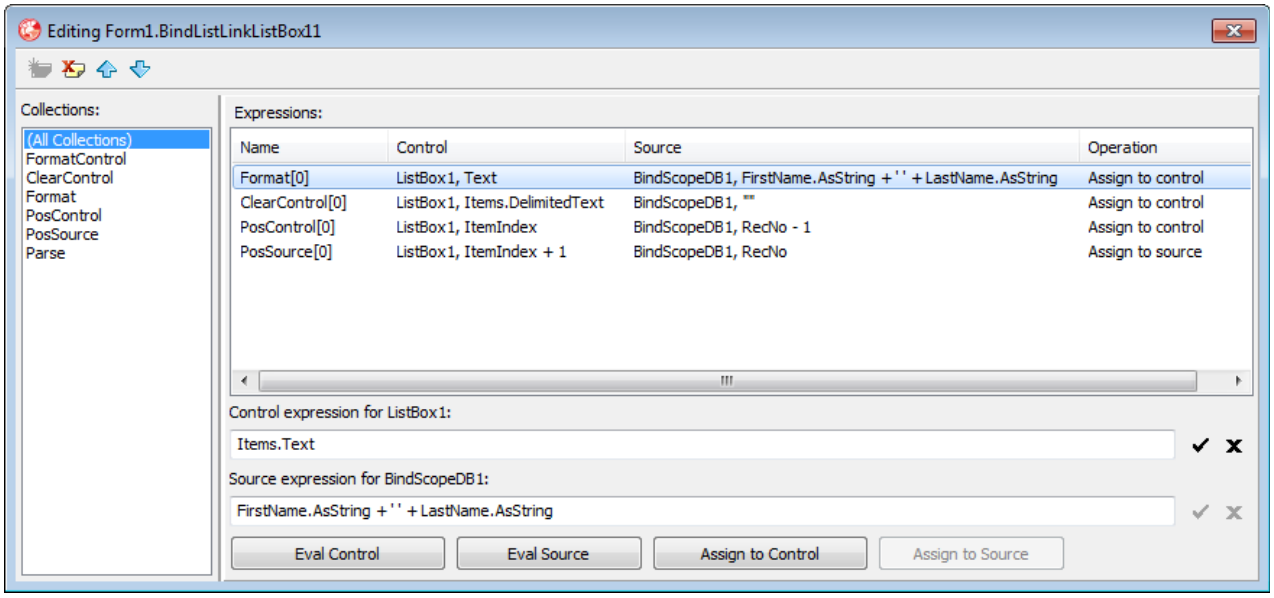

Figure 27. The expressions associated with a BindListLink LiveBinding in the VCLLiveBindings project

BindGridLink

The BindGridLink LiveBinding is the most capable of all of the LiveBindings in RAD Studio XE2, in that it has the capabilities of all of the unmanaged LiveBindings discussed so far. It supports expressions that can affect the grid as a whole, individual grid columns, individual grid cells, and the position of the current record in the grid as well as the source DataSet. In addition, it supports CellParse collections that can be used to write data from individual cells of the grid back to the source DataSet.

Consider the Expression editor for the BindGridLinkStringGrid11 LiveBinding shown in Figure 28. Clearly, there is a lot going on here. However, now that we've had a chance to look at a lot of different expressions, this should be pretty easy to read.

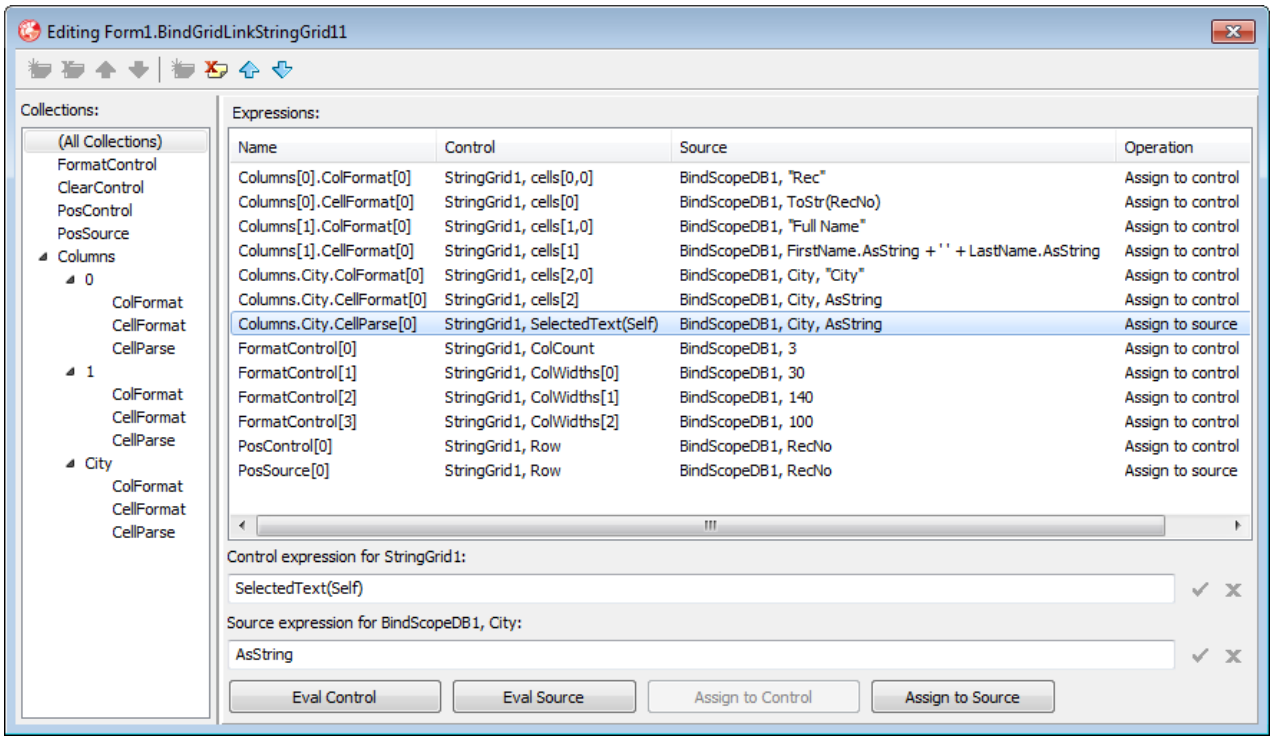

Figure 28. The Expression editor for the BindGridLinkStringGrid11 LiveBinding

We have seen much of what is going on here before. The PosControl and PosSource collections are keeping the StringGrid and the ClientDataSet synchronized on the same current record. The FormatControl expressions are configuring the overall look of the StringGrid, and the CellFormat expressions are displaying the data in the individual cells of the grid. One of the CellFormat expressions displays the RecNo property of the ClientDataSet, and a second displays the results of a concatenation of the FirstName and LastName fields. The third CellFormat expression displays the value of the City field in its corresponding column.

As mentioned, CellParse is used to write data from a cell back to the corresponding field of the ClientDataSet. In this particular grid, there is only one column that displays the data from a field, and therefore, this the only column for which a CellParse expression really makes any sense. This is the expression that is selected in the Expression editor shown in Figure 28.

As you can see, the ControlComponent is the StringGrid and the SourceComponent is the BindScopeDB. The direction of CellParse expressions is always control to source.

The Source expression is the AsString property of the underlying field, which is the City field (and which is identified by the SourceMemberName property of this expression). The Control expression is SelectedText(Self), and this requires a little explanation.

SelectedText is a bidirectional custom method that permits the retrieval of data from, and assignment of data to, the control. Which control SelectedText refers to is defined by the Self variable in this expression, and Self is a reference to the control in the expression's scope. When used in the context of a StringGrid, SelectedText permits the value of the current cell to be read from or written to. As a result, changes made to the City field in the StringGrid are written to the current record of the ClientDataSet.

LiveBindings in FireMonkey Applications

Why, you might ask, have I used a VCL application to demonstrate nearly every LiveBinding that I've covered so far when LiveBindings play such a crucial role in FireMonkey applications? The answer is this. I don't want you to think that LiveBindings are exclusively for implementing data awareness in FireMonkey applications. Sure, you can use them that way, but that is just one of their uses.

First of all, data awareness is only one use of LiveBindings. Consider the BindLinks associated with the two buttons in the VCLLiveBindings project as well as the BindPosition associated with the StatusBar. These LiveBindings represent a different way of thinking about the user interface. They let the user interface react to the state of the underlying controls, rather than being manually updated through the invocation of methods that have to be written with intimate knowledge of the controls being manipulated.

The second point is that LiveBindings are not a replacement for event handlers. Case in point— FireMonkey controls support event handlers, just as do VCL controls. LiveBindings are a new and different mechanism. There are things that event handlers do very well. At the same time, there are things that LiveBindings do that would be difficult for event handlers.

The third reason is that I think it's important to start using LiveBindings, where appropriate, in both VCL and FireMonkey applications. Once you start doing so, you will begin to think about LiveBindings differently. LiveBindings, as a framework, is going to be an important part of the new generation of rich user interface applications we are going to be expected to create.

The fourth reason is that configuring LiveBindings in FireMonkey uses all of the techniques that I have covered so far in this paper. Every one of the LiveBindings I described, along with their corresponding expression collections, appear in FireMonkey.

FireMonkey does have something, however, that the VCL implementation of LiveBindings currently does not have. FireMonkey has eight components not found in the VCL implementation. Seven of these are Link LiveBindings: TBindDBEditLink, TBindDBTextLink, TBindDBCheckLink, TBindDBGridLink, TBindDBListLink, TBindIDBImageLink, and TBindDBMemoLink. The final component is BindNavigator, and it is a component but not a LiveBinding. I will discuss the BindNavigator component later in this section.

The Link LiveBindings in FireMonkey greatly simplify the process of binding controls to DataSets, taking the responsibility for create all of the necessary expressions for you.

Code: The code for this project is available in the FireMonkeyLiveBindings project, which you can download along with this white paper.

Let take a look at a simple project that uses LiveBindings to provide a FireMonkey application with data awareness. The running application is shown in Figure 29.

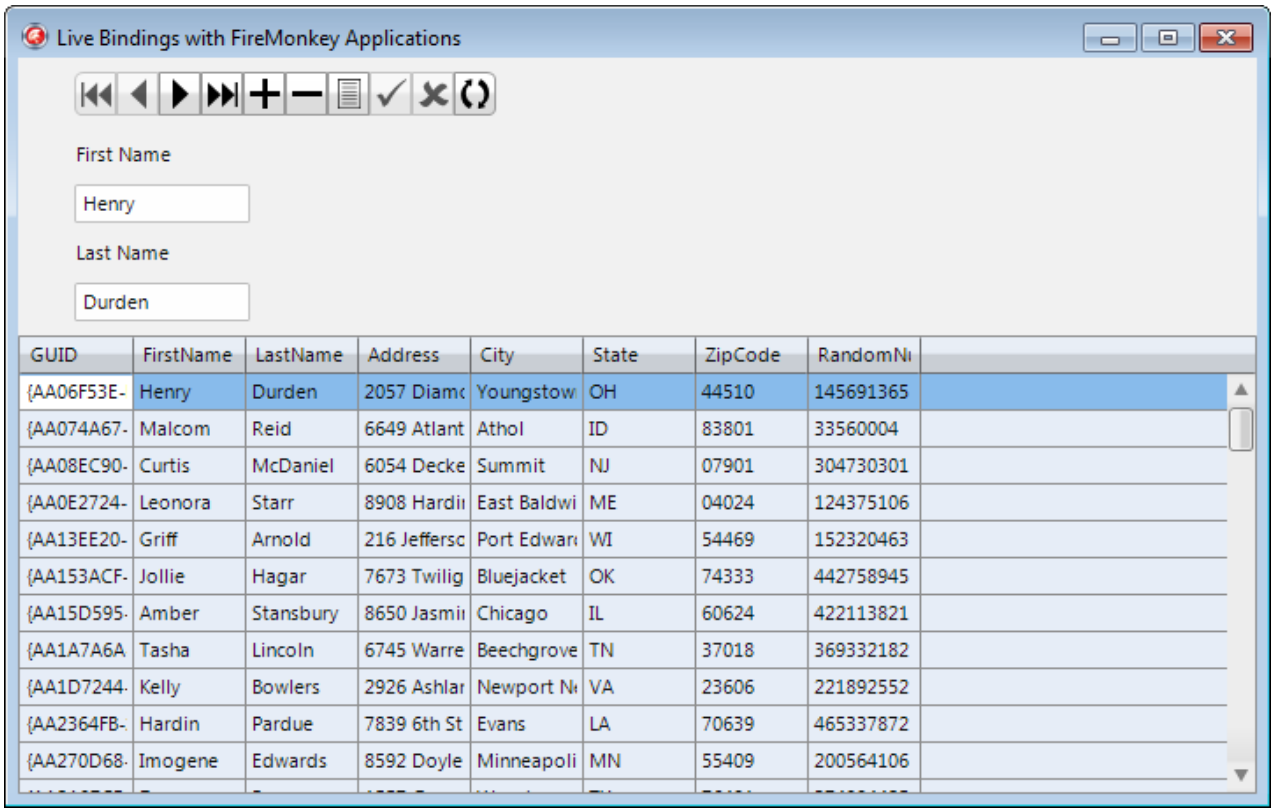

Figure 29. A data-aware FireMonkey application

This application is pretty plain, because I have not used LiveBindings to do anything special. However, with a little effort, all of the techniques that were used in the VCLLiveBindings project could have been applied here.

I did keep this application simple to demonstrate the incredible ease with which you can associate FireMonkey controls with data. In short, once you have a DataSet, a DataSource, and a BindScopeDB hooked up on a form, this interface can be recreated in a under a minute (if you work fast).

Here is what you are going to do. You are going to delete all of the visual controls from this form, and then re-create this form yourself. To do this, use the following steps:

1. Open the FireMonkeyLiveBindings project in RAD Studio.

2. Delete the StringGrid, the two Edits, and the BindNavigator from your form. Also, double-click the BindingsList and delete all of the LiveBindings that were previously created on this form. Your form should now look something like that shown in Figure 30.

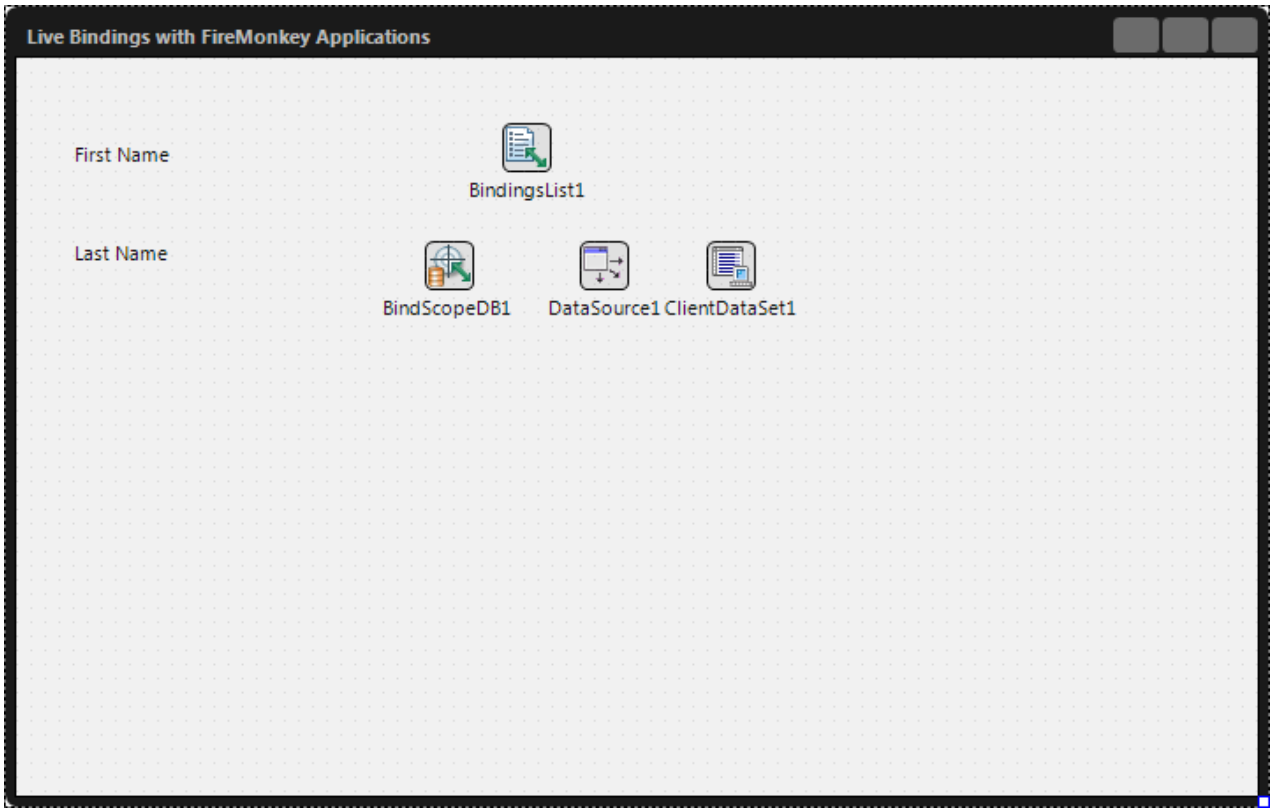

Figure 30. A basic form with a BindScopeDB that is hooked up to an active ClientDataSet through a DataSource

3. Next, we need to make sure that the ClientDataSet is configured and active. Select the ClientDataSet, and using the Object Inspector, set its FileName property to the sampledata.cds file, which you will find in the same directory as the FireMonkeyLiveBindings.dpr file. (Note that the FileName property editor for the ClientDataSet defaults its file filter to *.xml by default, so you will have to change the file filter to see the cds file.) Then, set the ClientDataSet's Active property to True.

4. Now, place a StringGrid on the form and set its Align property to alMostBottom. Adjust the size of the StringGrid so that it is more or less in the position of the StringGrid that appears in Figure 29.

5. Select the form (so that the StringGrid is not selected) and place two Edits, one beneath each of the Labels.

6. Select the form again and place a BindNavigator. Position it above the top-most Label.

7. Now you are ready to create the LiveBindings. Let's start with the StringGrid. Right-click the StringGrid and select New LiveBinding.... RAD Studio displays the New LiveBinding dialog box as shown in Figure 31.

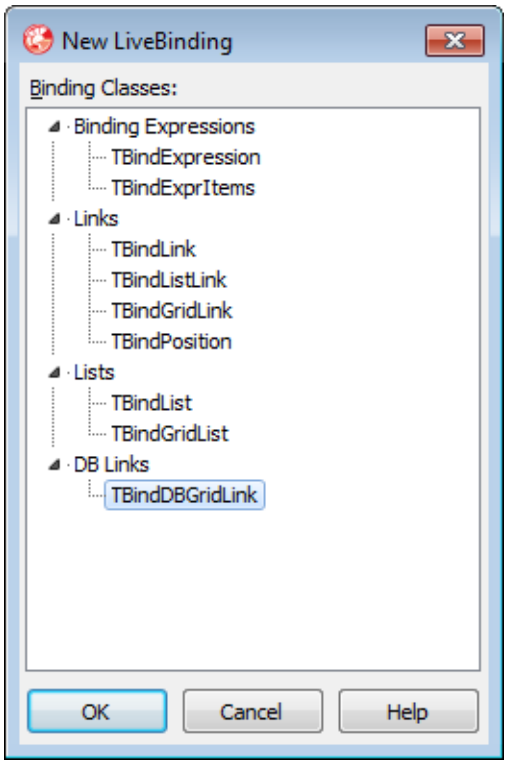

Figure 31. The New LiveBinding dialog box for a FireMonkey StringGrid

What you will see is similar, but not identical, to what you get when you create a new LiveBinding for a VCL component. Specifically, this dialog box includes an additional node: DB Links.

8. Select the TBindDBGridLink that appears under the DB Links node and click OK. A new LiveBinding named BindDBGridLinkStringGrid11 is created, and it is selected in the Object Inspector.

9. Using the Object Inspector, set the DataSource property of the new LiveBinding to BindScopeDB. Upon setting this property, the StringGrid is populated with the data from the ClientDataSet. Your form should now look something like that shown in Figure 32.

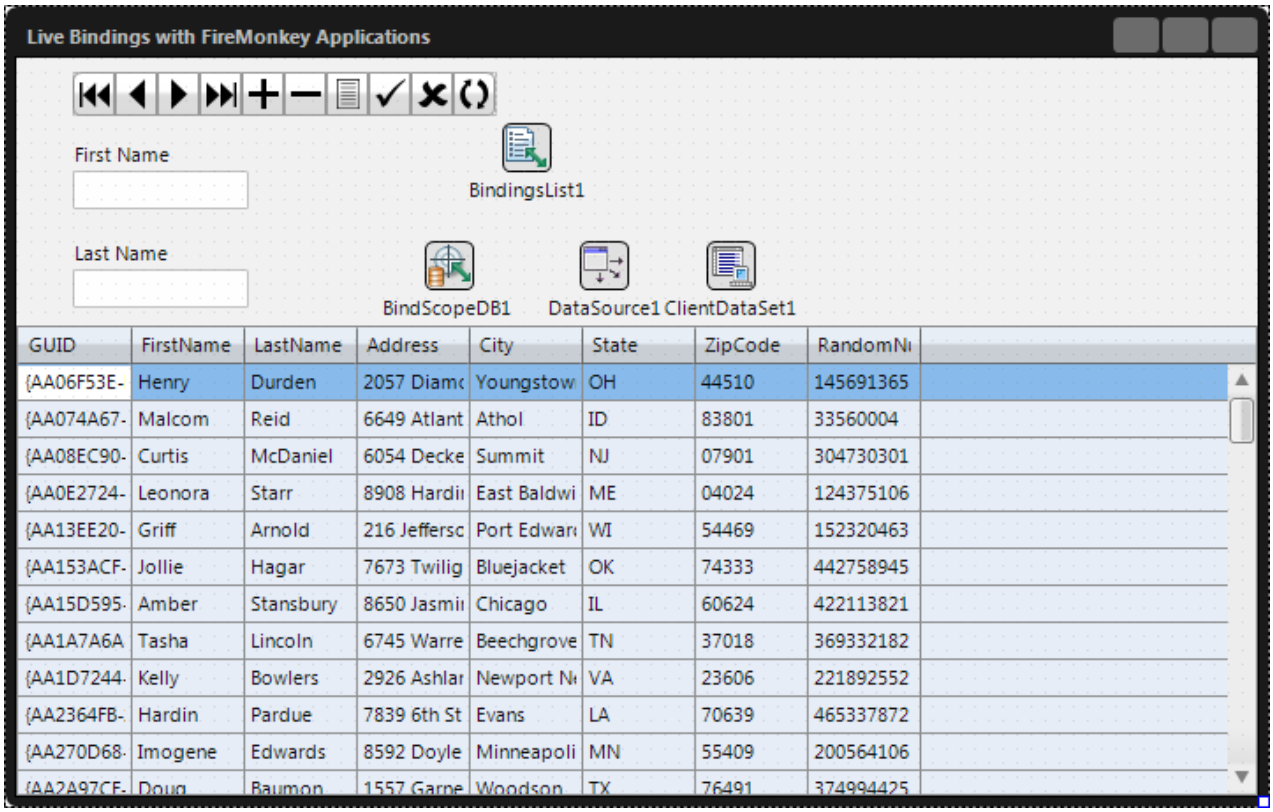

Figure 32. Once hooked up to a BindScopeDB, the new LiveBinding is fully functional

Actually, I made this a bit more difficult than I had to. The context menu for the StringGrid also included an option labeled Link to DB DataSource. If you had selected that option, the LiveBinding would have been created and configured that link for you in one step. Let's use that technique with the two Edits.

10. Right-click the Edit that appears below the First Name Label and Select Link to DB Field.... The New DB Link dialog box, shown in Figure 33, is displayed. Select FirstName and Click OK. The Edit is now fully configured. Repeat this step with the Edit that appears below the Last Name Label. This time select the LastName field from the New DB Link dialog box.

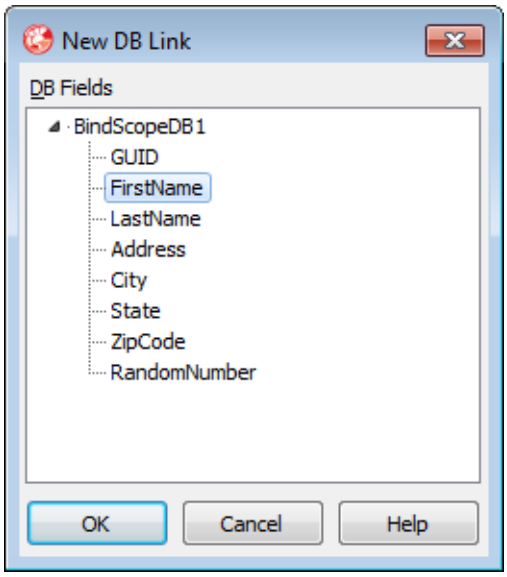

Figure 33. The New DB Link dialog box

11. Finally, let's configure the BindNavigator. Select the BindNavigator and set its BindScope property to BindScopeDB.

That's it. Your controls are fully data aware. You can now run the application, navigate the data, and even edit the data. If you close the application, the changes you made are changed. Admittedly, there are two event handlers on this form that helped. But here they are, and I think you'll agree that they are rather simple:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ClientDataSet1.Close;
  ClientDataSet1.FileName := ExtractFilePath(ParamStr(0)) + 'sampledata.cds';
  ClientDataSet1.LogChanges := False;
  ClientDataSet1.Open;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if ClientDataSet1.Active then
  begin
    ClientDataSet1.SaveToFile;
    ClientDataSet1.Close;
  end;
end;
```

So, what is different about adding LiveBindings in VCL versus FireMonkey applications? The answer is that the component editor created the Link LiveBindings and configured them almost the same way as you would do manually in VCL applications. The classes and

the properties are the same. But there is one difference. You cannot edit the properties of these LiveBindings. In fact, if you open one of these LiveBindings in the Expression editor, you will find that the Expression collections are read only, as shown in Figure 34.

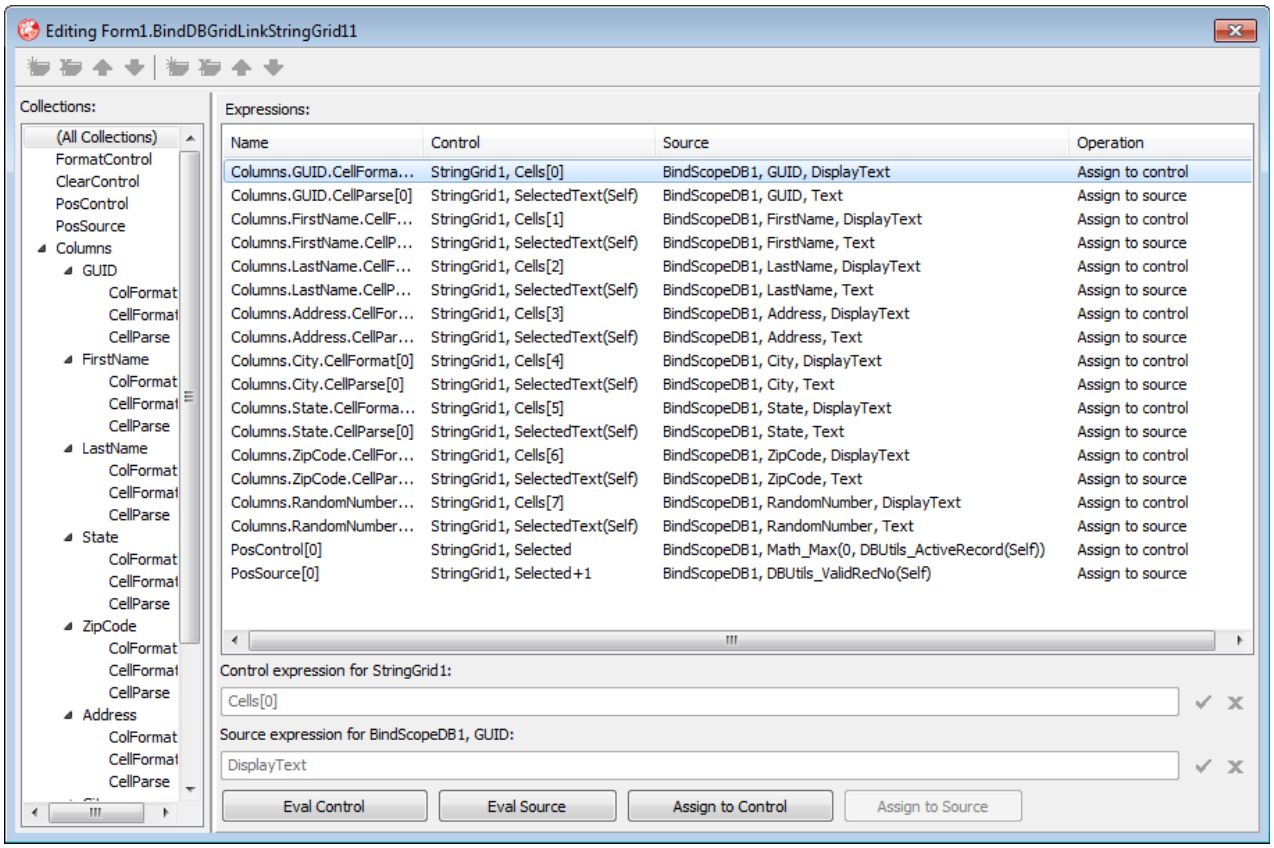

Figure 34. A LiveBinding created by the StringGrid component editor

I actually think that it is a shame that we cannot change these LiveBindings at design time. For example, while you can use the StringGrid's Column Editor component editor to configure the individual columns of the StringGrid, you cannot customize the individual expressions of this generated LiveBinding.

If you want to use these LiveBindings (as opposed to configuring Link LiveBindings manually), it is possible to change one or more of their properties, but you must do so at runtime. To do so, add an OnActivating event handler to your form, and use the GetDelete method of the Link LiveBinding to get a reference to the LiveBinding object. You can then use this reference to make changes to the contained expression collections.

There is only one more topic to cover where LiveBindings in FireMonkey applications are concerned. FireMonkey does not have any of the data-aware controls that appear on the

Data Control page of the VCL Tool Palette, which means that there is no DBNavigator in FireMonkey. That is where the BindNavigator comes in. If you want a predefined navigator in a FireMonkey application, use the BindNavigator.

THE FUTURE OF LIVEBINDINGS

I have been interested in LiveBindings since RAD Studio XE2 was released. It was clear to me that LiveBindings represented a new way to work with objects, and I have worked to promote their use through magazine articles and presentations, such as my LiveBinding presentation during the 24 Hours of Delphi event hosted by David Intersimone (David I). I have also blogged a bit about LiveBindings.

The response has been interesting, though not entirely positive. In short, there are three common complaints that developers have shared with me and I want to take this opportunity to address them.

The first comment that I have heard a number of times is that LiveBindings don't really add anything new. In other words, there is nothing that LiveBindings can do that you cannot already achieve using the existing mechanisms in RAD Studio, such as event handlers.

I don't agree. Sure, event handlers are powerful, and fast, but if we only look at LiveBindings as just a different way of doing what we've always done, we will come to this same conclusion. And that's just the thing. LiveBindings requires us to look at our goals from a different perspective. LiveBindings can do things that really don't fit into the event-driven model of traditional event handlers. Sure, in this version of RAD Studio they are somewhat limited, but that will change over time.

Here is an example of what I am talking about. I want to turn my attention back to a BindExpression event handler that I created for the VCLLiveBindings project, but have not yet discussed. To me, this LiveBinding represents a new way of working with components.

First of all, this BindExpressions includes a call to a method that produces side effects in the SourceExpression. This is that method, and as you can see, it is associated with the Form.

```
function TForm1.OpenCDS(Open: Boolean): string;
begin
  if Open then
  begin
    ClientDataSet1.Open;
    ClientDataSet1.LogChanges := False;
    exit('Close');
  end
  else
```

```
  begin
    ClientDataSet1.SaveToFile;
    ClientDataSet1.Close;
    exit('Open');
  end;
end;
```

This BindExpression is associated with an ActionItem named OpenCloseAction. Two Buttons use this ActionItem as their Action, which means that they share a number of properties, such as Caption, Enabled, images, and use the OnExecute event handler as their OnClick event handler. As a result, clicking either of these buttons both opens or closes the ClientDataSet, as well as causing the Caption of any object using this ActionItem to update. While this particular ActionItem is associated with two buttons on the Form, any number of different components (not just Buttons) could theoretically use this same ActionItem.

Turning back to the BindExpression, the ControlComponent is OpenCloseAction (the ActionItem), and this same object serves as the SourceComponent. The ControlExpression is Caption, and the SourceExpression is shown here:

```
Owner.OpenCDS(Self.Caption="Close")
```

This LiveBinding is shown in the Object Inspector in Figure 35.

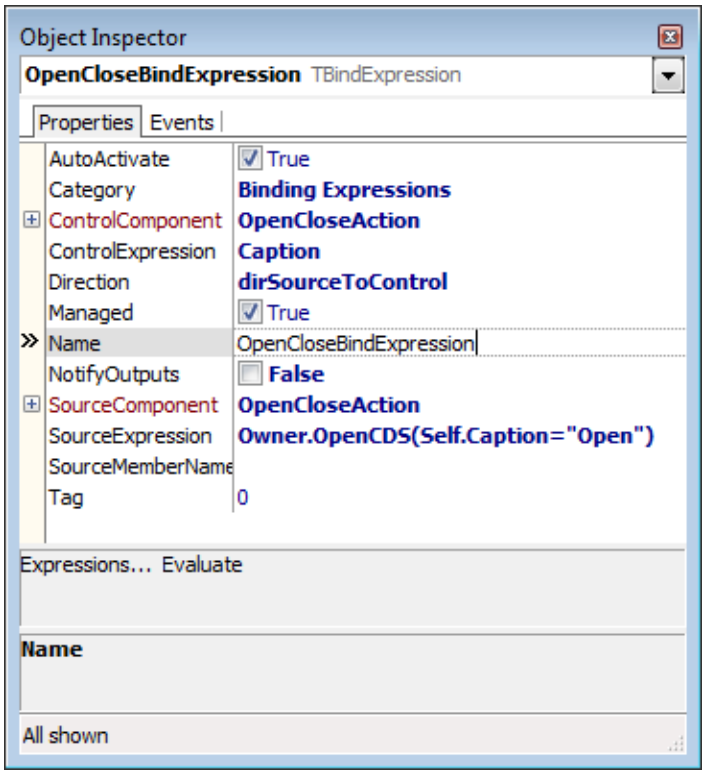

Figure 35. This LiveBinding, which produces side effects, represents a new way to enable component interaction

This is a managed LiveBinding, and accordingly, the OnExecute event handler for this ActionItem has to invoke Notify. However, it does so by calling the Value1EditChange event handler, which is the same event handler used by a number of managed LiveBindings on this form:

```
procedure TForm1.Value1EditChange(Sender: TObject);
begin
  BindingsList1.Notify(Sender, '');
end;
```

As you can see, when the user clicks the button, the expression engine is notified to evaluate the LiveBindings associated with Sender, which is the ActionItem in this case. The SourceExpression calls the OpenCDS method, which returns a new value for the Caption property of the ActionItem. This, in turn, causes the two buttons that are using this action to likewise adopt the caption. It also performs the rather simple side effect of closing or opening the ClientDataSet. However, there are really very few limits to the side effects that could have been implemented.

This is profoundly different than normal event handler usage. This event handler is completely agnostic, as far as the operation that will result from its invocation. All of the

behavior is define declaratively in the LiveBinding, along with the actions defined in any methods that are invoked during the evaluation of the SourceExpression. What's even more exciting is that in the future this type of effect might be achieved without any event handlers at all.

Earlier in this paper I noted that invoking a method that produces side effects from an expression is similar to the power of side effects produced by property accessor methods. Actually, these two techniques are more closely related than you might think. When designing a new class, you might actually implement a side effect in a property accessor method, and a LiveBinding could then produce that side effect as a result of its assignment of data to the associated property.

On the other hand, side effects produced by property accessor methods are often associated with keeping the internals of that component consistent. By comparison, the types of side effects that you can introduce in methods invoked through LiveBindings can have a much more global impact, keeping many elements of a form, or an entire application, synchronized.

The second comment people make about LiveBindings is that they are complicated. Sure, some LiveBindings require many different expressions, and if you are unfamiliar with this process, it can be challenging. But most of that is a combination of our collective unfamiliarity with these new tools and the limited design-time support that LiveBindings provide in this first release. I have no doubt that LiveBindings will get easier to use with each new release of RAD Studio. And, once you get comfortable with the process of configuring LiveBindings, I think much of the apparent complexity will vanish. It has for me, but it did take some time.

And, consider the following code listing. This is the listing for the implementation section of the main form in the VCLLiveBindings project. This is the only custom code involved in the entire project. All of the primary behaviors you see on the three different tabs are implemented by the LiveBindings themselves.

```
implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  ClientDataSet1.FileName :=
    ExtractFilePath(ParamStr(0)) + 'sampledata.cds';
  PageControl1.ActivePageIndex := 0;
end;

function TForm1.OpenCDS(Open: Boolean): string;
begin
```

```
  if Open then
  begin
    ClientDataSet1.Open;
    ClientDataSet1.LogChanges := False;
    exit('Close');
  end
  else
  begin
    ClientDataSet1.SaveToFile;
    ClientDataSet1.Close;
    exit('Open');
  end;
end;

procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  BindingsList1.Notify(TrackBar1, 'Position');
end;

procedure TForm1.Value1EditChange(Sender: TObject);
begin
  BindingsList1.Notify(Sender, '');
end;
end.
```

We can even get rid of the TrackBar1Change event handler, so long as we hook the TrackBar's OnChange property to use the Value1EditChange event handler, so this listing is actually more complicated than it needs to be.

The third complaint is that LiveBindings are insufficient for many tasks. Some of RAD Studio's biggest supporters have found that they cannot do with LiveBindings what they want to do. For example, longtime Delphi advocate and supporter Robert Love has noted that it is very difficult to create a bidirectional LiveBinding between a TList and a StringGrid. He's right. It can be done, but LiveBindings, at least in this incarnation, lack the support to make it practical.

The complaint is valid, but it is no reason not to use LiveBindings. First of all, this is the first release of LiveBindings. If developers need to do things with LiveBindings that are reasonable, but they cannot do those things now, it is very likely that this will change in future releases. While not going into specific details, RAD Studio Product Manager John Thomas has suggested that future releases will permit LiveBindings to bind to a much wider range of objects, and that doing so will be made much easier.

Delphi in particular, and RAD Studio in general, have been at the forefront of innovation for the more than 17 years, and good ideas are improved upon all the time. Delphi's RTTI (runtime type information) was truly groundbreaking when Delphi first shipped, and was a

critical technology in the implementation of the form designer. Over the years, RTTI has gotten more powerful and at the same time easier to use.

There are other examples as well, but I think it is safe to say that, yes, LiveBindings, in this release, have some limitations. These limitations, however, are unlikely to remain for long. LiveBindings is an important new technology, one that is sure to improve with age. On the other hand, LiveBindings have also added a whole new dimension to RAD Studio's capabilities. We should not avoid using them just because they can't do everything we want now.

I have a several personal concerns about LiveBindings as well. First, they are slow. The expression engine is an interpreter, and it appears as though it re-evaluates each expression every time. Consider the VCLLiveBindings project. If you click either of the buttons to close and then re-open the ClientDataSet you will notice a delay before the final expression has been evaluated. By comparison, event handlers are fast, since the code that appears in them is compiled to binary instructions at compile time.

What this means, at least for now, is that you should use LiveBindings where appropriate, but they are not a replacement for event handlers or design time configuration. For example, consider the BindExpression LiveBinding that associated the TrackBar and the ProgressBar. Instead of using a BindExpression, we could have used a BindExprItems, and added a second expression that synchronized the Max property of the ProgressBar to the Max property of the TrackBar. This, however, would be an unnecessary use of an expression. The Max property can easily be set at design time. Having it be set by an expression that is going to be evaluated many times at runtime incurs an unwelcome amount of overhead.

My second concern is that you cannot share a LiveBinding definition. The VCLLiveBindings project has two almost identical BindExpression classes, the ones associated with the two buttons. I'm not entirely sure that it makes sense, but I think it would be cool if a given LiveBindings could be employed by two or more controls.

Another concern that I have is that there is no debugger for the expression engine. Yes, you can use the Eval Control and Eval Source buttons on the Expression editor to examine the type and value of the associated expressions, but the error message that is returned if something is wrong is sometimes unhelpful. The whole thing fails if there is one error in the expression. You currently have to reassemble a complex expression to determine what the expression engine doesn't like.

Once again, however, I suspect that these issues are short term, and that they will go away as LiveBindings matures.

SUMMARY

LiveBindings is the new object binding technology first introduced in RAD Studio XE2. Not only does it enable data awareness in controls that do not support the classic Data Controls interface, they represent a new way of defining inter-object interaction. Rather than being an alternative to traditional event handlers, LiveBindings give developers a new dimension in which to work, one that promises to enable the rich interfaces that users are coming expect from modern applications.

In this paper, I have demonstrated the use and configuration of all of the LiveBindings components. My demonstrations were all design time, however. Yes, you can create and configure LiveBindings at runtime (just as you can set published properties both at design time and runtime), and you can even write code to directly use the expression engine.

For more examples of LiveBindings, including quite a bit of runtime code, please search the Embarcadero Developer Network for Jim Tierney's recorded LiveBindings presentations, including the two detailed presentations he recorded for CodeRage 6. You can also view my LiveBinding presentation from 24 Hours of Delphi. The Embarcadero Developer Network can be found at edn.embarcadero.com.

About the Author

Cary Jensen is Chief Technology Officer of Jensen Data Systems (JensenDaraSystems.com), a consulting, training, development, and documentation and help system company. Since 1988 he has built and deployed database and Internet applications in a wide range of industries. Cary is the best selling author of more than 20 books on software development, and is touring this spring with fellow author Marco Cantù on the Delphi Developer Days 2012 tour. A frequent speaker at conferences, workshops, and seminars throughout much of the world, he is widely regarded for his self-effacing humor and practical approaches to complex issues. Cary has a Ph.D. in Human Factors Psychology from Rice University, specializing in human/computer interaction. Cary's latest book, Delphi in Depth: ClientDataSets, is available from http://www.JensenDataSystems.com/cdsbook.

Acknowledgements

I want to acknowledge the help and support that I received in writing this paper. In particular, I want to thank Jim Tierney, Principal Engineer at Embarcadero Technologies, who patiently explained to me many of the inner workings of LiveBindings, and who performed the technical review on this paper. I also want to thank the many Embarcadero employees who agreed to review a draft of this paper, including Michael Devery, Pawel Glowacki, David Intersimone, Andreano Lanusse, Anders Ohlsson, Calvin Tang, John Thomas, and Jason Vokes. I also want to thank Loy Anderson of Jensen Data Systems, Inc. for copyediting several drafts of this paper. Finally, I want to thank Tim Del Chiaro, who managed this project from start to finish.

Embarcadero Technologies, Inc. is the leading provider of software tools that empower application developers and data management professionals to design, build, and run applications and databases more efficiently in heterogeneous IT environments. Over 90 of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero's award-winning products to optimize costs, streamline compliance, and accelerate development and innovation. Founded in 1993, Embarcadero is headquartered in San Francisco with offices located around the world. Embarcadero is online at www.embarcadero.com.