COM Corner: ActiveX Data Binding In Delphi

by Steve Teixeira

The idea of binding database fields to control properties is certainly not a new one for Delphi developers. After all, data-aware controls have been a basic part of Delphi since version 1.0. However, the ability to connect fields to properties has traditionally been limited to native VCL controls. ActiveX data binding is a different animal altogether, and the building of Delphi's ActiveX support has progressed steadily since the early days. Delphi 2 brought us an ActiveX control container, but with no data binding support. Delphi 3 provided the ability to easily create ActiveX controls which support data binding, but still no container data binding support. Finally, Delphi 4 brings it all together to provide control and container support for ActiveX data binding.

So What Is ActiveX Data Binding?

Like people, ActiveX data binding comes in two varieties: *simple* and *complex*.

As the name implies, *simple data binding* is a fairly straightforward system for binding database field values to ActiveX control properties: the control notifies the container before and after a property value changes and the container automatically modifies the property value whenever the value in the associated database field changes.

Complex data binding involves the manipulation of interfaces, `ICursor` and `ICursorMove` in particular, to negotiate cursor-level data navigation and manipulation between control and data. Complex data binding isn't directly supported by Delphi, so this article will focus on the creation and use of simple data bound controls.

There isn't a lot of work to be done in order to implement data binding in an ActiveX control. Data binding is supported on a per-property level. To make a property bindable requires only a few flags to be set in the type library and a couple of extra method calls in your control's implementation. Speaking in purely ActiveX terms, bindable properties will normally have at least the `VARFLAG_FBINDABLE`, `VARFLAG_FREQUESTEDIT` and `VARFLAG_FDISPLAYBIND` flags set in the IDL description of the control interface. Additionally, the property 'setter' function for the bindable property will normally call the `OnRequestEdit` and `OnChanged` methods on the container's `IPropertyNotifySinks`. The container is responsible for maintaining the correlation between a bound property and a data set field. If all that flew over your head, stay tuned and I promise that this will make more sense when I discuss the Delphi specifics in just a moment.

As an aside, allow me to mention a bit of important terminology: a *bindable* property is a control property that follows the description outlined in the previous paragraph. A *bound* property refers to a bindable property that is currently connected to a field in a data set. One additional note is that some containers don't support data binding, so your control should be able to function (albeit in a more limited fashion) when used in such an environment.

Creating A Simple Data Bound ActiveX Control

Being a Delphi programmer, the first step in creating a data aware ActiveX control is to create a suitable VCL control or choose an existing VCL control from which to create the ActiveX control wrapper. For the purposes of this illustration, I've created a simple VCL control called `TColorPick` that simply paints a rectangle in a color specified by the `Color` property. The color can be set at design-time via the Object Inspector or at run-time using the control's local menu or application code. It is the `Color` property that I will make data-aware when this VCL control is encapsulated as an ActiveX control. This VCL control also surfaces two events associate with the modification of the color property: `OnCanChange`, which fires when the color is about to be changed and allows the programmer to veto the change, and `OnChanged`, which is a notification that fires after the property has changed. The code for this VCL component is shown in Listing 1.

Now that we have a suitable VCL control, the next step is to create an ActiveX control based on this VCL control using Delphi's One-Step ActiveX feature. This is done by adding the `TColorPick` component to the palette, selecting `ActiveX Control` from the `ActiveX` page of the `New Items` dialog, choosing the `TColorPick` component and filling in the file names appropriately in the ActiveX Control Wizard. The Wizard will then generate the source code and type library for the new ActiveX control, which I call `DbXColorPick`.

Now that the VCL and ActiveX controls have been created, it's time to mark the `Color` property as bindable and make the necessary source code additions I described generically earlier. To set the appropriate type library flags, open the Type Library Editor by selecting `View | Type Library` from the main menu. In the type library editor, expand the control's primary interface (`IDbXColorPick`, in this case) node in the tree view in the left pane, and select the `Color` property. Select the `Flags` page in the right pane, and you will see a number of checkboxes that represent various IDL flags for the

```
unit ColorPickers;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs;
type
  TColorChangeEvent = procedure (Sender: TObject; NewColor:
    Tcolor; var CanChange: Boolean) of object;
  TColorPick = class(TCustomControl)
  private
    FColor: TColor;
    FOnCanChange: TColorChangeEvent;
    FOnChange: TNotifyEvent;
    procedure PopupClick(Sender: TObject);
  protected
    function CanChange(NewColor: TColor): Boolean; virtual;
    procedure Changed; virtual;
    procedure Paint; override;
    procedure SetColor(Value: TColor); virtual;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property Align;
    property Anchors;
    property Color: TColor
      read FColor write SetColor default clBlue;
    property Constraints;
    property DockSite;
    property DragCursor;
    property DragKind;
    property DragMode;
    property ParentShowHint;
    property ShowHint;
    property Visible;
    property OnCanChange: TColorChangeEvent
      read FOnCanChange write FOnCanChange;
    property OnChange: TNotifyEvent
      read FOnChange write FOnChange;
    property OnClick;
    property OnDblClick;
    property OnDragDrop;
    property OnDockDrop;
    property OnDockOver;
    property OnDragOver;
    property OnEndDock;
    property OnEndDrag;
    property OnEnter;
    property OnExit;
    property OnGetSiteInfo;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnStartDock;
    property OnStartDrag;
    property OnUnDock;
  end;
procedure Register;
implementation
uses Menus;
procedure Register;
begin
  RegisterComponents('Samples', [TColorPick]);
end;
constructor TColorPick.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Width := 100;
  Height := 100;
  FColor := clBlue;
  PopupMenu := NewPopupMenu(Self, '', paLeft, True,
    [NewItem('Change Color', 0, False, True, PopupClick, 0,
    '')]);
end;
function TColorPick.CanChange(NewColor: TColor): Boolean;
begin
  Result := True;
  if Assigned(FOnCanChange) then
    FOnCanChange(Self, NewColor, Result);
end;
procedure TColorPick.Changed;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;
procedure TColorPick.Paint;
begin
  Canvas.Brush.Color := FColor;
  Canvas.Rectangle(0, 0, Width, Height);
end;
procedure TColorPick.SetColor(Value: TColor);
begin
  if (FColor <> Value) and CanChange(Value) then begin
    FColor := Value;
    Invalidate;
    Changed;
  end;
end;
procedure TColorPick.PopupClick(Sender: TObject);
begin
  with TColorDialog.Create(Self) do begin
    if Execute then Self.Color := Color;
    Free;
  end;
end;
end.
```

➤ *Listing 1*

selected property. The flags pertaining to simple data binding are shown in table 1. In this case, I enable each of these flags for the `Color` property.

The final step in making the `Color` property bindable is to ensure that the `OnRequestEdit` and `OnChanged` methods of `IPropertyNotifySink` are called at the appropriate time. For this, I take advantage of the `OnCanChange` and `OnChanged` events that I created for the VCL control. The ActiveX Control Wizard already generated methods to handle these events, the implementation of which effectively translate calls to the VCL events into calls to the appropriate methods on the ActiveX control's events interface.

The code generated by the ActiveX Control Wizard for these two event handlers is shown in Listing 2.

Flag	Meaning
Bindable	The property supports data binding
Request Edit	The property will call the container's IPropertyNotifySink.OnRequestEdit method prior to modifying a property value.
Display Bindable	The property should be displayed to the user as bindable.
Default Bindable	The property best represents the control, and is therefore considered the default bindable property. This flag can only be used on one property on the object.

➤ *Table 1*

A bindable property must call `OnRequestEdit` prior to changing the property value so that the container has an opportunity to disallow modification of a property value when it sees fit (when a data set is read-only, for example). Rather than directly obtaining the container's `IPropertyNotifySink` interface and attempting to call it's methods, the `TActiveXControl` base class provides two overloaded methods which do the dirty work for you. These methods are declared as shown in Listing 3.

As you've probably deduced, the different versions of this method are provided as a convenience to the developer, allowing the developer to refer to the property either by name or by dispid. You should use the dispid

version of PropRequestEdit whenever possible because it is more efficient than the property name version because the implementation of the property name version obtains the dispid for the property using IDispatch.GetIDsOfNames and then calls the dispid version (use the type library editor to find the dispid for a particular property).

The implementation of the dispid version of PropRequestEdit does the work of iterating over all of the IPropertyNotifySinks sinked to the control and called their OnRequestEdit function. If any of these return an error value, PropRequestEdit returns False. Listing 4 shows the new implementation of the TDbXColorPick.CanChangeEvent that includes the call the PropRequestEdit.

As you might expect, TActiveXControl also contains a wrapper that handles the housekeeping involved in calling the IPropertyNotifySink.OnChanged method. This method, called PropChanged, is also overloaded for either a property name or dispid (Listing 5).

Once again, the dispid version of this method is more efficient. The new implemenation of the TDbXColorPick.ChangeEvent, which calls PropChanged, is shown in Listing 6.

By the way, notice that I did not insert my calls to PropRequestEdit and PropChanged in the Set_Color property setter that was generated for the ActiveX control by the ActiveX Control Wizard. It would have been incorrect to call PropRequestEdit and PropChanged in Set_Color because Set_Color is only called when the Color property is set via COM. If the Color property is set via the control's local menu, for example, the Set_Color method would not get called. You have to make sure that the locations in code you choose to insert your calls to PropRequestEdit and PropChanged are reside in code paths that will be executed *any time* the property value changes.

Just as I mentioned earlier, you set a few type library flags, and you call a couple of methods. That's all there is to creating a data bound ActiveX control! Your control is now ready for use in Delphi, VB, or another ActiveX control container. Check the notes at the end for compilation and installation details.

```
procedure TDbXColorPick.CanChangeEvent(Sender: TObject;
  NewColor: TColor; var CanChange: Boolean);
var
  TempCanChange: WordBool;
begin
  TempCanChange := WordBool(CanChange);
  if FEvents <> nil then
    FEvents.OnCanChange(OLE_COLOR(NewColor), TempCanChange);
  CanChange := Boolean(TempCanChange);
end;
procedure TDbXColorPick.ChangeEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnChange;
end;
```

➤ *Listing 2*

```
function PropRequestEdit(DispID: TDispID): Boolean; overload;
function PropRequestEdit(const PropertyName: WideString): Boolean; overload;
```

➤ *Listing 3*

```
procedure TDbXColorPick.CanChangeEvent(Sender: TObject;
  NewColor: TColor; var CanChange: Boolean);
var
  TempCanChange: WordBool;
begin
  TempCanChange := WordBool(CanChange);
  if PropRequestEdit(-501) and (FEvents <> nil) then
    FEvents.OnCanChange(OLE_COLOR(NewColor), TempCanChange);
  CanChange := Boolean(TempCanChange);
end;
```

➤ *Listing 4*

```
function PropRequestEdit(DispID: TDispID): Boolean; overload;
function PropRequestEdit(const PropertyName: WideString): Boolean; overload;
```

➤ *Listing 5*

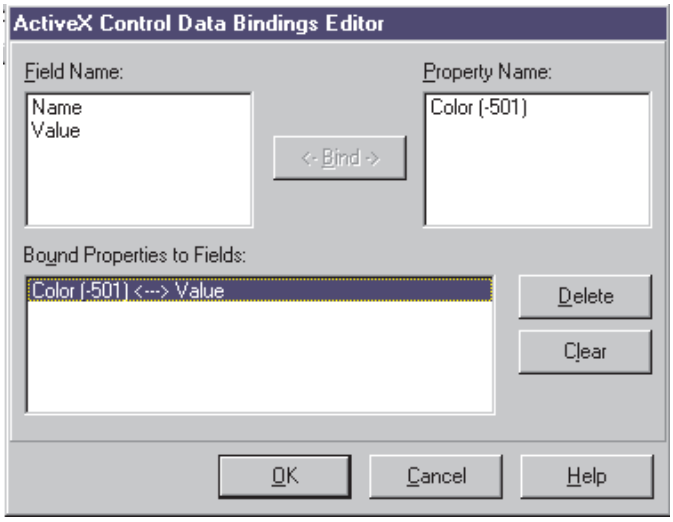
```
procedure TDbXColorPick.ChangeEvent(Sender: TObject);
begin
  PropChanged(-501);
  if FEvents <> nil then FEvents.OnChange;
end;
```

➤ *Listing 6*

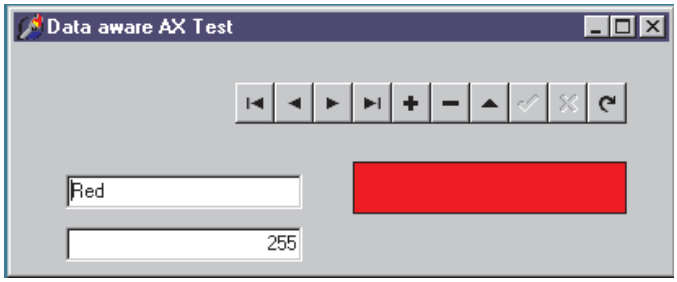
Using Simple Data Bound Controls In Delphi

If you inspect the Pascal file generated from the type library for this control, you will see that a VCL wrapper is created for the control like any other ActiveX control, but the wrapper descends from TDbOleControl rather than TOleControl as you might have been used to seeing. TDbOleControl is found in the DbOleCtl unit, and it represents a data-aware TOleControl. The most notable property of TDbOleControl is called DataBindings, which is a collection of TDataBindItem, where each collection item represents an association between one ActiveX control property and one field in a data set. When you import this control into Delphi and attempt to manipulate the DataBindings property, you will be presented with a property editor called the Data Bindings Editor, which enables you to establish links between the various bindable properties and field in a data set. The Data Bindings Editor is shown in Figure 1.

Once a link is established between the bindable property and a field, you can use the ActiveX control in a manner similar to Delphi's native ActiveX control. To ddemonstrate, I've created a database table with two fields: a string and an integer. The string



➤ *Figure 1*



➤ *Figure 2*

represents the name of a color and the integer represents the RGB value for the color. To browse this data, I have also created an application that allows you see the `DbXColorPick` ActiveX control in living, breathing action. The app is shown running in Figure 2.

The disk with this issue contains the source code for the `TColorPick` VCL control, the associated `DbXColorPick` ActiveX control, and the sample application and data. To go ahead and compile and use the ActiveX, follow a few simple steps. First, open the ActiveX project in Delphi 4 (sorry, not Delphi 3). Then compile it using `Project|Compile`. Then you need to register the OCX with Windows, either from Delphi (`Run|Register ActiveX server`) or from the command line (from `Start|Run` type `regsvr32 filename.ocx` after copying the OCX file into the Windows\System folder). Finally, import the ActiveX into Delphi (`Components|Import ActiveX`). You are then ready to try the sample project.

Summary

I hope I've given you some insights on what ActiveX data binding is and how you can create and use ActiveX controls with bindable properties in Delphi 4. As a control writer, Data binding support will make your ActiveX controls even more useful to those using tools like Visual Basic or Visual C++. As an application developer, the ability to use data-aware ActiveX controls in Delphi opens up even more possibilities in terms of third party controls available for your use. All that and the technology is a lot of fun to boot!

Steve Teixeira is an R&D Engineer at Inprise Corporation where he works on the Borland Delphi and Borland C++Builder products. He is also the co-author of *Delphi 4 Developer's Guide* from SAMS publishing. Steve thinks it's cool when you send him Delphi COM questions that make good articles. Got one? Why not email Steve at steixeira@inprise.com