Delphi Meets COM: Part 8

All good things...

by Dave Jewell

All good things must come to an end and, sadly, my contribution to the *Delphi Meets COM* series is no exception. This is the final instalment in my own introduction to COM from the perspective of a Delphi programmer. I'd like to take this opportunity to thank all the readers who emailed me to say how much they appreciated the series and especially to the gentleman who hoped that it would go on for ever!

From a financial point of view, the idea of this series going on forever is certainly attractive, but the truth of the matter is that I'm very much a beginner to COM myself and we'd soon end up scraping the barrel of what I know about this technology *[Heaven help the rest of us! Ed]*. The good news is that I'm passing you over into the capable hands of Steve Teixeira, who will be taking the reins from next month and covering the more advanced topics about which I feel less than authoritative! Amongst other things, Steve plans to cover advanced automation, threading models and DCOM. I'm sure that, like me, you'll find it fascinating stuff.

Property Pages Continued

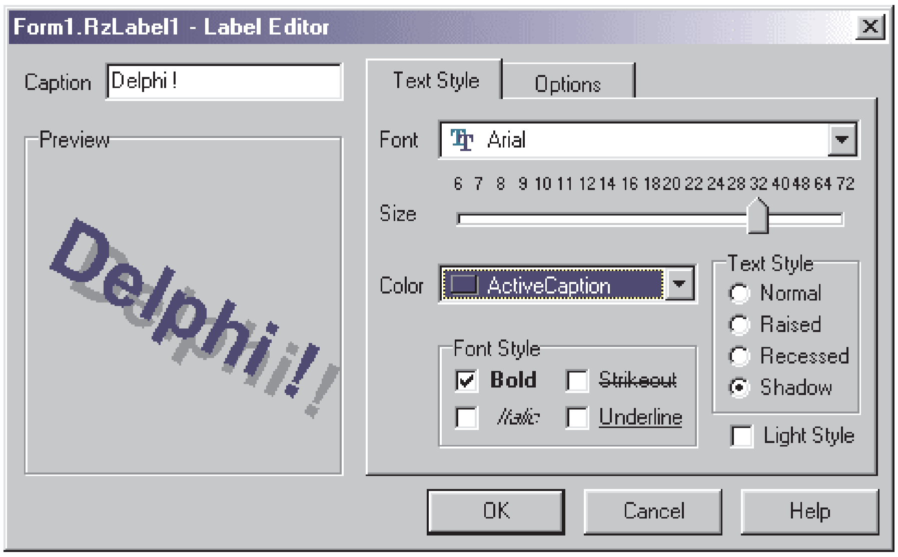
Last time round, I introduced the idea of property pages within the DAX framework and I promised that this month I'd explain how to create an association between the properties of your ActiveX component and the relevant VCL controls on a property page.

Creating such an association is very straightforward. If you examine the `DeskProp` unit (provided as part of last month's code and updated this month) you'll see that the `TDesktopProp` object has a couple of overridden methods called `UpdatePropertyPage` and `UpdateObject`. You don't have to override these methods, the code generated by Delphi will automatically do it for you. These routines have to be overridden because a property page would be useless without custom implementations of them both. Basically, the `UpdatePropertyPage` method transfers custom property information from the ActiveX component to the property page whereas `UpdateObject` does the reverse, copying property information from the property page to the underlying object.

`UpdatePropertyPage` is automatically called when a property page is first displayed and `UpdateObject` is called when the property page is dismissed. This raises an obvious question: how do we ensure that `UpdateObject` doesn't get called if the user dismisses the property dialog using the `Cancel` button rather than the `OK` or `Apply` buttons? The answer should be equally obvious, we don't! The DAX framework takes care of all this stuff for us, we don't care which button the user presses because our overridden `UpdateObject` method will only be called if `OK` or `Apply` has been clicked.

Let's follow through and finish implementing the property dialog for our somewhat whimsical desktop component. To begin with, you need to decide what VCL control types you're going to use for each property type in your control. For example, you'll almost certainly use an edit box for modifying text properties. If you're modifying a scalar property such as border width, angle, or something along those lines, Delphi offers a rich set of 'widgets' for tweaking the value of the property such as trackbars, up/down spin buttons and so forth. If you want to get some ideas for how to implement property pages, I'd recommend you take a look at the property pages designed by Ray Konopka, author of the excellent Raize Components. Ray has designed some excellent property pages to go with his own controls and, interestingly, he's even used his own components on those

➤ *Figure 1: Here's one of the property pages from Ray Konopka's Raize Components library The preview component is itself a live instance of a TRzLabel and, of course, there's nothing to stop you from using similar techniques in any ActiveX property pages which you author with Delphi.*


Form1.RzLabel1 - Label Editor

Caption: Delphi !

Font: Arial

Size: 6 7 8 9 10 11 12 14 16 18 20 22 24 28 32 40 48 64 72

Color: ActiveCaption

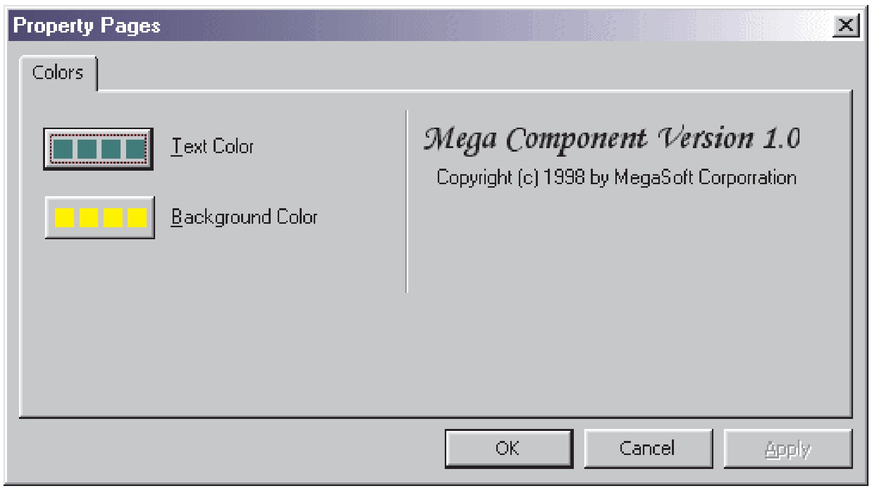
Font Style: [✓ Bold] Strikeout Italic Underline

Text Style: Normal Raised Recessed [✓ Shadow]

Light Style

same property pages. It might sound a bit counter-intuitive, but there's nothing to stop you from using your new-fangled slider control on a property page which sets attributes of the same component type. This technique is just as applicable to ActiveX control development as it is in straight VCL work.

In the present case, we'll look at just two properties of our desktop control, the text colour and text background colour properties. When editing colour values, it's nice to show (within the property page) the current colour assignments for the properties; it helps to give visual confirmation that the correct property is being modified. Accordingly, I added a couple of `TBitBtn` buttons to the property page and used them to access a standard colour dialog component. Because Delphi's button wrapper controls (by which I mean controls which wrap the Windows API-level `BUTTON` class) don't allow you to modify the surface colour of the button, I had to cheat slightly in order to represent the currently assigned colour values within the button. To do this, I used the Wingdings font to display the button captions, and made each caption out of a series of 'n' characters, which is represented as a solid box in this font. With a reasonably large font size, you can change the colour of the button by assigning to the button's `Font.Color` property. The effect is as shown in Figure 2.


Property Pages
Colors
Text Color
Background Color
Mega Component Version 1.0
Copyright (c) 1998 by MegaSoft Corporation
OK
Cancel
Apply

➤ *Figure 2: OK, so it's not as fancy as Ray's property editor, but it does the job! This illustrates one possible approach for changing colour value properties using a couple of TBitBtn components. Bear in mind that the OK, Cancel and Apply buttons are not part of your 'form' and are added by the DAX library framework.*

Listing 1 shows the complete code for our property page dialog. As you can see, the amount of code to be written is very small, partly because I've cheated (again!) and used `ForeColorClick` as a common event handler for both of the bitbutton controls. Based on what we've discussed so far, the code should be pretty straightforward. There are just two points I want to stress here. Firstly, be sure to only access the underlying ActiveX control through `OleObject` which is a property of the `TPropertyPage` class from which your own property page is derived. If you don't go through `OleObject`, then things won't work as advertised.

Secondly, and most importantly, notice the all-important call to `Modified` within the `ForeColorClick` event handler. Once again, this is a method of the `TPropertyPage` class. It's only by calling `Modified` that the DAX library knows you've actually altered the value of some property. DAX doesn't know what user interface you're going to provide for property editing and it therefore doesn't know (for example) whether or not the user clicked `Cancel` or `OK` when the colour selection dialog appeared. It's up to you to call `Modified` when the user actually makes some property change. If you don't call `Modified`, then the DAX library will assume no changes have been made: it won't bother to enable the `Apply` button

➤ *Listing 1*

```
unit DeskProp;
interface
uses SysUtils, Windows, Messages, Classes, Graphics,
  Controls, StdCtrls, ExtCtrls, Forms, ComServ, ComObj,
  StdVcl, AxCtrls, Dialogs, ExpBtn, Buttons;
type
  TDesktopProp = class(TPropertyPage)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Bevel1: TBevel;
    Label4: TLabel;
    ColorDialog1: TColorDialog;
    ForeColor: TBitBtn;
    BackColor: TBitBtn;
    procedure ForeColorClick(Sender: TObject);
  private
  protected
    procedure UpdatePropertyPage; override;
    procedure UpdateObject; override;
  public
  end;

const
  Class_DesktopProp: TGUID =
    '{C5B5EEE1-E9A7-11D1-8CDD-8D1116120B0F}';

implementation
{$R *.DFM}

procedure TDesktopProp.UpdatePropertyPage;
begin
  { Update your controls from OleObject }
  ForeColor.Font.Color := OleObject.TextColor;
  BackColor.Font.Color := OleObject.TextBackgroundColor;
end;

procedure TDesktopProp.UpdateObject;
begin
  { Update OleObject from your controls }
  OleObject.TextColor := ForeColor.Font.Color;
  OleObject.TextBackgroundColor := BackColor.Font.Color;
end;

procedure TDesktopProp.ForeColorClick(Sender: TObject);
begin
  with Sender as TBitBtn do begin
    ColorDialog1.Color := Font.Color;
    if ColorDialog1.Execute then begin
      Modified;
      Font.Color := ColorDialog1.Color;
    end;
  end;
end;

initialization
  TActiveXPropertyPageFactory.Create(
    ComServer, TDesktopProp, Class_DesktopProp);
end.
```

and any changes the user makes will be politely ignored when the OK button is clicked.

Creating ActiveX Control Libraries

Although I've probably mentioned this a while back, it needs to be appreciated that an OCX file is just, at the fundamental level, a Windows dynamic link library. Most users (and most developers!) tend to think of an OCX file as containing just a single control. However, you're at liberty to place multiple ActiveX controls into a single OCX file. In order to add a new control to an existing OCX project, just open the project, select New from the File menu, and choose ActiveX Control from the ActiveX page of the object repository, it's as simple as that.

From the perspective of the Delphi programmer, there are some positive benefits to be had from using this technique. First and foremost, if you're building your controls 'standalone' (ie the executables do not require any Delphi runtime packages) then you'll save a great deal of disk space by bundling multiple controls into a single executable. This is because only a single copy of the necessary VCL and DAX runtime code is required. Moreover, if you're planning to do things properly and automatically register your components at install-time on the end-user's machine, you'll only have to make a single call to the DllRegisterServer routine within the OCX file. This means that component registration will not only be simpler, but it will also be faster than would be the case if multiple OCX files were involved. Finally, if it becomes necessary to unregister your control library, this will also be simpler and faster than would otherwise be the case.

Against this, there are some obvious disadvantages. Firstly, there's the size of the resulting OCX file. Both Delphi 3 and Delphi 4 provide a sample ActiveX library project (it's in the directory Demos\ActiveX\DelCtrls) which illustrates how to group a number of the standard Delphi VCL controls into a single large OCX file. Under Delphi 4, the resulting OCX file weighs in at just over a megabyte! This is pretty big, but perfectly acceptable for deploying via CD. It isn't, however, the sort of thing that you'd want to download over the internet; at least, not by choice.

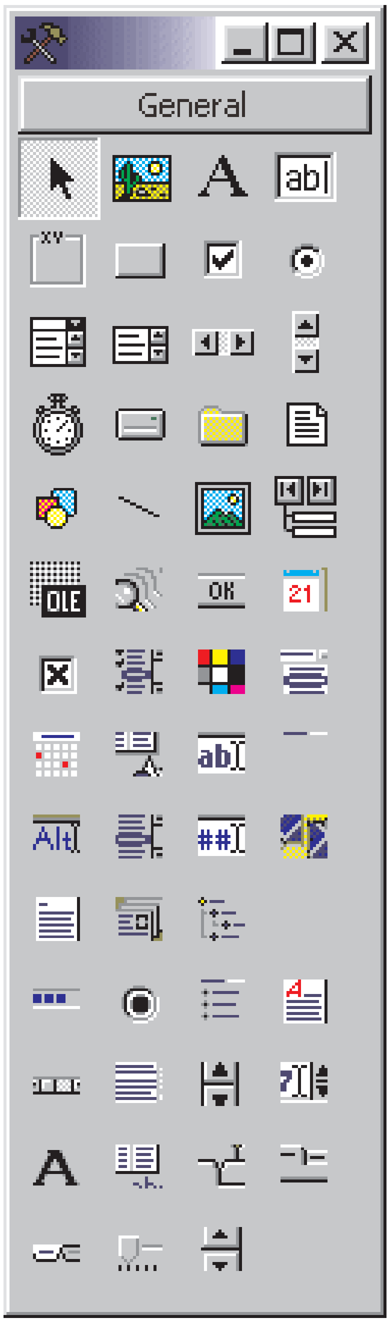
In actuality, when you bear in mind that there are some 34 controls in the project, the OCX file isn't that big. It averages out at around 29Kb per control, which is pretty good when you bear in mind that you've got the VCL/DAX runtime code in there as well. However, if any of the controls in the library require revision, then you're forced to ship the entire OCX file again.

There's another, less obvious disadvantage to putting multiple controls into a single OCX file, but it's something that can easily be rectified. As we discussed last time, whenever you use the ActiveX control wizard to create a new control, it will automatically include a new About unit into the project. This unit contains nothing more than a simple About box signed by 'Frank Borland' (Hmmm... I wonder whether that should be changed to Frank Inprise? It doesn't quite have the same ring to it, does it?) and a small amount of supporting code. In the case of the aforementioned DelCtrls project, you will end up with no less than 34 identical About forms, each of which is invoked from one of 34 identical units. This is obviously a very wasteful, nonsensical situation.

It's very easy to clean things up. If you examine each of the implementation units of your ActiveX control library, you'll see that the TActiveXControl class has a protected method called AboutBox. The implementation code for this method (generated by Delphi) then calls a routine called ShowXXXXAbout which is implemented in the About unit. The simplest way of tidying things up is to write a single, centralised About box procedure which determines the type of associated control on-the-fly and then adjusts the displayed form accordingly. For example, here's the AboutBox routine from Borland's ActiveX wrapper for the trackbar control:

```
procedure TTrackBarX.AboutBox;
begin
  ShowTrackBarXAbout;
End;
```

Now here it is after a little tweaking:



➤ *Figure 3: Here's what a 1Mb OCX file looks like in action! This is the result of building the Borland-supplied DelCtrls OCX and running it under Visual Basic.*

```
procedure TTrackBarX.AboutBox;
begin
  ShowActiveXAbout (Self);
end;
```

In the second case, we're calling a generic About box procedure, rather than one that's tied to a specific control. Additionally, we're passing the control instance (a derivative of TActiveXControl) to the routine. This would enable the generic code to determine what control-specific information to display on the About form.

As an alternative, you could always create your ActiveX controls without an About box (by leaving the About box checkbox unchecked in the ActiveX control wizard) and then manually add the necessary About box code yourself. However, I'd tend to discourage such an approach because it's error prone. The only thing that determines whether or not a control has an About box is the presence of the AboutBox method in the control object. If you wanted to add an About box after the ActiveX control wizard has done its stuff, you'd have to edit the type library.

Delphi 4 Goodies

I can't resist finishing this article without saying a little about the increasing amount of COM support being built into the Delphi IDE itself. Regular readers of my *Beating the System* column will know that I devoted two or three months to a discussion of the undocumented LibIntf unit, information which (to the best of my knowledge) has never been published elsewhere.

While digging around in the guts of the Delphi 3 IDE, I discovered that there were traces of COM-patibility (for want of a better word!) built into some parts of the IDE itself. For example, there's an undocumented, and largely unimplemented, COM interface to the internal TLibAppBuilder which provides access to the number of designer forms currently open, and to the forms themselves. At the time, I pondered whether this was a vestige of some failed attempt at a COM-based Open Tools API, or whether it was a hint at things to come?

Having recently got my hands on a release candidate of Delphi 4, I can now reveal that it appears to be the latter. In other words, Inprise seem to be moving towards a much more COM-based approach for accessing the IDE internals, a fact which will be significant to anyone who wants to write IDE add-ons. Although the original interface is still there, many new COM interfaces have been added to a new unit called BORIDE.PAS which lives in the same directory as the other elements of the Open Tools API.

Inprise seem to be drawing a distinction between the Open Tools API on the one hand and what they call the NTA (Native Tools API) on the other. Different interface names are labelled as either TNTAxxxx or TOTAxxxx according to which category they belong to. The code fragment in Listing 2 is typical of what I'm talking about. This is an interface declaration which encapsulates the IDE's design-time form editor. As you can see, there are facilities for finding a component by name, determining how many components on the form are selected, retrieving individual components from the current selection, and so on. This interface also provides support for creating new components on a form, with the specified size, position and type name. Presumably the Container argument to CreateComponent allows you to programmatically add a component to some pre-existing container such as a TPanel, or whatever. The GetFormResource allows the contents of a form to be copied to a stream.

```
IOTAFormEditor = interface(IOTAEditor)
  ['{F17A7BD2-E07D-11D1-AB0B-00C04FB16FB3}']
  { Return the form editor root component }
  function GetRootComponent: IOTAComponent;
  function FindComponent(const Name: string): IOTAComponent;
  function GetComponentFromHandle(ComponentHandle: TOTAHandle): IOTAComponent;
  function GetSelCount: Integer;
  function GetSelComponent(Index: Integer): IOTAComponent;
  function GetCreateParent: IOTAComponent;
  function CreateComponent(const Container: IOTAComponent;
    const TypeName: string; X, Y, W, H: Integer): IOTAComponent;
  procedure GetFormResource(const Stream: IStream);
end;
```

➤ *Listing 2*

At the moment, there is very little to go on, and I've established that none of the quoted GUIDs appear to have been entered into the system registry on my machine, so it will be interesting to see how all this is implemented. Exciting times ahead! The only thing one can say with some certainty is that this represents yet another way in which the all-pervasive COM technology is intruding into the life of every developer, and that's a very good reason why you should continue to read this series.

I'd like to conclude by recommending a recently published COM programming book called *Essential COM*. This book, published by Addison-Wesley and written by Don Box (ISBN: 0-201-63446-5), is probably one of the most readable introductions to COM that I've seen. Don is an acknowledged expert on COM and he takes you from the basics right through to the most intricate details, but always in a very readable, and often amusing, manner. The only problem with the book (from a Delphi standpoint) is that everything is written from the perspective of a C++ developer, and all the code fragments are written in that language. If you're reasonably familiar with C++, you'll get an excellent grounding in COM and if nothing else you'll readily appreciate what an excellent job Inprise have done in hiding the nasty details away behind the scenes.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is Technical Editor of *Developers Review* which is also published by iTec. You can contact Dave at Dave@HexManiac.com