# Delphi Meets COM: Part 7

## *Getting familiar with DAX...*

***by Dave Jewell***

In last month's instalment of *Delphi Meets COM*, we built a simple, for fun only, component whose only role in life was to 'tweak' certain visual aspects of the Windows desktop. We converted `TDesktop` from a native VCL control into an ActiveX component and, along the way, I showed you some of the pitfalls that lie in wait during the ActiveX conversion process.

This month, as promised, we're going to look in more detail at DAX (the Delphi ActiveX Framework) and the code that's generated by Delphi when converting a control into an ActiveX component. You might say to yourself 'Why do I need to know this stuff? Why can't I just rely on Delphi to do the conversion for me?'

Well, I've always believed that having a knowledge of the 'barefoot' Windows API will make you a better Delphi programmer, because it enables you to solve problems that can't be solved with Delphi alone. Having a knowledge of inline assembler code will likewise enable you to do things that (rarely!) Object Pascal will not let you do. In the same way, C++ programmers who use the Microsoft ATL (Active Template Library) admit that understanding the code generated by the ATL control wizard is a big help when the time comes to modify their ActiveX control.

The bottom line, then, is that although Delphi's ActiveX control wizard takes a lot of the 'grunt work' out of the conversion process, there's no substitute for understanding the code produced by the wizard so that you can make your own custom changes to the code as the need arises. Anyone who has despaired over the acres of MFC code generated by the Visual C++ wizards will know exactly what I'm talking about!

### The OCX Project File

Let's begin by taking a look at the .DPR (project file) created by the control wizard. The project file for `TDesktop` is shown in Listing 1. As with any ordinary library project, the identifier following the `library` keyword defines the name of the output file, AxDesktop.DLL in this case. However, since this is an ActiveX control project, an extension of .DLL would not be appropriate. Accordingly, the control wizard adds a special directive `{$E}` to the project file, which tells Delphi's linker to create the file with a .OCX extension. Other custom extensions are possible, but anything after the first three characters is ignored.

While on the subject of compiler directives, you'll note that the project includes not only a .RES file (as one would normally expect) but also a .TLB file. For a plain vanilla OCX project created by the ActiveX wizard, the .RES file will contain a version resource (if you asked for one), and a small bitmap which represents the control in development systems that 'import' it. You'll remember from previous months that the .TLB file contains the *actual* type library information that's used by the Delphi's visual type library editor. Recall that the XXXX_TLB.PAS unit is only a Pascal translation of what's contained in the type library: the type library editor will always modify the TLB data in response to the changes you make, and then regenerate the XXXX_TLB.PAS file accordingly. When you build your OCX control, the .TLB file gets stitched into the executable as a resource of type `TYPELIB` with an ID of 1. Valid OCX controls must always contain a type library with this ID so that COM can access the information contained therein.

In the present case, four units are referenced by the project's `uses` clause. We can forget about the `About1` unit: as mentioned last time round, this is a plain vanilla Delphi form and needn't detain us any longer. `ComServ`, which I've referred to in the past, is crucially important because it contains the code and declarations for the `ComServer` variable (type `TComServer`) which implements the COM server itself. The `ComServer` object is automatically created simply by including the `ComServ` unit in your `uses` list.

If you examine the source for the `ComServer` variable, you'll see that it contains a number of useful properties and methods. For example, the `ObjectCount` property indicates the number of 'farmed out' objects that are currently dependent on the OCX file while `StartMode` tells us why the server has been started. This can take one of several values but for an OCX control it's most likely to be set to `smAutomation`.

➤ *Listing 1*

```
library AxDesktop;
uses
  ComServ,
  AxDesktop_TLB in 'AxDesktop_TLB.pas',
  XDesktopImpl in 'XDesktopImpl.pas' {XDesktop: CoClass},
  About1 in 'About1.pas' {XDesktopAbout};
{$E ocx}
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;
{$R *.TLB}
{$R *.RES}
begin
end.
```

Bear in mind that, as discussed previously, an OCX control is really a tiny automation server.

It's also possible to start a COM server with the intention of registering or un-registering it, and this is also reflected in the possible values of the StartMode property. You'll notice that the constructor for TComServer examines the command line for certain well known strings and sets the StartMode property accordingly. The strings REGSERVER and UNREGSERVER are used to perform automatic server registration and un-registration. Thus, to register a new COM server (typically done from an installer program), you just execute the server passing /REGSERVER on the command line.

Hang on a minute, I hear you cry! An OCX control is, at the end of the day, just a Windows DLL, so how can one execute it, let alone give it a command line to chew on? The answer is that the ComServ unit has been cleverly written so that it can be included by both in-process servers (DLLs) and by out-of-process servers (EXE files). The command line mechanism is only applicable to EXE files. For OCX controls, the installer program needs to call the DllRegisterServer routine to register the control and DllUnregisterServer to remove the registration information from the system registry. Looking back to the project file, you'll see that these two routines (along with DLLGetClassObject and DllCanUnloadNow) are exported from the OCX library.

Note that, deeply wonderful though Object Pascal is, it's a restriction of the language that DLL routines can only be defined as exports in the main library file. Things would be neater if the exports statement could live inside the ComServ unit, but it can't. So there. In practice, there are arguments for and against this. On the positive side it does mean that all exports from a library are grouped together in one place, so you're unlikely to end up accidentally exporting something you didn't want to export.

## One OCX, Many Controls...

Before leaving the ComServ unit, let me stress that just because you've got a single OCX file doesn't mean that you're limited to a single OCX control. One OCX library can contain multiple controls and, with Delphi, there are good reasons for bundling multiple controls into one file, the primary one being that common units are shared and overall disk size is reduced. Just as the ComServer object is exported from ComServ, there's also a ComClassManager variable which is created within, and exported by, the ComObj unit. The ComClassManager is primarily concerned with managing the list of class factories contained within the overall COM server. You will remember from

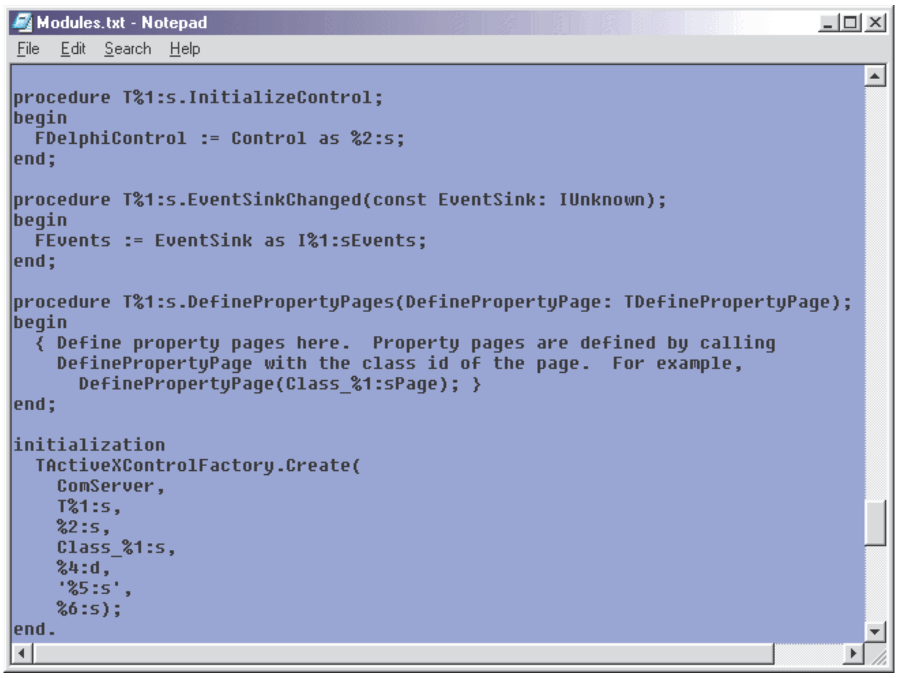
➤ *Listing 2*

```
unit XdesktopImpl;
interface
uses
  Windows, ActiveX, Classes, Controls, Graphics,
  Menus, Forms, StdCtrls, ComServ, StdVCL, AXCtrls,
  AxDesktop_TLB, Desktop;
type
  TXDesktop = class(TActiveXControl, IXDesktop)
  private
    FDelphiControl: TDesktop;
    FEvents: IXDesktopEvents;
  protected
    procedure InitializeControl; override;
    procedure EventSinkChanged(const EventSink: IUnknown);
      override;
    procedure DefinePropertyPages(
      DefinePropertyPage: TDefinePropertyPage); override;
    function Get_Font: Font; safecall;
    function Get_ItemCount: Integer; safecall;
    function Get_TextBackgroundColor: TColor; safecall;
    function Get_TextColor: TColor; safecall;
    procedure AboutBox; safecall;
    procedure Set_Font(const Value: Font); safecall;
    procedure Set_ItemCount(Value: Integer); safecall;
    procedure Set_TextBackgroundColor(Value: TColor);
      safecall;
    procedure Set_TextColor(Value: TColor); safecall;
    function Get_Visible: WordBool; safecall;
    procedure Set_Visible(Value: WordBool); safecall;
  end;
implementation
uses
  SysUtils, About1;
procedure TXDesktop.InitializeControl;
begin
  FDelphiControl := Control as TDesktop;
  FDelphiControl.Visible := False;
end;
procedure TXDesktop.EventSinkChanged(
  const EventSink: IUnknown);
begin
  FEvents := EventSink as IXDesktopEvents;
end;
procedure TXDesktop.DefinePropertyPages(
  DefinePropertyPage: TDefinePropertyPage);
begin
  { Define property pages here. Property pages are
    defined by calling DefinePropertyPage with the
    class id of the page, eg:
    DefinePropertyPage(Class_XDesktopPage); }
end;
function TXDesktop.Get_Font: Font;
begin
  GetOleFont(FDelphiControl.Font, Result);
end;
function TXDesktop.Get_ItemCount: Integer;
begin
  Result := FDelphiControl.ItemCount;
end;
function TXDesktop.Get_TextBackgroundColor: TColor;
begin
  Result := FDelphiControl.TextBackgroundColor;
end;
function TXDesktop.Get_TextColor: TColor;
begin
  Result := FDelphiControl.TextColor;
end;
procedure TXDesktop.AboutBox;
begin
  ShowXDesktopAbout;
end;
procedure TXDesktop.Set_Font(const Value: Font);
begin
  SetOleFont(FDelphiControl.Font, Value);
end;
procedure TXDesktop.Set_ItemCount(Value: Integer);
begin
  FDelphiControl.ItemCount := Value;
end;
procedure TXDesktop.Set_TextBackgroundColor(Value: TColor);
begin
  FDelphiControl.TextBackgroundColor := Value;
end;
procedure TXDesktop.Set_TextColor(Value: TColor);
begin
  FDelphiControl.TextColor := Value;
end;
function TXDesktop.Get_Visible: WordBool;
begin
  Result := FDelphiControl.Visible;
end;
procedure TXDesktop.Set_Visible(Value: WordBool);
begin
  FDelphiControl.Visible := Value;
end;
initialization
  TActiveXControlFactory.Create(ComServer, TXDesktop,
    TDesktop, Class_XDesktop, 1, '', 0);
end.
```

previous discussions that COM classes are generally created on behalf of the client using a class factory. Consequently, in an OCX control that exports half a dozen different controls, there will also be half a dozen different class factories which oversee the instantiation of the related objects. The job of the ComClassManager is to oversee this list of class factories.

Back in the project file, you'll see that the ActiveX control wizard also uses the XDesktopImpl unit. It's this unit which provides the nuts and bolts implementation of the XDesktop interface that's forward-declared in the AxDesktop_TLB file. The unusual syntax of the uses statement tells the project manager that it is this file that contains the implementation code for the specified CoClass. Source code to this unit is given in Listing 2.

The unit starts off (or ends, depending on your perspective) by creating a new TActiveXControl-Factory object in the unit initialization clause. It isn't necessary to maintain a reference for this object



```
procedure T%1:s.InitializeControl;
begin
  FDelphiControl := Control as %2:s;
end;

procedure T%1:s.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as I%1:sEvents;
end;

procedure T%1:s.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
begin
  { Define property pages here.  Property pages are defined by calling
    DefinePropertyPage with the class id of the page.  For example,
      DefinePropertyPage(Class_%1:sPage); }
end;

initialization
  TActiveXControlFactory.Create(
    ComServer,
    T%1:s,
    %2:s,
    Class_%1:s,
    %4:d,
    '%5:s',
    %6:s);
end.
```

➤ *Figure 1: The secret life of the Delphi IDE code generator. To customise the ActiveX control wizard you can modify these internal text resources within the IDE, provided you tread very carefully.*

because a list of class factories is maintained internally as I've already indicated. In order, the various arguments to the constructor identify the COM server object, the class which is to be 'manufactured' by this factory, the corresponding VCL class (TDesktop), a unique ClassID for the new class, a bitmap ID for toolbar images, a license string for control licensing (empty in this case because I told the ActiveX control wizard that I didn't want to use design-time licensing) and finally a set of status bits that are stored in the registry. After this call, the new class factor will be added to the list of factories managed by ComClassManager.

The actual declaration of the TXDesktop class begins with the syntax that you should now be familiar with: it specifies that TXDesktop is derived from TActiveXControl, and that it implements the IXDesktop interface, defined in the AxDesktop_TLB file.

As you can see from the code, the main purpose of the TXDesktop class is to act as a go-between, mapping any property, method or event calls down onto the underlying VCL-level component. Effectively, TXDesktop 'wraps' the component, providing an object which implements the functionality of TActiveXControl and can therefore be embedded in an ActiveX container.

I won't describe the Get_XXXX and Set_XXXX routines because you can see that they are just trivial wrappers around the underlying VCL component. The only exception here is the Get_Font and Set_Font routines. The TDesktop component only understands VCL-style TFont fonts, whereas the ActiveX component can only deal with OLE fonts. Thus some conversion is necessary. The GetOLEFont and SetOLEFont routines are implemented inside the AXCtrls unit and they allow you to map a TFont onto an OLE font object, and vice versa. In OLE, fonts are themselves implemented as automatable objects that implement an IFontDisp interface. Wheels within wheels!

The InitializeControl method of the implementation class is particularly important because it's here that you will typically want to execute any ActiveX-specific initialisation code after the underlying VCL component has been created. In other words, there might be some ActiveX specific initialisation you wish to place there that wouldn't be appropriate if the control was running natively under Delphi. As an example, the Delphi documentation states that you should bind the OLE event firing methods to the VCL control's event properties within the InitializeControl routine.

```
IPersistStreamInit
IPersistStorage
IOleObject
IOleControl
IOleInPlaceObject
IOleInPlaceActiveObject
IViewObject
IViewObject2
IPerPropertyBrowsing
ISpecifyPropertyPages
ISimpleFrameSite
```

➤ *Listing 3*

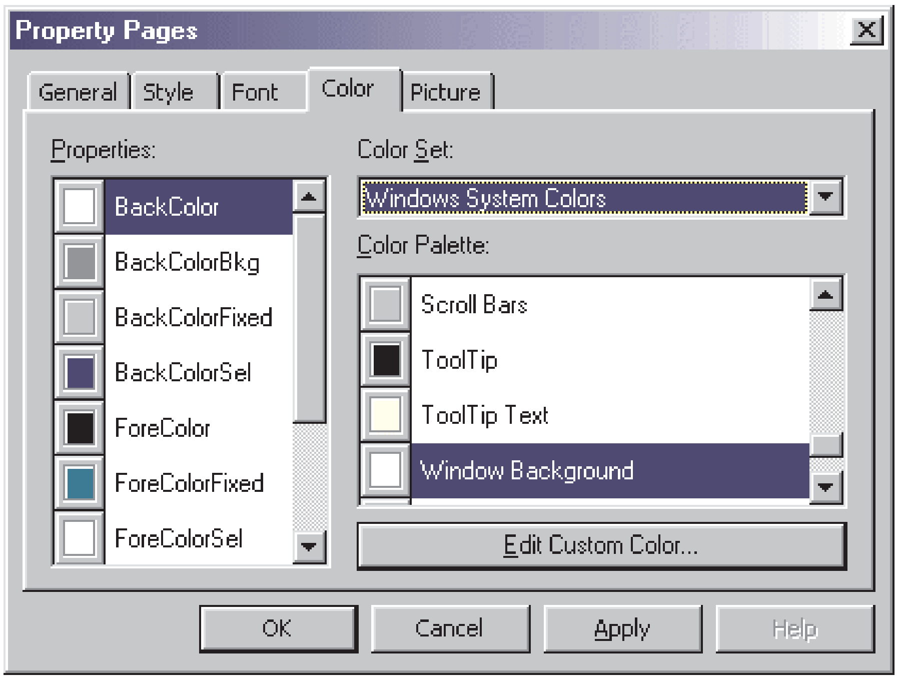
## The Undocumented MODULES Resource

Incidentally, you might be wondering how it is that the ActiveX control wizard is able to generate all this code. Where does it all come from? The fact is that most of the code generated by the wizard is pre-packaged boiler-plate code that's hidden away inside a number of undocumented resources within the Delphi 3 IDE. If you're feeling adventurous (or maybe you just don't like the way Inprise format their source code!), you can tweak these resources to customise the wizard's generated code to meet your exact needs. However, it should go without saying that care is needed and you shouldn't even think about it unless you create a lot of OCX controls using the wizard.

As an example of what the internal resources look like, take a look at Figure 1. This shows part of the MODULES resource (of type RT_RCDATA). As you can see, it's just a specially formatted text file that's been hidden away inside the IDE. Other internal resources you might care to examine are OCXRESOURCE and AXNEWSOURCE. All three of these 'files' (I use the term loosely) contain a series of code templates which are used by the IDE in various situations.

It should be obvious from examining Figure 1 that each template is made up of boiler-plate code containing embedded strings corresponding to (for example) the actual class name that's being generated. As the IDE processes the code template, the embedded string is replaced with the actual

➤ *Figure 2: Using Delphi, it's very easy to create great-looking property pages which are just as sophisticated as those to be found in Microsoft's ActiveX controls. This is from one of the components that ships with Visual Basic.*



class name. Thus, consider the following:

```
procedure T%1:s.InitializeControl;
begin
  FDelphiControl :=
    Control as %2:s;
end;
```

Comparing this code snippet with the generated code in Listing 2, we see that %1:s corresponds to the implementation class name for the OCX control (without its leading T) while %2:s corresponds to the class name of the underlying VCL control. The meta-arguments (for want of a better word) go all the way up to %6:s in the case of the call to

```
TActiveXControlFactory.Create
```

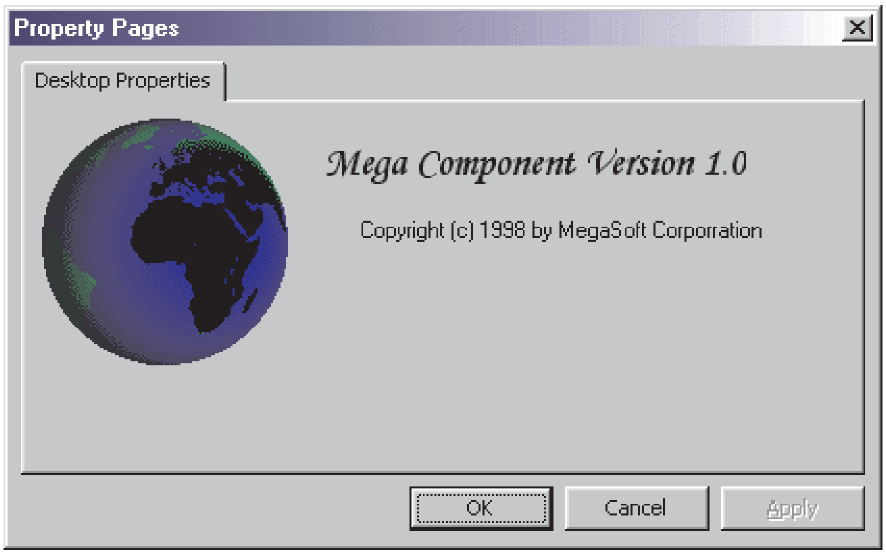
Finally, if you're planning to have a go at tweaking these resources, remember that each template is separated from the next by a '|' character. This is very important, leave them out and the IDE will be most disgruntled!

I mentioned earlier that the key DAX class in OCX implementation is TActiveXControl. It's this abstract class which implements all the functionality that's needed to embed a VCL control into an ActiveX container. The implementation code for TActiveXControl is contained in the AxCtrls unit which is nearly 3900 lines long! This is where the rubber hits the road! Aside from other things, this class provides support for events, property pages and property browsing, persistence, in place activation and embedding in a container. In order to provide all this functionality, the TActiveXControl implements all the interfaces shown in Listing 3.

In addition, as we've already seen, the implementation class must also implement the interface that's required by this specific control and it's for this reason that, as already mentioned, it makes sense to think of it as a go-between, linking the world of ActiveX controls to VCL land. Despite the 'under the hood' complexity of TActiveXControl, it has relatively few properties and methods. Its primary job is to implement all the required ActiveX control functionality within one object.

Before leaving the subject of the code generated thus far by the ActiveX control wizard, let me just draw your attention to one possible area of confusion in the AxDesktop_TLB unit. If you examine the source code here, you'll see that there appears to be a second TXDesktop class which is distinct from the class of the same name that's defined in the XDesktopImpl unit. What's the reason for this anomaly?

The TXDesktop class in the XDesktopImpl unit is, as previously stated, used to expose an underlying VCL component to the outside world as an ActiveX compatible control. The TXDesktop class in the AxDesktop_TLB unit does exactly the reverse job: it enables Delphi to host an ActiveX control as if it were a native VCL component, placing the component onto (by default) the ActiveX page of the component palette. Because the emphasis here is on creating ActiveX components rather than hosting them, we won't delve into this any further, but it's important to be aware that there are effectively two classes with the same name and exactly opposite roles. If you don't bear this in mind, then things will get very confusing!


Property Pages

Desktop Properties

Mega Component Version 1.0

Copyright (c) 1998 by MegaSoft Corporation

➤ *Figure 3: And here's one I prepared earlier, complete with spinning globe, courtesy of an animated GIF file and a freeware GIF viewer. Next month, I'll show you how, amongst other things, to tie the controls on a property page to your ActiveX properties.*

## Adding Property Pages To Your OCX

Up until now, I haven't covered the subject of property pages. The DAX framework makes it really very easy to add custom property pages to your control. Within the XDesktopImpl unit (Listing 2), you'll see that the ActiveX control wizard overrides the DefinePropertyPages method whether or not you plan to add property pages to your control. It places a comment into the empty method, providing a hint on how to define your own property page. It's not terribly clear from the code, but the single argument is actually a procedural pointer to a routine that takes a single argument, a GUID. Assuming that you've already created a property page for your control, you can just call this passed procedure, giving it the GUID which corresponds to the property page you want to use.

So let's go ahead and add a property page to our desktop tweaking component. First we select New from the File menu, choose the ActiveX page in Delphi's object repository and then click the Property Page icon. Delphi will create a new, empty property page together with a corresponding form unit and add it to the ActiveX project. The great thing about implementing property pages with Delphi is that the design process is identical to that of creating a normal Delphi form. Just add the required components to the form, glue them all together with some event handling code and off you go.

If you examine the newly created property page unit, you'll see that Delphi has created a new GUID for the exclusive use of the property page. It's this GUID which you need to specify when calling `DefinePropertyPage` from the `DefinePropertyPages` method of the implementation class. Doing this will 'connect' your property page to the associated ActiveX control. You will also need to add the property page unit to the `uses` clause of the implementation class unit.

Once this is done, you can just go ahead and design your properties page. Bear in mind that the `OK`, `Cancel` and `Apply` buttons will not appear on the design-time form. These buttons only appear on the runtime property dialog as viewed from the 'containing' development system, see Figure 2 for an example of what I mean. Also bear in mind that a control can (and typically will) be associated with more than one page, each page will appear with its own 'tab' on the overall properties dialog. To implement multiple property pages, just repeat the above process and call `DefinePropertyPage` from the `DefinePropertyPages` method for as many property page GUIDs as you've got.

Incidentally, being able to create the property page for your ActiveX control using Delphi is a real opportunity to rub the noses of your C++ friends in the dirt. Errrm, sorry. I meant to say it's a real opportunity to demonstrate how easy it is to create cool visual effects with Delphi, which amounts to pretty much the same thing. In Figure 3, the picture of Earth is actually spinning round, courtesy of the freeware `TGIFImage` component created by Theodor Kleynhans. This component supports animated GIF files and makes it very easy to add knock-em-dead effects to About boxes and property pages. If you don't have this component, then get it! You can download it from Theodor's website at: http://members.gem.co.za/~theodor/index.html *[But do beware the GIF licensing nightmare! Ed]*.If you want to build the OCX control on this month's disk, then you'll need to download and install this component first.

That said, the property page in Figure 3 has a conspicuous lack of editable properties. In next month's instalment, I'll continue our discussion of ActiveX property pages by showing you how to associate ActiveX properties with the controls on a property page. See you then.

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