The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Property Editor Question

Q I am in the process of writing a property editor. How do I get the class name of the object whose properties I am editing?

A The TPropertyEdit class (and all its descendants) defines a GetComponent method to give you access to the component selected on the form designer (or one of the components, if there are any components selected). So GetComponent(0).ClassName will give the class name and GetComponent(0).ClassType gives you the class reference.

Page Control Query

Q Is there an easy way of finding the active tab selected in a TPageControl component?

A You can either index into the page control's read-only Pages property with its ActivePageIndex property:

```
PageControl1.Pages[
  PageControl1.ActivePageIndex]
```

or simply use the ActivePage property (PageControl1.ActivePage).

Both of these return the TTabSheet object that is active in the page control. I think the key word there is page, instead of tab.

Customised Alias

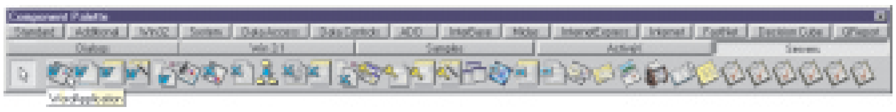
Q I need to access one of two similar local databases, depending on a given parameter when the application starts (read from an INI file). The alias name will be the same for both but, inside the alias, I need to redirect the path to one directory or the other. How can I programmatically alter the path of an alias for a standard database?

A Drop a TDatabase object on the form and set the properties like this (you can use the Database Editor by right-clicking the TDatabase object). Set AliasName to be your BDE alias name, chosen from the list. Set DatabaseName to be some different string (note that in the Database Editor, the DatabaseName property is set with the edit control labelled Name). Now make sure your TTable or TQuery components use the value of the database component's DatabaseName property, rather than the real BDE alias.

At runtime, whilst the database component is not connected (Connected is False), you can use:

```
Database1.Values['PATH'] :=
  'c:\TheRequiredPath';
```

When you open a table or query, the database component will then connect to the appropriate database.



➤ *Figure 1: Delphi 5's COM server components.*

Customised Status Bar

Q Is it possible to change the colour of the text in a status bar's panels?

A I couldn't find an easy way, so the best I can think of is to set the Style property of the TStatusPanel in question to psOwnerDraw, then use the OnDrawPanel event to do whatever customised drawing you want.

COM Server Components

Q Before purchasing Delphi 5, I am interested in finding out a little more about the components on the Servers page of the component palette. How many components are there, and what exactly do they do?

A There are 32 COM server components on the Servers page. They represent the COM objects surfaced from the applications that make up Microsoft Office 97, and described in the various type libraries that accompany

➤ *Table 1*

TWordApplication	TWordDocument	TWordFont
TWordParagraphFormat	TWordLetterContent	TBinder
TExcelQueryTable	TExcelApplication	TExcelChart
TExcelWorksheet	TExcelWorkbook	TExcelOLEObject
TDoCmd	TAccessHyperlink	TAccessForm
TAccessReport	TAccessReferences	TPowerPointApplication
TPowerPointSlide	TPowerPointPresentation	TOutlookApplication
TAppointmentItem	TContactItem	TJournalItem
TMailItem	TMeetingRequestItem	TNoteItem
TPostItem	TRemoteItem	TReportItem
TTaskItem	TTaskRequestItem	

that package. Figure 1 shows my undocked component palette's Servers page. Table 1 is a list of all the components installed there.

The fact that all these components are installed onto the component palette is very convenient, but not anything we could not do for ourselves. You see, Delphi 5's type library importer goes considerably further than those that were supplied with Delphi 3 and 4. When you import a type library now, as well as manufacturing a unit containing all the interfaces, dispatch interfaces, enumerations and so on, it also manufactures components.

For every coclass in the type library, Delphi optionally creates a component class based upon the new `TOleServer` class to provide simplified access to COM Automation. A component registration call is also added to the unit's `Register` routine. You can see the IDE checkbox that dictates whether this happens in Figure 2. The equivalent command line tool TLIBIMP.EXE has a `-L+` command line switch that does the same thing.

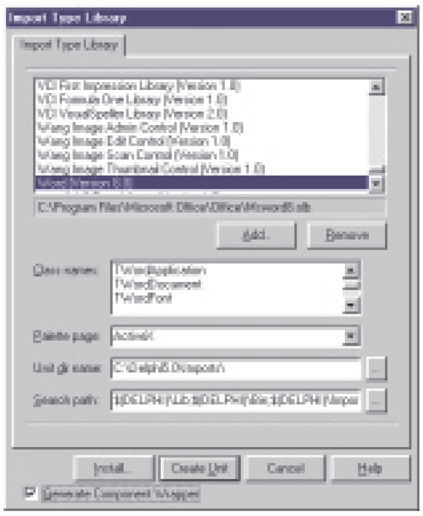
What Inprise have done is run the type library importer across all the type libraries that come with Office 97, and installed all the resulting components on the Servers page of the component palette (the component glyphs used also come from the type libraries). Compiled versions of all these type library import units are stored in Delphi's Imports directory, whilst the source files can be found in Delphi's OCX\Servers directory.

In short, it doesn't *really* matter that the components exist on the palette. If they didn't, Delphi 5 offers the ability to make them exist. What does matter is the introduction of the very handy `TOleServer` class, and the improvements to the type library importer. So what do these `TOleServer` descendants do?

Firstly, they offer various ways of connecting to the target COM interface. The `ConnectKind` property can be set to `ckRunningInstance` to ensure that no new instance is started. The component will connect to an instance that is already running, or if none can be found will raise an exception. A value of `ckNewInstance` will start a new instance to connect to, even if one is already running. `ckRunningOrNew` will try to connect to a running instance, but if none can be found will start a new instance. `ckRemote` attempts to connect to an instance of the server running on the machine named by the `RemoteMachine` property. Finally, `ckAttachToInterface` will not bind to a server, but instead waits for you to supply an appropriate interface reference using a call to the `ConnectTo` method.

The component defines its own versions of all the methods and properties defined in the corresponding COM interface. Any methods that have optional parameters defined in the type library have multiple overloaded versions to allow optional parameters to be omitted. Additionally, if the coclass specifies that it has an events interface, the component will surface as many of the events as it can into the Object Inspector. This means you can easily handle COM server events without worrying about connection point interfaces and all the rest of the misery that goes with it.

Just to give you an idea of how these COM server components operate, and how they differ from normal COM programming, here


Import Type Library
Import Type Library
VCI First Impression Library (Version 1.0)
VCI Formula One Library (Version 1.0)
VCI VisualSpeller Library (Version 2.0)
Wang Image Admin Control (Version 1.0)
Wang Image Edit Control (Version 1.0)
Wang Image Scan Control (Version 1.0)
Wang Image Thumbnail Control (Version 1.0)
Word (Version 8.0)
C:\Program Files\Microsoft Office\Office\Msword8.olb
Add... Remove
Class names: TWordApplication
TWordDocument
TWordFont
Palette page: ActiveX
Unit dir name: C:\Delphi5.0\Imports\
Search path: $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Impor
Install... Create Unit Cancel Help
Generate Component Wrapper

➤ *Figure 2: Delphi 5 importing a type library.*

are some comparative listings (from the sample Delphi 5 project WordAuto.Dpr). Listing 1 shows some late bound Automation of Microsoft Word using `Variant` variables. The code starts a fresh copy of Word, making it visible, makes a new document with a bit of text in, saves the document (without adding the filename to Word's file history list) and then shuts Word. The call to `SaveAs` is quite short thanks to Delphi supporting optional named arguments with Automation through a `Variant`.

Automation with a `Variant` is quite easy to write, but normally less efficient than using direct interfaces. Also, you have to wait until runtime to find out whether you spelt any methods or properties incorrectly.

Listing 2 shows the equivalent logic expressed, but using early bound Automation with direct v-table access through interface references. As you can see, optional arguments have to be specified, although you can use `EmptyParam` variable (in Delphi 4 or later) to specify the default value. Consequently, the calls to `SaveAs` and `Quit` suddenly become longer.

Listing 3 works after dropping a `TWordApplication` called `WordApplication` on the form, with its `ConnectKind` property set to `ckNewInstance`, then dropping a `TWordDocument` called `WordDocument` on the form with `ConnectKind` set to `ckAttachToInterface`.

➤ *Listing 1*

```
uses
  ComObj;
procedure TForm1.btnVariantClick(Sender: TObject);
var WordApplication, WordDocument: Variant;
begin
  WordApplication := CreateOleObject('Word.Application');
  WordApplication.Visible := True;
  WordDocument := WordApplication.Documents.Add;
  WordApplication.Selection.TypeText('Hello world');
  WordDocument.SaveAs(FileName := 'C:\Doc.Doc', AddToRecentFiles := False);
  WordApplication.Quit
end;
```

The differences between Listing 2 and Listing 3 are the lack of interface reference variable declarations (the components have their own declarations in the form class), and the shorter parameter lists to some of the method calls. The Quit method has three parameters defined in the interface, but since all are optional, the TWordApplication component defines four versions of it (see Listing 4). The document's SaveAs method is defined a whopping total of twelve times with different parameter lists.

```
procedure TForm1.btnInterfaceClick(Sender: TObject);
var
  WordApplication: _Application;
  WordDocument: _Document;
  FileName, VariantFalse: OleVariant;
begin
  WordApplication := CoWordApplication.Create;
  WordApplication.Visible := True;
  WordDocument := WordApplication.Documents.Add(EmptyParam, EmptyParam);
  WordApplication.Selection.TypeText('Hello world');
  FileName := 'C:\Doc.Doc';
  VariantFalse := False;
  WordDocument.SaveAs(FileName, EmptyParam, EmptyParam, EmptyParam, VariantFalse,
    EmptyParam, EmptyParam, EmptyParam, EmptyParam, EmptyParam, EmptyParam);
  WordApplication.Quit(VariantFalse, EmptyParam, EmptyParam);
end;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
procedure TForm1.btnComponentClick(Sender: TObject);
var
  FileName, VariantFalse: OleVariant;
begin
  WordApplication.Visible := True;
  WordDocument.ConnectTo(WordApplication.Documents.Add(EmptyParam, EmptyParam));
  WordApplication.Selection.TypeText('Hello world');
  FileName := 'C:\Doc.Doc';
  VariantFalse := False;
  WordDocument.SaveAs(FileName, EmptyParam, EmptyParam,
    EmptyParam, VariantFalse);
  WordApplication.Quit;
end;
```

Fancy Font Dialog

Q I have added a TFontDialog component to a form and have encountered the following problem. TFontDialog has three events available, OnApply, OnClose and OnShow. I find that OnShow works fine, and clicking the dialog's Apply button triggers the OnApply event just fine. However, OnClose is triggered for both OK and Cancel, meaning that I cannot distinguish between these buttons. Am I missing something or is this a Delphi bug?

A The way this component is designed to be used is as follows. OnShow and OnClose are designed to allow you to set up any special stuff you might need when the dialog is first shown, and then tidy it up when the dialog closes. That's it.

For actually taking the user's font request and dealing with it, you have two choices. Firstly, if you do not set up an OnApply event handler (and do not have the fdApplyButton option set), you will not get an Apply button. This means that you call the dialog's Execute method and decide whether to do anything based upon this method's Boolean return value.

The second option is where you have an OnApply method, and so get an Apply button. Here you put the code that deals with font changes in the OnApply event handler. To invoke the dialog, you still call the Execute method and, if it returns True, manually execute the OnApply event handler. The code in there can check things out to ensure pointless code is not executed, such as setting an object's font that has already been set by the user pressing the Apply button.

You are in this latter case of having an Apply button. A simple sample project called FontDlg.Dpr is included on this month's CD-ROM. It has a speedbutton to invoke a font dialog, and a couple of controls on the form that have fonts. I chose an edit control and a rich edit. If the edit control is active when the button is pressed, the code will set its font (assuming Cancel is not pressed in the font dialog). If the rich edit is active, the selected text will have its font changed. The code can be seen in Listing 5.

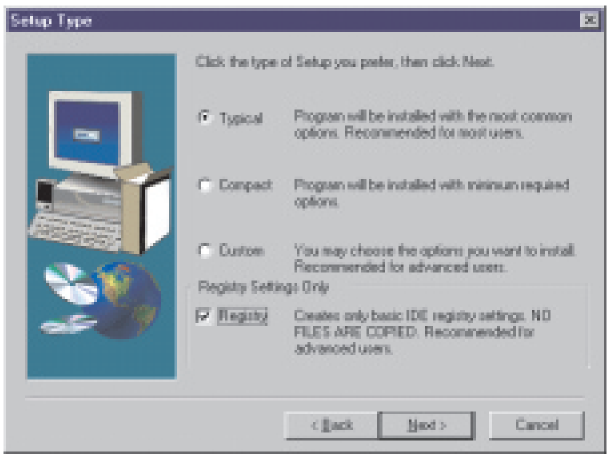
When the speedbutton is clicked, it calls the dialog's Execute method. Assuming it returns True (OK was pressed), and the OnApply event handler exists, the OnApply handler is called. Of course it could already have been called thanks to the Apply button being pressed, but this simple example doesn't check before re-executing the code in that event handler.

```
procedure Quit; overload;
procedure Quit(var SaveChanges: OleVariant); overload;
procedure Quit(var SaveChanges: OleVariant; var OriginalFormat: OleVariant);
  overload;
procedure Quit(var SaveChanges: OleVariant; var OriginalFormat: OleVariant;
  var RouteDocument: OleVariant); overload;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  if ActiveControl is TEdit then
    FDlg.Font := TEdit(ActiveControl).Font
  else if ActiveControl is TRichEdit then
    TRichEdit(Activecontrol).SelAttributes.Assign(FDlg.Font);
  //Launch dialog. If it returns True, and
  //we have an OnApply handler, call it
  if FDlg.Execute and Assigned(FDlg.OnApply) then
    FDlg.OnApply(FDlg, 0)
end;
procedure TForm1.FDlgApply(Sender: TObject; Wnd: HWND);
begin
  //Could check here that there is in fact a need to apply
  //the font, by comparing, but for brevity, I'll skip it
  if ActiveControl is TEdit then
    TEdit(ActiveControl).Font := TFontDialog(Sender).Font
  else if ActiveControl is TRichEdit then
    TRichEdit(ActiveControl).SelAttributes.Assign(TFontDialog(Sender).Font)
end;
```


Setup Type

Click the type of Setup you prefer, then click Next.

[✓ Typical] Program will be installed with the most common options. Recommended for most users.

Compact Program will be installed with minimum required options.

Custom You may choose the options you want to install. Recommended for advanced users.

Registry Settings Only

[✓ Registry] Creates only basic IDE registry settings. NO FILES ARE COPIED. Recommended for advanced users.

➤ *Figure 3: Registry-only installation.*

Multi-User Delphi Installation

Q I installed Delphi 4 on a Windows NT4 workstation while logged in as one user, making sure the shortcuts are put in the correct place to be seen by all users. When I log back into the same machine as another user and run Delphi none of the components appear and all the search directories vanish. I can reinstall all the components and set the directories again, but this is not an ideal situation. Is there a known solution to this?

A Each user has their own area in the Windows registry. As various different people log in, the `HKEY_CURRENT_USER` hive maps itself onto to a different branch under `HKEY_USERS`. Delphi stores all its information in the registry between sessions, including what component packages are installed. When you log in as another user, you miss out on the previous user's session information which is no longer accessible. This means that when Delphi tries to read the list of installed component packages from the registry, it finds none, and so the component palette stays empty.

You have three ways of fixing this problem. Firstly, you could reinstall when logged in as the other user, making sure you use exactly the same installation options and directories as before. This is by no means an ideal situation.

The second option applies to Delphi 4 and later. The installation program offers a registry only installation option for exactly this type of situation. Once you have installed Delphi normally as one user, log in as another user and install again. This time however, remember to check the `Registry` installation option (see Figure 3). Assuming you keep all the other installation options the same, this will populate this second user's area of the registry with all the Delphi settings needed to make it work, copying a minimum set of files across.

The third option is an alternative way of doing the same as the second option, and applies to any 32-bit version of Delphi, not just 4. Log in as the original user, launch REGEDIT.EXE and navigate to Delphi 4's area in the registry:

```
HKEY_CURRENT_USER\Software\
  Borland\Delphi\4.0
```

Now export the whole Delphi 4 registry hierarchy to a `.REG` file by choosing `Registry | Export Registry File...`. Log in as another user, locate the `.REG` file in Windows Explorer and double click it. This will merge all those entries into the current user's area of the registry.

Components On The Clipboard

Q I am trying to put multiple components onto the clipboard. `ClipBoard.SetComponent` works fine for a singular component, but fails for more than one. Is there an easy fix or should I hack the VCL code?

A Well, in this case you don't need to do much hacking, as I've done it for you ☺. Before embarking on the solution, let's get a little background information on the clipboard, and writing data to it. Incidentally, all this applies to all versions of Delphi.

Any unique form of data that goes on the clipboard has to be marked with a registered format. A number of standard formats exist, such as text (`CF_TEXT`), bitmap (`CF_BITMAP`), palette (`CF_PALETTE`), metafile picture (`CF_METAFILEPICT`) and so on. The constants for these formats are defined for you in

➤ *Listing 6*

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  I: Integer;
  Buf: array[0..255] of Char;
  Fmt: String;
begin
  ListBox1.Items.Clear;
  for I := 0 to ClipBoard.FormatCount - 1 do begin
    GetClipboardFormatName(ClipBoard.Formats[I], Buf, SizeOf(Buf));
    Fmt := StrPas(Buf);
    if Fmt = '' then
      case ClipBoard.Formats[I] of
        CF_TEXT:             Fmt := 'CF_TEXT';
        CF_BITMAP:           Fmt := 'CF_BITMAP';
        CF_METAFILEPICT:     Fmt := 'CF_METAFILEPICT';
        CF_SYLK:             Fmt := 'CF_SYLK';
        CF_DIF:              Fmt := 'CF_DIF';
        CF_TIFF:             Fmt := 'CF_TIFF';
        CF_OEMTEXT:          Fmt := 'CF_OEMTEXT';
        CF_DIB:              Fmt := 'CF_DIB';
        CF_PALETTE:          Fmt := 'CF_PALETTE';
        CF_PENDATA:          Fmt := 'CF_PENDATA';
        CF_RIFF:             Fmt := 'CF_RIFF';
        CF_WAVE:             Fmt := 'CF_WAVE';
        CF_OWNERDISPLAY:     Fmt := 'CF_OWNERDISPLAY';
        CF_DSPTEXT:          Fmt := 'CF_DSPTEXT';
        CF_DSPBITMAP:        Fmt := 'CF_DSPBITMAP';
        CF_DSPMETAFILEPICT:  Fmt := 'CF_DSPMETAFILEPICT';
        CF_UNICODETEXT:      Fmt := 'CF_UNICODETEXT';
        CF_ENHMETAFILE:      Fmt := 'CF_ENHMETAFILE';
        CF_HDROP:            Fmt := 'CF_HDROP';
        CF_LOCALE:           Fmt := 'CF_LOCALE';
        CF_DSPENHMETAFILE:   Fmt := 'CF_DSPENHMETAFILE';
      else
        Fmt := 'Unknown clipboard format'
      end;
    ListBox1.Items.Add(Fmt)
  end;
end;
```

```
var
  CF_COMPONENTS: Word;
procedure ClipBoardSetComponents(Components: array of
  TComponent);
var
  ClipStream, TxtStream: TMemoryStream;
  Loop: Integer;
  Data: THandle;
  DataPtr: Pointer;
begin
  ClipStream := TMemoryStream.Create;
  try
    for Loop := Low(Components) to High(Components) do
      ClipStream.WriteComponent(Components[Loop]);
    { Reset stream pointer to beginning }
    ClipStream.Position := 0;
    { Allocate memory block to give to clipboard }
    Data := GlobalAlloc(GMEM_MOVEABLE and GMEM_DDESHARE,
      ClipStream.Size);
    if Data = 0 then
      OutOfMemoryError;
    { Lock it for writing }
    DataPtr := GlobalLock(Data);
    try
      ClipStream.Read(DataPtr^, ClipStream.Size);
    finally
      { Unlock it }
      GlobalUnlock(Data)
    end;
    { Clipboard takes ownership of memory block, so we can
      forget it }
    ClipBoard.SetAsHandle(CF_COMPONENTS, Data);
  finally
    ClipStream.Free;
  end;
end;
initialization
  CF_COMPONENTS := RegisterClipboardFormat('Delphi
    Components');
end.
```

➤ *Listing 7*

the Windows unit. Most clipboard formats also have a textual description available (apart from some standard ones).

The Delphi ClipBrd unit defines two new clipboard formats. CF_PICTURE is defined with the string Delphi Picture and CF_COMPONENT is defined with the string Delphi Component. At any time, data may be stored in the clipboard in a number of formats simultaneously. ClipList.Dpr is a simple project that has a listbox on it. A timer is used to trigger some code every half a second that enumerates the clipboard formats, getting their numeric values, and then asking Windows for their textual descriptions. This information is fed into a listbox, so that as you copy things onto the clipboard you can see what formats are available (Listing 6 shows the simple code).

My first thought on the subject was that the Delphi form designer can copy multiple components onto the clipboard, so I wondered what clipboard format was being used. Running this ClipList program and then copying some components onto the clipboard from a form designer, it told me that Delphi stores components on the clipboard in several formats. As well as text format (CF_TEXT), Delphi uses a format described as Delphi Components. This is *just* different enough to the format registered by the ClipBrd unit (Delphi Component) to mean that extra coding will be required.

The code which implements the solution registers a clipboard format with the same string as used by Delphi, and assigns it to a clipboard format variable, CF_COMPONENTS. That's a good start, but how does the clipboard take and give information? To add arbitrary information to the clipboard requires you to allocate memory, using GlobalAlloc with specified flags. Then you copy the data in question into the memory block. But since GlobalAlloc returns a memory handle, you have to first call GlobalLock to get a pointer to the memory, copy the data into the memory block, and then call GlobalUnlock afterwards.

So once we have some data to put in the clipboard, we know what to do, but we first need to work out how to get several components into a block of memory that we can work with. The routine ClipBoardSetComponents takes an open array of TComponent references. It loops through the components passed in to it and then passes them on to the WriteComponent method of a TMemoryStream object.

This means we now have several memory versions of the components in question. We can allocate a suitably sized block of memory, copy the contents of the stream to it (with appropriate lock and unlock calls), and pass it off to the clipboard. The ClipBoard object has a SetAsHandle method to achieve this. Listing 7 shows what we have here.

With this code in place, an application can copy some components to the clipboard with a call such as the following:

```
ClipBoardSetComponents(
  [Label1, Memo1,
   ListBox1]);
```

With the components on the clipboard, you can switch over to a copy of Delphi, and paste the components on to the form designer. That deals with the question as it was posed, but presumably the questioner also wants to know how to read multiple components from the clipboard. This means we need an implementation of ClipBoardGetComponents to complete the pair.

The logic will go like this. We have to open the clipboard for reading, then ask the clipboard for the data in the CF_COMPONENTS format. It gives the data over as a memory handle that needs locking (and later unlocking). Once we have access to the raw data, the memory block should be copied over to a VCL memory stream. With the stream ready to be read from, we can repeatedly read components from it, setting their owner and parent (if appropriate) until there are no more left.

To see if there are any more components left to be read requires some care. The block of memory owned by the clipboard will not exactly match the sum of the size of all components stored there. This is because Windows memory allocations are typically rounded up, leaving several spare bytes. To check if another component is there to be read, the code checks for the presence of the standard VCL stream signature, TPF0. Listing 8 shows a possible implementation.

There are possible problems that could crop up. For example if you try and make, say, a form own a component with the same name as one it already owns, an

```
procedure ClipBoardGetComponents(Owner, Parent: TComponent);
var
  Data: THandle;
  DataPtr: Pointer;
  ClipStream: TMemoryStream;
  Comp: TComponent;
const
  FilerSignature: array[1..4] of Char = 'TPF0';
begin
  ClipBoard.Open;
  try
    Data := GetClipboardData(CF_COMPONENTS);
    if Data = 0 then
      Exit;
    DataPtr := GlobalLock(Data);
    if DataPtr = nil then
      Exit;
    try
      ClipStream := TMemoryStream.Create;
      try
        ClipStream.WriteBuffer(DataPtr^, GlobalSize(Data));
        ClipStream.Position := 0;
        try
          repeat
            { Check for VCL stream signature before
              proceeding }
            if PLongint(Longint(ClipStream.Memory) +
              ClipStream.Position)^ <>
              Longint(FilerSignature) then
              Exit;
            Comp := ClipStream.ReadComponent(nil);
            if Comp is TControl then
              TControl(Comp).Parent :=
                Parent as TWinControl;
            try
              Owner.InsertComponent(Comp)
            except
              Comp.Free;
              raise
            end
          { We will probably leave thanks to the signature }
          { before check this condition is met, as Windows }
          { memory is rounded up in size, so there will be }
          { slack }
          until ClipStream.Position = ClipStream.Size
        except
          Exit
        end
      finally
        ClipStream.Free
      end
    finally
      GlobalUnlock(Data)
    end
  finally
    ClipBoard.Close
  end
```

➤ *Listing 8*

exception will be raised. Also, there are requirements that must be met before it works. Before you can pass an instance of a component into an application, its class must be passed to a call to either `RegisterClass` or `RegisterClasses`. However, having done this, you can copy a selection of components from a form designer onto the clipboard, and then paste the contents of the clipboard into an application, using a call to `ClipBoardGetComponents`.

There are restrictions on how successfully this works. When Delphi copies components onto the clipboard, it copies a textual version as well as the binary streamed component version. This means you can copy from a form designer and paste into Notepad (and *vice versa*). My code only stores the binary version of components onto the clipboard.

Also, when copying components to the clipboard, `ClipBoard-SetComponents` will not cater for children of anything copied, unlike a form designer. The stream's `WriteComponent` method writes out the specified component, along with anything it owns, but not any of its children. Both these issues could be worked out, given some time, but hopefully this code should solve the immediate problem at hand.

A simple demo program, ClipComp.Dpr, has a button that copies some components from a panel onto the clipboard and then pastes them back onto the form. Listing 9 shows the button event handler and shows that the original components have to have their `Name` properties set to blank strings to avoid exceptions. The `Clip-BoardSetComponents` and `ClipBoard-SetComponents` routines are defined in the `ClipHelp` unit.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ClipBoardSetComponents([Label1, Edit1, Image1]);
  Label1.Name := '';
  Edit1.Name := '';
  Image1.Name := '';
  ClipBoardGetComponents(Self, Self)
end;
...
initialization
  RegisterClasses([TImage, TEdit, TLabel])
end.
```

➤ *Listing 9*

Custom TLabel Component

Q I have some `TLabel` fields showing my homepage and email information. When I move the mouse cursor on these fields, the `OnMouseMove` event handler changes their font attributes to look like they are clickable hyperlinks. However, when my mouse is *not* on top of these fields, I want their font attributes reset back to their original values. What should I do?

A Whilst the `OnMouseMove` event allows you to change the behaviour of a control when the mouse moves over it, there is no event that triggers when the mouse leaves. However, if you make a new component based upon `TLabel` (or even `TCustomLabel`, if many of the `TLabel` properties are of no relevance) you can handle some dedicated component messages which should help you fulfil your requirements.

Delphi controls have two messages that are sent to them when the mouse is entering the control or leaving it: `cm_MouseEnter` and `cm_MouseLeave`. If you write a message handler for each of these, then the component can look after the font attribute changing for you. Have a look at the code in Listing 10 from the `HLLabel` unit, which compiles in any version of Delphi. It shows a simple class inheriting from `TCustomLabel` with the two message handlers in place. Two properties dictate what colour the hyperlink will display as and what font style will be used.

The component also reacts to being clicked on. In the overridden `Click` method, the URL displayed in the label's caption is passed along to a call to `ShellExecuteEx` in

```
THyperLinkLabel = class(TCustomLabel)
private
  { Property state holders }
  FHyperlinkColour, FOldColour: TColor;
  FHyperlinkStyle, FOldStyle: TFontStyles;
protected
  procedure Click; override;
  { VCL message handlers }
  procedure CMMouseEnter(var Msg: TMessage);
    message cm_MouseEnter;
  procedure CMMouseLeave(var Msg: TMessage);
    message cm_MouseLeave;
public
  constructor Create(AOwner: TComponent); override;
published
  { New properties }
  property HyperlinkColour: TColor
    read FHyperlinkColour write FHyperlinkColour
    default clBlue;
  property HyperlinkStyle: TFontStyles
    read FHyperlinkStyle write FHyperlinkStyle;
  { Make these hidden properties/events show up on the
    Object Inspector }
  property Caption;
  property Font;
  property ParentShowHint;
  property ShowHint;
  property OnClick;
end;
constructor THyperLinkLabel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FHyperlinkColour := clBlue;
  FHyperlinkStyle := [fsUnderline];
  FOldColour := Font.Color;
  FOldStyle := Font.Style;
end;
procedure THyperLinkLabel.Click;
var
  { For PChar version of URL. Written for Delphi 1
    compatibility }
  URLBuf: array[0..255] of Char;
begin
  inherited Click;
  StrPCopy(URLBuf, Caption);
  ShellExecute(Application.Handle,
    nil, URLBuf, nil, nil, sw_ShowNormal)
end;
procedure THyperLinkLabel.CMMouseEnter(var Msg: TMessage);
begin
  inherited;
  FOldStyle := Font.Style;
  FOldColour := Font.Color;
  Font.Style := FHyperlinkStyle;
  Font.Color := FHyperlinkColour;
end;
procedure THyperLinkLabel.CMMouseLeave(var Msg: TMessage);
begin
  inherited;
  Font.Style := FOldStyle;
  Font.Color := FOldColour;
end;
procedure Register;
begin
  RegisterComponents('Clinic', [THyperLinkLabel]);
```

➤ *Listing 10*

order to get the installed web browser to navigate to it.

The logic in the listing is (hopefully) short and simple enough to be self-explanatory. Just install the component unit and try it out.

Figure 4 shows a simple form at runtime with two `THyperLinkLabel` components on it. As you can see, the first one looks like a normal label because the mouse is not on it. The second label has the mouse pointing at it and so looks like a hyperlink.

User-Defined Exceptions

Q I have a requirement to create a new exception class that contains more than just the `Message` property. I have tried to inherit this new class from the `Exception` class, but my attempts to override the constructor fail because it is a static method. How do I do it?

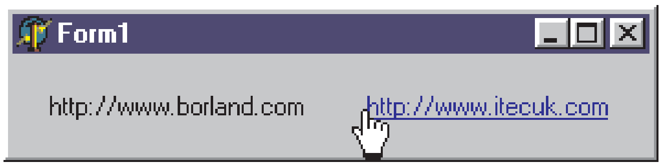
A You only need to override a virtual (polymorphic) constructor, which the `Exception` class does not have. `TComponent` has a virtual constructor, to allow the generic form streaming code in the VCL to recreate a form, and all the components on it, when it is either loaded at design-time or created at runtime. No other VCL classes have polymorphic constructors.

Since `Exception` (or anything inherited from it in the VCL) has a static constructor, you just redefine it. There is no need to use `virtual` or `override` (required only in the case of polymorphic methods). If you wish to chain back on to the original constructor, you can do this by using the inherited reserved word in conjunction with a call to the old constructor, with the relevant parameters. If you have the VCL source, you should carefully examine the definition of `EDBEngineError` in the `DBTables` unit (or `DB` unit in Delphi 1 and 2), which is reproduced in Listing 11.

Listing 12 shows a custom exception object with a modified constructor, along with the constructor's implementation and a sample statement raising the exception.

More On Active OLE Object

Q I read your answer to the *Active OLE Object* question in Issue 46 with great interest because it seemed to address some of the problems I am currently trying to resolve. However, it doesn't quite answer everything and I wonder if you could give me some ideas



➤ *Figure 4: Hyperlink-style label.*

on how to extend this to a DCOM environment and also using an NT service. I will explain briefly what I'm trying to achieve.

I am developing an application which involves an NT service application that is constantly communicating with some external hardware. This application stores data in a database and also acts as an automation server to the GUI client application.

The client will run either on the same machine as the server, or on a remote machine that connects using Dial-Up Networking. The client will be able to view various reports from the database, but also communicate with the external hardware via COM interfaces provided by the server.

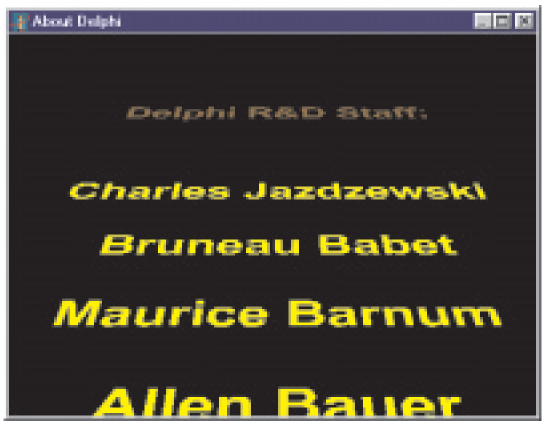
I have an early implementation of this working except that the server is currently a standard executable. The client uses `CreateRemoteComObject` to 'connect' to the server and this will launch the server if it is not already running.

My question is, how do I achieve this if the server is a service? The service should be set up to start automatically, so I presume that the client should be able to use the ROT to connect to it, but can you do this with a remote machine?

A This is a good question. My current understanding is that the ROT is only valid for the current machine, so we will look at a slightly different solution.

One way around it might be to have a small DCOM sentinel application, whose job is to be kick-started by the local/remote clients. It could then talk to the single service-hosted Automation object to get information to pass back to clients (maybe in `Variant` byte arrays or whatever). That way, when the clients are started from various machines, your server machine will have small DCOM applications popping up every now and again, and one main Automation object in a service, doing the work.

➤ *Figure 5: Star Wars credits.*



Delphi R&D Staff:

Charles Jazdzewski

Bruneau Babet

Maurice Barnum

Allen Bauer

Delphi 5 Easter Eggs

Q Does Delphi 5 have any new Easter Eggs?

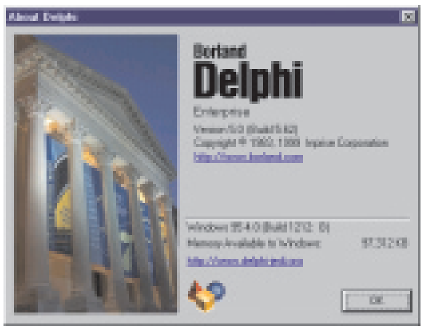
A The Delphi Easter Eggs have always been found in the About box. Delphi 5 still supports holding the Alt key down and typing `DEVELOPERS`, `TEAM` or `QUALITY` to get varying lists of Inprise employees. However, I am pleased to say that the appearance of these credit lists is now much improved. Instead of simply scrolling some names from the bottom to the top of the About box, they have been revamped with OpenGL support (if installed, otherwise they look the same as before).

The list now looks very much like the introductory scrolling text scene in the Star Wars films (see Figure 5). In fact, so much is it based on Star Wars that the scrolling does not actually appear in the About box itself, but in a separate form that is placed exactly over the About box (notice that the form in Figure 5 has a maximise icon on its caption). According to WinSight, the form's class name is `TSWForm`. No prizes for guessing what `SW` stands for.

Whilst this OpenGL scrolling is in itself quite nice, you should find it more fun when you use the cursor keys to spin the text left and right, or make the text angle more flat or straight up. Incidentally, if you spin any of the scrolling lists so far around that you would expect to be able to see the text backwards, you won't. Instead, you see the text: *Use the Source, Luke*. This is a play on the famous Obi-Wan Kenobi Star Wars quote, suggesting that you gain much by exploring the VCL source. Danny Thorpe used this quote in the introduction of his (sadly out of print) book *Delphi Component Design*.

One additional key combination available in Delphi 5 is `Alt+JEDI`. This gives brief information about the Delphi JEDI (Joint Endeavour of Delphi Innovators) project. The About dialog normally has one hyperlink label that takes you to www.borland.com. Once you invoke the `Alt+JEDI` Easter Egg another hyperlink label becomes visible, which takes you to www.delphi-jedi.org (Figure 6).

➤ *Figure 6: Delphi's About box with an extra hyperlink.*



```
EDBEngineError = class(EDatabaseError)
private
  FErrors: TList;
  function GetError(Index: Integer): TDBError;
  function GetErrorCount: Integer;
public
  constructor Create(ErrorCode: DbiResult);
  destructor Destroy; override;
  property ErrorCount: Integer read GetErrorCount;
  property Errors[Index: Integer]: TDBError read GetError;
end;
```

➤ *Above: Listing 11* ➤ *Below: Listing 12*

```
EClinicError = class(Exception)
private
  FRectangle: TRect;
public
  constructor Create(const Rect: TRect);
  //Making the rectangle record available
  property Rectangle: TRect read FRectangle;
end;
...
constructor EClinicError.Create(const Rect: TRect);
begin
  FRectangle := Rect;
  //Calling one of the original constructors to set up the message
  inherited CreateFmt('Bad rectangle (%d,%d)-(%d,%d)',
    [Rect.Left, Rect.Top, Rect.Right, Rect.Bottom])
end;
...
raise EClinicError.Create(Application.MainForm.BoundsRect)
```