Beating the System: Exploring Delphi's Closed-Tools API

IDE integration for Rocket Scientists!

by Dave Jewell

This month's *Beating The System* is affectionately dedicated to Danny Thorpe, Borland development engineer and author of the excellent *Delphi Component Design* published by Addison-Wesley. In the introduction to his book, Danny expresses the wish that there might be more advanced Delphi books around with names such as *Delphi for Rocket Scientists*, *Delphi for DemiGods* and so forth. Not having access to the source code, I wouldn't claim to have anything like the Delphi knowledge that Danny possesses, but nevertheless I hope there might be one or two rocket scientists out there who find this interesting!

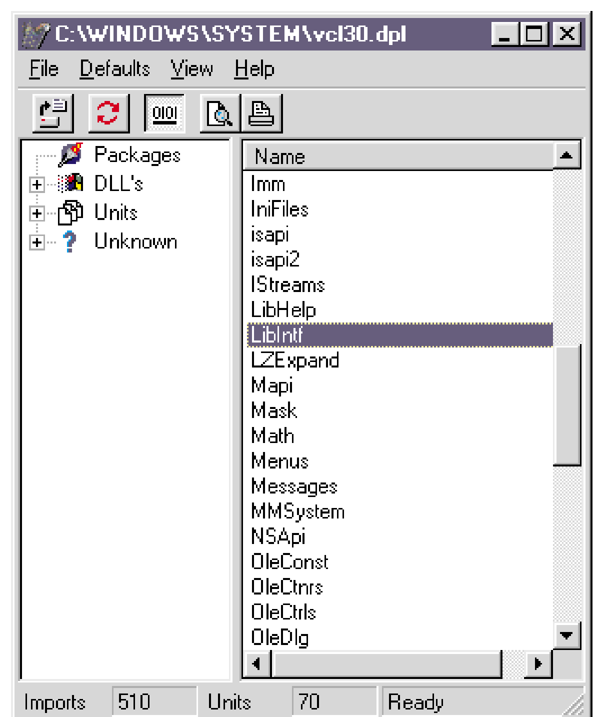
The subject of this month's column is primarily of interest to those who are developing wizards and other add-ins which integrate into the Delphi 3.0 IDE. If you've done much work with the Open Tools API, you may occasionally have been frustrated that some much-needed functionality wasn't there, or that you could only achieve some effect in a very round about way. What I want to discuss here is perhaps one of the best kept secrets of Delphi programming, the mysterious `LibIntf` unit. Using `LibIntf`, you get immediate access to a whole bunch of new routines that aren't directly available through the documented Open Tools API. In a real sense, `LibIntf` represents the Closed Tools API!

The History Of LibIntf

As the name suggests, `LibIntf` is primarily related to the component library, but it also contains quite a number of other, IDE-related goodies. In essence, `LibIntf` is the interface between the IDE and the component library. Back in the days of Delphi 1.0 and Delphi 2.0, the component library existed as a single, huge DLL which had to be rebuilt each time that one or more components were added to the library. Not only was this slow, but it was also not particularly reliable. When Borland advised you to make a backup copy of the component library before adding new components, they meant it!

What you might not appreciate is that the old COMPLIB.DCL library contained a lot more than just components. It also contained a number of other undocumented units, the main one being a large unit called `LibMain`. Together, they implemented much of the design-time 'on-form' behaviour of components, including things like grid painting and snapping to grids, grab handles and so forth, along with Delphi's various built-in property editors. The IDE communicated with the library by passing it a pointer to an object of type `TILibAppBuilder`. This object was derived from `TInterface` (so that its various methods could be called across a DLL boundary) and provided all the necessary methods for the library to pull the IDE's strings, so to speak. Similarly, when the IDE first initialised the library DLL, it received back a pointer to an object of type `TILibrary` which in turn allowed the IDE to directly access and manipulate the contents of the library. Thus, the library and the IDE were partners in an elaborate, symbiotic relationship where each partner had access to the other's closely guarded secrets...

With the arrival of Delphi 3.0, much of this 'you show me yours, and I'll show you mine' behaviour became unnecessary. The component library now exists only as a number of packages and each of these packages contains very little besides the components and VCL



➤ *Figure 1: There's more in the VCL30.DPL package than meets the eye. Using the Merlin executable file viewer, you can see our quarry hiding amongst the undergrowth...*

library routines you'd expect to find in there. All the 'real' IDE code is now located firmly inside the IDE which, of course, is where it ought to be.

Does this mean that these secret back-doors into Delphi no longer exist? Funnily enough, no. The `TILibrary` and `TILibAppBuilder` interfaces are still very much alive, except that the latter has now been externally renamed to `TIDelphiIDE`, which better reflects its purpose.

The Source, The Whole Source And Nothing But...

So you think you've got all the VCL source code? If you use something like Merlin's Executable Viewer tool to peek inside the VCL30.DPL package, you'll find that it contains a few units which aren't available in source code form. This includes `LibHelp`, `TLHelp32` and `Proxies`, but the one we're interested in is `LibIntf`. It's this unit which contains the declarations for `TILibrary` and `TILibAppBuilder`. Interestingly, even if you search the 'undocumented Delphi' books, such as those by Ray Lischner and the aforementioned book by Danny Thorpe, you'll find no mention of `LibIntf`...

If `LibIntf` is contained in the `VCL30` package, then surely all we have to do is figure out how to call the various methods it contains? Again, no. `LibIntf` is just a set of declarations for abstract classes, ie classes which contain one or more virtual abstract methods. In other words, there's no real code in there. `LibIntf` is just a template which defines a number of interfaces, it doesn't contain the code necessary to implement those interfaces. If you've used units such as `DsgnIntf`, `EditIntf`, `ExptIntf` etc you'll be familiar with this approach. All these units contain only the interface declarations: the code is inside the IDE itself.

So how can we use `LibIntf` if we haven't got the declarations? More to the point, what does `LibIntf` do for us? Ok, enough teasing! If you take a look at Listing 1, you'll see a partial listing of the contents of the `LibIntf` unit. This contains the declaration for the `TIDelphiIDE` interface, along with a few other assorted bits and pieces. Let me stress right away that this code won't compile. Not only that, but you shouldn't even *try* to compile it! I'm showing you this code in order that you, the programmer, know what methods are in the `TIDelphiIDE` object and what parameters they take. This code is for your benefit not that of the compiler.

Somewhat counter intuitively, we don't need to supply any source code for `LibIntf`, nor do we need to pull apart the code inside the IDE, even though we don't have a DCU file containing the real code. Provided that we know what the interface looks like, we can go ahead and use it in our own experts. All we need to do is add `LibIntf` to our `uses` clause and off we go!

Let me stress that this will only work in an expert. If you add `LibIntf` to the `uses` clause of an ordinary application (it wouldn't make sense to do this anyway, for obvious reasons) the compiler will complain that it can't find the `LibIntf` unit. There is obviously some sleight of hand operating behind the scenes when the compiler is in 'expert-writing mode.'

If you look at Listing 1 again, you'll see that there's a variable, `DelphiIDE`, which is initialised to zero. As with the declaration of the `ToolServices` object in the `ExptIntf` unit, we can rely on this object having been initialised to point to a real `TIDelphiIDE` object by the time our expert gets a look in.

In the remainder of this article, I'm going to discuss some of the `DelphiIDE` methods, relating them to the operation of a small LibIntf

➤ *Listing 1*

```
unit LibIntf;
interface
uses
  Windows, Classes, Graphics, ToolIntf, ExptIntf,
  FileIntf, VirtIntf;
type
  TDesignDialog = ( ddAlign, ddSize, ddScale, ddTabOrder,
    ddCreationOrder, ddSaveTemplate );
  TDesignerOptions = record
    DisplayGrid: Boolean;
    SnapToGrid: Boolean;
    GridSizeX: Integer;
    GridSizeY: Integer;
    ShowComponentCaptions: Boolean;
  end;
  TIPaletteItem = class (TInterface)
  public
    function CreateComponent (Owner, Parent: TComponent;
      Module: TIModule):TComponent; virtual; abstract;
    procedure Paint (Canvas: TCanvas; X, Y: Integer);
      virtual; abstract;
  end;
  TIDelphiIDE = class (TInterface)
  public
    procedure ActiveFormModified; virtual; abstract;
    procedure ComponentRenamed (const CurName, NewName:
      String); virtual; abstract;
    procedure ExecDesignDialog (DesignDialog:
      TDesignDialog); virtual; abstract;
    procedure FormActivated; virtual; abstract;
    function GetAppHandle: hWnd; virtual; abstract;
    function GetPathAndBaseExeName: String;
      virtual; abstract;
    function GetBaseRegKey: String; virtual; abstract;
    function GetToolSelected: Boolean; virtual; abstract;
    function GetCurCompClass: TIPaletteItem;
      virtual; abstract;
    function GetPaletteItem (ComponentClass:
      TComponentClass): TIPaletteItem; virtual; abstract;
    function GetCurTime: Integer; virtual; abstract;
    procedure GetDesignerOptions (var Options:
      TDesignerOptions); virtual; abstract;
    function GetMainWindowSize: TRect; virtual; abstract;
    function LockState: Boolean; virtual; abstract;
    procedure ModalEdit (EditKey: Char; ReturnWindow:
      Pointer); virtual; abstract;
    procedure OpenForm (const FormName: String; Show:
      Boolean); virtual; abstract;
    procedure RaiseException (const Message: String);
      virtual; abstract;
    procedure ResetCompClass; virtual; abstract;
    procedure SelectionChanged; virtual; abstract;
    procedure ShowClassHelp (const ClassName: String);
      virtual; abstract;
    procedure SelectItemName (const Name: String);
      virtual; abstract;
    procedure ValidateActiveModule; virtual; abstract;
    procedure AddExpert (Expert: TIExpert);
      virtual; abstract;
    procedure RemoveExpert (Expert: TIExpert);
      virtual; abstract;
    function GetToolServices: TIToolServices;
      virtual; abstract;
    procedure ExpertsLoaded; virtual; abstract;
    function GetFileSystem (const Ident: String):
      TIVirtualFileSystem; virtual; abstract;
    function MakeBackupFileName (const FileName: String):
      String; virtual; abstract;
    function CreateBackupFile: Boolean; virtual; abstract;
    function WinHelp (const HelpFile: String;
      Command, Data: Integer): Boolean; virtual; abstract;
  end;
const
  DelphiIDE: TIDelphiIDE = nil;
implementation
end.
```

Sample Expert which I've written. The code for this expert is shown in Listing 2. There's also a registration file and a unit which implements the usual `TIExpert` interface: you'll find a complete set of files on the disk. You can see the expert running in the various screenshots that accompany this article. The expert doesn't actually do anything, it merely serves as a vehicle for testing the individual methods of the `TDelphiIDE` class.

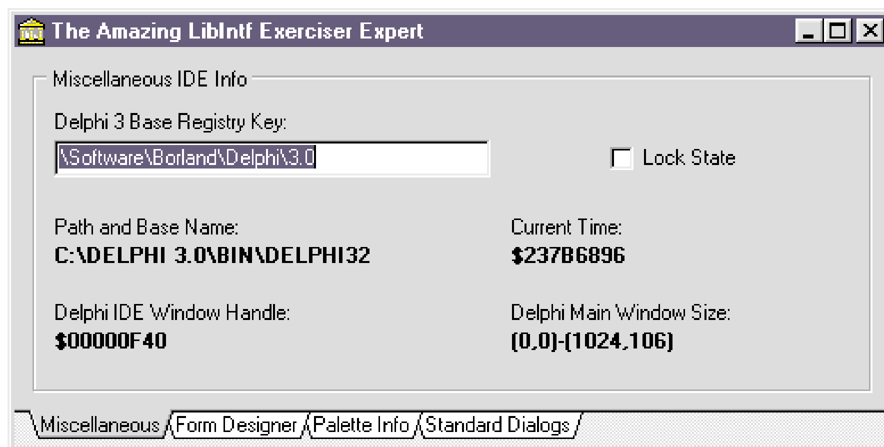
Miscellaneous Information Routines

Firstly, take a look at Figure 2. This page of the `LibIntf` expert returns miscellaneous information relating to the state of the Delphi 3.0 IDE. Let me stress that I've written the user interface of this expert in such a way that it rather gives the impression that you can use it to change internal IDE variables, but this definitely isn't the case! You can edit edit boxes, check check boxes and push radio buttons to your hearts content, but the information being presented by the expert is read-only.

In Figure 2 you'll see that the first item (top left corner) is the base registry key used by the Delphi 3.0 IDE to keep track of configuration information. This registry key is relative to the `HKEY_CURRENT_USER` registry tree. The information is returned by referencing the `GetBaseRegKey` method of the `DelphiIDE` variable. This information is obviously useful if you're writing your own expert and you want to store your own configuration as a part of the Delphi 3.0 registry tree.

You should bear in mind that some of the `DelphiIDE` methods discussed here have equivalent functionality to the official, documented methods. For example, the `GetBaseRegKey` method corresponds directly to the `TIToolServices.GetBaseRegistryKey` method. As you might imagine, the relationship between these different interfaces is fairly incestuous and it's sometimes difficult to know which is the organ-grinder and which is the monkey!

The next method is `GetPathAndBaseExeName`. As the name suggests, this returns the full pathname of the installed IDE executable, though without the terminating .EXE. Again, this might be useful if you wanted to store your own information in a subdirectory hanging off the main Delphi directory tree. Next, we come to `GetAppHandle`. This method actually maps down onto a call to `Application.Handle` within the IDE and therefore returns a window handle. Bear in mind that this is not the window handle of the main IDE window, it's the window handle of the hidden API-level window that's a characteristic of all Delphi applications, including the IDE itself.



➤ *Figure 2: This is the Miscellaneous page of my LibIntf 'exerciser' expert. Amongst other things, LibIntf returns the current date/time, size and position of the main Delphi window and the base registry key used to store IDE configuration info.*

In Figure 2 you can also see a `Lock State` checkbox. The on/off state of this checkbox is determined by the return value of the `LockState` method. If you look on the `Edit` menu of the Delphi IDE, you'll see a menu item entitled `Lock Controls`. This IDE option is used to lock the size and position of controls in the form designer while allowing you to edit properties via the Object Inspector in the usual way, thus guarding against inadvertent 'finger-trouble'! If controls are currently locked, then `LockState` returns `True`, otherwise `False`. It's worth noting that internally the `LockState` method works by merely returning the `Checked` property from the aforementioned `Lock Controls` menu item! This elegant technique eliminates the need for the IDE to maintain a separate `Boolean` variable for the lock state, and removes the necessity of keeping the menu item synchronised with a `Boolean` variable: effectively the menu item *is* the variable.

Moving on, the `GetCurTime` function returns the current time and date, as reported by an internal routine within the IDE. I believe that this information is made available to interested experts and add-ons in order that they can perform comparisons of file date/time modifications. For example, if you look at the ISTREAMS.PAS file (part of the Open Tools API) you'll find reference to methods with names such as `SetModifyTime` and `GetModifyTime`. If an expert needs to know the date/time, it's reasonable to ask why it doesn't just call the `Now` function in `SYSUTILS`? However, the `GetCurTime` routine returns date/time information in DOS format, rather than the `TDateTime` format. Thus, if you want to use the deeply cool `FormatDateTime` function to express 'IDE time' in human-readable format, you'll have to do something like this:

```
Str := FormatDateTime('dddd,
  mmmm d, yyyy, hh:mm AM/PM',
  FileDateToDateTime(
    GetCurTime));
```

Finally, the `GetMainWindowSize` method returns a `TRect` which corresponds to the size and location of the main Delphi IDE window in screen co-ordinates (pixels).

Unlike the GetAppHandle call, this really is the visible, main IDE window which contains the component palette and speed-bar. In Figure 2, you will notice that the window has an initial width of 1024 because I'm using a 1024 by 768 resolution display: at initialisation time, the IDE sets up this window to exactly fit the width of the screen.

Form Designer Information

You're right of course, I ought to have put the Lock Controls checkbox into the Form Designer page of the LibIntf expert. Sorry, you don't get your money back, but you're welcome to massage my code as much as you like!

Figure 3 shows the Form Designer information returned by the DelphiIDE object. There are only five items here, each corresponding to a field within a record of type TDesignerOptions. The contents of this record are returned by the GetDesignerOptions method. When using this method, be sure that you have checked the Aligned Record Fields option in your compiler settings. Calling the method will deliver 16 bytes of information, whether you're expecting it or not.

Incidentally, it turns out that there are some 'almost-right' versions of the LibIntf interface

➤ *Listing 2*

```
unit LibTestForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, LibIntf, StdCtrls, ExtCtrls, Tabs, Menus;
type
  TLibExTest = class(TForm)
    TabSet1: TTabSet;
    Notebook1: TNotebook;
    GroupBox1: TGroupBox;
    Label1: TLabel;
    BaseReg: TEdit;
    GroupBox2: TGroupBox;
    Label6: TLabel;
    Label3: TLabel;
    DisplayGrid: TCheckBox;
    SnapToGrid: TCheckBox;
    ShowComponentCaptions: TCheckBox;
    GridSizeX: TEdit;
    GridSizeY: TEdit;
    GroupBox5: TGroupBox;
    ComboBox1: TComboBox;
    Image2: TImage;
    cbLockState: TCheckBox;
    Label5: TLabel;
    PathAndBaseName: TLabel;
    Label7: TLabel;
    AppHandle: TLabel;
    Label8: TLabel;
    IDETime: TLabel;
    Label9: TLabel;
    WinSize: TLabel;
    Label4: TLabel;
    Image1: TImage;
    GroupBox4: TGroupBox;
    Label2: TLabel;
    HelpClassName: TEdit;
    ClassHelpOK: TButton;
    GroupBox3: TGroupBox;
    ComboBox2: TComboBox;
    Label10: TLabel;
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure HelpClassNameChange(Sender: TObject);
    procedure ClassHelpOKClick(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure TabSet1Change(Sender: TObject;
      NewTab: Integer; var AllowChange: Boolean);
    procedure ComboBox1Change(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
  public
  end;
implementation
{$R *.DFM}
procedure TLibExTest.FormCreate(Sender: TObject);
var r: TRect;
    Opts: TDesignerOptions;
begin
  TabSet1.Tabs := NoteBook1.Pages;
  with DelphiIDE do begin
    BaseReg.Text := GetBaseRegKey;
    GetDesignerOptions (Opts);
    DisplayGrid.Checked := Opts.DisplayGrid;
    SnapToGrid.Checked := Opts.SnapToGrid;
    ShowComponentCaptions.Checked :=
      Opts.ShowComponentCaptions;
    GridSizeX.Text := IntToStr (Opts.GridSizeX);
    GridSizeY.Text := IntToStr (Opts.GridSizeY);
    cbLockState.Checked := LockState;
    PathAndBaseName.Caption := GetPathAndBaseExeName;
    AppHandle.Caption := '$' + IntToHex (GetAppHandle, 8);
    IDETime.Caption := '$' + IntToHex (GetCurTime, 8);
    r := GetMainWindowSize;
    WinSize.Caption := Format('(%d,%d)-(%d,%d)',
      [r.Left, r.Top, r.Right, r.Bottom]);
    ComboBox1.ItemIndex := 0;
    ComboBox2.ItemIndex := 0;
    ComboBox1Change (Self);
  end;
end;
procedure TLibExTest.HelpClassNameChange(Sender: TObject);
begin
  ClassHelpOK.Enabled := HelpClassName.Text <> '';
end;
procedure TLibExTest.ClassHelpOKClick(Sender: TObject);
begin
  DelphiIDE.ShowClassHelp (HelpClassName.Text);
end;
procedure TLibExTest.FormPaint(Sender: TObject);
  var Item: TIPaletteItem;
begin
  with DelphiIDE do
    if GetToolSelected then begin
      Label4.Visible := True;
      Image1.Visible := True;
      Item := GetCurCompClass;
      try
        Item.Paint (Image1.Canvas, 0, 0);
      finally
        Item.Free;
      end;
    end else begin
      Label4.Visible := False;
      Image1.Visible := False;
    end;
end;
procedure TLibExTest.TabSet1Change (Sender: TObject;
  NewTab: Integer; var AllowChange: Boolean);
begin
  NoteBook1.PageIndex := NewTab;
end;
procedure TLibExTest.ComboBox1Change(Sender: TObject);
var Item: TIPaletteItem;
    Cls: TComponentClass;
begin
  with DelphiIDE do begin
    case ComboBox1.ItemIndex of
      0: Cls := TMainMenu;
      1: Cls := TPopupMenu;
      2: Cls := TLabel;
      3: Cls := TEdit;
      4: Cls := TMemo;
      5: Cls := TButton;
      6: Cls := TCheckBox;
      7: Cls := TRadioButton;
      8: Cls := TListBox;
      9: Cls := TComboBox;
     10: Cls := TScrollBar;
     11: Cls := TGroupBox;
     12: Cls := TRadioGroup;
     13: Cls := TPanel;
    else
      Cls := Nil;
    end;
    if Cls = Nil then
      Image2.Visible := False
    else begin
      Item := GetPaletteItem (Cls);
      try
        Item.Paint (Image2.Canvas, 0, 0);
      finally
        Item.Free;
      end;
      Image2.Visible := True;
      Image2.Invalidate;
    end;
  end;
end;
procedure TLibExTest.Button1Click (Sender: TObject);
begin
  { This will do nothing if there's no active form }
  DelphiIDE.ExecDesignDialog(TDesignDialog(
    ComboBox2.ItemIndex));
end;
end.
```

circulating on the Internet (see my comments about LibIntf at the end of this article). The GetDesignerOptions record is one area where there's an error. You must be sure to name the final field as ShowComponentCaptions and not as ShowComponentCaption as I've seen it elsewhere. If you get this wrong, the compiler will understandably complain that it can't find the field in question.

Palette Information

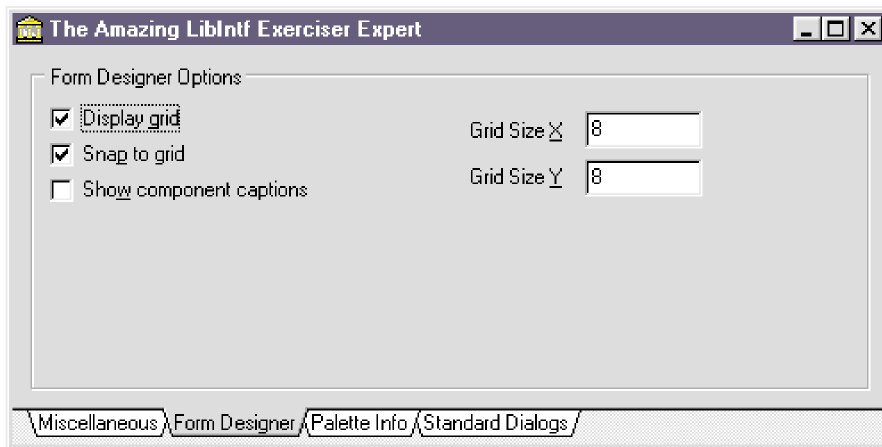
The next page, Palette Info, is rather more interesting and is shown in Figure 4. On the left hand side, you can see a combo box which I've 'hard-wired' with the names of the fourteen component classes which reside on the Standard page of the component palette. Each time you select a different component class, the small glyph to the right of the combo box will be updated with the corresponding component bitmap. This is achieved through the use of the GetPaletteItem method. Basically, you pass this method the name of an installed component palette class and the method will give you back an object of type TIPaletteItem.

Armed with a TIPaletteItem, we can call the Draw method for that item and the corresponding component bitmap will appear at the specified position. Note carefully that the TIPaletteItem is a brand new object, created for you, and must be destroyed when you've finished using it. Seasoned expert writers will be familiar with this approach when using other aspects of the Open Tools API. This is all very well, you're saying, but isn't it a bit klunky having to hard-wire the list of component classes that we're interested in? That, gentle reader, is the subject of next month's column when I'll explore the undocumented TILibrary class and demonstrate how you can programmatically interrogate the component library to discover what goodies it contains...

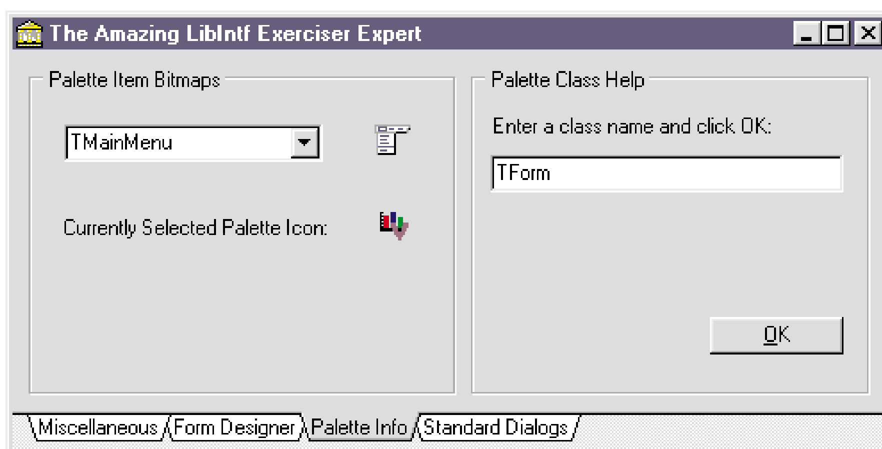
Beneath the combo box, you'll see a label marked Currently Selected Palette Icon together with another component glyph. If you run the LibIntf expert for yourself, the chances are that you won't see these two items. That's because you most likely won't have a component selected on the component palette and my expert code hides the label and picture component in these circumstances. To select a component, just click once on the required palette item before running the expert.

An expert can determine whether or not a component is selected by calling the GetToolSelected method. If it is, then True will be returned. If a component is selected, then calling GetCurCompClass will return another object of type TIPaletteItem as previously discussed. If you don't want to bother checking GetToolSelected, then you can just go ahead and call GetCurCompClass: it will simply return Nil if there's no selected palette component. As before, be sure to delete the farmed-out object when you've finished with it.

To the right hand side of all this information, you can see a group box entitled Palette Class Help. This makes use of the ShowClassHelp method, another rather nice facility that enables you to programmatically invoke Windows Help for a designated Delphi class name. This doesn't just apply to components, but it also works with TForm, TApplication and so on.

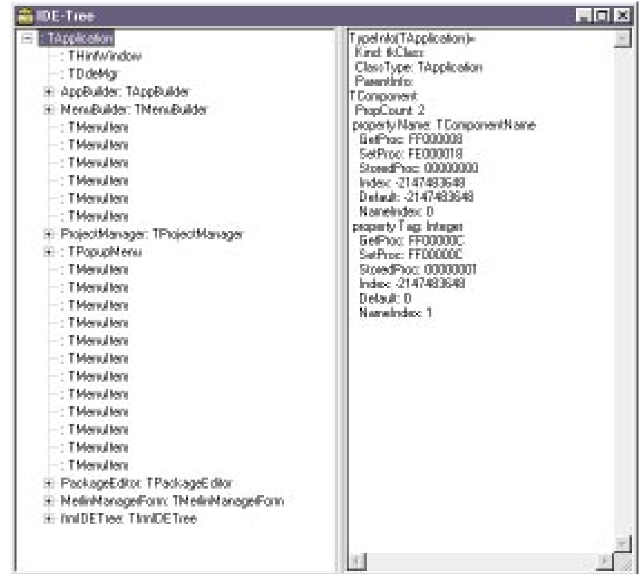


➤ *Figure 3: The Form Designer options are neatly packaged up into a single record that's returned via one method call. As ever, bear in mind that this page is informational only: changing the default values won't actually alter the corresponding IDE state.*



➤ *Figure 4: The DelphiIDE interface will let you access palette item bitmaps and invoke help information for a specified class. The problem, of course, is that at run-time you don't necessarily know what's in the library. Next month, I'll fix that by showing how to navigate all installed library components.*

➤ *Above, Figure 5: Simple demo of ExecDesignDialog.*



➤ *Right, Figure 6: Tickets, please, for the IDE Guided Tour. Using the freeware MIDETree package included on this month's disk you can browse the run-time objects hanging off the IDE's Application object.*

In fact, if you've got one or more third-party component libraries and those libraries have installed Delphi help information in the correct manner, then you can even specify third party class names. For instance, I've got the popular Raize class library and `ShowClassHelp` will happily accept `RzDriveComboBox`, `RzLabel` and so on. If you're writing a Delphi component library and you want to supply it in conjunction with a set of design-time property editors, this is a simply way of providing quick access to help on your library components.

Standard Dialogs

No, we're not talking about the so-called Windows standard dialogs. As you'll no doubt appreciate, Delphi has a set of standard, design-time dialogs such as `Align`, `Size` and `Tab Order`. Figure 5 shows my expert's simple interface to the `ExecDesignDialog` method. You call this method using one of the values from the `TDesignDialog` type and the corresponding design dialog will immediately appear on screen. Once again, this is a useful facility for building into an expert. In fact, as you get more into the inner workings of the IDE, you'll realise that it's possible not only to invoke design dialogs from inside an expert, but it's even possible to drive those same dialogs on 'auto-pilot' using a little extra wizardry. Space doesn't allow me to go into such techniques in detail, but I'll give you a little clue by including a public-domain package called `MIDETree` on this month's disk. This package, written by Martin Waldenburg, will enable you to browse through the hierarchy of run-time objects which hang off the IDE's `Application` object. I think you'll find it quite fascinating. You can see it running in Figure 6.

Until Next Time...

I've devoted this month's column to a discussion of the mysterious `LibIntf` unit and some of the more useful methods associated with the `DelphiIDE` object. In next month's column, I'll continue the story by showing you another important `LibIntf` class, `TILibrary`. Using next month's code, you'll be able to do all sorts of unspeakable things with the component library...

In case you can't wait until then, I've included another small file, LIBINTF.ZIP, on this month's disk. The interface file it contains will give you the 'full monty' as far as `LibIntf` is concerned, but this particular file was taken straight off the Web and I can't vouch for its accuracy. I've already mentioned one problem with the `ShowComponentCaption/s` field, and there may be other inaccuracies too.

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