Beating the System: Exploring Delphi's Closed-Tools API, 3

To Boldly Go...

by Dave Jewell

When I first contemplated doing this mini-series on the undocumented `LibIntf` unit, I did have some reservations about how useful readers would find it. Happily, my misgivings have been laid to rest by the very positive feedback I've received via email. In particular, I was delighted when Mark Miller (of Raptor/CodeRush fame) told me that the first part of this series had helped him to overcome a registry-related problem in CodeRush. High praise indeed! It's nice to know that I've helped contribute in some small way to what's already a great product.

The TIForm Class

In this final article, I'll tie up some of the loose ends of the `LibIntf` unit by describing the `TIForm` class. As the name suggests, `TIForm` is concerned with the design-time forms displayed by the IDE. Using the `TIForm` class, you can not only obtain a lot of information about what's on a particular form, but you can also programmatically do things like alter the name of the form, change component tab order, rename a form method and so forth. As ever, you need to realise that the `LibIntf` unit represents the means by which the IDE itself manipulates design-time forms, and that the official Open Tools API routines are, in the main, wrappers which map down onto the `LibIntf` classes below.

The interface declaration to the `TIForm` class is shown in Listing 1. Once again, you're not meant to compile this, it's simply there to show you the methods that are available. The first point to make is that I've stripped out the `abstract` and `virtual` keywords from this class for the sake of brevity. All the methods shown are, of course, virtual methods. If they weren't, you wouldn't be able to call them, since the real code is lurking inside the IDE and only calls through a VMT will work.

```
TIForm = class(TIFile)
protected
  function GetDesigner:TFormDesigner;
public
  procedure Align (Affect: TAffect);
  procedure CreateComponent (Item: TICompClass);
  function  FindCompClass(const CompName:string): String;
  function  GetAncestorName: String;
  function  GetCompCount: Integer;
  procedure GetDependentForms (Proc: TGetFormProc);
  function  GetDesignClassName: String;
  procedure GetFormDependencies (Proc:TGetFormProc);
  function  GetNVComp (Index:integer): Pointer;
  function  GetCompInfo (Index: Integer): TICompInfo;
  function  GetModule: TIModule;
  function  GetCompName(Index: Integer): String;
  function  GetFileSystem: String;
  function  GetFormInterface: TIFormInterface;
  function  GetFormName: String;
  function  GetFormImage: Word;
  procedure GetFormUnits (Proc: TGetStrProc);
  function  GetState: TFormState;
  function  GetTabCompCount: Integer;
  function  GetTabCompInfo(Index: Integer; var Name: String; var Comp: Pointer):
    Boolean;
  function  GetObjectMenuItemCount: Integer;
  function  GetObjectMenuItem (Index: Integer): String;
  procedure ObjectMenuAction (Index: Integer);
  procedure Hide;
  procedure GoDormant;
  procedure RenameFormMethod (const CurName: String; const NewName : String);
  procedure RemoveDependentLinks;
  procedure Scale (Factor: Integer);
  procedure SetFileSystem (const FileSystem: String);
  procedure SetFormName (const AName: String);
  procedure SetSelection (const Name: String);
  procedure SetNVComp (Comp: Pointer; Order: Integer);
  procedure SetTabCompOrder(Comp:pointer; Order: Integer);
  procedure Show;
  procedure ShowAs (ShowState: TShowState);
  procedure Size (Affect: TSizeAffect; Value: Integer);
end;
```

➤ *Listing 1*

```
if NoteBook1.Pages [NewTab] = 'Form Designer' then
  AllowChange := InitFormDesignerPage;
if not AllowChange then
  Exit
else
  NoteBook1.PageIndex := NewTab;
```

➤ *Listing 2*

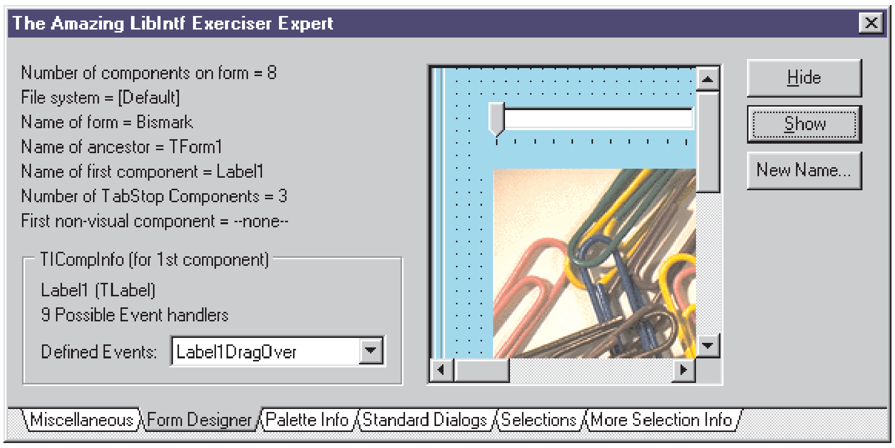
The simplest way to get a `TIForm` interface is to call the `GetActiveForm` method of `TILibrary`. This will give you a `TIForm` object instance that corresponds to the current active form. If no form is active, then you'll get `Nil` back. To ensure that this doesn't happen in my expert, you'll notice that I've added a check to this month's code such that you can't select the `Form Designer` page of the expert unless a form is active. I did this simply by checking for an active form in the `InitFormDesignerPage` method and using this to allow or disallow the page change (Listing 2).

Incidentally, if you're thinking that my little expert is becoming a bit of a mess, I share your opinion! This code evolved over the course of these three articles and could be rewritten in a much more elegant fashion. The object of the exercise isn't to create the world's most elegantly-written expert, but rather to demonstrate some of the more interesting classes and methods available through LibIntf. You'll notice, for example, that this month I've entirely removed the original functionality from the Form Designer page and replaced it with an interface that demonstrates the TIForm interface.

One more point before we look at TIForm in detail. Referring to Listing 1, you'll see that the GetDesigner method is a protected routine, meaning that we can't call it directly to get the form designer associated with the form. *Quelle horreur!* In fact, this turns out not to be a problem because TIForm has a property called Designer which isn't shown in Listing 1. Once you've got an object of type TIForm, you can just access the Designer property to get an object of type TFormDesigner.

Note that if you call the ClassName method on object instances of the various classes defined in LibIntf, (or indeed, many of the official Open Tools classes) you'll find that the returned class name is rarely what you'd expect it to be. That's because the abstract classes are simply used as templates which allow the compiler to call methods of the real, 'concrete' classes which reside in the IDE. As an example, call ClassName on a TIForm object, and you'll get back a classname of TLibForm. Similarly, call ClassName on the Designer property of a TIForm and you'll find that the real class-name is TWindowDesigner. TLibForm and TWindowDesigner are real, non-abstract classes inside the IDE and, for obvious reasons, they respectively descend from TIForm and TFormDesigner.

Right then, on with the show. As with other parts of LibIntf, some of the methods in TIForm return the same information that can be obtained by more orthodox means. For example, the GetCompCount method will tell you how many components are currently installed on the form, and for each


The Amazing LibIntf Exerciser Expert

Number of components on form = 8
File system = [Default]
Name of form = Bismark
Name of ancestor = TForm1
Name of first component = Label1
Number of TabStop Components = 3
First non-visual component = --none--

TIComplnfo (for 1st component)
Label1 (TLabel)
9 Possible Event handlers
Defined Events: Label1DragOver

➤ *Figure 1: Here's the new Form Designer page, not to be confused with the Form Designer page you saw in the first part of this series. It demonstrates how to get the design-time image of a form, interrogate the event information relating to all the controls on a form, and much more.*

component in the range 0..GetCompCount - 1 you can call the GetCompName method to give you the name of a specific component. Be sure not to call GetCompName if there are no components on the form: if you do, you'll get an *index out of bounds* exception.

Similarly, the GetFormName method can be called to return the name of the form associated with the TIForm interface. You can likewise call the GetAncestorName routine to return the name of the form's immediate ancestor class. Typically, this will be TForm unless you're using form inheritance.

Improving Your Image

A more interesting routine is GetFormImage. If you call this method, you'll get back an API-level bitmap handle which contains a design-time image of the form. You're probably aware that the plain-vanilla TForm class has a GetFormImage method of its own, but of course that particular method can only give you a *run-time* image of a form.

By contrast, the GetFormImage method of TIForm will give you a *design-time* image, complete with design grid, assuming that you've got the grid display turned on. What you won't see are any 'grab handles' around selected components, these don't get included in the bitmap and neither do any non-visual components.

Being able to access the design-time image of a form isn't likely to be much use in end-user software, but after all, the whole point of the LibIntf unit is that it lets us do interesting things in Delphi experts and property editors which are written for other developers. Using GetFormImage, you could potentially create fancy Delphi tutorials or property editors which show the effect of certain actions by retrieving the image from a hidden form design window.

As you'd expect, the GetFileSystem method returns the identifying string for the file system associated with the form. If the default file system is being used, then an empty string is returned. Similarly, you can use the SetFileSystem method to associate a new file system with a form. An exception will be thrown if you refer to a file system that hasn't previously been registered. Because the IDE's representation of a design-time form is held wholly in memory, you can call SetFileSystem for a form and then immediately use the new file system to save the form.

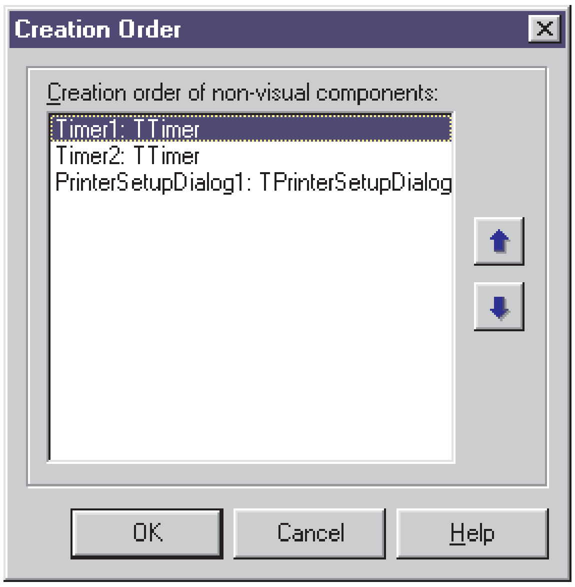
Another interesting method is GetTabCompCount. This returns the number of components on the form which happen to have a TabStop property, ie those components which are descendants of TWinControl. This method is used by (for example) the IDE's Tab

Order dialog which allows you to modify the tab order of components on a form.

In a similar vein, you can use the GetNVComp method to return a reference to a non-visual component such as an open file dialog, a timer or whatever. There are a few points to make about the use of this method. Firstly, there's no method which tells you how many non-visual components are on the form, so it's natural to ask what index values you pass to the GetNVComp. The answer is that you first call GetCompCount to determine the total number of components on the form and you then call GetNVComp for every possible component, ie 0..GetCompCount - 1. If the component in question is not non-visual, then GetNVComp will detect this internally and give you back a Nil result. Thus, if you wanted to fill a listbox with non-visual component names (as in the IDE's Creation Order dialog), you'd just loop through all the possible component indexes, adding an entry to the list-box every time GetNVComp returns non-Nil.

So what do you get back from GetNVComp? For some odd reason, this method is defined as returning a pointer, although what you're actually getting is a reference to a TComponent. Thus, you must cast it to TComponent before you can do anything sensible with it. Also, bear in mind that, unlike the official Open Tools API, the object reference you get back is an honest-to-goodness component handle, it isn't a proxy object that's been manufactured for you and you shouldn't call Free on it.

Well, actually, you can call Free on it, but if you do so, you'll delete the component from the form! It should be obvious from this mini-series that there are many ways in which LibIntf gives you access to the *real* components that the IDE adds to a real form, in contrast to the TWindowDesigner form which masquerades as the real form at design-time. Having got direct access to a *real* component, you'll appreciate that it's very easy to get access to the real form itself without going through the normal


Creation Order

Creation order of non-visual components:

Timer1: TTimer
Timer2: TTimer
PrinterSetupDialog1: TPrinterSetupDialog

➤ *Figure 2: So you want to create your own, souped-up Creation Order dialog? Using the information in this article, you'll be able to do that by making use of the GetNVComp and SetNVComp methods*

contortions imposed by the Open Tools API. The possibilities for clever hackery are boundless!

Another component-fetching routine is GetTabCompInfo. As the name suggests, this fetches information on tabbed components and should only be used in conjunction with GetTabCompCount. Having determined how many tabbed components are on the form, you can access them by passing an index in the range 0..GetTabCompCount - 1. The routine will return True on success and False on failure. If successful, the var parameters will return the name of the tabbed component and (as with GetNVComp) an anonymous pointer which must be cast to a TComponent (or at the very least, a TObject) in order to identify the type of component that's being dealt with.

Related to GetTabCompInfo is the SetTabCompOrder routine. As the name suggests, this routine alters the tab order of the designated component. As with some of the other TIForm methods, it unaccountably expects you to pass the designated tabbed component as a Pointer, so you must cast the first parameter to a Pointer to keep the compiler happy. Also, you should take care when calling this routine otherwise you will end up with several tabbed components all having the same tab order. The best approach is to do what the IDE's Tab Order dialog does: build a temporary list of all tabbed components, allow the user to rearrange them as required, and then call SetTabCompOrder for all components in the list, updating the order for everything in one go.

Bear in mind that there is nothing 'magic' about the way that SetTabCompOrder works, it doesn't do much more than set the TabOrder property of the designated component. However, you shouldn't try doing the job yourself because SetTabCompOrder also marks the designer as modified, which is important.

Similar arguments apply to the SetNVComp method which can be used to change the creation order of non-visual components. Again, this must be done 'in one go.' However, the situation is more complex with SetNVComp because LibIntf doesn't contain a mechanism allowing you to reference non-visual components with a contiguous set of indexes, you have to perform the mapping yourself.

Beyond The Event Horizon...

A particularly interesting routine is GetCompInfo, which takes an index parameter in the range 0..GetCompCount. In return, it will give you an object of type TICompInfo. In this particular case, the TICompInfo object has been created on your behalf and you *must* delete it when you've finished using it. The TICompInfo interface opens up a whole new scenario, the declaration for which is shown in Listing 3.

```
TICompInfo = class (TInterface)
public
  procedure ClearEvent (Index: Integer);
  function  GetClassName: String;
  function  GetEventCount: Integer;
  function  GetEventInfo (Index: Integer): PPropInfo;
  function  GetEventValue (Index: Integer): String;
  function  GetComponentHandle: Pointer;
end;
```

➤ *Listing 3*

```
function TCompInfo.GetNamePath: String;
function TCompInfo.GetSubInfoCount: Integer;
function TCompInfo.GetSubInfo (Index: Integer): TCompInfo;
```

➤ *Listing 4*

The GetClassName method of TICompInfo is relatively boring and simply returns the class name of the associated component. In the same way, the GetComponentHandle function (which claims to return a pointer) actually returns a TComponent handle for the component. The other methods are more interesting. Calling GetEventCount will tell you how many event types are defined for this component type. Let me reiterate the point: it tells you how many different types of event handlers are defined for this type of component. It does *not* tell you how many event handlers have actually been set up for this specific component instance.

If you want to know what event handlers are actually set up for the component, then you can call the badly named GetEventValue method. What this method actually gives you is a string that returns the name of an event handler or an empty string if no event handler has been established.

Let's take a simple example to make this clearer. Suppose you've got a TLabel control installed on the form. If you select the control and then use Object Inspector to look at the defined events, you'll see that there are nine possible event types for TLabel controls. Sure enough, if you call GetEventCount on the TICompInfo interface for a TLabel component, then nine is what you'll get back. Now let's suppose that you've got an OnClick handler set up for this particular label component. You know there are nine possible event types so you call GetEventValue nine times with an index in the range 0..8. Eight of those times, you'll get back an empty string but one time you'll get back the name of your installed event handler which might be something like Label1Click.

In this way, you can determine how many events are defined for the component, how many event handlers have been set up and what their names are, as methods of the form. But what if you want to know the actual names of the event types themselves, the parameters taken by a specific event type, and so forth? In other words, how can you 'discover' programmatically that the Label1Click handler actually corresponds to the OnClick event, and that this event type takes a single parameter, Sender, of type TObject? The answer is to use the GetEventInfo method. This takes an index parameter (in the range 0..GetEventCount - 1) and returns a pointer to the TPropInfo data structure associated with this event. This encodes not only the name of the event (OnClick, OnCloseQuery, etc, but also any parameters and their types).

It's best to think of the Index parameter to GetEventInfo and GetEventValue as a scalar type which defines a particular event. Having determined that an event

A Word About Proxies

In the first article of this series, I explained that although Borland supply no source code to LibIntf and although it isn't an official part of the VCL library, you can make use of it from your own Delphi experts simply by adding LibIntf to the uses clause of a unit. This works because the 'packaged' version of the VCL library (VCL30.DPL to be precise) has a dummy LibIntf unit which contains abstract declarations of the various LibIntf classes. Inside an expert, the all-important CompLib and DelphiIDE variables get initialised with object instances that reference the 'real' LibIntf code which is resident inside the IDE itself.

It's been pointed out to me that because LibIntf is located inside VCL30.DPL, there's nothing to stop you putting LibIntf into your uses clause when building an ordinary application (ie not a Delphi expert) that uses packages. That's certainly true, but such a scenario means that your copy of LibIntf will no longer be 'shared' with the IDE, meaning that the CompLib and DelphiIDE variables will remain set to zero, which is just as it should be.

All this raises the question of whether there are any other hidden units inside VCL30.DPL which can likewise be made use of? As it turns out, there are! One of them is called Proxies and you can include it in just the same way as for LibIntf. You can find the interface definition for Proxies in the file PROXIES.INT which is located in your Delphi DOC directory. The routines defined in this unit relate to the way in which the IDE maintains design-time 'proxies' for each component that you add to a form. For example, when you create a new form method in the IDE, the Proxies routine CreateSubClassMethod gets called. When you rename a form method, then the corresponding RenameSubClassMethod gets called, and so on. Although the Proxies unit isn't documented by Borland, Ray Lischner has done an excellent job of describing what it does in his new book *Hidden Paths of Delphi* (available from the Borland User Group on 01980 630032 and your local bookshop).

handler is set up through a call to GetEventValue, you should use the *same* index value when calling GetEventInfo to return the actual type information associated with this event. In the same way, you can use the ClearEvent method to clear a designated event handler type that's currently associated with a component. Calling ClearEvent doesn't physically delete the event handler code, it simply disassociates it from the component.

But Wait: There's More!

While investigating the inner workings of the TICompInfo class, I discovered that there are actually three other TICompInfo methods whose presence was apparently unknown to Sergey Orlic, the Moscow-based Borland Product Manager who allegedly reverse-engineered the original LibIntf class. The declarations for these methods are given in Listing 4.

The GetNamePath function returns the name path for the component associated with the TICompInfo interface. Under normal circumstances, this will be exactly the same as the component name. If you read the Delphi on-line documentation relating to GetNamePath (which is also a method of TPersistent and TComponent) you'll see that its real purpose is to provide support for collections in the Object Inspector. In the same way, the remaining two methods are also associated with components which contain collections.

For most common controls, the GetSubInfoCount method will return 1 indicating that there are no 'sub-components'. However, for components such as THeaderControl and TStatusBar, a larger value will be returned. You can obtain a TICompInfo interface for sub-components by calling the GetSubInfo method. As with the GetCompInfo routine, the TICompInfo object is created on your behalf and you must free the object when you've finished using it.

I intended to document the TIModule class, but I'm fresh out of space again. Sorry! I'll keep this in reserve and maybe discuss it at some future point along with a detailed description of some of the other 'ghost' units that haunt the VCL30.DPL package (see the boxout *A Word About Proxies*).

This month's source code for my LibIntf exerciser expert is given in Listing 5. As before, it's essentially a 'delta' of the previous month's code in order to save space. There are two or three other methods that this code 'exercises' which didn't make it into my discussion. In particular, the code shows how to use the LibIntf methods which show and hide a design-time form, and enable you to specify a new name for a form. the full source is on this month's disk of course.

Well, that concludes our tour of LibIntf, I hope you've found it interesting. For those who develop Delphi add-ins, experts, property editors and so on, I hope that, as with CodeRush, it's given you some ideas for getting around limitations with the official Open Tools API. Even if you're not developing any Delphi add-ins, you should at least have a greater appreciation

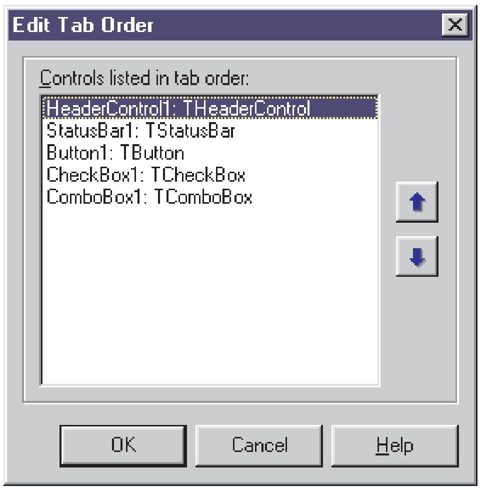
➤ *Listing 5*

```
function TLibExTest.InitFormDesignerPage: Boolean;
var
  Idx: Integer;
  Str: String;
  CInfo: TICompInfo;
  ActiveForm: TIForm;
begin
  { Assume no active form }
  Result := False;
  ActiveForm := CompLib.GetActiveForm;
  if ActiveForm = Nil then begin
    ShowMessage(
     'Can''t access this page without an active form');
    Exit;
  end;
  { OK - we've got an active form }
  Result := True;
  with ActiveForm do begin
    CompCount.Caption := Format(
     'Number of components on form = %d', [GetCompCount]);
    Str := GetFileSystem;
    if Str = '' then Str := '[Default]';
    FileSystem.Caption := Format('File system = %s', [Str]);
    FormName.Caption :=
      Format('Name of form = %s', [GetFormName]);
    Ancestor.Caption :=
      Format('Name of ancestor = %s', [GetAncestorName]);
    Image3.Picture.Bitmap.Handle := GetFormImage;
    if GetCompCount > 0 then
      Str := GetCompName (0) else Str := '--none--';
    FirstCompName.Caption :=
      Format('Name of first component = %s', [Str]);
    TabCompCount.Caption :=
      Format('Number of TabStop Components = %d',
      [GetTabCompCount]);
    { Find first non-visual component, if any }
    Str := '--none--';
    for Idx := 0 to GetCompCount - 1 do
      if GetNVComp (Idx) <> Nil then begin
        Str := TComponent (GetNVComp (Idx)).Name;
        break;
      end;
    FirstNVComp.Caption :=
      Format('First non-visual component = %s', [Str]);
    { Do something to demonstrate the TICompInfo interface }
    GroupBox2.Visible := GetCompCount > 0;
    if GroupBox2.Visible then begin
      CInfo := GetCompInfo (0);
      try
        CIName.Caption := TComponent(
          CInfo.GetComponentHandle).Name;
        CIName.Caption := CIName.Caption +
          Format(' (%s)', [CInfo.GetClassName]);
        CIEventCount.Caption := Format(
          '%d Possible Event handlers',
          [CInfo.GetEventCount]);
        EventCombo.Items.Clear;
        for Idx := 0 to CInfo.GetEventCount - 1 do begin
          Str := CInfo.GetEventValue (Idx);
          if Str <> '' then
            EventCombo.Items.Add (Str);
        end;
        if EventCombo.Items.Count = 0 then
          EventCombo.Items.Add ('--none--');
        EventCombo.ItemIndex := 0;
      finally
        CInfo.Free;
      end;
    end;
  end;
end;
procedure TLibExTest.HideClick(Sender: TObject);
begin
  CompLib.GetActiveForm.Hide;
end;
procedure TLibExTest.ShowClick(Sender: TObject);
begin
  CompLib.GetActiveForm.Show;
end;
procedure TLibExTest.NewNameClick(Sender: TObject);
var
  NewName: String;
begin
  with CompLib.GetActiveForm do begin
    NewName := InputBox ('LibIntf Expert',
      'Enter a new name for the form', GetFormName);
    if (NewName <> '') and
      (NewName <> GetFormName) then begin
      SetFormName (NewName);
      InitFormDesignerPage;
    end;
  end;
end;
end.
```

of the 'wheels within wheels' that lie at the heart of the IDE.

Let me stress one last time that you shouldn't use `LibIntf` indiscriminately. Neither `LibIntf` nor the Open Tools API is documented by Borland, but at least the existence of the Open Tools API is acknowledged! I strongly recommend that you use `LibIntf` only when absolutely necessary in order to maximise the likelihood of your add-in working with future versions of the Delphi IDE.

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.



```
HeaderControl1: THeaderControl
StatusBar1: TStatusBar
Button1: TButton
CheckBox1: TCheckBox
ComboBox1: TComboBox
```

➤ *Figure 3: The same arguments apply to the Tab Order dialog. Want to know how Borland implemented it? It's basically a judicious mixture of GetTabCompCount and GetTabCompInfo with a light sprinkling of SetTabCompOrder.*