The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Bad Uninstall

Q After uninstalling a MultiEdit demo program using the uninstall utility I keep getting the following message: *Could not find Expert C:\....\MEWEX32.DLL.* I could not find any way of getting rid of this message from within Delphi, and no reference appears in any of the relevant INI files. Grepping around proved equally fruitless. Simple question: how do I get rid of this silly message?

A Run REGEDIT.EXE and look in: HKEY_CURRENT_USER\Software\Borland\Delphi3.0\Experts. If you are using Delphi 2.0, make the appropriate modification to the registry path. In this key you should see a reference to the no longer resident expert. Select it and delete the entry. That should see to it.

Control Looping

Q Is there an easy way to walk through the different objects on a panel or a form? I have a form. On that form are various edit controls, some of which are created dynamically and some are made in design mode. What I want to do is loop through the text fields and put all the values in a string.

A You could set up an array or `TList` of all the appropriate edit controls and loop through the filled entries. Or, if you simply want to iterate through all the edit controls on a form, you can loop through the `Components` array property of the form, or loop recursively through its `Controls` property.

Which one is appropriate depends on what you need. The array and `TList` options are the most flexible, but also the most tedious to set up, especially if there are many controls to loop through. Let's take a look at some of these approaches with the project Loopy.Dpr from this month's disk. This project has three edit controls and employs a number of mechanisms to loop through them and display their contents. Listing 1 shows a simple, but dull way, using an array that gets manually populated.

```
...
EditArray: array[1..3] of TEdit;
...
procedure TForm1.FormCreate(Sender: TObject);
begin
  EditArray[1] := Edit1;
  EditArray[2] := Edit2;
  EditArray[3] := Edit3;
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  Loop: Integer;
  Msg: String;
begin
  Msg := '';
  for Loop := 1 to 3 do
    Msg := Msg + EditArray[Loop].Text;
  ShowMessage(Msg)
end;
```

➤ *Listing 1*

```
procedure TForm1.FormCreate(Sender: TObject);
var Loop: Integer;
begin
  for Loop := 1 to 3 do
    EditArray[Loop] := FindComponent('Edit' + IntToStr(Loop)) as TEdit;
end;
```

➤ *Listing 2*

In this particular program, because the names of the edits are all the same, with incrementing numbers at the end, the setup code can be abbreviated to that shown in Listing 2.

Listing 3 shows the equivalent when using a `TList`. Again, because of the simple naming convention Delphi follows, a `FindComponent` loop can be used to fill the list. A `TList` has the advantage of being extendible, Delphi arrays must have their size fixed at compile-time.

The other methods of tackling the problem involve the `Components` and `Controls` arrays that all components (and forms) possess. You can get access to all components on a form by looping through the forms `Components` array, and for every entry, looping through its `Components` array and so on. All components placed on the form at design-time are owned by the form (their `Owner` property will be the form object). This means that they will all appear in the form's `Components` array. Dynamically created components will appear in the form's `Components` property so long as you passed the form as the parameter to the constructor.

You can get access to all the `TControl` descendants on a form by doing the same thing with the `Controls` array. Controls that are children of the form will be in the form's `Controls` property. Controls

that are children of some component on the form will be in the Controls array property of that component.

Listing 4 shows two routines that do this and then write the name and class of the found components in some TStrings object. The code that calls these routines passes in the Items property of a TListbox (see Figure 1 to view the results).

We can take advantage of this approach to loop through all the components and perform some operation on all the edit controls, or whatever is required. Listing 5 has code that loops through the form's Components array, concatenating edit control contents and displaying them.

Of course, when looping through the Components array, we have no programmatic control over the order of the edits we encounter. If you need to ensure that the components will be found in a particular order, you will need to view the form in text mode. In Delphi 2 and 3 you can right-click on the form and choose View as Text. Delphi 1 users can select File | Open File... and then choose Form file from the List files of type: combobox.

The textual view of a form has child controls nested within their parents. In any given parent, the order in which you see the children listed dictates the order in which they will appear in the relevant Components and Control arrays.

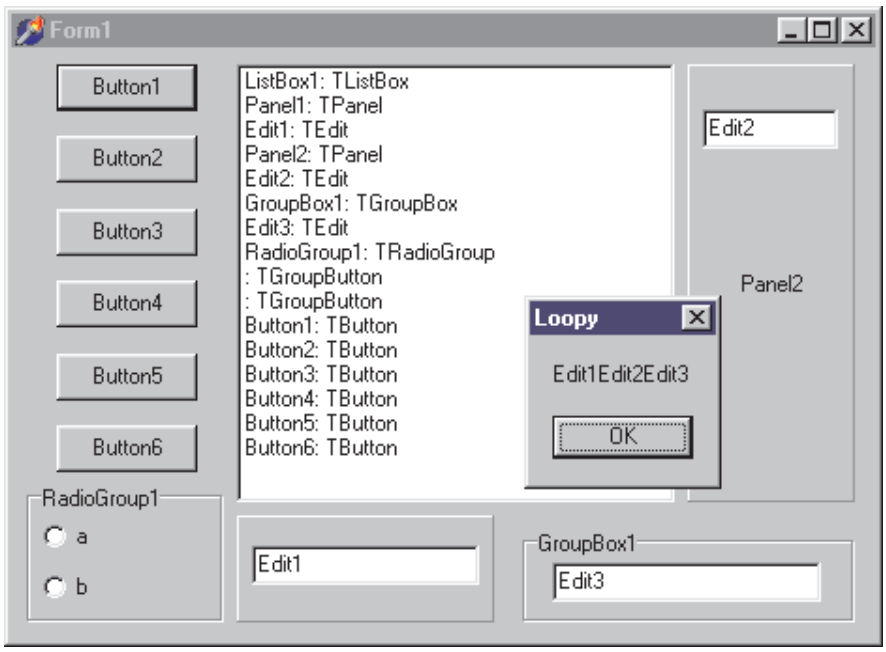
To iterate through all the Controls array properties, looking for some components, requires some recursion. Listing 6 shows what is required.

```
...
EditList: TList;
...
procedure TForm1.FormCreate(Sender: TObject);
var Loop: Integer;
begin
  EditList := TList.Create;
 {EditList.Add(Edit1);
  EditList.Add(Edit2);
  EditList.Add(Edit3); }
  for Loop := 1 to 3 do
    EditList.Add(FindComponent('Edit' + IntToStr(Loop)));
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  EditList.Free;
  EditList := nil
end;
procedure TForm1.Button2Click(Sender: TObject);
var
  Loop: Integer;
  Msg: String;
begin
  Msg := '';
  for Loop := 0 to EditList.Count - 1 do
    Msg := Msg + TEdit(EditList[Loop]).Text;
  ShowMessage(Msg)
end;
```

➤ *Listing 3*

```
procedure ListControls(Control: TWinControl; List: TStrings);
var Loop: Integer;
begin
  for Loop := 0 to Control.ControlCount - 1 do begin
    with Control.Controls[Loop] do
      List.Add(Format('%s: %s', [Name, ClassName]));
    if Control.Controls[Loop] is TWinControl then
      ListControls(TWinControl(Control.Controls[Loop]), List)
  end
end;
procedure ListComponents(Component: TComponent; List: TStrings);
var Loop: Integer;
begin
  for Loop := 0 to Component.ComponentCount - 1 do begin
    with Component.Components[Loop] do
      List.Add(Format('%s: %s', [Name, ClassName]));
    ListComponents(Component.Components[Loop], List)
  end
end;
...
ListBox1.Items.Clear;
ListControls(Form1, ListBox1.Items)
```

➤ *Listing 4*



➤ *Figure 1*

BDE Errors

Q I need a place to look up Borland Database Engine Error Codes. I recently got an error 2109 error whilst initialising the Database Engine. I cannot find any listing of error codes for the BDE. Where are they?

A The best reference to these creatures is the BDE import unit (or units in Delphi 1). In the 16-bit world, the BDE is represented in all its glory by three

```
procedure TForm1.Button3Click(Sender: TObject);
var
  Loop: Integer;
  Msg: String;
begin
  Msg := '';
  for Loop := 0 to ComponentCount - 1 do
    if Components[Loop] is TEdit then
      Msg := Msg + TEdit(Components[Loop]).Text;
  ShowMessage(Msg)
end;
```

➤ *Listing 5*

```
procedure TForm1.Button4Click(Sender: TObject);
  procedure LoopEdits(Ctl: TWinControl; var Msg: String);
  var
    Loop: Integer;
  begin
    for Loop := 0 to Ctl.ControlCount - 1 do begin
      if Ctl.Controls[Loop] is TEdit then
        Msg := Msg + TEdit(Ctl.Controls[Loop]).Text;
      if Ctl.Controls[Loop] is TWinControl then
        LoopEdits(TWinControl(Ctl.Controls[Loop]), Msg)
    end
  end;
var
  Msg: String;
begin
  Msg := '';
  LoopEdits(Self, Msg);
  ShowMessage(Msg)
end;
```

➤ *Listing 6*

```
{=====================================================================}
{                       Error Categories                              }
{=====================================================================}

const
...
  ERRCAT_SYSTEM       = $21;   { 33 System related (Fatal Error) }
...
  ERRBASE_SYSTEM      = $2100; { System related (Fatal Error) }
...
{=====================================================================}
{                     Error Codes By Category                         }
{=====================================================================}
...
{  ERRCAT_SYSTEM }
{  ============= }
...
ERRCODE_CANTLOADIDAPI = 9;     { Cannot load IDAPIxx.DLL }
...
DBIERR_CANTLOADIDAPI  = (ERRBASE_SYSTEM + ERRCODE_CANTLOADIDAPI);
```

➤ *Listing 7*

units. You do not get the source to these, but you do get what is important, the interface sections. These can be found in Delphi's DOC directory. DbiProcs.Int contains import declarations for all the documented IDAPI functions. DbiTypes.Int defines all the data structures used by those functions. Finally, DbiErrs.Int defines constants for all the errors. In Delphi 2 and 3 all three units are combined into the BDE unit. The DOC directory contains BDE.Int.

[Another good reference to BDE error codes and lots of other BDE information is 'Delphi Database Development', by Blue, Kaster, Lief and Scott, published by M&T Books, ISBN 1-55851-469-4. Ed]

When the BDE generates an error, it puts it in a certain category and gives it a sub-code. The error codes you typically see have both these pieces of information. The category is the high byte and the sub-code is the low byte. Unfortunately, trying to translate a number back into a constant is a little arduous, but let's struggle on.

Let's take your error number, 2109. This is more than likely a hexadecimal number, so we will assume it is in fact $2109. This means a category of $21 and a sub-code of 9. Checking the Pascal interface files, the important information is shown in Listing 7.

So the BDE constant for the error code is `DbiErr_CantLoadIDAPI`. Unfortunately, you could have probably guessed about as much as this constant name and its surrounding comments tell you. There was a problem and so one or more of the IDAPI DLLs couldn't be loaded. Now you need to figure out why. Usually it's because the BDE is not installed, or not correctly installed. Or, maybe the 16-bit BDE is installed and you're running a 32-bit program, or vice-versa.

The good thing about knowing how to get these constants is that if you are writing exception handling code for database operations, you can use them to aid clarity. A well-named constant is always more readable that a literal numeric value. See the next entry for an example that uses the BDE error constants.

Trapping Database Exceptions

Q How can I trap key violations that occur when my users are editing tables in data aware controls? If the user is entering data in a grid, there doesn't seem to be any of my code to put a `try..except` block around.

You can set up a global exception handler via the `Application` object's `OnException` event (as detailed in *Changing the default exception handler* in Issue 14's *Delphi Clinic*) and trap for the problem there. The KeyViol.Dpr project on the disk serves as an example of how to do this. This is a simple program that tries to intercept key violations, as one example of a BDE error that could be caught.

When a key violation occurs, an `EDBEngineError` exception object is manufactured with a list of error objects held within its Errors array property. There is always at least one `TDBError` object in the array, sometimes more. A `TDBError` object has five properties. `ErrorCode` is the full BDE error number (see the previous entry for details of BDE errors in general). `Category`

and Subcode are the high and low bytes of ErrorCode respectively. NativeError is a database server error code number that needs to be interpreted separately, and finally Message is the server generated or the BDE generated error message.

To see whether the EDBEngineError represents a key violation we can examine the first TDBError and see if it has the right information. We can check that ErrorCode has a value of DbiErr_KeyViol, or ErrBase_Integrity + ErrCode_KeyViol. Alternatively we could check that Category has a value of ErrCat_Integrity and SubCode is ErrCode_KeyViol. Choices, choices.

The global exception handler from the project is shown in Listing 8. If a key violation is encountered it simply writes this fact on the form's caption bar. All other exceptions are left to be dealt with in the usual way.

```
procedure TForm1.DoException(Sender: TObject; E: Exception);
begin
  if (E is EDBEngineError) and
    (EDBEngineError(E).Errors[0].ErrorCode = DbiErr_KeyViol) then
    Caption := E.Message
  else
    Application.ShowException(E);
end;
```

➤ *Listing 8*

Captionless Dragging

Q How can I allow my users to drag my form without using the caption bar (which will be absent)? The question also extends to how I can allow arbitrary controls to be dragged around on a form.

A There seems to be at least two ways to accomplish this. One way is to send the same message to the target window that a form receives when the user drags it with the mouse. The other is to pretend that the target window is part of a caption bar (that is to say part of the non-client area of the window).

This is the way the system works by default. Every time the mouse is moved, messages are sent to the window under the mouse (or that has captured the mouse) indicating a mouse movement (wm_MouseMove) and to check for a possible new mouse cursor requirement (wm_SetCursor). Additionally, a message is sent to find out whether the mouse is over part of the non-client area of the window. If it is reported to be over the caption bar and the left mouse button is pushed (generating a wm_NCLButtonDown) and then moved, a special wm_SysCommand message is generated. You can see these messages in Figure 2.

➤ *Figure 2*

If you, as a user, choose Move from any system menu, you are able to move the window around the screen with the arrow keys on your keyboard. You can match this programmatically by sending a wm_SysCommand message with a WParam value of sc_Move. The particular wm_SysCommand message seen in Figure 2 is almost, but not quite, the same. It has a WParam value of sc_Move+2, which equates to mouse dragging of the window.

What all this means is that we can drag any window around the screen by the mouse in one of two ways. Either we can trap for wm_NCHitTest messages in the appropriate component class and return a value of HTCaption, or we can send the window a wm_SysCommand with a WParam of sc_Move + 2, whenever the mouse is pressed on the component.

The latter option is by far the easiest general case. The Drag.Dpr application on this month's disk shows how to make a generic OnMouseDown event handler (which in this case is shared between the form and a panel on the form) to achieve dragging controls upon demand. Notice that if there is any mouse capture enabled, it is necessary to disable it. See Listing 9.

The other approach requires message handling for the appropriate windows. Listing 10 comes from Drag2.Dpr and shows a suitable wm_NCHitTest for a form. This allows the form to be dragged by the caption bar as normal, and also the client area. Note however that

➤ *Listing 9*

```
procedure TForm1.GenericMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  ReleaseCapture;
  (Sender as TControl).Perform(wm_SysCommand, sc_Move + 2, 0);
end;
```

➤ *Listing 10*

```
TForm1 = class(TForm)
  public
    procedure WMNCHitTest(var Msg: TMessage);
      message wm_NCHitTest;
  end;
...
procedure TForm1.WMNCHitTest(var Msg: TMessage);
begin
  inherited;
  if Msg.Result = HTClient then
    Msg.Result := HTCaption
end;
```

care is taken to ensure that the border icons and normal border resizing are still allowed to function normally. Only the client area is modified to allow dragging.

Resetting Autoincrement Fields

Q When using Paradox autoincrement fields in Delphi, how do you reset the value back to zero when you clear the file of records?

A Use the BDE's `DbiDoRestructure` API to firstly convert the autoincrement field to a long integer field, and then convert it back to an autoincrement. Similarly if you want to seed the autoincrement field with a value other than zero, first create the table with a long integer field, add a record and put the value that you want into the field, restructure to convert to an autoincrement and then empty the table.

For more information on the use of `DbiDoRestructure`, check the *Tips & Tricks* column in Issue 27.

Acknowledgements

Thanks to Steve Axtell of Inprise's European Technical Team for the autoincrement information used this month.