The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

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

TPageControl.OnChanging

Q Is there any way of obtaining the new page in a `TPageControl`'s `OnChanging` event? I was using a `TTabbedNotebook` before which gave me the current and new tabs before changing pages. Using this information I could decide if the user was allowed to go to this new page. Delphi 3's `TPageControl` does not appear to reveal where a user is going as the tabs on a `PageControl` are pressed.

A That's right. The `OnChanging` event is triggered in response to a `TCN_SELCHANGING` notification message which doesn't reveal the about-to-be-selected page. I can't figure a way of getting around this. I can only assume you have to treat that event as one which stops people *leaving* the current page, rather than preventing people *going* to a new page.

Removing Components

Q What's the proper way to remove third party components from Delphi 3? There doesn't seem to be a "remove components" option and the Help just doesn't! I got in all kinds of a mess trying to do this, with Delphi GPFing so badly it hung my system several times. I also noticed that some components had become marked as "hidden" on the component palette, what is this for?

A There seem to be two ways of removing components to one extreme or another.

First, you can remove the registration unit from the design-time package concerned. From `Component | Install Packages...` locate the package in the list and press the `Edit` button (say `Yes` when it offers to open the package into the Package Editor). Then locate the relevant component source unit(s) in the Package Editor and remove it/them with the `Remove` button. Now press the Package Editor's `Compile` button. The component(s) should now be gone.

Or, you can hide the component. From `Component | Configure Palette...` locate the relevant page of the palette where the component resides. Select the component from the list of components on that page. Press the `Delete` key on the keyboard or the `Hide` button. The component is now hidden.

Your question asks why you would want to hide a registered component. You might want to do this in order to trim down your component palette's component selection, but don't really want to remove them completely. This makes it easy to get these components back again. Simply go back to the component palette configuration dialog, select `All` from the `Pages` list, locate and select the hidden components and press the `Show` button. Consider the old `TReport` component: it's actually installed in Delphi 3, but hidden.

Incidentally if you want to remove a whole package full of components, in the Install Packages dialog, you select the package and press `Remove`.

Version Information Problem

Q I have used the Delphi 3 project option to include version information in my project. I also turned on the option to auto-increment the build number, but it doesn't seem to work. Whenever I modify my project and recompile it, the `FileVersion` remains the same. What is the problem?

A The key word in the option you turned on is *build*. The option causes the `FileVersion` to increment the `Build` portion of the `FileVersion` whenever you choose `Project | Build All`. This explains why the auto-increment option is disabled in the options available for a package: you can only compile a package, the `Build All` option is not available.

MessageDlg Caption

Q Is there a way of changing the caption of a `MessageDlg` over and above the few default strings available?

A This goes back to the `MessageDlg` question from Issue 23. You can use the lesser known `Dialogs` unit routine `CreateMessageDialog`. You can wrap this up in a routine of your own. Listing 1 shows an appropriate call to it: notice that the key routine takes much the same parameters as `MessageDlg`. Figure 1 shows the dialog in action.

SQL Server Blob Size

Q We are using Delphi 3 and MS-SQL 6.5. When storing Blobs larger than 32Kb the Borland Database Engine (BDE) returns the error message "Invalid blob length." A number of people on the Internet appear to be having the same problem, but we haven't seen a solution yet.

A To retrieve more than 32kb from a MS SQL Server table Blob, you have three possibilities.

➤ *Figure 1*

```
uses Dialogs;
...
with CreateMessageDialog('This text goes in the box', mtInformation, [mbOk]) do
  try
    Caption := 'My new dialog caption';
    ShowModal;
  finally
    Free
  end;
```

➤ *Listing 1*

```
var
  PI: TProcessInformation;
procedure TForm1.Button1Click(Sender: TObject);
var SI: TStartupInfo;
begin
  GetStartupInfo(SI);
  Win32Check(CreateProcess(nil, PChar(Edit1.Text),
    nil, nil, False, 0, nil, nil, SI, PI));
  WaitForInputIdle(PI.hProcess, Infinite);
  Button2.Enabled := True
end;
procedure TForm1.Button2Click(Sender: TObject);
const
  Msgs: array[Boolean] of String = ('Stopped', 'Still running');
var
  Flag: Boolean;
begin
  Flag := WaitForSingleObject(PI.hProcess, 1) = Wait_TimeOut;
  ShowMessage(Msgs[Flag])
end;
```

➤ *Listing 2*

First, use a `TTable`. Second, use a live `TQuery`. Lastly, set the BDE's `BLOB SIZE` setting to more than 32Kb.

For the first two choices, the table must be uniquely indexed and you must ensure that you refer to a fully qualified table name (eg `dbo.authors`) to guarantee that the BDE will find the unique index for the table you are selecting from. If you are the owner of the table (and therefore may want to use a non-qualified table name) the BDE will only find the unique index if your login name is the same as your database user name.

The reason for the use of the unique index is because only the first 32Kb comes back with the record. The BDE then needs to fetch the rest of the Blob in 32Kb chunks but it needs to uniquely identify the record to do this.

The only way to ensure a `TQuery` makes use of a unique index is to set `RequestLive` to `True` (the same applies to Oracle). If you *must* use a non-live query, you will find in the BDE Configuration application or BDE Administrator a setting for all SQL Aliases: `BLOB SIZE`. This defaults to 32Kb but can be set higher if larger Blobs are required.

Bear in mind that for non-live queries Blobs are buffered in the client's memory, so setting this higher will increase the memory overhead on the client. You may also need to use the BDE's `BLOBS TO CACHE` setting in order to ensure success.

Apparently these steps are unnecessary for InterBase, since it stores Blobs independently of the records that they are attached to. Each Blob has its own unique identifier so the BDE does not have to make use of a unique identifier of the record itself to retrieve them.

Changing Table Language

Q How do I programmatically (at run-time) change the language driver of a Paradox table?

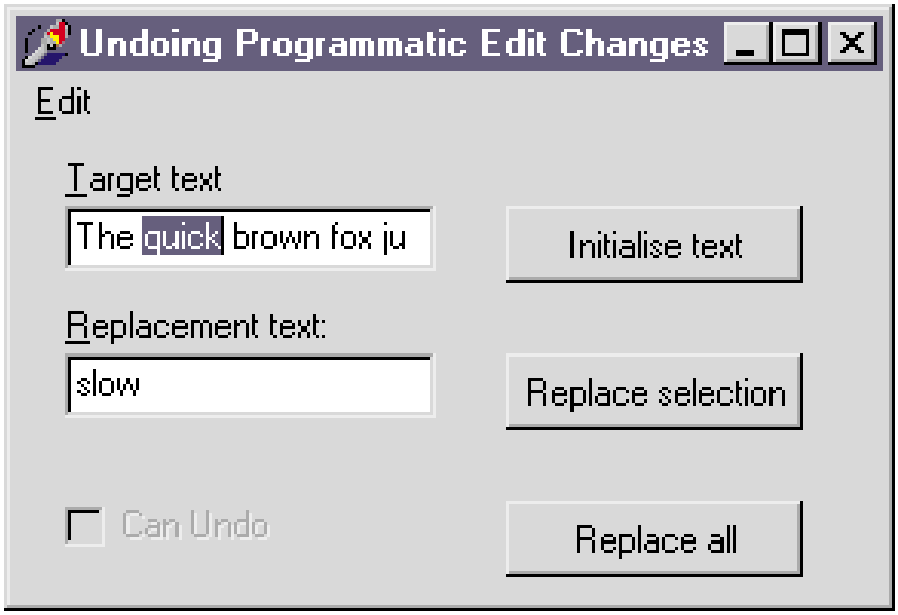
A You need to restructure the table to change the table language, therefore I would not recommend doing this "on the fly." However if the table is not being used, you can call `dbiDoRestructure` (a particularly cumbersome routine) to do the job. `dbiDoRestructure` is an IDAPI call defined in Delphi 1's DBIProcs unit or the 32-bit BDE unit. Information on how to use this API can be found in last issue's *Tips & Tricks* column.

Are You Running?

Q How can I check if an executable is running from a 32-bit Delphi application? I know how to check if forms are running by using the class or the caption, but I can't find out how to simply check if an EXE is running. I don't care how many times the exe is running. I have tried to record the program handles as I create the EXE, using `CreateProcess`, then check to see if these programs still exist using `GetWindow` but it doesn't work.

A You need to understand that Windows maintains many lists of many things. The way it refers to most of these is via handles. However, there are many types of handles. Windows, forms and dialogs are identified by window handles, executables are represented by process handles, icons are referred to via icon handles, etc.

`CreateProcess` can give you a process handle thanks to the `HProcess` field of the `TProcessInformation` record variable you pass to it. However `GetWindow` and `FindWindow` take window handles. These



➤ *Figure 2*

are completely different entities. To get success, pass the process handle to `WaitForSingleObject` and check the return value. If it returns `Wait_TimeOut` then the launched executable is still running. If, on the other hand, it returns `Wait_Object_0` then the program has terminated.

Listing 2 (a snippet from the `AppLnch` application on this month's disk) shows the idea. Note that `Win32Check` is a Windows API test routine that generates an exception with a nice message if the API fails. It is new in Delphi 3.

Opening Projects

Q How does Delphi determine what files to open when opening a project? Sometimes it seems to open the main form, sometimes it opens lots of forms and sometimes lately, rather annoyingly, it only opens the project source file.

A Generally speaking this is dictated by the state of the desktop saving option. This can be found on the `Preferences` page of Delphi 1's `Options | Environment...` dialog, Delphi 2's `Tools | Options...` dialog and Delphi 3's `Tools | Environment Options...` dialog. At the top right there is an option for auto-saving the desktop. If this is on, whenever a project is closed, Delphi generates an INI file with the projects name, but with a DSK extension. This file contains information about what files were open, and how various IDE windows looked when the project was closed. This allows the project to be brought back up looking much the same as it did when it was closed, with all the same files and forms opened.

Incidentally, if you are using Delphi 1 you will probably want to ensure the radio buttons to the left of the auto-save desktop option are set to save a Desktop file only (as opposed to Desktop and Symbols, the Delphi 1 default) to conserve disk space. See the *Delphi Clinic* in Issue 6 for more information on symbol files.

So if a desktop file is located when a project is opened, it is used. If no desktop file is found the general practice of the IDE is to open the main form and its corresponding form unit. Delphi 3.01 (and possibly 3.00 as well, but I don't have it installed to check) can make a slight variation on this theme which will show up if you have old Delphi 1 projects to work on.

First, some background information. Delphi 2 and 3 add one additional statement to a new project source file when compared with Delphi 1. The form creation statements are preceded by a call to `Application.Initialize`.

```
procedure TForm1.btnInitTextClick(Sender: TObject);
begin
  edtTarget.Text := 'The quick brown fox jumps over the lazy dog';
  edtTarget.SelStart := 4;
  edtTarget.SelLength := 5
end;
procedure TForm1.btnReplaceSelClick(Sender: TObject);
begin
  edtTarget.SelText := edtSource.Text
end;
procedure TForm1.btnReplaceAllClick(Sender: TObject);
begin
  edtTarget.Text := edtSource.Text
end;
```

➤ *Listing 3*

```
TEditUndo = class(TEdit)
{ These private fields will be initialised to False, an empty }
{ string and zero respectively, without any intervention }
private
  { To avoid unwanted calls to our additional code }
  FInternalOverwrite: Boolean;
  { Saved version of old Text property }
  FText: String;
  { Saved version of old SelStart property }
  FSelStart,
  { Saved version of old SelLength property }
  FSelLength,
  { Length of replacement text }
  FReplaceLength: Integer;
  procedure SaveContents(NewText: PChar);
  procedure RestoreContents;
protected
  { Called if Text property written to }
  procedure WMSetText(var Msg: TMessage); message wm_SetText;
  { Called if SelText property written to }
  procedure EMReplaceSel(var Msg: TMessage); message em_ReplaceSel;
  { Called when Ctrl-Z pressed }
  procedure EMUndo(var Msg: TMessage); message em_Undo;
end;
```

➤ *Listing 4*

```
procedure TEditUndo.SaveContents(NewText: PChar);
begin
  FText := Text;
  FSelStart := SelStart;
  FSelLength := SelLength;
  { Need to keep record of length of replacement }
  { text to ensure highlighting works when you }
  { repeatedly press the Undo key combination }
  FReplaceLength := StrLen(NewText)
end;
procedure TEditUndo.RestoreContents;
var
  TmpText: String;
begin
  { Need to ensure we can undo the undo }
  { i.e. perform a redo operation }
  { Swap saved text with current text }
  TmpText := Text;
  { Writing to Text will generate a wm_SetText message }
  { which we don't want to trap ourselves this time }
  FInternalOverwrite := True;
  Text := FText;
  FText := TmpText;
  FInternalOverwrite := False;
  { Restore old highlight}
  SelStart := FSelStart;
  SelLength := FSelLength;
  { Update other fields accordingly }
  FSelLength := FReplaceLength;
  FReplaceLength := SelLength;
end;
```

➤ *Listing 5*

Assuming you don't take advantage of this routine (see the *Startup And Shutdown Points* article in Issue 23 for details of how you could do this) it is only ever made use of by a Delphi program if you are writing an Automation or COM server, and so isn't generally required.

If you load up a project without a call to `Application.Initialize` and without an associated desktop file into Delphi 3.01 it will not bother loading the main form at all. It will leave you looking at the less-than-interesting project file, forcing you to use the project manager or the `Ctrl-F12` or `Shift-F12` keystroke combinations to get to any of your forms.

One possible way of alleviating this problem is to add:

```
{$ifdef Win32}
 Application.Initialize;
{$endif}
```

into Delphi 1 project files just before the form creation statements. An easier way would be to turn on auto-desktop saving.

Undoing In Edits

Q I am aware that `Alt-Backspace` or `Ctrl-Z` performs an undo operation on an edit control, but it seems to be restricted to undoing user interface modifications. For example, you can highlight text, press the `Delete` key and `Ctrl-Z` will undo it. However if you programmatically modify the `Text` or `SelText` property of an `Edit`, these normal undo keys do nothing. Can this be fixed?

A You are right in stating that programmatic edit changes are not undoable. You can verify this explicitly by asking the edit control at any time if it is able to offer an undo facility via its `em_CanUndo` message. Something like:

```
LongBool(Edit1.Perform(
  em_CanUndo, 0, 0))
```

will evaluate to either `True` or `False`.

To rectify the situation, we can make a new component inherited from `TEdit` and manually set up our own "undo buffer." When you write to the `Text` property, a `wm_SetText` message is sent to the underlying edit control. With `SelText` an `em_ReplaceSel` message is sent. We can write new message handlers for both these messages and save the old value before allowing the replacement to occur.

When the undo operation is requested, `em_Undo` is sent to the edit. We can also write a handler

for that and do our own undoing (if possible) if the control is unable (or unwilling) to help.

The new component is in the file EditUndo.Pas and is shown in action in the EditTest.Dpr project (see Figure 2). The top edit control is a TEditUndo component, whose native Windows undo ability is constantly reported in the check-box via a timer.

The first button sets up some text and highlights a portion of it as shown. The second button over-writes the highlighted text and the third button replaces the whole contents (see Listing 3). Finally, the Edit | Undo menu item simply sends an em_Undo message to the active control:

```
ActiveControl.Perform(
  em_Undo, 0, 0)
```

Admittedly this is not a good catch-all way to implement an undo menu option for edit-like devices. It doesn't cater for StringGrids or DBGrids, for example. A better lump of code was discussed in the *Delphi Clinic* back in Issue 5 *[That's a handy advert for our back issues CD-ROMs, thanks Brian! See Page 63. Editor]*.

The TEditUndo class has a few extra private fields and methods in addition to the message handlers already mentioned. See Listing 4.

The em_ReplaceSel and wm_SetText message handlers both use the SaveContents method to save the current value of the Text property along with SelStart and SelLength and also the length of the new piece of text (which is passed across to SaveContents). The em_Undo handler calls RestoreContents to set things back how they were, assuming the edit itself cannot do it.

Listing 5 shows these two helper routines. You might notice that RestoreContents performs its restoration by writing to the Text property, which will cause a wm_SetText message to be sent back to the control, which would call SaveContents even though we wouldn't want to do any saving at this point. To avoid unwanted saving, a Boolean data field is made use of. The wm_SetText message handler only calls SaveContents if FInternalOverwrite is False.

The em_Undo message handler only calls RestoreContents if the edit refuses to perform its own undo operation and FText has something in it, otherwise the default edit functionality is invoked.

Acknowledgements

Thanks to Steve Axtell for the database information used in this month's column.