The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

Just email Brian Long, our Delphi Clinic Editor, on 76004.3437@compuserve.com or write/fax us at The Delphi Magazine

Radio Group Focus

Q When I try and set focus to a radio group, with the intention of focusing on the current radio button within it, the focus disappears. What seems to happen is that the radio group itself takes focus. I would explicitly set focus to one of its radio buttons, but they are represented by the `Items` property, a `TStrings` object, and so I can't find a window handle. What do I do?

A Well, when you get the group box focused, a press of the `Tab` key sets focus on the current radio button. You need a program statement that achieves the same thing. Something like this:

```
RadioGroup1.SetFocus;
SelectNext(ActiveControl,
  True, True);
```

`SelectNext` focuses on the next component on the form in the tab order. It appears the radio buttons come after the group box.

Drag And Drop On Grid Cells

Q I have a `DBGrid` that I would like to drag and drop a value onto. It works fine when I drop the value into the currently selected row; however, I would like to be able to drop it into the record the drag cursor is over when the mouse button is released. Is there any way to change the selected record in the grid based on the location of the drag cursor?

A To change the record normally, you click with the mouse. Here is one possible solution, where the `OnDragDrop` event handler simulates a mouse click at the cursor position. Rather than limit this answer to a `TDBGrid`, I have also implemented a handler for a `TStringGrid`. In my example project (DRAGDROP.DPR on the disk) there is a string grid and a database grid, which will both receive a value dragged over from a file list box (`SourceControl`). The two event handlers are shown in Listing 1.

The string grid writes a value to a cell using the two-dimensional string array property, `Cells`, using `Selection` to identify the currently selected cell. The database grid uses the `SelectedField` property to write a value to the target field: it also ensures the dataset the field belongs to is in `Edit` mode first. The `OnDragOver` method is simply:

```
Accept :=
  Source = SourceControl;
```

256 Colour TImages

Q When I set the `Stretch` property of a `TImage` which contains a 256 colour bitmap to `True`, its colours get mashed. Is this a VCL bug, and is there a fix?

A I have heard that this does not occur on some video drivers, and so the problem could be a `TImage` problem or a Windows video driver feature. Rather than decide who is to blame, let's see if we can fix it.

There are two approaches here, and both involve using an intermediate bitmap to copy the `TImage`'s bitmap onto. It seems that when the bitmap is copied onto a canvas directly, the palette is not fixed up, or 'realized' correctly, however if it is copied onto another bitmap, it magically is. If you are brave enough to modify the VCL source (which you may not even have, depending on what products you have purchased), you can try the first approach, otherwise you can use the substitute component presented below.

First Method. Find the EXTCTRLS.PAS file and locate the `TImage.Paint` method. Add to it a second local variable:

```
Bmp: TBitmap;
```

Now go down to the end of the method and find:

➤ *Listing 1*

```
procedure TForm1.StringGridDragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  if Source <> SourceControl then Exit;
  with Sender as TStringGrid do begin
    Perform(wm_LButtonDown, 0, MakeLong(X, Y));
    Perform(wm_LButtonUp,   0, MakeLong(X, Y));
    Cells[Selection.Left, Selection.Top] := SourceControl.FileName;
  end;
end;
procedure TForm1.DBGridDragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  if Source <> SourceControl then Exit;
  with Sender as TDBGrid do begin
    Perform(wm_LButtonDown, 0, MakeLong(X, Y));
    Perform(wm_LButtonUp,   0, MakeLong(X, Y));
    SelectedField.DataSet.Edit;
    SelectedField.AsString := SourceControl.FileName;
  end;
end;
```

```
with inherited Canvas do
  StretchDraw(Dest,
    Picture.Graphic);
```

Before these lines insert the block of code in Listing 2. If this compiles okay, copy the resultant EXTCTRLS.DCU file into your DELPHI\LIB directory (backing up the old one first) and then rebuild the component library (`Options | Rebuild Library`).

You can see that if the `Stretch` property is set, the code copies the picture to another bitmap before drawing it on the image's canvas. The use of `inherited` against a property in the original and modified code is worth exploring here, as it causes a headache when attempting to put similar code in a new component. The `TImage` is derived from a `TGraphicControl` which has a `Canvas` property, referring to the screen space where it will draw. `TImage` redefines `Canvas` to refer to the bitmap's canvas instead. When it comes to draw itself in the `Paint` method, it must use the word `inherited` to access the real canvas, declared in the `TGraphicControl` class.

Unfortunately there is no way for a class inherited from `TImage` to get access to this proper canvas, two levels up the inheritance tree, so we have to use other sneaky devices to achieve the goal.

Second Method. The `TNewImage` component in Listing 3 (the file IMAGE2.PAS) traps the `wm_Paint` message (well it's not a real message: a `TImage` does not have a window handle, but let's not get too involved here) and obtains a Windows device context handle from the message parameters. It uses this to set up a temporary canvas that can be used by the new `Paint` method. Notice that to allow the entirety of the component to be seen at design time, the new code only executes if a stretched bitmap is present. In other cases, the usual surrounding dashed line will be seen.

Using this approach has a side benefit, as now things other than bitmaps can be stretched as well: try and load an icon or a metafile into a `TImage` and set the `Stretch` property to `True`. In case you have lots of `TImage` components in use that you want to replace with `TNewImages`, but can't face the ordeal of deleting the originals, adding `TNewImages`, setting the `Stretch` property and loading a `Picture`, here is an alternative. Load your form as a text file using `File | Open File`, and choosing `Form Files` from the `file types` combo box. Now do a search and replace of `TImage` with `TNewImage` and close the file. Finally open the form's unit file normally and do the same search and replace through the form class definition. Problem solved.

This approach comes in very handy if you start working on a `TTable` component, use the `Fields Editor` to set up all the field objects and start writing code, only to realise that you should have started with a `TQuery`. If you were to delete the `TTable`, all the field objects would also be deleted. It is much easier to change the definition of a `TTable` to be a `TQuery` and alter the differing property (ie change the `TableName` property to be an SQL property formatted appropriately – you can find what the right format is by examining an existing `TQuery` component in text mode).

➤ *Listing 2*

```
if Stretch then begin
  Bmp := TBitmap.Create;
  try
    Bmp.Height := Picture.Height;
    Bmp.Width  := Picture.Width;
    Bmp.Canvas.Draw(
      0, 0, Picture.Graphic);
    inherited Canvas.StretchDraw(
      Dest, Bmp);
  finally
    Bmp.Free;
  end;
end
  else
```

➤ *Listing 3*

```
unit Image2;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, ExtCtrls;
type
  TNewImage = class(TImage)
  private
    FCanvas: TCanvas;
    FBmp: TBitmap;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure WMPaint(var Msg: TWMPaint);
      message wm_Paint;
    procedure Paint; override;
  end;
procedure Register;
implementation

constructor TNewImage.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { Can't draw on the TImage canvas - that turns out to
    be the bitmap object's canvas }
  FCanvas := TCanvas.Create;
  { Temporary bitmap to cause palette realization }
  FBmp := TBitmap.Create;
end;

destructor TNewImage.Destroy;
begin
  FBmp.Free;
  FCanvas.Free;
  inherited Destroy;
end;

procedure TNewImage.WMPaint(var Msg: TWMPaint);
begin
  { Identify what the real canvas is }
  FCanvas.Handle := Msg.DC;
  { Do normal stuff, like call Paint }
  inherited;
  { Now forget about it }
  FCanvas.Handle := 0;
end;

procedure TNewImage.Paint;
begin
  { Only do new stuff if it is a stretched image }
  if (Picture.Graphic = nil) or not Stretch then
    inherited Paint
  else begin
    FBmp.Height := Picture.Height;
    FBmp.Width  := Picture.Width;
    FBmp.Canvas.Draw(0, 0, Picture.Graphic);
    FCanvas.StretchDraw(ClientRect, FBmp);
  end
end;

procedure Register;
begin
  RegisterComponents('Samples', [TNewImage]);
end;
end.
```

Clipboard Stuff

Q How can I implement generic cut/copy/paste menubar functionality right across `TEdit`, `TDBEdit`, `TStringGrid`, `TDBGrid`, `TMemos` and `TDBMemo` controls?

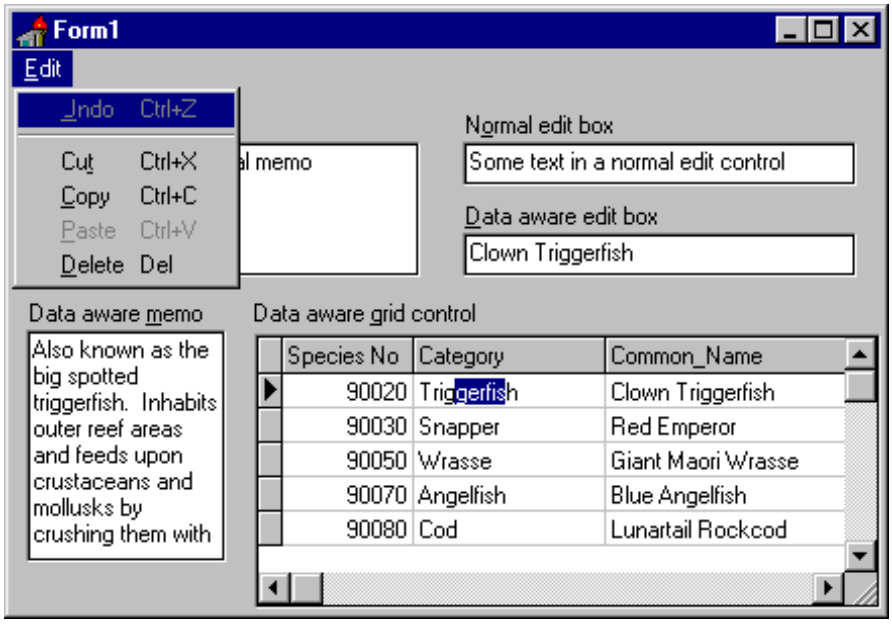
A To make common code work across these components we need some functionality common to them all. This is ok for edits and memos, as they're all based at some point on `TCustomEdit`, but what about the grids?

Well, when it comes to editing on a grid the component makes use of a specialised in-place edit control called a `TInPlaceEdit`. This is also based on `TCustomEdit`, and so we need to know how to get a handle on this object, so we can call its clipboard-type functionality.

The code in Listing 4 contains two routines used in the program shown in the screenshot. When the edit menu is invoked, the `EditMenuClick` event handler is invoked to identify if there is a `TCustomEdit` derivative around to work with. If the active component is an edit or memo, then the target has been found. However if it is a grid, it is more involved. Even when a grid has an in-place editor active, the grid is still the active component as far as the form is concerned. If a `TCustomGrid` is active, the code cycles through its components until it finds a visible `TInPlaceEdit`.

When a `TCustomEdit` descendant is located it is assigned to a data field called `EditCtl` which I have added to the form's declaration. If an edit control is found, the various menu items need to be enabled or disabled, depending on the current state of both it and the clipboard; ie whether there is any selected text, if there is any text in the clipboard, etc.

All the menu items that hang off the `Edit` menu use the same event handler. To distinguish between the menu item that triggered the event, they have all had their `Tag` properties set to unique values. Providing `EditCtl` refers to a valid object, the code performs a standard edit control clipboard type of operation, such as `CutToClipBoard` or `ClearSelection`. The one exception is for the `Undo` menu item, which uses a windows message to achieve its goal instead.



Updates From Issue 4

Several readers contacted us with more elegant or efficient ways of doing a left zero fill. Jack Bakker and Niek de Ruitjer reminded us about Delphi's `Format` routine, which will do the job quite nicely:

```
Format('%.5d', [123]); {00123}
```

See the online help for 'Format Strings' for more details (the features are quite comprehensive so it should meet most needs).

Also, Alan Gregory send in a less dirty solution for updating a file listbox. Simply call its `Update` method.

Acknowledgements

Thanks to Roy Nelson of Borland for the `TDBImage` VCL fix and idea for the `TDBImage` replacement.

➤ *Listing 4*

```
procedure TForm1.EditMenuClick(Sender: TObject);
var
  Loop: Byte;
begin
  EditCtl := nil;
  if ActiveControl is TCustomEdit then
    EditCtl := ActiveControl as TCustomEdit
  else if (ActiveControl is TCustomGrid) then
    with TCustomGrid(ActiveControl) do
      { When editing in a grid, the grid is the active
        control not the in-place editor, so we need to find
        the editor in the grid. If grid owns any controls,
        cycle through them checking for editor}
      if ControlCount > 0 then
        for Loop := 0 to Pred(ControlCount) do
          if Controls[Loop] is TInPlaceEdit then
            { Editor is visible when being used }
            if Controls[Loop].Visible then begin
              EditCtl := TInPlaceEdit(Controls[Loop]);
              Break;
            end;
  if Assigned(EditCtl) then begin
    Undo1.Enabled :=
      Bool(EditCtl.Perform(em_CanUndo, 0, 0));
    Cut1.Enabled := EditCtl.SelLength > 0;
    Copy1.Enabled := Cut1.Enabled;
    Paste1.Enabled := ClipBoard.AsText <> '';
    Delete1.Enabled := EditCtl.SelLength > 0;
  end else begin
    Undo1.Enabled := False;
    Cut1.Enabled := False;
    Copy1.Enabled := False;
    Paste1.Enabled := False;
    Delete1.Enabled := False;
  end;
end;

procedure TForm1.MenuClick(Sender: TObject);
begin
  if Assigned(EditCtl) then with EditCtl do
    case (Sender as TComponent).Tag of
      1: Perform(em_Undo, 0, 0);
      2: CutToClipBoard;
      3: CopyToClipBoard;
      4: PasteFromClipBoard;
      5: ClearSelection;
    end;
end;
```