The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Delphi 3 Display Glitch

Q I have just bought a new PC and have installed my Delphi 3.01 on it. It all functions okay, but the Component Palette is displayed wrongly. There seems to be some internal image maintenance problem, which is hard to describe but garbles up all the component palette bitmaps. I have sent you a screen dump to clarify what I mean. How can I fix this?

A The screen dump is shown in Figure 1, you can see how bad things are by noting that the mouse is over the `TEdit` component type which is being displayed as a combination of half a `TStringGrid` and half a `TPOP` component. I have seen this problem elsewhere and sometimes the Component Palette images get so scrambled that it looks like a television that has not been tuned. It appears that the problem stems from which version of the common control library (file COMCTL32.DLL) you have installed.

Delphi 3's README says you need at least version 4.70 of this library for some of the properties of some components on the Win32 page of the component palette to function and for `TToolBar` and `TCoolBar` to work at all. This is because the underlying Windows control set was improved and extended in that version of the DLL.

Apparently, Delphi 3 was tested and developed with an even later version of the control library (4.71) and due to certain internal DLL differences, it seems that the image list support goes a bit doo-lally on some video drivers. To remedy the situation, you will need a later version of the DLL. You can get this by installing a very recent version of Internet Explorer. This certainly solved the problem on my machine.

➤ *Figure 1*

To check which version you have installed, use Windows Explorer to navigate into your Windows\System or WinNT\System32 directory, then locate and select COMCTL32.DLL. Next, either right-click on it and choose `Properties`, hold `Alt` down while double-clicking it, or press `Alt-Enter`. The second page of the resultant dialog has version information on it. The `File Version` should be reading at least 4.71.

Troublesome RichEdits

Q How do I copy the contents of one `TRichEdit` component to another? I have tried copying the `Text` property across, and also the `Lines` property, but this loses all the rich-text formatting, regardless of the value of the `PlainText` property.

A You can easily do this using the clipboard, using the `CopyToClipboard` and `PasteFromClipboard` methods, but this erases anything already stored in the clipboard by the user. You could also save to a temporary file to write to and read back from. Listing 1 shows how to do this using the Windows APIs that firstly tell you where the Windows `TEMP` directory is and secondly make up a temporary file name. The straightforward `SaveToFile` and `LoadFromFile` methods are used before the file is finally deleted.

The temporary file solution is fine, but is not particularly efficient. An in-memory solution would be preferable. Probably the best way is to write the `Lines` property of one of them out to a stream (such as a memory stream) and then read it back in to the other `Lines` property.

Even though the `Lines` property is declared as type `TStrings`, it is in fact an object of a type inherited from `TStrings`, tailored to working with rich edit controls. The `TRichEditStrings` object deals with adding all the formatting characters into the text string, as normally the attributes and text are stored separately.

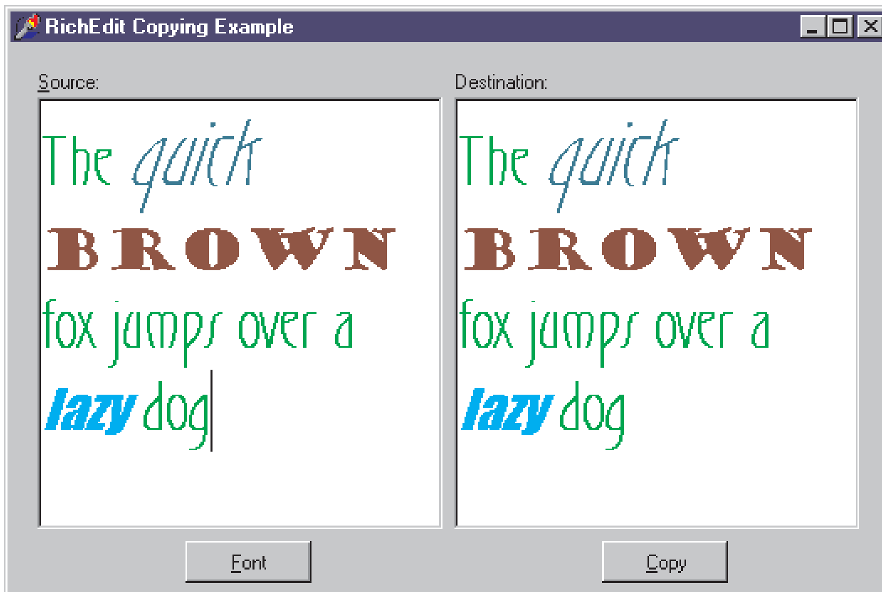
The RICHEDIT.DPR project on the disk demonstrates using a temporary memory stream as a way of

➤ *Listing 1*

```
var
  PathName, FileName: array[0..Max_Path] of Char;
...
Win32Check(Bool(GetTempPath(SizeOf(PathName), PathName)));
Win32Check(Bool(GetTempFileName(PathName, '~XX', 0, FileName)));
try
  reSource.Lines.SaveToFile(FileName);
  reDest.Lines.LoadFromFile(FileName)
finally
  DeleteFile(FileName)
end
```

```
procedure TForm1.btnCopyClick(Sender: TObject);
var
  stmStorage: TMemoryStream;
begin
  stmStorage := TMemoryStream.Create;
  try
    reSource.Lines.SaveToStream(stmStorage);
    stmStorage.Position := 0;
    reDest.Lines.LoadFromStream(stmStorage)
  finally
    stmStorage.Free
  end
end;
```

➤ *Listing 2*



➤ *Figure 2*

copying the content of a rich edit control. It is shown in Figure 2 after the `Copy` button has been pushed (whose event handler is shown in Listing 2).

Rounding Inconsistency

Q I was taught that when rounding fractional numbers, a *.5 value always rounds up to the next larger whole number. Delphi does not seem to agree with me all the time. Both 2.5 and 1.5 round to 2 when using the `Round` function.

A You describe one approach to rounding. What you are seeing is another approach called Banker's Rounding. Unlike your regular rounding where you always round up, banker's rounding rounds to the nearest even number. This gives a more even distribution of rounding since, on average, half the *.5 numbers are rounded up and half are rounded down. This does not affect numbers with fractional parts greater or less than .5 which are automatically rounded to the closest number.

As the name suggests, Banker's Rounding comes from financial situations, where monetary values are often formed by rounding accurate figures down to values with fewer decimal places. A fractional digit of 0 needs no rounding. 1, 2, 3 and 4 round downwards; 6, 7, 8 and 9 round upwards. To ensure there is no overall bias either way the number 5 rounds either up or down, towards the nearest even value.

Unfortunately, the Delphi online help does not suggest that this is the case. It gives the declaration for the `Round` function, taking a parameter `X` of type `Extended` and then says, *"If X is exactly halfway between two whole numbers, the result is the number with the greatest absolute magnitude"* which is not true.

Incidentally, in the question you say that Delphi is performing different rounding to what you expect. However, it is your numeric coprocessor that is implementing this Banker's Rounding, `Round` is implemented by floating point coprocessor instructions. To emphasise this, the project ROUND.DPR on the disk performs a rounding operation both by using the `Round` RTL function, and also by calling the coprocessor directly with assembler. Listing 3 shows the two button event handlers that give identical results.

If you wish to avoid this Banker's Rounding and have the `Round` function always round upwards, then you will need to add a very small value, such as 0.000001, onto the argument before passing it to `Round`.

Icons, Metafiles And Graphic Fields

Q I have a program that stores pictures read from disk into a database. This works fine for bitmap files, but whenever I try to put an icon or metafile into the database I get an `EInvalidGraphic` exception saying *Bitmap image is not valid*. Is there an easy fix?

A Since this question and the *Pictures In Databases* one from last month's *Delphi Clinic* are related to exactly the same sort of thing, I ensured that the IMGTEST.DPR project on last month's disk could reproduce the problem. The open dialog component used in that project allows you to choose bitmaps or icons or metafiles.

The problem arises if you try to load any non-bitmap file into a graphic field. As was mentioned previously, Delphi has various classes inherited from `TGraphic` that allow you to work with bitmaps, metafiles and icons. Delphi 3 added a new class for JPEG files in the `JPEG` unit. Other file formats could be supported if you could find or write implementations of appropriate classes.

Incidentally, talking of the `JPEG` unit, it is easy to use it at runtime

by adding it into your `uses` clause, but you can also enable the design-time environment to make use of `JPEG` images (for example in `TImages`). Adding the `JPEG` unit into a design-time package can do this. Simply make a new package (`File | New... | Package`) and give it a file name and location (and maybe a description). Next, from the package editor, choose `Add`, push `Browse` and choose `Delphi compiled unit (*.dcu)` from the `Files of type:` combobox. Now navigate your way into Delphi's LIB directory and choose the JPEG.DCU file. Finally hit the Package Editor's `Install` button. I have supplied a sample package source file on the disk (JPEGENABLER.DPK) to save you the trouble of doing this. Simply open it (`File | Open…`), choose `Delphi package source (*.dpk)` from the `Files of type:` list, select the file and hit the `Install` button.

Now back to the main plot... The code in Listing 4 uses the `LoadFromFile` method of a `TBlobField`. Internally, this does much the same as the expanded version that was shown in last month's Listing 7, making use of a blob stream to do the file content storage. It is the blob stream that starts the chain of events that ends up giving you the exception, but the data aware control (the `TDBImage`) is what actually causes the problem. During the blob stream's destruction it calls a field notification method that eventually tells any data-aware controls connected to the field that the underlying field value has changed. The `TDBImage` refreshes its view of the value by calling its `LoadPicture` method, which passes the field object to the `Assign` method of the `TPicture` object represented by its `Picture` property. Following this, the `TBlobField`'s `AssignTo` method attempts to save its contents in a `TBitmap` object which, if the field has an icon or metafile in it, will fail. This gives the exception.

I guess one solution would be not to use a `DBImage`, but to use a normal `TImage` instead. As you go from record to record, you could read the appropriate graphic field's contents into a blob stream

```
procedure TForm1.Button1Click(Sender: TObject);
var AFloat: Extended;
    AnInt: Longint;
begin
  AFloat := StrToFloat(
    InputBox('Rounding Test', 'Enter a floating point number', '2.5'));
  AnInt := Round(AFloat);
  Label1.Caption := Format('Round(%f) = %d', [AFloat, AnInt])
end;
procedure TForm1.Button2Click(Sender: TObject);
var AFloat: Extended;
    AnInt: Longint;
begin
  AFloat := StrToFloat(
    InputBox('Rounding Test', 'Enter a floating point number', '2.5'));
  asm
    { Push AFloat onto co-processor stack }
    FLD     AFloat
    { Round float to an integer }
    FRNDINT
    { Pop integer into AnInt }
    FISTP   AnInt
    { Ensure co-pro ops are complete before proceeding }
    FWAIT
  end;
  Label1.Caption := Format('Round(%f) = %d', [AFloat, AnInt])
end;
```

➤ *Listing 3*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDlg.Execute then begin
    Table1.Insert;
    Table1Graphic.LoadFromFile(OpenDlg.FileName);
    Table1Common_Name.Value :=
      ExtractFileName(OpenDlg.FileName);
    Table1.Post
  end
end;
```

➤ *Listing 4*

```
function LoadStreamIntoGraphic(Graphic: TGraphic; Stream: TStream): Boolean;
begin
  Result := True;
  try
    Stream.Position := 0;
    Graphic.LoadFromStream(Stream);
    { Often, an icon object will assume it represents the }
    { stock application icon if it has unknown data in it }
    if Graphic is TIcon then
      if TIcon(Graphic).Handle = LoadIcon(0, IDI_APPLICATION) then begin
        Graphic.Assign(nil);
        Abort
      end
  except
    Result := False
  end
end;
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
var
  BlobStream: TBlobStream;
 {$ifndef DelphiLessThan3}
  JPEGImg: TJPEGImage;
 {$endif}
begin
  { Only execute this code if graphic field was modified or record was changed }
  if not Assigned(Field) or (Field = Table1Graphic) then begin
    BlobStream := TBlobStream.Create(Table1Graphic, bmRead);
    try
      if not LoadStreamIntoGraphic(Image1.Picture.Bitmap, BlobStream) then
      if not LoadStreamIntoGraphic(Image1.Picture.Metafile, BlobStream) then
      if not LoadStreamIntoGraphic(Image1.Picture.Icon, BlobStream) then
     {$ifndef DelphiLessThan3}
      begin
        JPEGImg := TJPEGImage.Create;
        try
          { A TPicture does not have a JPEG property, so we have to do without }
          Image1.Picture.Graphic := JPEGImg;
          LoadStreamIntoGraphic(Image1.Picture.Graphic, BlobStream)
        finally
          JPEGImg.Free
        end
      end
     {$endif}
    finally
      BlobStream.Free
    end
  end
end;
```

➤ *Listing 5*

and then use the LoadFromStream method of the TGraphic object in the TImage.

The IMGTEST2.DPR project implements this alternative approach. Listing 4 is still used to get data into the table, but now a TImage is used instead of a TDBImage. Listing 5 shows the code found in the data source's OnDataChange event handler along with the utility routine it employs. Assuming the OnDataChange event was triggered due to the graphic field being changed, or the current record being scrolled, it proceeds with its logic.

Unfortunately, due to a small wrinkle, the logic was not quite as plain and simple as I would have liked. All the images that my supplied code puts into the table can be read very easily, but those fish images already present in the table have a Paradox file format graphic header in front of them that causes problems. So in the case of bitmap images, the logic differs from all the other types. I'll come back to these differences later.

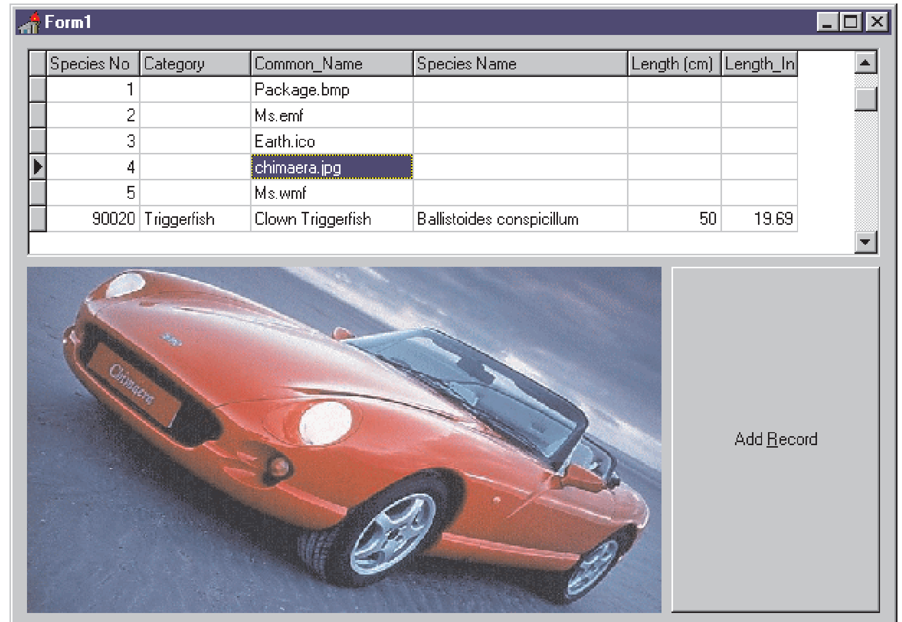
In the general case, a blob stream is manufactured to extract the contents of the field and an attempt is made to load it into a TBitmap, TMetafile and TIcon in turn. The three TPicture properties used to do this (Bitmap, Metafile and Icon) are programmed such that if the underlying graphic object is not of the appropriate type, it is disposed of and an object of the specified type is manufactured.

Finally in this routine, if none of the available graphic types seem to recognise the image, then (assuming Delphi 3 or later is being used) an attempt is made to load the image into a TJPEGImage object. Since TPicture objects do not inherently understand JPEG images, the code has to take this into account and set the JPEG object up manually.

The LoadStreamIntoGraphic routine is reasonably straightforward. It ensures the stream pointer is reset to the start of the stream and tries to load the image from the stream into the graphic object. In addition it deals with a problem that sometimes arises. When the TIcon type is told to read an image from a stream, it sometimes mistakes a non-icon image for a reference to Windows' stock application icon (due to the occurrence of a zero value in the data block that would normally be the icon header). Since references to stock icons are unlikely to be stored in a database table, if the code sees a stock icon being used it clears it immediately.

So that caters for the approach where you have no Paradox-stored bitmaps in your table. However, when working with something like the BIOLIFE table, problems arise. Due to the aforementioned Paradox graphic header, the current code fails to load these bitmaps into the image. So onto the final project, IMGTEST3.DPR. This one changes the logic for the bitmap load attempt. Listing 6 shows the main differences and Figure 3 shows the program running, displaying a JPEG image from the table.

One final note. The JPEG code was compiled using conditional compilation. Since it was introduced in Delphi 3 and will be present in future versions, none of the predefined symbols seemed adequate to ensure JPEG support was available where sensible. So the DelphiLessThan3 symbol used in Listing 5 is a hand-crafted symbol, made with the following compiler directives:



➤ *Figure 3*

```
function LoadFieldIntoBitmap(Bitmap: TBitmap; Field: TBlobField): Boolean;
begin
  Result := True;
  try
    Bitmap.Assign(Field)
  except
    Result := False
  end
end;
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
  ...
  if not LoadFieldIntoBitmap(Image1.Picture.Bitmap, Table1Graphic) then begin
    BlobStream := TBlobStream.Create(Table1Graphic, bmRead);
    try
      if not LoadStreamIntoGraphic(Image1.Picture.Metafile, BlobStream) then
        if not LoadStreamIntoGraphic(Image1.Picture.Icon, BlobStream) then
          ...
    finally
      BlobStream.Free
    end
  end
end;
```

➤ *Listing 6*

```
{$ifdef Ver80} { Delphi 1.0x }
 {$define DelphiLessThan3}
{$endif}
{$ifdef Ver90} { Delphi 2.0x }
 {$define DelphiLessThan3}
{$endif}
{$ifdef Ver93}
 { C++ Builder 1.0x }
 {$define DelphiLessThan3}
{$Endif}
```

Thunking: Calling 16-Bit Code From 32-Bit

The business of calling 16-bit code from 32-bit Windows 95 apps seems to be riddled with problems. I first wrote about how to do this using the undocumented `QT_Thunk` API in an article in Issue 12 of *The Delphi Magazine* (August 1996). This was followed by an additional, simplifying, routine in the *Tips & Tricks* column of Issue 13, and then by a bug-fix in Issue 16's *Delphi Clinic*.

Now I have another update. I was notified by several readers, who had clearly purchased shiny new machines more recently than I had, that my code failed to work on the newer Windows 95 release, OSR2. It seems that in Windows 95 OSR2, the `QT_Thunk` API was modified slightly. Because of the way I was accessing the API, the previously supplied code failed to work.

Having at last obtained a copy of OSR2, I have analysed the problem and amended the source files appropriately. On the disk accompanying this issue are updated versions of all the files from the original article, which have been successfully tested on both versions of Windows 95 (and also a pre-release version of Windows 98). The files are in QTTHUNK.ZIP in the CLINIC directory.

The online version of the article located at www.itecuk.com/delmag/thunk95.htm has also been updated to reflect these changes (and the source code is also there now), so you may wish to check it out, in order to make sure you aren't missing anything important. I think the primary practical difference is a warning about using typed constant `PChars` in calls to the `Call16BitRoutine`: use zero-based arrays of `Char` instead to avoid Access Violations.

Thanks are due to Simon Chang, Diego Barros, Jinglei Duan and James A Whelan for alerting me to the problem.

January 1998 *Surviving Client/Server* Erratum

In the January 1998 *Surviving Client/Server* column dealing with freeform text indexing, there was a bug in the memo scanning code. If the last character of the memo was not punctuation or whitespace, then the last keyword of the memo was lost. The corrected code can be found on this month's disk in the SURVIVE directory, in the file NEWJAN98.ZIP, (in the `TMemoScanner.Scan` method).

Steve Troxell