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

Outline Bug

Q There is a problem with the `TOutline` component in Delphi 1 and 2. Place an outline on a form and use its `Lines` property to add one line to it. Run the program and either double click the line, or press the plus key, which normally expands an expandable node. Obviously it will have no effect here. Now press the `End` key, which usually takes you to the last item at the current hierarchy level. You get a *List index out of bounds* exception.

A This bug seems fairly easy to fix in the VCL source code. It is caused by the `KeyDown` method of `TCustomOutline` not checking if an expanded node actually has child nodes. It just moves to the last child node and if there are no children then the last will return something that is indeed "out of bounds."

In `TCustomOutline.KeyDown`, in OUTLINE.PAS, you need to change the code for the `VK_END` case to be as shown in Listing 1.

Back To Front Math Unit

Q The new `MinValue` and `MaxValue` routines in the Math unit don't work. The following line gives 0.7 as the maximum passed value:

```
ShowMessage(FloatToStr(
  MaxValue([5, 3, 7, 9.2,
  0.7, 8])));
```

A This problem has been spotted and reported. The implementations of both `MinValue` and `MaxValue` use the wrong comparison operator. If you have the source, you can fix it, otherwise you will need to use what would appear to be the wrong routine to get the right effect. This will presumably be remedied in a maintenance release, so be prepared to change them back to the right way round.

Autosizing TDBGrid

Q How can I get a `DBGrid` to set its own width so as to have no white space past its rightmost column?

A To do this requires identifying the width of all the columns in the grid, the width of the scroll bar and the width of the borders, if there are any. Of course, if this turns out to be wider than the form, then it will need to be truncated somewhat. The column widths are available inside the `TDBGrid` itself, but are not surfaced as `public` properties. The following extract from a simple `TDBGrid` derivative (found in OPTGRIDU.PAS) shows the implementation of a property called `OptWidth` that can be used to find the best value. The GRIDFIT.DPR project on the disk shows how to use it.

The project allows any query to be typed in and executed and the query's `AfterOpen` event handler sets the grid's width to the `OptWidth` value. One point worth mentioning is that the query is opened in the `OnCreate` handler. To ensure that this property works at this early stage in the program's lifetime, I send a `wm_Paint` message to the grid's window procedure (not by using Windows calls, but by using the Delphi `Perform` method). This forces it to calculate its initial column widths. In Delphi 1, this is needed since, at that point, all columns in the grid are considered to be all the size of the default column width. See Listing 2.

No New Record In A TDBGrid

Q On a `TDBGrid`, how can I stop the user opening up a blank record when they press `Tab` or the down arrow on the last field of the last record?

A My proposed solution involves setting the form's `KeyPreview` property to `True` and writing an `OnKeyDown` handler for it as shown in Listing 3.

If the down arrow key is pressed, the keypress is let through if the dataset is not on the last record. If the `Tab` key is pressed on the last tab stop, then again a check is made on the state of the dataset before letting the key though. GRIDS.DPR shows this in action.

Reading Form Properties

Q I need to read a form's properties from a program not written in Delphi. The form file format is not clear and so I am having

➤ *Listing 1*

```
VK_END:
  begin
    Node := TOutlineNode(FRootNode.List.Last);
    while Node.Expanded { add this } and (Node.List.Count > 0)
      { end of new bit } do
      Node := TOutlineNode(Node.List.Last);
    SelectedItem := Node.Index;
    Exit;
  end;
```

trouble. How can I get access to the property data? I have no problems with launching a small Delphi program that generates a text file, but don't know how.

A You didn't specify whether you meant from a form file, or a form resource bound into an executable. So we'll do both.

When a form is created at runtime in a Delphi application, the form resource is read in and the property values are read in. There is code in the VCL to do this. When you open a form file in the Delphi editor, or use the command line CONVERT.EXE tool, you can translate between a form file and a text file. This facility is also in the VCL. The important routines are `ObjectTextToBinary`, `ObjectBinaryToText`, `ObjectResourceToText` and `ObjectTextToResource`, all of which take an input stream and an output stream. On the disk is a project called READFRM.DPR which shows how to generate a text file from a form file (`ObjectResourceToText`) and also from a form resource (`ObjectBinaryToText`). The program is shown in Figure 1.

If compiled in Delphi 1, the code uses a `THandleStream`. In Delphi 2, it uses the new `TResourceStream`. The two routines from the form unit REDFRMU are shown in Listing 4.

Optimising TPaintBox Painting

Q I am drawing on the `Canvas` of a `TPaintBox`. Under certain circumstances, for efficiency, I want to clear and then repaint only a part of the canvas (like `InvalidateRect` in the Windows API, but that doesn't work since the paint box hasn't got a window handle). In other words I want the `Canvas.ClipRect` to be smaller than the visible area of the `PaintBox`. How do I do that?

A You are correct in saying that `TPaintBox`, for example `TSpeedButton` and `TImage` among others, does not have a window handle available through a `Handle` property. This is because it is not a `TWinControl` descendant. However, those clever people at Borland allow all these components to manage themselves by acting as though they have a window handle. Likewise, `TGraphicControl` descendants (such as `TPaintBox`) don't have real canvases (or device contexts, as API programmers refer to them). They get to use a portion of their `TWinControl` owners' canvas.

All `TControl` descendants have a window procedure method, `WndProc`, which despatches messages to message handlers. All this leads up to the fact that a `TPaintBox` has a `wm_Paint` message handler that can be overridden to do whatever optimised painting you like.

The `wm_Paint` message is passed with a device context handle. We can make a new clipping region by

➤ *Listing 2*

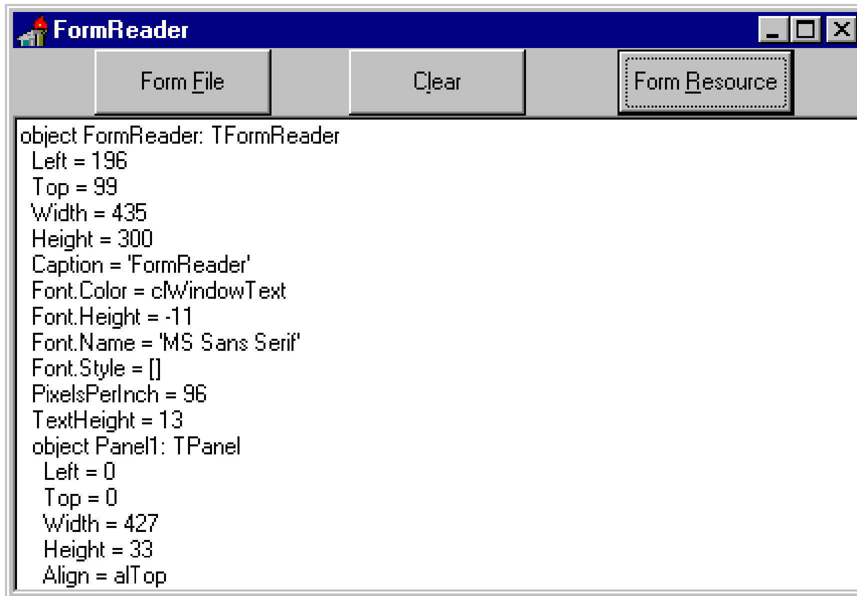
```
function TDBOptGrid.GetOptWidth: Integer;
var Loop: Integer;
begin
  Result := GetSystemMetrics(sm_CXVScroll);  { Vertical scroll bar }
  { Left and right borders }
  if BorderStyle = bsSingle then
    Inc(Result, 2 * GetSystemMetrics(sm_CXBorder));
  { Each column, possibly including the indicator }
  for Loop := 0 to Pred(ColCount) do
    Inc(Result, Succ(ColWidths[Loop]));
  { Make sure it fits in parent }
  if Parent is TForm then begin
    if Result > TForm(Parent).ClientWidth then
      Result := TForm(Parent).ClientWidth;
  end else
    if Result > Parent.Width then
      Result := Parent.Width;
end;
```

➤ *Listing 3*

```
{$B-}
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
  { Tabs are fine until we are at the last cell. Are we at the last tab stop? }
  function AtLastTabStop(G: TDBGrid): Boolean;
  var Loop: Word;
  begin
    Result := True;
    with G do begin
      if SelectedIndex = Pred(FieldCount) then
        Exit;
      for Loop := Succ(SelectedIndex) to Pred(FieldCount) do begin
        Result := Fields[Loop].ReadOnly;
        if not Result then
          Exit;
      end;
    end;
  end;

  procedure NextRecord(var Key: Word; DataSet: TDataSet);
  begin
    Key := 0;
    if not DataSet.EOF then
      DataSet.Next;
  end;

begin
  if (ActiveControl is TDBGrid) and
    (TDBGrid(ActiveControl).DataSource <> nil) then
    case Key of
      vk_Tab: if not (ssShift in Shift) then
        with TDBGrid(ActiveControl).DataSource, DataSet do
          if AtLastTabStop(TDBGrid(ActiveControl)) then begin
            NextRecord(Key, DataSet);
            if not EOF then
              TDBGrid(ActiveControl).SelectedIndex := 0;
          end;
      vk_Down: if not (ssCtrl in Shift) then
        with TDBGrid(ActiveControl).DataSource, DataSet do
          NextRecord(Key, DataSet);
      else Exit;
    end;
end;
```



➤ *Figure 1*

```
procedure TFormReader.Button1Click(Sender: TObject);
var
  InStream, OutStream: TFileStream;
begin
  InStream := TFileStream.Create(FormFile, fmOpenRead);
  try
    OutStream := TFileStream.Create(TextFile, fmCreate);
    try
      { Translate a form file to a text file }
      ObjectResourceToText(InStream, OutStream);
    finally
      OutStream.Free;
    end;
  finally
    InStream.Free;
  end;
  FormDescription.Lines.LoadFromFile(TextFile);
end;
procedure TFormReader.Button2Click(Sender: TObject);
var
  InStream: {$ifdef VER80}THandleStream{$else}
                        TResourceStream{$endif};
  OutStream: TFileStream;
begin
{$ifdef VER80}
  InStream := THandleStream.Create(AccessResource(HInstance,
    FindResource(HInstance, 'TFormReader', rt_RCData)));
{$else}
  InStream := TResourceStream.Create(HInstance, 'TFormReader', rt_RCData);
{$endif}
  try
{$ifdef VER80}
    if InStream.Handle = 0 then
      raise EResNotFound.CreateResFmt(SResNotFound, [ClassName]);
{$endif}
    try
      OutStream := TFileStream.Create(TextFile, fmCreate);
      try
      { Translate an exe-based resource to a text file }
        ObjectBinaryToText(InStream, OutStream);
      finally
        OutStream.Free;
      end;
    finally
      InStream.Free;
    end;
  finally
    FileClose(Handle);
  end;
  FormDescription.Lines.LoadFromFile(TextFile);
end;
```

➤ *Listing 4*

modifying the given canvas, using the `IntersectClipRect` API. If we make a new component (Listing 5), we can add a read/write `ClipRect` property which can be used in the `wm_Paint` handler to achieve our goal.

There's an example project on the disk called PBOX.DPR that sets up `ClipRect` to be the right half of the paint box. It then draws two lines across the paint box, but of course only the right half of the lines show up.

Restricting Property Access

Q I wish to change the access specification of a property from `published` to `private` so that I can set default properties of a derived control and ensure that these defaults do not get changed, how do I do this?

A You don't. You cannot make a given element of an object more restrictive in a descendent – it's effectively taking things away, and that is not allowed. However, you can define a new property of the same name (a dummy property), which is read-only. If it is made `published` then it hides the old one from the Object Inspector. To access the original property in your new component, use the `inherited` keyword. Listing 6 shows an example which allows users of a button descendant to programmatically read the `ModalResult` property value, but not set it.

General Help Page

Q I don't want to provide context-sensitive in my application, however I do want the `F1` key to jump to one help page, no matter where it is pressed in my program. Noticing that the `HelpContext` property of all components is zero, I tried adding a page in my help file with a context number of zero and then associating the help file with my project. This had no effect.

A Unfortunately, the help file is not invoked if the help context of the focused component is

zero. The steps you need to follow are:

1. Decide which items need context-sensitive help, ie which things, when focused on, can have `F1` pressed to get help. All the potential items (components which can take the focus) have a `HelpContext` property.

2. Give each such component a consistent, unique help context (`HelpContext` property), for example 100, and add a help page to your help file with the same context number.

3. Associate your help file with the program using `Options | Project | Application | Help file`.

Hiding Tabs

Q I need to display a varying number of pages of a `TTabbedNotebook` to reflect different types of record in a database. How do I hide specified pages and restore them at will?

A Depending on your preferences, you may want to hide the tab or just disable it, so we'll need two solutions. One to remove the page (`TTabPage` class) from the `TTabbedNotebook` (but being prepared to put it back later), one to disable the tab (which is implemented as a `TButton` derivative in the TABNOTBK unit so we can't see its name).

A `TTabbedNotebook`'s page objects are kept as the objects in its `TStrings Pages` property. There is a help topic that appears to state the contrary of this (`Help | Topic Search, string lists, Adding Objects to a String List`): *"...the pages in a notebook cannot have associated objects"*. What this is saying is that you can't put your own arbitrary objects in the list, because there are already page objects there. One thought for solving the problem would be to say:

```
TabbedNotebook1.Pages.Delete(
  HiddenPageNumber);
...
TabbedNotebook1.Pages.Insert(
  HiddenPageNumber);
```

and this will indeed remove and restore the page, but since all the components on a given page are children of the page itself, they will all be deleted along with the page. To avoid this we might try assigning the page object to a temporary page object and writing `nil` in its place in the `Pages.Objects` property, so the object doesn't get deleted. This also fails because writing to `Pages.Objects` has no effect (if you have the VCL source, see `TStrings.PutObject`'s implementation in CLASSES.PAS – it's empty. The `Pages TStrings` descendent doesn't override it). The best I can manage is to iterate through all the controls on the page and make them children of another object – a surrogate page object or foster parent. Then when the page is deleted there are no children to delete. When restoring the page we need to do the reverse and iterate through the spare page's controls and add them back to the new page that's made when re-inserting.

Some code that does this for one page at a time and also caters for other problems that are encountered along the way is shown in Listing 7. This can also be seen in action by running the MPD.DPR project on this month's disk. This project also demonstrates a similar routine that has had various modifications to work with `TNotebook` and `TTabset` combos.

Now let's turn to disabling a page. A `TTabbedNotebook` is a component that has page components (`TWinControl` descendants) and tabs (`TGraphicControl` descendants in Delphi 1). Since `TGraphicControls` have an `Enabled` property, we can disable the appropriate tab. We also need to prevent the tab's keyboard support selecting the disabled tab and this is done in the

➤ *Listing 5*

```
unit NewPBox;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls;
type
  TNewPaintBox = class(TPaintBox)
  private
    FClipRect: TRect;
  protected
    procedure WMPaint(var Msg: TWMPaint); message wm_Paint;
  public
    property ClipRect: TRect read FClipRect write FClipRect;
  end;
procedure Register;
implementation
procedure TNewPaintBox.WMPaint(var Msg: TWMPaint);
begin
  with FClipRect do
    IntersectClipRect(Msg.DC, Left, Top, Width, Height);
  inherited;
end;
procedure Register;
begin
  RegisterComponents('Samples', [TNewPaintBox]);
end;
end.
```

➤ *Listing 6*

```
type
  TProtectAProperty = class(TButton)
  protected
    function GetModalResult: Integer;
  public
    property ModalResult: Integer read GetModalResult;
  end;
function TProtectAProperty.Get ModalResult: Integer;
begin
  Result := inherited ModalResult;
end;
```

tabbed notebook's `OnChange` handler. Credit for this technique goes to Roy Nelson at Borland. Unfortunately for 16-colour display users, the colour used by the tabs' text when disabled matches that of the tab itself so the text disappears. Code to disable/enable a page, taken from the MPD2.DPR project on the disk, is shown in Listing 8, where `PresentChk` is a checkbox.

In Delphi 2 a `TTabbedNotebook` is implemented by a `TTabControl` which does not support individual page disabling. If we instead use a `TPageControl` things are rather easier. A `TPageControl` has a `Pages` property which is an array of `TTabSheet` controls, each of which has an `Enabled` property.

To cater for Delphi 1 and 2 (which require different controls to be used for page disabling) I have two different forms and form units (MPD2U2A and MPD2U2B) which are conditionally compiled in dependent on platform. Unfortunately Win32 does not provide enough information to stop a disabled tab being selected (there is an `OnChanging` event but you don't get told which tab might get selected).

Disabling a page in a notebook/tabset combo involves much the same concept, although finding the target tab is, of course, much easier. To grey the tab text, you can make the tabset owner-draw and use a different font colour.

```
procedure TForm1.HidePage(Notebook: TTabbedNotebook; PageNum: Word;
  Hide: Boolean);
const Proxy: TTabPage = nil;
      ProxyName: String = '';
var   Page: TTabPage;
begin
  if Hide then begin
    { Can't assign new value to Pages.Objects[i] as it's
      implemented as the base TStrings no-op, meaning the real page will be
      deleted, losing child controls. Also, Assign is not implemented. Make
      surrogate parent window instead. }
    Proxy := TTabPage.Create(Self);
    { Save page name }
    ProxyName := Notebook.Pages[PageNum];
    Page := Notebook.Pages.Objects[PageNum] as TTabPage;
    { Transfer all page children to foster parent }
    while Page.ControlCount > 0 do
      Page.Controls[0].Parent := Proxy;
    { Delete target page }
    Notebook.Pages.Delete(PageNum);
  end else begin
    { Stop bad flicker as page is inserted }
    Notebook.Hide;
    { Can't use InsertObject - doesn't work }
    { Use Insert, which makes the TTabPage object }
    Notebook.Pages.Insert(PageNum, ProxyName);
    { Avoid tab-button refocus problem (caused by page being inserted where
      PageIndex is currently set) by moving PageIndex one left, then moving
      one right (catering for already being at the beginning - the right
      hand side of the addition resolves to 1) }
    Notebook.PageIndex := Notebook.PageIndex + (Byte(PageNum = 0) * 2) - 1;
    Notebook.PageIndex := Notebook.PageIndex + (Byte(PageNum > 0) * 2) - 1;
    Page := Notebook.Pages.Objects[PageNum] as TTabPage;
    { Move children back to natural parent }
    while Proxy.ControlCount > 0 do
      Proxy.Controls[0].Parent := Page;
    { Destroy foster parent }
    Proxy.Free;
    Notebook.Show;
  end;
end;
```

➤ *Above: Listing 7*

➤ *Below: Listing 8*

```
procedure TForm1.PresentChkClick(Sender: TObject);
var
  Loop: Integer;
  Tab : TControl;
begin
  PagesGrp.Enabled := PresentChk.Checked;
{$ifdef Windows}
  { Delphi 1.0x }
  { A TTabbedNotebook has pages (TWinControls) and tabs (TGraphicControls) }
  for Loop := 0 to Pred(Form2.Notebook.ControlCount) do begin
    Tab := Form2.Notebook.Controls[Loop] as TControl;
    if (Tab is TGraphicControl) and
      { Apparently dangerous typecast is actually safe. Caption is in
        every TControl, just protected. It is published in a TButton, and
        so is accessible }
      (TButton(Tab).Caption = PagesGrp.Items[PagesGrp.ItemIndex]) then
      begin
      Tab.Enabled := PresentChk.Checked;
      Break;
    end;
  end;
{$else}
  Form2.NoteBook.Pages[PagesGrp.ItemIndex].Enabled := PresentChk.Checked;
{$endif}
end;
```

Acknowledgements

Thanks to Roy Nelson for the code for tab manipulation. Also to Bob Swart for the outline bug, and for passing along a message from Ron Johnson regarding the Math unit bug.

Errata

It has been pointed out that the implementation of the `Power` routine in Issue 7 left a little to be desired – there were one or two special cases that weren't catered for. Another copy of the PowTest project and the PowerU unit appear on this issue's disk.