Tips & Tricks

Windows Resource Limits – What Limits?

One of the most common problems that newcomers to Delphi experience is the dreaded 'Error creating window' message. The reason for this message is not a limitation of Delphi, but a limitation of the operating system (Windows 3.1x), as I will continue to explain.

This problem is most easily demonstrated when using the `TTabbedNotebook` component, or in MDI applications. Normally the scenario is that the overzealous Delphi developer fills the notebook pages or MDI child windows with too many controls. When running the application and paging through the notebook, or creating new MDI child windows, the error message gets displayed.

So why does it happen? Windows controls, eg list boxes and edit controls, make use of your application's local heap to varying degrees. In particular, edit controls store their text in the local heap, although memo controls don't (thanks to some VCL setup code). The local heap is a limited resource and over-use may result in the failure to allocate the necessary data structures in the local heap, leading to a failure to create a window.

A second factor is that the Windows operating system has a global resource limit: a maximum number of windows that can be created. This seems to be a problem particularly with RAD tools, where many windows can be easily created. According to the Microsoft KnowledgeBase document Q112860 *How to Optimize Memory Management in VB 3.0 for Windows*, Windows 3.1 allows around 600 windows to be created at any one time. Effectively, this limit is around 300-400 per application. This might seem more than enough, but bear in mind that every icon and its title on the desktop, the desktop itself, every other open application and its associated controls, as well as every windowed control in your application, uses window handles.

Here's an example. The `TTabbedNoteBook` component only creates window handles as each page becomes active. However, once the page is displayed, the windows of previously active pages are not destroyed. Should your application contain a `TabbedNotebook` with 5-10 pages, each page containing up to 40 edit controls (as is sometimes the case in address book applications), around 300 window handles for this application alone could exist after paging has occurred. At this point an exception will occur: 'Error creating Window' (`EOutofResources`). This exception will occur in a similar way irrespective of which limit is reached first.

So how do we solve the problem? Probably the easiest solution is to recycle controls (first proposed by Mike Toms of the Borland/Digital support desk, UK) by re-using existing controls. This involves changing the parent property of the control as you turn pages. In `TabbedNotebooks` this is very easy. However, the developer needs to keep track of the re-sizing and re-positioning of the controls. Listing 1 shows some example code which demonstrates the principle.

The second solution is slightly more robust and involves surprisingly little code when implemented for `TabbedNotebooks` – see Listing 2. When looking at this code you may notice the strange cast of `TWinControl` to `THintWindow`. This is because it is the only component which gives public access to the protected method `DestroyHandle` (`ReleaseHandle` just calls `DestroyHandle`). The `DestroyHandle` releases the window handle of the control, but it still keeps all the other properties of the control safely in the OOP component wrapper. C++/C/VB developers might say, "Yes, *but* what about the contents of edit controls and list boxes etc?". Well the VCL creators were ahead of us – it is all preserved regardless of the updates between creation and destruction. This code could also be used in a `TNotebook` with a `TTabset` with little modification: see Listing 3.

In MDI applications things are slightly more difficult, but the principle stays the same. The idea is to destroy the window controls but still continue to display an image of the window's client area when the window is not active. The code in Listing 4 demonstrates the principle but could be improved upon. Note the use of the `GetFormImage`; again the creators of the VCL provided an easy way of solving the problem by getting a bitmap of the form's image. The MDI child's client area is filled with a `TPanel` which contains all the

➤ *Listing 1*

```
procedure TForm1.TabbedNotebook1Change(Sender:
  TObject; NewTab: Integer; var AllowChange: Boolean);
var PresentPage, NewPage : TWinControl;
begin
  if Sender is  TTabbedNotebook then begin
    with TTabbedNotebook(Sender) do begin
      PresentPage :=
        TWinControl(Pages.Objects[PageIndex]);
      LockWindowUpdate(Handle);
      NewPage := TWinControl(Pages.Objects[NewTab]);
      while PresentPage.ControlCount > 0 do
        PresentPage.Controls[0].Parent := NewPage;
      LockWindowUpdate(0);
    end;
  end;
end;
```

➤ *Listing 2*

```
procedure TForm1.TabbedNotebook1Change(Sender:
  TObject; NewTab: Integer; var AllowChange: Boolean);
var aPage: TWinControl;
begin
  if Sender is TTabbedNotebook then
    with TTabbedNotebook(Sender) do begin
      aPage := TWinControl(Pages.Objects[PageIndex]);
      LockWindowUpdate(Handle);
      THintWindow(aPage).ReleaseHandle;
      TWinControl(Pages.Objects[NewTab]).HandleNeeded;
      LockWindowUpdate(0);
    end;
end;
```

controls. This is a technique used in OWL 2.5 and MFC, which makes it easier to destroy the child controls by destroying their parent, the `Panel`.

I hope you will find this useful in your development.

Contributed by Roy Nelson,
Borland UK European Technical Team

Big Arrays

64k limitation on arrays too small? Can't wait for Delphi 32? Try the array object on this issue's free disk (file ARRAYU.PAS), which is limited only by the size of Windows' accessible memory. Apart from a call to the constructor and destructor and a slightly different declaration, this object can be treated just like any other array, only much bigger.

Let's take an example: you want an array of 100,000 `Doubles`. Trying `array[1..100000] of Double` won't work. Instead, do the following: take the ARRAYU.PAS file on the disk and change the first type definition from:

```
TItemType = Longint;
```

to:

```
TItemType = Double;
```

This type defines what the array object will contain. Where you would declare the array itself as:

```
MyArray: array[1..100000] of Double;
```

instead declare:

```
MyArray: TArray;
```

Before you use it for the first time, call:

```
MyArray := TArray.Create(1, 100000);
```

where the two parameters are the upper and lower limits, and after you are done with it call `MyArray.Free`. Apart from that, the object can be accessed like any normal array:

```
if MyArray[1] < 5 then MyArray[2] := 20.2;
```

With Delphi 32 not that long off, some conditional compilation can be employed to cater for this object and proper large arrays when available. There is an example project on the disk called HUGEARAY.DPR which shows the array in use. It allows you to allocate an array of as many `Longints` as you like, populate them with their element number and then list as many of them as you please. Here's the code snippet that sets the array up:

```
Arr := TArray.Create(LowerBound.Value,
                     UpperBound.Value);
for Loop := Arr.Low to Arr.High do
  Arr[Loop] := Loop;
```

Some (currently read-only) properties of the object that are useful include `Size`, `Low` and `High`, which return the number of elements, lowest element number and highest element number respectively.

It wouldn't take forever to modify the `Size` property so that you could dynamically extend and contract the array at runtime with a combination of reallocating the memory occupied by the array (`GlobalRealloc`) and copying the old array into it (`hmemcpy`).

Contributed by Brian Long (CompuServe 76004,3437)

➤ *Listing 3*

```
procedure TForm1.TabSet1Change(Sender: TObject;
  NewTab: Integer; var AllowChange: Boolean);
var aPage: TWinControl;
begin
  LockWindowUpdate(Handle);
  if Sender is TTabset then
    with TTabSet(Sender) do begin
      if TabIndex > -1 then begin
        aPage := TWinControl(Tabs.Objects[TabIndex]);
        THintWindow(aPage).ReleaseHandle;
        TWinControl(Tabs.Objects[NewTab]).HandleNeeded;
      end;
    end;
  LockWindowUpdate(0);
end;
```

➤ *Listing 4*

```
procedure TMDIChild.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
end;
procedure TMDIChild.FormDeactivate(Sender: TObject);
begin
  aBitmap := GetFormImage;
  LockWindowUpdate(Handle);
  THintWindow(Panel1).ReleaseHandle;
  LockWindowUpdate(Handle);
end;
procedure TMDIChild.FormPaint(Sender: TObject);
begin
  if Assigned(aBitmap) then Canvas.Draw(0,0,aBitmap);
end;
procedure TMDIChild.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  FormResize(nil);
end;
procedure TMDIChild.FormResize(Sender: TObject);
var ParentForm: TForm;
begin
   if Assigned(aBitmap) then aBitmap.Free;
   aBitmap := nil;
   LockWindowUpdate(Handle);
   Panel1.HandleNeeded;
   ParentForm := GetParentForm(Panel1);
   ParentForm.ActiveControl := Panel1;
   with Panel1 do begin
     BringToFront;
     Visible := false;
     Visible := True;
   end;
   LockWindowUpdate(0);
end;
```

Hiding The Titlebar

A lot of people are asking if it is posible to hide and then show the titlebar from a window in a Delphi application *while the application is running*. The two procedures in Listing 5 show how to do this.

Contributed by Claus Ziegler from Denmark (email ziegler@winboss.dk)

Creating Paradox Auto-Increment Fields

Listing 6 shows a method for creating a Paradox table containing an ***auto-increment*** field within a Delphi application, using SQL with a `TQuery`. Does anyone know how to do this for a `TTAble`, just using BDE calls?

Stephen Thompson (Steve@DDEN.DEMON.CO.UK)

➤ *Listing 6*

```
with Query1 do begin
  DatabaseName := 'DBDemos';
  with SQL do begin
    Clear;
    Add('CREATE TABLE "PDoxTbl.db" (ID AUTOINC,');
    Add('Name CHAR(255),');
    Add('PRIMARY KEY(ID))');
    ExecSQL;
    Clear;
    Add('CREATE INDEX ByName ON "PDoxTbl.db" (Name)');
    ExecSQL
  end;
end;
```

➤ *Listing 5*

```
Procedure TYourFormName.HideTitlebar;
Var Save : LongInt;
Begin
  If BorderStyle = bsNone then Exit;
  Save := GetWindowLong(Handle,gwl_Style);
  If (Save and ws_Caption) = ws_Caption then Begin
    Case BorderStyle of
      bsSingle,
      bsSizeable :
        SetWindowLong(Handle,gwl_Style,Save and
        (Not(ws_Caption)) or ws_border);
      bsDialog :
        SetWindowLong(Handle,gwl_Style,Save and
        (Not(ws_Caption)) or ds_modalframe or
        ws_dlgframe);
    End;
    Height := Height - getSystemMetrics(sm_cyCaption);
    Refresh;
  End;
end;

Procedure TYourFormName.ShowTitlebar;
Var Save : LongInt;
begin
  If BorderStyle = bsNone then Exit;
  Save := GetWindowLong(Handle,gwl_Style);
  If (Save and ws_Caption) <> ws_Caption then Begin
    Case BorderStyle of
      bsSingle,
      bsSizeable :
        SetWindowLong(Handle,gwl_Style,Save or
        ws_Caption or ws_border);
      bsDialog :
        SetWindowLong(Handle,gwl_Style,Save or
        ws_Caption or ds_modalframe or ws_dlgframe);
    End;
    Height := Height + getSystemMetrics(sm_cyCaption);
    Refresh;
  End;
end;
```

Update: Custom Clipboard Formats

by Xavier Pacheco

Thanks to Hallvard Vassbotn for pointing out an oversight in Issue 3's *Custom Clipboard Formats* article. In the article, the `TBirthDay.PasteFromClipBoard` method declared a variable `Size`, which was used in the statement:

```
if SizeOf(FPersonRec)  GlobalSize(Data) then
  Size := GlobalSize(Data);
```

however, the following statement neglected to make use of the `Size` variable:

```
Move(DataPtr^, FPersonRec, SizeOf(FPersonRec));
```

The correct code should read:

```
if SizeOf(FPersonRec)  GlobalSize(Data) then
  Size := GlobalSize(Data)
else
  Size := SizeOf(FPersonRec);
Move(DataPtr^, FPersonRec, Size);
```

Also, the outer `try..except` statement in that same method was unnecessary. The memory referenced by the `Data` variable does not have to be freed in this method since Windows will manage this memory block already. Listing 1 below shows the corrected version of the `PasteFromClipBoard` method (also on the disk).

➤ *Listing 1*

```
procedure TBirthday.PasteFromClipBoard;
var
  Data: THandle;
  DataPtr: Pointer;
  Size: Integer;
begin
  { Get the data on the clipboard }
  Data := ClipBoard.GetAsHandle(CF_BIRTHDAY);
  { Exit is unsuccessful }
  if Data = 0 then Exit;
  { Lock the Global memory object }
  DataPtr := GlobalLock(Data);
  try
    if SizeOf(FPersonRec) > GlobalSize(Data) then
      Size := GlobalSize(Data)
    else
      Size := SizeOf(FPersonRec);
    { Copy contents of DataPtr to Buffer }
    Move(DataPtr^, FPersonRec, Size);
  finally
    GlobalUnlock(Data); {Unlock global memory object}
  end;
end;
```