The Unofficial Newsletter of Delphi Users - Issue #19 - January 1997

Phew!!! I'm exhausted! This is probably one of the bigger issues of UNDU that I have put out. It's chock-full of all sorts of nifty Delphi stuff. There is expanded sections for both Tips & Tricks as well as The Component Cookbook, so make sure you check out everything there. You won't be sorry!

Since this issue is so much larger and I had so much good input from all of you, I figured I would award a total of 3 prizes this month to 3 randomly chosen contributors. The first prize is a copy of **MagiKit** (reviewed in this issue) which goes to Paul Harding for his tips on Paradox Lock Violations and Checking of Someone Else is Running. The second prize is a copy of **Delphi-In-Depth** from Osborne/McGraw-Hill which goes to Grahame Marsh for his articles on Extending the TPageControl and A Big Bitmap Viewer. The third prize is a copy of the **Delphi Informant Works CD** which goes to James Sager for his technique on Aligning Text in a Grid.

Thanks again for everyone's great articles and tips!

Index of Past Issues

Below is a complete index of all principle articles in past issues of the Unofficial Newsletter of Delphi Users. Provided that you have the prior issues in the same directory as this issue, you can click on any of these hotspots to go directly to that article. To return to the index, you can click on the **Back** button, or you can use the **History** list. Once you jump to one of these issues, you can navigate through the issue as you would normally, but you will need to go to the **History** list to get back to this index. There will be an updated index included in all future issues of UNDU.

Return to Front Page

Where To Find UNDU

When each issue of UNDU is complete, I put them in the following locations:

1. UNDUs official web site at `http://www.informant.com/undu`. This site houses all the issues in both HTML and Windows HLP format.

2. *Borlands* Delphi forum on CompuServe (**GO DELPHI**) in the "Delphi IDE" file section. This forum will only hold the issues in Windows HLP format.

Tips & Tricks

Have you ever wanted to tap into the Win95 shell to utilize those fancy copy dialogs, or to throw things in the Recycle Bin so you can change your mind later and get them back? Now you can by Using the SHFileOperation to Copy/Move/Delete/Rename Files.
In past issues, I have presented techniques for doing splash screens. Michael Barnes takes the technique a bit further by allowing you to create splash screens that are not rectangular in shape. Learn how this is done in How to create a Polygon Splash screen.
Paul Harding brings a few more tips in this issue with an update to his code to determine if a certain application is running. Check it out in Is Someone else running?. Also, check out his tip on Lock Violation errors in Paradox.

A while back I had several people ask me how to print raw textual data to a printer without fiddling with the Windows printing mechanism. Learn more about it in the tip on Printing Directly to a printer.

Some other tips & tricks include: Refreshing MDI Menus, Extending the Background Bitmap Technique, Paradox File Size Limits, Safer use of Enumerated Types, and Simplifying Code management with Include.

Finally, a number of readers have requested help on the use of the TreeView control. I put together a brief tutorial on the subject in the article on A Look at the TreeView Control.

Return to Front Page

The Component Cookbook

Welcome sir to ***Chez Delphi****. Will that be a table for two sir? Excellent… Please walk this way… Here we are, sir, a nice, romantic table for two overlooking the water. We have four specials this evening that are sure to satisfy your* ***palette****. The first is delicately braised Text, Aligned in a Grid with baby carrots and wild rice. Our second special is TPageControl Flambé in a delicate brandy sauce with hearts of Artichokes. Next is a generous portion of Bits of Maps (we like to call them Big Bitmaps). It is accompanied by pearl onions and savory rice. And last, but certainly not least, is our special of the evening… Masks ala Transparency with steamed vegetables and herb polenta. Magnifique!*
At Chez Delphi we always try to have the finest menus!

Return to Front Page

UNDU Subscriber List

The subscriber list is a method by which I can notify the readers when a new issue is out. I will maintain a list of readers email addresses and when a new issue is released, I will fire off a batch mailing to notify everyone that it is available and where they can find it.

This is what you need to do to get on the subscriber list… Simply send me an email to my email address (**RobertV@compuserve.com**) and put the words **SUBSCRIBE UNDU** anywhere in the subject line or in the main body of the message. If you no longer wish to be notified of future issues (i.e. you are on the list and want off…) just send an email with the words **UNSUBSCRIBE UNDU**.

Return to Front Page

How to Make a Polygonal Splash Screen

by Michael Barnes - MBarnes195@worldnet.att.net

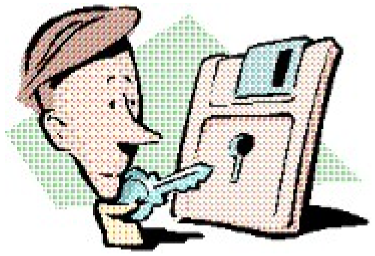
Back in issue #16, UNDU presented a way of creating a Splash Screen and even a Splash Screen with a Time Delay. Splash screens don't need to be rectangular however. The tip below shows how you can make a splash screen that consists of polygonal regions.
All you need to do is create an array of TPoints and use the **SetWindowRgn** API call to change the shape of your splash screen form. This particular sample also shows how you can replicate a small bitmap across the form to fill in your polygonal region. For more on extending the concept of repeating bitmaps across a form, see the tip on Extending Background Bitmaps.

```
unit SplashUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls;

type
  TSplash = class(TForm)
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormPaint(Sender: TObject);
  private
    BackBmp: TBitmap;
  public
    { Public declarations }
  end;

var
  Splash: TSplash;

implementation

{$R *.DFM}

procedure TSplash.Timer1Timer(Sender: TObject);
begin
  Timer1.Enabled := False;
end;

procedure TSplash.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  CanClose := Not Timer1.Enabled;
end;

procedure TSplash.FormCreate(Sender: TObject);
var
   R: HRgn;
   P: array [0..11] of TPoint;
```

```
begin
   P[0] := Point(1, 1);
   P[1] := Point(90, 1);
   P[2] := Point(90, 90);
   P[3] := Point(180, 90);
   P[4] := Point(310, 210);
   P[5] := Point(310, 90);
   P[6] := Point(400, 90);
   P[7] := Point(400, 300);
   P[8] := Point(310, 300);
   P[9] := Point(180, 180);
   P[10] := Point(180, 300);
   P[11] := Point(1, 300);
   R := CreatePolygonRgn(P, 12, ALTERNATE);
   SetWindowRgn(Handle, R, True);
   BackBmp := TBitmap.Create;
   BackBmp.LoadFromResourceName(hInstance, 'MYBITMAP');
end;

procedure TSplash.FormDestroy(Sender: TObject);
begin
  BackBmp.Free;
end;

procedure TSplash.FormPaint(Sender: TObject);
var
  r, c: Integer;
  FormDC: HDC;
  FormHandle: HWND;
begin
  FormHandle := Splash.Handle;
  FormDC := GetDC(FormHandle);
  for r := 0 to Splash.Height div BackBmp.Height do
    for c := 0 to Splash.Width div BackBmp.Width do
      BitBlt(FormDC, c*BackBmp.Width, r*BackBmp.Height,
        BackBmp.Width, BackBmp.Height, BackBmp.Canvas.Handle, 0, 0, SRCCOPY);
end;

end.
```

Return to Tips & Tricks

Return to Front Page



Lock Violations in Paradox

by Paul Harding - CIS: 100046,2604

If you run your application across a network of mixed Windows 3.x and Windows 95 machines with more than one user accessing a Pardox table, you may find an inexplicable "**Lock violation**" error.
This actually comes about when a 16 bit application (compiled in Delphi 1.0) is being run where the data table is stored on a Windows 95 machine. It seems that you can't get away with this, so the solution is to move your data tables on to the 16 bit machine (Windows 3.x) and then you can access that data simultaneously from both a Windows 3.x and a Windows 95 machine. By the way, you don't get this problem if you compile your app for the Win 95 machine under Delphi 2 so as to generate 32 bit code.

Return to Tips & Tricks

Return to Front Page

Printing Directly to a Printer

by Robert Vivrette - RobertV@compuserve.com

Several readers have in the past inquired about how to print directly to a printer without going through the Windows printing engine. The key use for such a technique is when the user wants to send raw data (from a text file for example) directly to the printer without having to mess with fonts or formatting. Also, sometimes you may have directed some print output to a file rather than the printer. This technique allows you to send such a file directly to the printer.

The technique is really quite simple. All you need to do is use AssignFile to open a file with the filename specified as 'LPT1' (or any other printer port). The operating system recognizes this as a special file type and directs all the output to that printer port. The Write & Writeln procedures are used to actually send the data. If you need to eject the page when you are done, just send a ASCII code of #12. Most printers recognize this as a formfeed code.

```
var
  F : TextFile;
begin
  AssignFile(F,'LPT1');
  Rewrite(F);
  Writeln(F,'Hello');
  Writeln(F,'There!');
  Writeln(F,#12);
  CloseFile(F);
end;
```

Return to Tips & Tricks

Return to Front Page

Questions From UNDU Readers

I often get a wide variety of emailed questions from readers of UNDU. Some of them have been quite interesting and the solutions are equally interesting. Anyway, I figured "Why not let everyone help on the solution?"
Each month I will present a few questions here that readers have submitted to me and open them up to all the readers of UNDU. If you know the answer to a question, feel free to send it in to **RobertV@compuserve.com**. I will chose the best solution to the question and post it in the following issue. This way, everyone gets to see the answer!
The solutions can be anything including even shareware components that might solve a particular problem.

This months questions:

Steven Lucey asks *"How do you make forms so that they will display correctly no matter the resolution or font size (large or small) at runtime?"*

Steven Gill asks *"I am trying to work out how to add bitmaps to StringGrids. I want to use the first column as a status column with a graphic indicating the status.   What's a simple way to do this?"*

Return to Front Page

Refreshing Delphi MDI Menus

by Gerard Sillekens - g.sillekens@tip.nl

When programming a MDI application with Delphi you can assign the WindowMenu property of the MDIForm. The menu you assign to it will include a list with currently open MDI child windows (or forms). This works well, until you change menu properties such as the visible- or checked property. After you have done a change of propertes the list with open MDI child-windows disappears. In a few words: When you change menu-items dynamically the list of child windows disappears.

The solution:

When examining the problem I mentioned that there is a different solution for Delphi 1 and Delphi 2. In both versions of Delphi the solution is sending a windows message to the MDI main-form. In the code fragments below I send the messages by a method of the MDI main form (named TMDIMainForm). I have used a conditional define to handle both versions of Delphi.

```
procedure TMDIMainForm.RefreshMenus;
begin
  {$IFDEF WIN32}
  SendMessage(ClientHandle, WM_MDIREFRESHMENU, 1,0);
  {$ELSE}
  SendMessage(ClientHandle, WM_MDISETMENU, 1, 0);
  {$ENDIF}
end;
```

You should then call RefreshMenus each time after changes have been made to a menu item (or its properties).

About Gerard

Gerard Sillekens works for the Dutch Ministry Of Defense as a system developer. He uses both Delphi and Borland Pascal 7 with Objects as programming languages.

Return to Tips & Tricks

Return to Front Page

Speed Daemon by Integrationware Inc.

Speed Daemon (SD) is a source code profiler for Delphi. With it you can analyze your Delphi applications by tracking how much time individual routines consume out of the total run of the program. In addition, SD also reports on the number of times routines were called. SD is compatible with both Delphi 1.0 and 2.0. It is capable of profiling single-threaded and multi-threaded applications and can also be used on DLL's and OLE Automation Servers.

Using SD is quite simple. The package provides a Wizard application that guides you through the steps necessary for profiling the application. First, you chose the Delphi Project file that you want to profile and indicate if it is a 16-bit or 32-bit application. Then you specify an output file and the order that you want the results sorted by. Then SD takes over and sets integrates itself into your application. Really, what it does is makes a duplicate copy of the application and all of its source code files (off in a private directory) and inserts the profiling hooks as required. This modified code is then compiled and executed allowing you to take it through its paces. The modified program executes at virtually full speed (it does seem a little slower) and every function that you do in the program is recorded faithfully by the profiling hooks SD had inserted. When you shut down the application, a text file is generated with the results of the profiling session, which looks something like this:

```
Speed Daemon for Delphi - Profile results for BITEST.EXE

General statistics:
====================
Program start time:       1:09:05 PM on 12/31/96
Total execution time:     13.44 seconds
Total number of threads:  1
                                    Function Time      Func + Child
Function Name                Hits    Time / %age       Time / %age  Avg Time
========================= ======= ================ ================ ========
TBigImage.EncodeRow2D        6500    2.33 / 36.55%    5.21 / 81.53%    0.00
TBigImage.FillCodingLine     6501    1.42 / 22.31%    1.42 / 22.31%    0.00
TBigImage.LoadTiff              1    1.10 / 17.25%    1.10 / 17.25%    1.10
TBigImage.AddToBuffer      146014    0.79 / 12.37%    0.79 / 12.37%    0.00
TBigImage.RecordModeH       84212    0.66 / 10.30%    0.80 / 12.55%    0.00
TBigImage.SaveTiff              1    0.07 /  1.11%    5.29 / 82.75%    5.29
```

```
    TBigImage.WriteTiffHeader        1    0.00 /  0.00%    0.01 /  0.11%    0.01
    TBigImage.EncodeRow1D            1    0.00 /  0.00%    0.00 /  0.00%    0.00
```

As you can see, the output shows the particular function names along with the number of times each function was called. After that is the total time spent in the function and the percentage of time the is over the total run time. There also are figures which identify the time a function took along with any child functions it may have called. The average time for each function call is also specified.

The Speed Daemon package is really two parts. First, there is the part that does the actual profiling. The profiling hooks inserted into the code tie into this "profiling engine" in order to generate the appropriate statistics. The second half of the package is the wizard that sets things up for you.

The wizard provided provides very little control over what portions of your code are actually setup for profiling. From what I could see, it takes all of the units mentioned in your DPR file and adds profiling hooks into each function in those units. As a result, however, if a unit is not mentioned in the DPR, no profiling is done on that code. Further, if the code you want to profile is in a component, the wizard won't catch that either. For me, the wizard was next to useless.

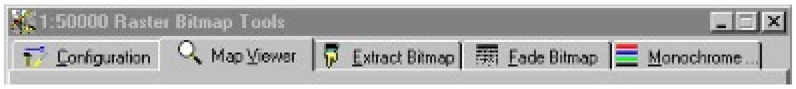
However, looking over the changes the wizard made to the copy of my project, I realized that I could insert the necessary hooks to the profiling engine myself and have complete control over what areas of the code are profiled. I did this in order to profile my component whose output is indicated above. I even found that it really wasn't necessary to profile an entire function or procedure but could actually just do a portion of it, perhaps just some time-sensitive loop for example. Also, the Wizard's concept of "copy the project then modify the copy for profiling" didn't really fit well with the way I do things. As a result, I left all the profiling hooks in the code, but set them off with conditional defines. That way, I could turn on or off profiling anytime by inserting {$DEFINE PROFILING} at the top of the unit. This made the profiling and testing process much more interactive.

All-in-all I am very impressed with Speed Daemon. To my knowledge, it is the only profiling package for Delphi applications. Although the Wizard application didn't help me, I found that the profiling engine itself is quite flexible and stable. Every time I tested the application, the numbers barely twitched. That made it easier to see speed improvements when I modified the code.

In conclusion, if you don't expect too much out of Speed Daemon's wizard, and don't mind a little manual insertion of a few lines of code, you will be very happy with the capabilities Speed Daemon adds to your Delphi project.

For additional information on Speed Daemon, you can contact **Jim Flury** at **Integrationware Inc**. His email address is `JFlury@integrationware.com`. You can also visit their web-site at `http://www.integrationware.com`.

Return to Front Page



Extending the TPageControl

by Grahame Marsh - Grahame.S.Marsh@corp.courtaulds.co.uk

The TPageControl component in the VCL has not fully implemented the capabilities of the Win95 control. A couple of useful features associated with the tabs on the control are omitted. The component provided here is an extension to the standard VCL implementation by adding the ability to:

- Add glyphs on to each tab of the control
- Use all of the tab styles (i.e. buttons)
- Use the owner draw style

The code to do this is straightforward: two new published properties are added, the Styles property which can take on all of the available page control styles and a Glyphs property which is of type TImageList for the glyphs. Two new events are also added, the OnDrawItem event is just like the OnDrawItem event of a TListBox component and allows the user to draw on the Canvas property, and the OnGlyphMap property allows the order of the glyphs in the TImageList to be draw differently for the order of the tabs (by default tab 0 gets glyph 0, tab 1 gets glyph 1 etc).

In the source code there are a couple of twists I would point out. First, I wanted the multiline style to be available not as a boolean property as it is in the TPageControl, but as a setting in the Styles property. It has been said in many places, that once you have declared a property published so that it appears in the object inspector, that you cannot unpublish it. Well it seems that if you redeclare the property in a read-only fashion:

```
interface

  property MultiLine: boolean read GetMultiLine;

implementation

  function TExPageControl.GetMultiLine: boolean;
  begin
    Result := inherited MultiLine;
  end;
```

then the property disappears from the property editor. You can still access the original property using the **inherited** keyword for both read and write.

The OnDrawItem event is just like the OnDrawItem of a TListBox (that's where I got the basis of the code from). You are provided with a Canvas to draw on and the state of the control (odSelected, odDisabled etc).

Finally, I have provided an event that enables you to alter the way that the TImageList glyphs are assigned to each tab. This event is declared as:

```
type
  TGlyphMapEvent = procedure(Control: TWinControl;
                             PageIndex: integer;
                             var GlyphIndex: integer) of object;
```

when called, PageIndex and GlyphIndex will always contain the same number. The event will be called once for each tab in the page control. You can therefore return a different GlyphImage number than the default. Calling UpdateGlyphs forces the links to be re-established.

Source Code for this project

Return to Component Cookbook

Return to Front Page

A Big Bitmap Viewer

by Grahame Marsh - Grahame.S.Marsh@corp.courtaulds.co.uk

In a recent article in UNDU I mentioned that I was working with some large bitmap images of maps of the UK. This images are 4000 by 4000 pixels by 256 colors and weigh in at 16,001,078 bytes apiece!. The first viewer I wrote for these maps was to use a TImage, loading the bitmap with something like:

```
var
  B : TBitmap;
...
  B := TBitmap.Create;
  try
    B.LoadFromFile (AFilename);
    Image.Picture.Graphic := B;
  finally
    B.Free
  end;
```

This worked great, as always with Delphi, I had a viewer up and running in a few moments. But even on a machine with 32 MB RAM, it took several 10's of seconds to load the first image, and a minute to load the second. On a portable with only 8 MB RAM it took 10 minutes to load one! All the while there was a great deal of disc thrashing as the operating system obeyed my request for memory and used its swap file.

Well, I could have started to try to optimize the computer configuration, or even buy more RAM. However, I looked for a Delphi solution first. The two tricks that this component illustrates are, most importantly, using Win 95 memory mapped files and, of least importance, using the API StretchDIBits function to draw a bitmap without creating a TBitmap.

I will only dwell on the first of these two, as the purpose of this article is to be illustrative of the use of memory mapped files rather than provide a universally wanted component. There are three steps to map a file into the processes address space:

First, you open the file with the permissions, you need. In this case I only need read-only access:

```
var
  FData                 : Pointer;
  FileHandle,MapHandle : THandle;
...
  FileHandle := FileOpen(FFilename, fmOpenRead + fmShareDenyNone);
  if FileHandle = INVALID_HANDLE_VALUE then
    raise Exception.Create('Failed to open ' + FFilename);
```

Secondly, you create a map (note that the access permissions must be compatible with the way you opened the file.) Also, note that having successfully mapped the file, the file handle is no longer needed and so it can be closed:

```
try
  MapHandle := CreateFileMapping(FileHandle,nil,PAGE_READONLY,0,0,nil);
  if MapHandle = 0 then
    raise Exception.Create('Failed to map file')
```

```
finally
  CloseHandle(FileHandle)
end;
```

Third, and finally, you need to obtain a view of the map.   This gives you a pointer to the start of the file in memory.   Again, having obtained the view pointer using a compatible access mode, the map handle is no longer needed and can be closed):

```
try
  FData := MapViewOfFile(MapHandle,FILE_MAP_READ,0,0,0);
  if FData = nil then
    raise Exception.Create('Failed to view map file')
finally
  CloseHandle(MapHandle)
end;
```

When finished you close the view of the file with:

```
UnmapViewOfFile(FData);
```

This reduced image load times to seconds, with no disc drive thrashing and no great use of system memory!   Memory mapped files are here to stay, use 'em and love 'em.

Source for TPageControl

```
// Extended Page Control - adds some extra styles
// Grahame Marsh 1996

unit ExPage;

interface

uses
  Windows, Messages, Classes, Controls, ComCtrls, StdCtrls, CommCtrl,
  SysUtils, DsgnIntf;

// tab styles - search win32 api help for TCS_ for info on each style
type
  TTabStyle  = (tabButton, tabMultiline, tabRightJustify,
               tabIconLeft, tabLabelLeft, tabOwnerDraw);
  TTabStyles = set of TTabStyle;

// event to allow different mapping of glyphs from the imagelist component
type
  TGlyphMapEvent = procedure(Control: TWinControl; PageIndex : integer; var GlyphIndex
: integer) of object;

type
  TExPageControl = class (TPageControl)
  private
    FCanvas : TControlCanvas;      // canvas for drawing on with tabOwnerDraw
    FGlyphs : TImageList;          // link to a TImageList component
    FTabStyles : TTabStyles;       // tab style
    FOnDrawItem : TDrawItemEvent;  // Owner draw event
    FOnGlyphMap : TGlyphMapEvent;  // glyph mapping event
    procedure SetGlyphs (Value : TImageList);
    procedure SetTabStyles (Value : TTabStyles);
    function GetMultiline : boolean;
    procedure CNDrawItem (var Msg : TWMDrawItem); message CN_DRAWITEM;
    procedure GlyphsChanged (Sender : TObject);
  protected
    procedure CreateParams(var Params: TCreateParams); override;
    procedure UpdateGlyphs; virtual;
    procedure CreateWnd; override;
    procedure DrawItem(Index: Integer; Rect: TRect; State: TOwnerDrawState); virtual;
    // for owner draw
    property Canvas : TControlCanvas read FCanvas write FCanvas;
    // republish Multiline as read only
    property MultiLine : boolean read GetMultiline;
  published
    constructor Create (AOwner : TComponent); override;
    destructor Destroy; override;
    // link to TImageList
    property Glyphs : TImageList Read FGlyphs write SetGlyphs;
    // tab styles property
    property Styles: TTabStyles read FTabStyles write SetTabStyles default [];
    // owner draw event
    property OnDrawItem : TDrawItemEvent read FOnDrawItem write FOnDrawItem;
    // glyph map event
    property OnGlyphMap : TGlyphMapEvent read FOnGlyphMap write FOnGlyphMap;
  end;
```

```
// redeclare TTabSheet so it can have a component editor declared here

type
  TExTabSheet = class (TTabSheet)
                end;

procedure Register;

implementation

const
  DefaultTabWidth = 100;

// constructor must create a TControlCanvas for the owner draw style

constructor TExPageControl.Create (AOwner : TComponent);
begin
  inherited Create (AOwner);
  FCanvas := TControlCanvas.Create;
end;

// remove link with glyphs and free the canvas

destructor TExPageControl.Destroy;
begin
  FGlyphs.OnChange := nil;
  FCanvas.Free;
  inherited Destroy
end;

// CreateParams called to set the additional style bits

procedure TExPageControl.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams (Params);

  with Params do
  begin
    if tabButton in FTabStyles then
      Style:= Style or TCS_BUTTONS;

    if tabRightJustify in FTabStyles then
      Style := Style or TCS_RIGHTJUSTIFY;

    if tabIconLeft in FTabStyles then
      Style := Style or TCS_FORCEICONLEFT;

    if tabLabelLeft in FTabStyles then
      Style := Style or TCS_FORCELABELLEFT;

    if tabOwnerDraw in FTabStyles then
      Style := Style or TCS_OWNERDRAWFIXED;
  end;
end;

// CreateWnd also must set links to the glyphs

procedure TExPageControl.CreateWnd;
begin
  inherited CreateWnd;
  if Assigned (FGlyphs) then SetGlyphs (FGlyphs);
```

```
end;

// if the glyphs should change then update the tabs

procedure TExPageControl.GlyphsChanged (Sender : TObject);
begin
  UpdateGlyphs;
end;

// multiline property redefined as readonly, this makes it
// disappear from the object inspector

function TExPageControl.GetMultiline : boolean;
begin
  Result := inherited Multiline
end;

// link the tabs to the glyph list
// nil parameter removes link

procedure TExPageControl.SetGlyphs (Value : TImageList);
begin
  FGlyphs := Value;
  if Assigned(FGlyphs) then
    begin
      SendMessage (Handle, TCM_SETIMAGELIST, 0, FGlyphs.Handle);
      FGlyphs.OnChange := GlyphsChanged
    end
  else
    SendMessage (Handle, TCM_SETIMAGELIST, 0, 0);
  UpdateGlyphs;
end;

// determine properties whenever the tab styles are changed

procedure TExPageControl.SetTabStyles (Value : TTabStyles);
begin
  if FTabStyles <> Value then
  begin
    FTabStyles := Value;

    if tabRightJustify in FTabStyles then
      FTabStyles := FTabStyles + [tabMultiLine];

    inherited Multiline := tabMultiline in FTabStyles;

    if tabLabelLeft in FTabStyles then
      FTabStyles := FTabStyles + [tabIconLeft];

    if (tabIconLeft in FTabStyles) and (TabWidth = 0) then
      TabWidth := DefaultTabWidth;

    ReCreateWnd;
  end
end;

// update the glyphs linked to the tab

procedure TExPageControl.UpdateGlyphs;
var
  TCItem : TTCItem;
  Control,
  Loop : integer;
begin
  if FGlyphs <> nil then
  begin
    for Loop := 0 to pred(PageCount) do
    begin
      TCItem.Mask := TCIF_IMAGE;
      TCItem.iImage := Loop;
      Control := Loop;
      // OnGlyphMap allows the user to reselect the glyph linked to a
      // particular tab
      if Assigned (FOnGlyphMap) then
        FOnGlyphMap (Self, Control, TCItem.iImage);

      if SendMessage (Handle, TCM_SETITEM, Control, longint(@TCItem)) = 0 then
        raise EListError.Create ('TExPageControl error in setting tab glyph')
    end
  end
end;

// called when Owner Draw style is selected:
// retrieve the component style, set up the canvas and
// call the DrawItem method

procedure TExPageControl.CNDrawItem (var Msg : TWMDrawItem);
var
  State: TOwnerDrawState;
begin
  with Msg.DrawItemStruct^ do
  begin
    State := TOwnerDrawState (WordRec (LongRec (itemState).Lo).Lo);
    FCanvas.Handle := hDC;
    FCanvas.Font := Font;
    FCanvas.Brush := Brush;
    if integer (itemID) >= 0 then
      DrawItem (itemID, rcItem, State)
    else
      FCanvas.FillRect (rcItem);
    FCanvas.Handle := 0
  end;
end;

// default DrawItem method

procedure TExPageControl.DrawItem (Index: Integer; Rect: TRect; State:
TOwnerDrawState);
begin
  if Assigned(FOnDrawItem) then
    FOnDrawItem (Self, Index, Rect, State)
  else begin
    FCanvas.FillRect (Rect);
    if odSelected in State then
      FCanvas.TextOut (Rect.Left + 6, Rect.Top+4, Tabs[Index])
    else
      FCanvas.TextOut (Rect.Left + 2, Rect.Top, Tabs[Index])
  end
end;

//-- COMPONENT EDITOR ------------------------------
```

```
// provide a new PageControl component editor so that the
// glyphs are handled correctly

type
  TExPageControlEditor = class(TDefaultEditor)
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;

procedure TExPageControlEditor.ExecuteVerb(Index: Integer);
var
  PageControl: TExPageControl;
  Page: TExTabSheet;
  Designer: TFormDesigner;
begin
  if Component is TTabSheet then
    PageControl := TExTabSheet(Component).PageControl as TExPageControl
  else
    PageControl := TExPageControl(Component);

  if PageControl <> nil then
  begin
    Designer := Self.Designer;
    if Index = 0 then
    begin
      Page := TExTabSheet.Create(Designer.Form);
      try
        Page.Name := Designer.UniqueName(TExTabSheet.ClassName);
        Page.Parent := PageControl;
        Page.PageControl := PageControl
      except
        Page.Free;
        raise
      end;
      PageControl.ActivePage := Page;
      PageControl.UpdateGlyphs
    end else begin
      Page := PageControl.FindNextPage(PageControl.ActivePage,Index = 1, False) as
TExTabSheet;
      if (Page <> nil) and (Page <> PageControl.ActivePage) then
        PageControl.ActivePage := Page
    end;

    Designer.SelectComponent(Page);
    Designer.Modified
  end
end;

function TExPageControlEditor.GetVerb(Index: Integer): string;
begin
  case Index of
    0 : Result := 'New (ex)page';
    1 : Result := 'Next page';
    2 : Result := 'Previous page'
  end
end;

function TExPageControlEditor.GetVerbCount: Integer;
begin
  Result := 3
end;
```

```
procedure Register;
begin
  RegisterClasses([TExTabSheet]);
  RegisterComponents ('Custom', [TExPageControl]);
  RegisterComponentEditor(TExPageControl, TExPageControlEditor);
  RegisterComponentEditor(TExTabSheet, TExPageControlEditor)
end;

end.
```

Return to Article

Return to Component Cookbook

Return to Front Page

Source for Big Bitmaps

```
// BITMAP VIEWER
// Enables bitmaps which are disc files to be displayed in a
// TImage-like control without loading the bitmap into memory.
// Great for very large bitmaps.

// Grahame Marsh 1996

unit BitView;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Dialogs, Forms, DsgnIntf;

type
  TBMPFilename = type string; // filename will have its owner property editor

type
  TBigBitmapViewer = class (TGraphicControl)
  private
    FFileName: TBMPFilename;  // bitmap filename, own type so it can have own prop
editor
    FPalette : HPalette;      // handle to bitmap palette
    FData    : pointer;       // pointer to start of data in memory mapped file
    FBitmapWidth,             // copy of bitmap width info for convience
    FBitmapHeight,            // copy of bitmap height info for convience
    FColours : integer;       // number of colours in palette
    FCentre,                  // centre the bitmap in the control
    FStretch,                 // stretch the bitmap to fill the control
    FAutoSize,                // automatically size the control to display the bitmap
    FActive  : boolean;       // true opens the viewer, false it's closed
    FFileHeader : PBitmapFileHeader;  // pointer to TBitmapFileHeader struct
    FInfoHeader : PBitmapInfoHeader;  // pointer to TBitmapInfoHeader struct
    FInfo : PBitmapInfo;      // pointer to the TBitmapInfo struct
    FPixelStart : pointer;    // pointer to the start of the pixel data
    procedure SetActive (Value : boolean);
    procedure SetAutoSize (Value : boolean);
    procedure SetFilename (const Value : TBMPFilename);
    procedure SetStretch (Value : boolean);
    procedure SetCentre (Value : boolean);
    procedure SetDummyInt (Value : integer);
  protected
    procedure OpenViewer; virtual;
    procedure CloseViewer; virtual;
    procedure GetPalette;
    procedure Paint; override;
    procedure Changes; virtual;
  public
    constructor Create (AOwner : TComponent); override;
    destructor Destroy; override;
// open the viewer
    procedure Close;
// close the viewer
    procedure Open;
// pointer to the file header info
```

```
    property BitmapFileHeader : PBitmapFileHeader read FFileHeader;
// pointer to the bitmap info header
    property BitmapInfoHeader : PBitmapInfoHeader read FInfoHeader;
// pointer to the bitmap info
    property BitmapInfo : PBitmapInfo read FInfo;
// pointer to the bitmap pixel data array
    property PixelStart : pointer read FPixelStart;
// palette handle
    property Palette : HPalette read FPalette;
  published
//READ-WRITE PROPS
// size control to bitmap
    property AutoSize : boolean read FAutoSize write SetAutoSize default false;
// bitmap centred
    property Centre : boolean read FCentre write SetCentre default false;
// filename of bitmap
    property Filename : TBMPFilename read FFilename write SetFilename;
// stretch bitmap
    property Stretch : boolean read FStretch write SetStretch default false;
// READ-ONLY PROPS
// number of colours in the bitmap palette
    property Colours : integer read FColours write SetDummyInt stored false;
// bitmap width
    property BitmapHeight : integer read FBitmapHeight write SetDummyInt stored false;
// bitmap height
    property BitmapWidth : integer read FBitmapWidth write SetDummyInt stored false;
// TGraphicControl PROPS NOW PUBLISHED
    property Align;
    property DragCursor;
    property DragMode;
    property Enabled;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property Visible;
    property OnClick;
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnStartDrag;
// viewer activate - stream active last!
    property Active : boolean read FActive write SetActive default false;
  end;

// filename property editor .. fileopen dialog box
type
  TBMPFilenameProperty = class (TStringProperty)
  public
    procedure Edit; override;
    function GetAttributes: TPropertyAttributes; override;
  end;

procedure Register;
```

```
implementation

const
  BitmapSignature = $4D42;

procedure InvalidBitmap;
begin
  raise Exception.Create ('Bitmap image is not valid')
end;

procedure NotWhenActive;
begin
  raise Exception.Create ('Not on an active big bitmap viewer')
end;

constructor TBigBitmapViewer.Create (AOwner : TComponent);
begin
  inherited Create (AOwner);
  Width := 150;
  Height := 150
end;

destructor TBigBitmapViewer.Destroy;
begin
  CloseViewer;  // ensure file view is freed
  inherited Destroy
end;

procedure TBigBitmapViewer.GetPalette;
var
  SysPalSize,
  Loop,
  LogSize : integer;
  LogPalette : PLogPalette;
  DC : HDC;
  Focus : HWND;
begin
// fetch palette for colour bitmaps only
  if FColours > 2 then
  begin

// create palette from bitmap info
    LogSize := SizeOf (TLogPalette) + pred(FColours) * SizeOf(TPaletteEntry);
    LogPalette := AllocMem (LogSize);
    try
      with LogPalette^ do
      begin
        palNumEntries := FColours;
        palVersion := $0300;
{$IFOPT R+}
  {$DEFINE R_PLUS}
  {$R-}
{$ENDIF}
        Focus := GetFocus;
        DC := GetDC (Focus);
        try
          SysPalSize := GetDeviceCaps (DC, SIZEPALETTE);
          if (FColours = 16) and (SysPalSize >= 16) then
          begin
            GetSystemPaletteEntries (DC, 0, 8, palPalEntry);
            loop := 8;
            GetSystemPaletteEntries (DC, SysPalSize - loop, loop, palPalEntry[loop])
          end else
            with FInfo^ do
              for loop := 0 to pred (FColours) do
              begin
                palPalEntry[loop].peRed   := bmiColors[loop].rgbRed;
                palPalEntry[loop].peGreen := bmiColors[loop].rgbGreen;
                palPalEntry[loop].peBlue  := bmiColors[loop].rgbBlue
              end
        finally
          ReleaseDC(Focus, DC)
        end
{$IFDEF R_PLUS}
  {$R+}
  {$UNDEF R_PLUS}
{$ENDIF}
      end;
      FPalette := CreatePalette (LogPalette^)
    finally
      FreeMem (LogPalette, LogSize)
    end
  end
end;

procedure TBigBitmapViewer.OpenViewer;
var
  FileHandle,
  MapHandle : THandle;
begin
  if FActive then exit;

// open file
  FileHandle := FileOpen (FFilename, fmOpenRead + fmShareDenyNone);
  if FileHandle = INVALID_HANDLE_VALUE then
    raise Exception.Create ('Failed to open ' + FFilename);

// create file map
  try
    MapHandle := CreateFileMapping (FileHandle, nil, PAGE_READONLY, 0, 0, nil);
    if MapHandle = 0 then
      raise Exception.Create ('Failed to map file')
  finally
    CloseHandle (FileHandle)
  end;

// view file map
  try
    FData := MapViewOfFile (MapHandle, FILE_MAP_READ, 0, 0, 0);
    if FData = nil then
      raise Exception.Create ('Failed to view map file')
  finally
    CloseHandle (MapHandle)
  end;

// set pointers into file view
  FFileHeader := FData;

// test for valid bitmap file:
  if FFileHeader^.bfType <> BitmapSignature then
  begin
```

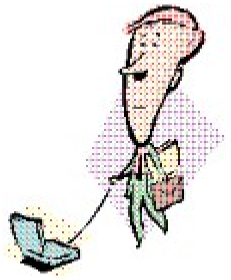
```
    UnmapViewOfFile (FData);
    FData := nil;
    InvalidBitmap
  end;

// set up a few other pointers
  FInfoHeader := pointer (integer (FData) + sizeof (TBitmapFileHeader));
  FInfo := pointer (FInfoHeader);
  FPixelStart := pointer (integer(FData) + FFileHeader^.bfOffBits);

// get number of colours
  with FInfoHeader^ do
    if biClrUsed <> 0 then
      FColours := biClrUsed
    else
      case biBitCount of
        1,
        4,
        8 : FColours := 1 shl biBitCount
      else
        FColours := 0
      end;

  // get bitmap size into easy to access variables
  FBitmapHeight := FInfoHeader^.biHeight;
  FBitmapWidth := FInfoHeader^.biWidth;

// fetch the palette
  GetPalette;

// other setups
  FActive := true;
  Changes
end;

procedure TBigBitmapViewer.Paint;
var
  OldPalette : HPalette;
  Dest       : TRect;
begin
  with Canvas do
    if (csDesigning in ComponentState) and not FActive then
    begin
      Pen.Style := psDash;
      Brush.Style := bsClear;
      Rectangle (0, 0, Width, Height)
    end else begin
      if FPalette <> 0 then
        OldPalette := SelectPalette (Handle, FPalette, false)
      else
        OldPalette := 0;

      try
        RealizePalette (Handle);

        if FStretch then
          Dest := ClientRect
        else
          if Centre then
            Dest := Rect ((Width - FBitmapWidth) div 2,
                          (Height - FBitmapHeight) div 2,
                          FBitmapWidth, FBitmapHeight)
          else
            Dest := Rect (0, 0, FBitmapWidth, FBitmapHeight);

        with Dest do
          StretchDIBits (Handle,
                         Left, Top, Right, Bottom,
                         0, 0, FBitmapWidth, FBitmapHeight,
                         FPixelStart, FInfo^,
                         DIB_RGB_COLORS, SRCCOPY)
      finally
        if OldPalette <> 0 then
          SelectPalette (Handle, OldPalette, false)
      end
    end
end;

procedure TBigBitmapViewer.CloseViewer;
begin
  if FActive then
  begin
    FActive := false;
    if FData <> nil then
    begin
      UnmapViewOfFile (FData);  //  remove the memory mapped file view
      FData := nil
    end;
    if FPalette <> 0 then
      DeleteObject (FPalette)   // free the palette
  end
end;

procedure TBigBitmapViewer.Open;
begin
  Active := true
end;

procedure TBigBitmapViewer.Close;
begin
  Active := false
end;

procedure TBigBitmapViewer.SetActive (Value : boolean);
begin
  if Value <> FActive then
    if Value then
      OpenViewer
    else
      CloseViewer
end;

procedure TBigBitmapViewer.SetAutoSize (Value : boolean);
begin
  if Value <> FAutoSize then
  begin
    FAutoSize := Value;
    Changes
  end
end;

procedure TBigBitmapViewer.SetStretch (Value : boolean);
begin
  if Value <> FStretch then
  begin
    FStretch := Value;
    Changes
  end
end;

procedure TBigBitmapViewer.SetCentre (Value : boolean);
begin
  if Value <> FCentre then
  begin
    FCentre := Value;
    Changes
  end
end;

procedure TBigBitmapViewer.SetFilename (const Value : TBMPFilename);
begin
  if Value <> FFilename then
  begin
    if FActive then
      NotWhenActive;
    FFilename := Value
  end
end;

procedure TBigBitmapViewer.SetDummyInt (Value : integer);
begin
end;

procedure TBigBitmapViewer.Changes;
begin
  if (BitmapWidth >= Width) and (BitmapHeight >= Height) then
    ControlStyle := ControlStyle + [csOpaque]
  else
    ControlStyle := ControlStyle - [csOpaque];

  if AutoSize and (BitmapWidth > 0) and (BitmapHeight > 0) then
    SetBounds (Left, Top, BitmapWidth, BitmapHeight)
  else
    Invalidate
end;

//--- filename property editor

procedure TBMPFilenameProperty.Edit;
begin
  with TOpenDialog.Create(Application) do
  begin
    Filename := GetValue;
    Filter := 'Windows bitmaps (*.BMP)|*.BMP';
    Options := Options + [ofPathMustExist, ofFileMustExist, ofHideReadOnly];
    try
      if Execute then
        SetValue(Filename)
    finally
      Free
    end
  end
end;

function TBMPFilenameProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog, paRevertable]
```

```
end;

// - register component and filename editor

procedure Register;
begin
  RegisterComponents ('Custom', [TBigBitmapViewer]);
  RegisterPropertyEditor (TypeInfo (TBMPFilename), nil, '', TBMPFilenameProperty);
end;

end.
```

Return to Article

Return to Component Cookbook

Return to Front Page



So… You Think You Are Computer-Illiterate?

The following is an excerpt from the Wall Street Journal by Jim Carlton. I thought you would enjoy it as much as I did!
This was forwarded to me by P. Wyatt

1. Compaq is considering changing the command "**Press Any Key**" to "**Press Return Key**" because of the flood of calls asking where the "**Any**" key is.
2. AST technical support had a caller complaining that her mouse was hard to control with the dust cover on. The cover turned out to be the plastic bag the mouse was packaged in.
3. Another Compaq technician received a call from a man complaining that the system wouldn't read word processing files from his old diskettes. After trouble- shooting for magnets and heat failed to diagnose the problem, it was found that the customer labeled the diskettes then rolled them into the typewriter to type the labels.
4. Another AST customer was asked to send a copy of her defective diskettes. A few days later a letter arrived from the customer along with Xeroxed copies of the floppies.
5. A Dell technician advised his customer to put his troubled floppy back in the drive and close the door. The customer asked the tech to hold on, and was heard putting the phone down, getting up and crossing the room to close the door to his room.
6. Another Dell customer called to say he couldn't get his computer to fax anything. After 40 minutes of troubleshooting, the technician discovered the man was trying to fax a piece of paper by holding it in front of the monitor screen and hitting the "send" key.
7. Another Dell customer needed help setting up a new program, so a Dell tech suggested he go to the local Egghead. *"Yeah, I got me a couple of friends,"* the customer replied. When told Egghead was a software store, the man said, *"Oh, I thought you meant for me to find a couple of geeks."*
8. Yet another Dell customer called to complain that his keyboard no longer worked. He had cleaned it by filling up his tub with soap and water and soaking the keyboard for a day, then removing all the keys and washing them individually.
9. A Dell technician received a call from a customer who was enraged because his computer had told him he was "bad and an invalid". The tech explained that the computer's "bad command" and "invalid" responses shouldn't be taken personally.
10. An exasperated caller to Dell Computer Tech Support couldn't get her new Dell Computer to turn on. After ensuring the computer was plugged in, the technician asked her what happened when she pushed the power button. Her response, *"I pushed and pushed on this foot pedal and nothing happens."* The "foot pedal" turned out to be the computer's mouse.
11. Another customer called Compaq tech support to say her brand-new computer wouldn't work. She said she unpacked the unit, plugged it in, and sat there for 20 minutes waiting for something to happen. When asked what happened when she pressed the power switch, she asked *"What power switch?"*
12. True story from a Novell NetWire SysOp:
    Caller: *"Hello, is this Tech Support?"*
    Tech: *"Yes, it is. How may I help you?"*
    Caller: *"The cup holder on my PC is broken and I am within my warranty period. How do I go about getting that fixed?"*
    Tech: *"I'm sorry, but did you say a cup holder?"*
    Caller: *"Yes, it's attached to the front of my computer."*
    Tech: *"Please excuse me if I seem a bit stumped, It's because I am. Did you receive this as part*

of a promotional, at a trade show? How did you get this cup holder? Does it have any trademark on it?"
Caller: *"It came with my computer, I don't know anything about a promotional. It just has '**4X**' on it."*

At this point the Tech Rep had to mute the caller, because he couldn't stand it. The caller had been using the load drawer of the CD-ROM drive as a cup holder, and snapped it off the drive!

Return to Front Page

Extending the Background Bitmap Technique

by Robert Vivrette - RobertV@compuserve.com

A few weeks back, Dave Weld of "*It Just Works Software*" (dave@itjustworks.com) commented on the Background Bitmaps on Forms article in issue #17 of UNDU. His comment was: *"Using your tip on the background image works but only places it in the upper left hand corner (0,0). How do you get it to tile like windows wallpaper to fill the form background? Or can you?"*

You sure can Dave… All you need to do is to repeat the bitmap across the form in the form's paint method. Just modify FormPaint to read as follows:

```
procedure TForm1.FormPaint(Sender: TObject);
var
  X,Y,W,H : Integer;
begin
  W := BackgroundBitmap.Width;
  H := BackgroundBitmap.Height;
  for x := 0 to ClientWidth div W do
    for y := 0 to ClientHeight div H do
      Canvas.Draw(x*W,y*H,BackgroundBitmap);
end;
```

Return to Tips & Tricks

Return to Front Page

Paradox File Size Limits

by Brad Evans - brade@iinet.com.au

How big can a Paradox table be? My first answer was huge, a million records, 100 million records, a Gig, perhaps even 2 Gig. Well that's what I thought until I started getting table full error messages at about 750,000 records.

It seems Paradox tables can only grow to a certain size before becoming full. Out of the box Paradox tables can grow to 134,215,680 bytes (about 131mb). So if you want tables bigger than 131mb then it won't work without a little planning.

The BDE configuration program allows a figure for block size. Each table can have exactly 65,535 blocks. Thus a table with a 2k block size can grow to 131mb, a table with a 4K block size can grow to 262mb.

Unfortunately the block size is fixed when the table is created. So for all the tables you have already created they can only grow to be 131mb in size. However if you change the block size and recreate your tables you can set their maximum size to whatever you like.

Forewarned is forearmed. If you think your tables will grow set their block size now. I use a block size of 8K and have not found a performance hit at all.

About the Author

Brad Evans is a Programmer for Realtime Computing in Perth Western Australia. Brad develops leasing applications full time using Borland Delphi 2.0. You can reach Brad through e-mail at brade@iinet.com.au or on the web at http://www.iinet.com.au/~brade/.

Return to Tips & Tricks

Return to Front Page

Is Someone Else Running?

by Paul Harding - 100046.2604@compuserve.com

In issue #12 of UNDU I presented an article on Am I Running Already?. This new tip is a bit different and lets you determine if another program is running.

How Do I Find Out if Another Program is Running?

If your program needs to know if another different application is running (for example, you may need to know if some communications software is running in the background before your application can proceed), you can use the API call FindWindow like this:

```
if FindWindow(myClassName, nil) > 0 then {the application is running)
```

But how do I know what myClassName is? Well, you can build up a list of class names for all the windows that are in memory. To do this, we need to use the Windows API calls GetClassName and GetNextWindowHandle. If your application has focus, then the handle of your own form is easy to get - its just Self.Handle. Then if we make a loop for the GetNextWindowHandle call, we will be able to step through the windows getting their handles and by using GetClassName we can get their Class Names, too.

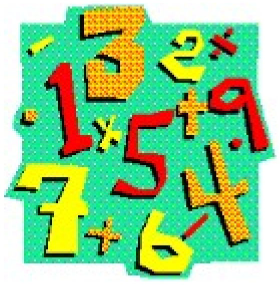
So start a new project, and drop a Button and a ListBox onto the form. With the following code on the buttons click, you can fill the list box with currently running Class Names:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  myClassName: Array[0..63] of Char;
  Handle: THandle;
begin
  ListBox1.Items.Clear;
  ListBox1.Font.Name := 'Courier';
  ListBox1.Items.Add(Format('%-7s %-64s',[' Handle','Class Name']));
  ListBox1.Items.Add(Format('%-7s %-64s',[' ------','----------']));
  {start off the list with the current application}
  Handle := Self.Handle;
  GetClassName(Self.Handle, myClassName, 64);
  ListBox1.Items.Add(Format('%7d %-64s',[Handle,StrPas(myClassName)]));
  {now list all the others}
  While Handle > 0 do
    begin
      Handle := GetNextWindow(Handle, GW_HWNDNEXT);
      GetClassName(Handle, myClassName, 64);
      if myClassName[0] <> '#' then
        ListBox1.Items.Add(
          Format('%7d %-64s',[Handle,StrPas(myClassName)]));
    end;
end;
```

Notice that if you run this code outside of the Delphi environment you will get a much shorter list. Now all you need to do is run the other application that you are interested in in the background, and then run this one. Click the button and the other apps Class Name appears in the window as well as all the current ones. It is this Class Name that appears in the list that you need for the call to FindWindow. If you find the other application is not running, but you want to launch it, then use the easy to use WinExec call: just look up WinExec in the online help for more information about that.

Return to Tips & Tricks

Return to Front Page



A Safer Way of Working With Enumerated Types

by Andy Cogan - Epiphany Software

In perusing The Unofficial Newsletter of Delphi Users, issue #16. I enjoyed the discussion of enumerated types. However, I noticed an important omission in the discussion. In my code, I often need to iterate through enumerated types; there is a bad way and a good way to do it.

For example, to iterate through the days of the week in the example, you could do it this way:

```
var
  Day: DayOfWeek;
begin
  for Day := Mon to Sun do
  ....
end;
```

but a better, less maintenance intensive way is this way:

```
var
  Day: DayOfWeek;
begin
  for Day := Low(DayOfWeek) to Hi(DayOfWeek) do
  ....
end;
```

Return to Tips & Tricks

Return to Front Page

Aligning Text in a Grid

by James Sager - jsager@ao.net

For a long while (almost the same day that I purchased Delphi) I have been interested in being able to align text while it's being input into a string grid. With some digging I found that the TCustomGrid has an internal component called TInplaceEdit that handles the input into a grid. Much later on I found a component called AlignEdit (the author's name escapes me, but it's out there on the web) which came with code, permitting me to see how to change the alignment of text within a TEdit component (TInplaceEdit is ultimately derived from the TEdit component). Well, I put 2 and 2 together and got 3 1/2 - just a little shy of my desired result. I have a person that I converse with on the net from time to time who has been an invaluable source of Delphi information for me - his name is Norbert Triebenbacher (N.Triebenbacher@t-online.de). I sent all the applicable information to him and asked him for some help. Lo and Behold...
TIEAlignStringGrid is born. To set the alignment of the inplace editor you need only call the procedure SetAlignment with a TAlignment parameter and the inplace editor can be left, right or center justified. The call can be in the OnClick procedure or any where you might want it to be (I use it in the OnSelectCell event). You can align each individual cell if you desire.

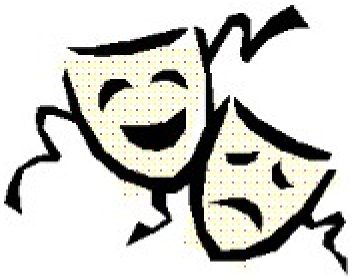
Anyway, I hope this can be of help to someone else. I noticed that not too many programmers were too interested in doing this but, perhaps having it handed to them, they might change their minds.

Happy Programming!

Source Code for this Project

Return to Tips & Tricks

Return to Front Page

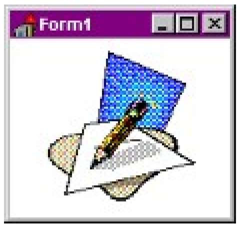


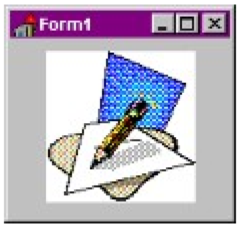
True Transparent Images With Masking

by Robert Vivrette - RobertV@compuserve.com

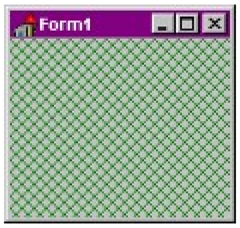
You are all excited about your new Delphi program as you deliver it to a co-worker for testing. The program is accented by the use of several graphic images on the main form. It wasn't too difficult to include those graphics... Just drop a TImage on the form, select a graphic and that's it!

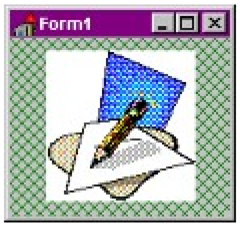
But wait... When the program runs on your co-workers machine, there is a big white box around the graphic. His system colors are different from yours. On your machine, the default window background was white, but his is gray and instead of the graphic integrating into the form's background, it is boxed in by the white background of the graphic.



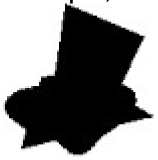


"No problem…", you say, *"I'll just hard code the form color to be white regardless of the system settings."* Okay, I want you to put your left hand out and slap the back of it with your right hand while scolding yourself… *"Bad!!! Bad!!! Bad!!!"* Programmers who hard code things like this in their programs just to make **their** life easier should have their Delphi club card taken away!
Borland has made life a little easier by providing a **BrushCopy** procedure. This procedure replaces one color in a graphic with the color of the form. You could simply use this procedure when painting the graphic on the screen, but suppose the background is not all one color. Maybe you are allowing the graphic to sit on top of other graphic components, such as a clickable graphic on a map? In this case (and in a number of other cases) **BrushCopy** doesn't do you much good.
For the purpose of illustration, look at these two images. The form on the left is filled with a hatch pattern. Obviously, any graphic you place there will disturb the background regardless of which color you choose to replace.

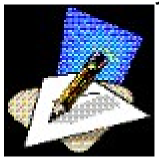




What you need is a way of selectively placing a graphic image on a form without disturbing some areas of the form's contents. This is done with a technique called masking. With this technique, you provide an additional graphic called a "mask" that defines the areas of your image that you want painted. For example, for our Pen and Paper graphic shown above, the mask would look something like this:



Think of this mask as a kind of "cookie-cutter". Only the region of the image that falls within this black area of the mask will make it to the screen. In addition, you need to make sure the graphic you are using has a black background. Black is needed because it is represented by zero's in the Windows color scheme. That way, when you combine the colors together, the black doesn't mathematically skew other colors when your image gets to the form.



The masking solution I have put together here is based mostly off of the TImage component. Normally, I would have descended a new component off of Delphi's stock TImage, added the mask property, and then modified the paint routine. Unfortunately, the end result had a few bad behaviors that I couldn't get around. As a result, I have essentially duplicated a TImage and modified it accordingly. Not good use of inheritance I know…
In the original TImage, the functional part of the paint routine looked like this:

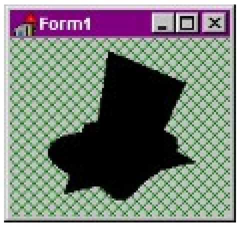
```
  with inherited Canvas do
    StretchDraw(Dest, Picture.Graphic);
```

In the new TMaskedImage component, the functional part of the paint routine is extended a bit to look like this:

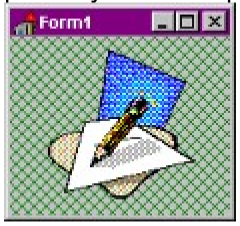
```
  with inherited Canvas do
    begin
      CopyMode := srcAnd;
      StretchDraw(Dest, Mask.Graphic);
      CopyMode := srcPaint;
      StretchDraw(Dest, Picture.Graphic);
    end;
```

In a TImage, the graphic is specified in the **Picture** property. All I have done to add support for the mask is to add an identical **Mask** property. The mask graphic is selected into this property in exactly the same way you would select an image for the Picture property.

In the modified painting code, you will see that the first thing that is done is to set CopyMode to *srcAnd*. While painting the mask, this will have the effect of punching out a section of the form's canvas. The area of the mask that is black will cause the form to become black and the area of the mask that is white will leave the form alone. With *srcAnd*, black added to any color becomes black. White added to any color remains that original color. After setting the copy mode, we paint the mask image onto the form and here is the result:



Now we set CopyMode to *srcPaint*. While painting the final image, this has the effect of mathematically combining the colors on the form with the colors in the image. Since we have punched out a section of black on the form, the colors of the image that are in that area are simply copied to the form. But the other areas of the form are left untouched because the image has black in those areas. With *srcPaint*, black plus any color is equal to that color. White plus any color is equal to white.



So there you have it! The only real disadvantages of this technique is; Firstly that you have to do a little preparation work to make the Mask and Picture ahead of time. Also, the painting process copies two images to the form for every paint (Borland's BrushCopy does this as well). If you are painting many images on the screen or a single image over and over again, you may see some flicker. I have developed a way around this, but it is a bit more complex and I wanted to keep the example here simple.
As a side note, the Win32 API does have a MaskBlt routine that does much of this for you. The problem is that it currently is not supported by Win95… only WinNT. However, Delphi 2.0 does have wrapper routines for MaskBlt so if you are developing for WinNT, you can use MaskBlt instead!

Source for TMaskedImage

Return to Component Cookbook

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #19 - January 1997

Simplifying Code Management with "Include"

by Robert Vivrette - RobertV@compuserve.com

Procedures and Functions are a normal part of any Delphi program (and virtually all other languages as well for that matter). Recently I was working on a component that encoded and decoded Tiff graphics files. One of the overriding concerns for this code was that it had to be FAST… no excuses!

Down in the bowels of the code was about 10 lines of code that was responsible for sending data to an output buffer. When that buffer filled up, it would be written to the disk and then reset to retrieve new data. These 10 lines were naturally a candidate for a procedure call, because I used them over and over again from different parts of the main code.

When the code was finished, I began looking for ways to optimize the speed of the routine. Saving a 5 meg TIFF file took about 4 seconds but I felt it could be faster. By profiling the code (using Speed Daemon, reviewed elsewhere in this issue) I discovered that I was calling this procedure an average of about 150,000 times during those 4 seconds.

So far, nothing new to any of you. However, some of you may not be aware of this but there is actually a little bit of overhead calling procedures and functions. When a call to a procedure or function occurs, the CPU must push the current code address onto the stack, as well as managing the transfer of parameters into the procedure. I knew that if I just repeated these 10 lines of code in the body of the main code without having to call a procedure, I would save roughly 150,000 of these stack manipulations. I was right… It shaved about a half second off of my 4 second TIFF writing time. Normally, the microscopic overhead for calling a function or procedure is negligible, but in this case it added up because I was calling the procedure so many times.

But now, I have roughly 15 copies of these 10 lines of code strewn throughout the TIFF writing code. Obviously a code management nightmare. If I decide to change the way I manage this buffer, I need to change all 15 copies of this code.

Hang on… we are almost there…

In Delphi there is a compiler directive, `{$INCLUDE}`, that allows you to include a separate source file and insert it into your code. In Delphi 1.0 however, that separate source file had to be a complete and separately valid code block (complete with Begin.. End) and also that the include in the main code could not appear inside a procedure. For example, this would be invalid in Delphi 1.0:

```
Procedure DryCleanMySuit;
begin
  A := B * 10;
  MyLabel := 'Hey there folks';
  {$INCLUDE AFILE.INC}
  ShowMessage('It is a nice day out…');
end;
```

In short, this meant that in Delphi 1.0 I couldn't put those 10 lines of code in a separate file and include them multiple times within the body of my TIFF writing code. When I switched over to Delphi 2.0 last year, I checked the help file to see if it would now be allowed. Nope… it said the same thing. So just a week or so ago I figured I would try it anyway. Lo and Behold… it worked. They changed the way the compiler reads the source so that you could use the `{$INCLUDE}` compiler directive anywhere and with just code fragements. They just hadn't changed the reference in the help system.

Now, I have my 10 lines of code in a separate include file (*.INC) and I use the `{$INCLUDE}` directive 15 or so times in the body of the Tiff writing code. When the Delphi compiler gets to these lines, it just reads the 10 lines in when it gets to each of the compiler directives. Now I just have 1 copy of the code that I need to maintain, but I gain the speed advantage of not having to put it in a procedure.

So the moral of the story is… Don't believe everything the help system tells you!

Return to Tips & Tricks

Return to Front Page

Probing into the Shell's File Operations

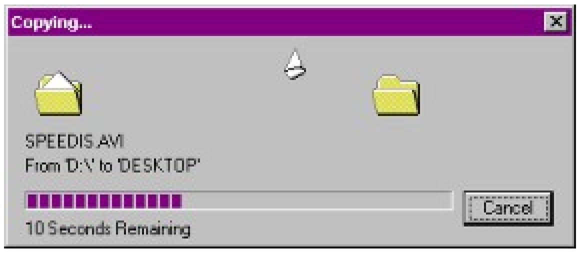
For those of you who use Win95, you no doubt are familiar with the way you can move, copy, delete, and rename files and folders on the desktop. However, did you know that most (if not all) of these facilities are available to you in Delphi 2.0?
The secret to this technique is the **SHFileOperation** API call. With this function, you can do just about anything you can do from the Win95 shell including Copying multiple files to multiple destinations, renaming files, even deleting files and sending them to the Recycle Bin. The function takes a single parameter which is a structure of type TShFileOpStruct defined in the **ShellAPI** unit.

```
TSHFileOpStruct = record
  Wnd: HWND;
  wFunc: UINT;
  pFrom: PAnsiChar;
  pTo: PAnsiChar;
  fFlags: FILEOP_FLAGS;
  fAnyOperationsAborted: BOOL;
  hNameMappings: Pointer;
  lpszProgressTitle: PAnsiChar;
end;
```

It looks a little imposing, but is actually quite simple to work with. Take the following button click event as an example (**Note**: You need to have **ShlObj** and **ShellAPI** in your Uses clause).

```
procedure TForm1.Button1Click(Sender: TObject);
var
  F : TShFileOpStruct;
begin
  F.Wnd := Handle;      // if 0, then no parent and can task switch away
  F.wFunc := FO_COPY;
  F.pFrom := 'D:\SPEEDIS.AVI';
  F.pTo := 'C:\WINDOWS\DESKTOP\SPEEDIS.AVI';
  F.fFlags := FOF_ALLOWUNDO or FOF_RENAMEONCOLLISION;
  if ShFileOperation(F) <> 0 then ShowMessage('Copy Failed');
end;
```

This code will copy the file called SPEEDIS.AVI from the root of the D: drive over to the desktop in the Windows directory. Running this code causes the file to be copied appropriately and is accompanied by one of the dialog boxes we all know and love.



All that is being done in this example is to declare a local variable of type TShFileOpStruct (which I labeled as **F**). Then you fill in the fields in the record to define what you want ShFileOperation to do. Then you call ShFileOperation and pass the TShFileOpStruct variable as its only parameter. If the return value is zero the function worked, and if it is non-zero, then the call failed. Here's what the above code sample

is doing:
The **Wnd** field of the structure is a handle used to determine who the parent is of any dialog box that might appear as a result of the call. In my opinion the help system on the ShFileOperation call is misleading about this parameter. From trail and error, I have determined that if you assign the Wnd field to be the form's Handle, then the dialog will become a child window of the form. This will mean that you will not be able to switch focus back to the form until the dialog is gone. However, if you put a zero in the Wnd field, it tells the shell that there is no parent. In this case you will be able to click on your form and bring it to the foreground on top of the dialog. Keep in mind also that the dialogs will not appear in some cases… for example, if a file copy operation is less than two or three seconds in length, the copy dialog won't even appear… the same behavior as in the shell.

The **wFunc** field can be FO_COPY, FO_DELETE, FO_MOVE, or FO_RENAME and defines which file operation will be performed. The **pFrom** and **pTo** fields have different meanings depending on which file operation is chosen.

The **fFlags** field holds various flags (there are 10 mentioned in the Win32 API Help file) that control the file operation. In the example above I am using FOF_RENAMEONCOLLISION and FOF_ALLOWUNDO. The FOF_RENAMECOLLISION flag is used in a move, copy, or rename operation and gives the file a new name if it collides with an existing one. For example, if you were to execute the above code several times, you would have multiple copies of the destination file instead of it copying over the same file each time. The file names are distinguished by pre-pending *"Copy #1 of "*, *"Copy #2 of "*, etc, before the file name. The FOF_ALLOWUNDO flag tells the shell to preserve Undo information if possible. For example after executing the code above, you could go back to the desktop and press CTRL-Z (the shortcut for UNDO on the desktop) and the copied file would be removed.

As a second example, consider the following code:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  F : TShFileOpStruct;
begin
  F.Wnd := Handle;
  F.wFunc := FO_DELETE;
  F.pFrom := 'C:\WIN95\DESKTOP\SPEEDIS.AVI';
  F.fFlags := FOF_ALLOWUNDO;
  if ShFileOperation(F) <> 0 then ShowMessage('Delete Failed');
end;
```

In this example, we are using the FO_DELETE flag to tell ShFileOperation to delete the SPEEDIS.AVI file from the desktop. However, because the FOF_ALLOWUNDO flag is specified, the deleted file is placed into the recycle bin, where you could retrieve it if necessary. If you didn't include this flag, the file would be deleted without Undo information being saved by the operating system.

As you can see, there are a lot of powerful features of the ShFileOperation function. You can find additional information by looking up **SHFileOperation** and **ShFileOpStruct** in the Win32 API Help file that comes with Delphi.

Return to Tips & Tricks

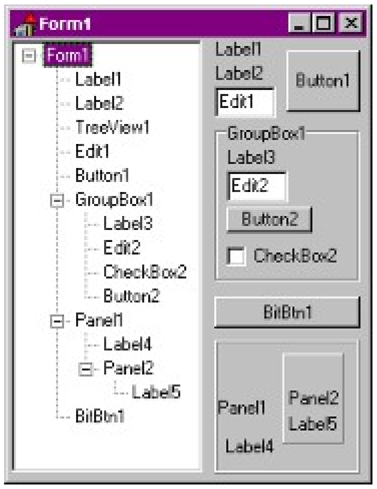
Return to Front Page

TreeViews

by Robert Vivrette - RobertV@compuserve.com

TreeViews are very powerful tools if you need to see data in a hierarchical fashion. Often times this is used to show parent/child relationships such as with directory trees for example. With a directory tree, each file or folder exists somewhere within the overall structure. It would have a parent directory, and it might have sub-directories underneath it. Those sub-directories might also have sub-directories, and so on. A TreeView component is excellent at viewing such data.
But you can also use TreeViews to view other kind of organizational data. Take components on a form as a second example. Each component has a parent, and each component may have zero or more children… perfectly suited for viewing in a tree structure.

To illustrate the use of TreeViews for something like this, I have put together a little sample code. I created a form and added all sorts of controls to the right half of the form. Each control has some relationship to the main form. Some components are more distantly connected to the form than others (i.e. those in a group box or on a panel). I then added a TreeView control to the left side to display the relationship of the controls on the form. Here is what the finished project looks like.



The code to achieve this is really quite simple and only consists of the following two procedures:

```
procedure TForm1.AddAComponent(A: TComponent);
var
  b : Integer;
begin
  for b := 0 to TreeView1.Items.Count-1 do
    if TreeView1.Items[b].Text = A.Name then exit;
  if A.HasParent then AddAComponent(A.GetParentComponent);
  for b := 0 to TreeView1.Items.Count-1 do
    if TreeView1.Items[b].Text = A.GetParentComponent.Name then
      TreeView1.Items.AddChild(TreeView1.Items[b],A.Name);
end;

procedure TForm1.FormCreate(Sender: TObject);
var
```

```
  a : Integer;
begin
  TreeView1.Items.Add(TreeView1.Selected,Name);
  For a := 0 to ComponentCount-1 do AddAComponent(Components[a]);
end;
```

When the form is created, it adds a root node to the TreeView that is named the same as the form it is sitting on. A TreeView components maintains a list of all of the nodes it has through its items property. This property is really just a list of nodes, and each node knows its relationship to other nodes (i.e. what children it has, who its parent is, and so on). By adding this first node, we give the rest of the nodes something to "hang" off of.

Next, I run through the form's components list and add each component to the TreeView by passing that component to the AddAComponent procedure. This procedure is recursive in nature, meaning that it is able to call itself. More on that a little later.

Once in the AddAComponent procedure, we first need to determine if the component has already been added to the tree. All we do is run through the list of nodes and compare the text property of each node to the name of the component that has been passed in. If the component has already been added, then we exit.

Next, we determine if the component has a parent. If it does, then we call AddAComponent for that parent. This is the recursive nature of this procedure. The procedure calls itself for each parent of the component passed in so that all components on the form (and their parents) are visited at least once. Recursion has often been likened to a stack of plates. Each plate you put on buries the other plates one step further down. To get back to the plates at the bottom of the stack, you need to pull off the plates above it. In the same way, a recursive procedure calls itself and buries itself down a few layers. Then, as each copy of the procedure exits, it unwinds itself back to the starting point of the stack.

After the procedure returns from its recursive call to add the components parent, it is time to add ourselves to the TreeView. At this point we can be certain that all of our parents back to the form level have already been added to the TreeView (as a result of the recursion). So, all we do is run the TreeViews list of nodes and look for our parent's name. Once we find it, we add ourselves as a child of that particular node.

There is so much more to the TreeView control than I can really cover in this article. In future articles, I will talk about some of the additional features of the TreeView component and show you some of its other uses. Until then, do look at the help system in Delphi on TreeViews, as there is quite a bit of good examples and documentation (make sure you have the latest help files from Borland's Web site though!)

Return to Tips & Tricks

Return to Front Page

Source for Align Text In Grid

```
unit IEAGrid;

interface

uses
 SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
 Forms, Dialogs, Grids;

type
  TNewInplaceEdit = class(TInplaceEdit)
  private
    { Private declarations }
    FAlignment: TAlignment;
  protected
    { Protected declarations }
    procedure CreateParams(var Params: TCreateParams); override;
    procedure SetAlignment(Value: TAlignment);
  public
    { Public declarations }
    constructor Create(AOwner:TComponent);
  published
    { Published declarations }
    property Alignment: TAlignment read FAlignment write SetAlignment
                default taLeftJustify;
  end;

  TAlignmentEvent = procedure(Sender: TObject; ACol, ARow: Longint;
                              AState: TGridDrawState;
                              var Alignment: TAlignment ) of object;

  TIEAlignStringGrid = class(TStringGrid)
  private
    { Private declarations }
    FDefaultCellAlignment : TAlignment;
    FCellAlignment : TAlignmentEvent;
    procedure SetDefaultCellAlignment(AValue: TAlignment);
  protected
    { Protected declarations }
    function CreateEditor: TInplaceEdit; override;
    function CanEditShow: Boolean; override;
    procedure DrawCell(ACol, ARow: Longint; ARect: TRect;
              AState: TGridDrawState); override;
  public
    { Public declarations }
  published
    { Published declarations }
    property DefaultCellAlignment: TAlignment read FDefaultCellAlignment
                      write SetDefaultCellAlignment default taLeftJustify;
    property OnGetCellAlignment: TAlignmentEvent read FCellAlignment
                        write FCellAlignment;
 end;

procedure Register;

implementation

procedure Register;
```

```
begin
  RegisterComponents('New Components', [TIEAlignStringGrid]);
end;

{TNewInplaceEdit procedures}

procedure TNewInplaceEdit.CreateParams(var Params: TCreateParams);
const
  Alignments : array[TAlignment] of Longint = (ES_LEFT,ES_RIGHT,ES_CENTER);
begin
  inherited CreateParams(Params);
  Params.Style := Params.Style or ES_MULTILINE or Alignments[FAlignment];
end;

procedure TNewInplaceEdit.SetAlignment(Value: TAlignment);
begin
  if FAlignment <> Value then
    begin
      FAlignment := Value;
      RecreateWnd;
      SetSel(0,-1);
    end;
end;

constructor TNewInplaceEdit.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);
  FAlignment := taLeftJustify;
end;

{TIEAlignStringGrid procedures}

procedure TIEAlignStringGrid.SetDefaultCellAlignment(AValue: TAlignment);
begin
  if AValue <> FDefaultCellAlignment then
    begin
      FDefaultCellAlignment := AValue;
      Update;
    end;
end;

function TIEAlignStringGrid.CreateEditor: TInplaceEdit;
var
  InplaceAlignment: TAlignment;
begin
   Result := TNewInplaceEdit.Create(Self);
end;

function TIEAlignStringGrid.CanEditShow: Boolean;
var
  InplaceAlignment: TAlignment;
begin
  Result := inherited CanEditShow;
  if (InplaceEditor <> nil) and Result then
    begin
      InplaceAlignment := FDefaultCellAlignment;
      if Assigned(FCellAlignment) then
        FCellAlignment(Self,Col,Row,[gdSelected,gdFocused],InplaceAlignment);
     (InplaceEditor as TNewInplaceEdit).Alignment := InplaceAlignment;
    end;
end;
```

```
procedure TIEAlignStringGrid.DrawCell(ACol, ARow: Longint; ARect: TRect;
     AState: TGridDrawState);
var
  Alignment: TAlignment;
  TheText: array[0..255] of char;
  TheRect: TRect;
begin
  if DefaultDrawing = TRUE then
    begin
      { Select background color }
      if AState = [gdFixed] then
        begin
         Canvas.Brush.Color := FixedColor;
        end
      else
        begin
          if (AState = [gdSelected])
            or ((AState = [gdSelected, gdFocused]) and
               (goDrawFocusSelected in Options)) then
            Canvas.Brush.Color := clActiveCaption
          else
            Canvas.Brush.Color := Color;
       end;
      Canvas.Font := Font;
      Canvas.FillRect(ARect);
      StrPCopy(TheText, Cells[ACol, ARow]);
      TheRect := ARect;
      with ARect do
        begin
          Alignment := FDefaultCellAlignment;
          if Assigned(FCellAlignment) then
            FCellAlignment(Self, ACol, ARow, AState, Alignment);
          if Alignment = taRightJustify then
            InflateRect(TheRect, -3, -2)
          else
            InflateRect(TheRect, -2, -2);
          case Alignment of
            taLeftJustify:
                 DrawText(Canvas.Handle,TheText,-1,TheRect, DT_LEFT);
            taRightJustify:
                 DrawText(Canvas.Handle,TheText,-1,TheRect, DT_RIGHT);
            taCenter   :
                 DrawText(Canvas.Handle,TheText,-1,TheRect, DT_CENTER);
          end;
        end;
    end
  else
    inherited DrawCell(ACol, ARow, ARect, AState);
end;

end.
```

Return to Article

Return to Front Page

A Look at MagiKit

MagiKit, by A.I.T Films Ltd., is a wonderful set of graphic components that you can add to your Delphi VCL. MagiKit is compatible with versions 1.0 and 2.0 of Delphi, and includes 4 base components for use in your Delphi applications: Magic Buttons, Magic Labels, and Magic Panels, and MagicPalette. Don't think this is just some run of the mill set of components that gives you a few new features. No, this is a very full featured set, and each component has a dizzying array of properties you can set to modify its appearance. MagicButtons, MagicLabels and MagicPanels all have a wonderful component editor wizard that walks you through all the available options with each.

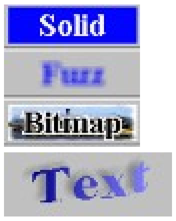
Lets take a look at each of the principle components. Drop a TMagicButton on your form call up the wizard and in just a few seconds you can make buttons like this:

Note: *Please keep in mind that when I created these and when they run in a Delphi application, they are very nicely colored and shaded. In the process of capturing these graphics, and depending on the restrictions of WinHelp or HTML color capabilities, it is likely that what you see here is probably pretty ugly and patchy. In fact it will likely be an ugly 16 colors instead of a beautiful 256. Please use your imagination a bit as they really are gorgeous. The neat thing is that these were all done on a 256 color depth setting in Windows! Very impressive!*

Anyway, back to buttons. The buttons you can create, have all sorts of interesting behaviors. You can define how they will look when up, down, disabled, and even when the mouse fly's over it. This latter option gives a neat effect of having the button reach out to you a bit when the mouse goes over it. Adds a little pizzazz to those humdrum Delphi apps.

Buttons can be rectangular, round, or any kind of polygon shape you can image. All can have shading, highlights, bitmaps on the backgrounds… whatever you want. Honestly speaking I think there are easily over 100 adjustable properties for just MagicButtons alone. Accessing them through the Object Inspector can be a bit annoying, but the wizard for each component really makes it a breeze.

MagicText is equally powerful, but allows you to have incredible control over shading, highlighting, borders, outlines… you name it. You can have curved text, warped text, even text where each letter is at a different angle, or in a different type face. Whew!! Take a look at some of the effects you can achieve:



Anyway, you get the idea… MagiKit has some of the most customizable components I have yet seen in a Delphi product. Try it… you won't be disappointed!
For more information on MagiKit, please contact A.I.T. Films Ltd, at ait@netvision.net.il. You can also obtain a free demo of MagiKit at http://www.ait.co.il.

Return to Front Page

Source for ImageMask

```
unit MaskImg;

// This is a masking image component. It's purpose is to provide an image component
// that allows accurate transparency with the use of a mask. This code is based
// primarily on the TImage component included with the Delphi VCL. From that, I
// added a Mask property (with its access methods) and modified a portion of the
// Paint routine. I initially tried to just descend from a TImage and add the Mask
// property there, but ran into all sorts of unacceptable behaviors. As a result
// the component is descended from TGraphicControl the same as TImage.
//
// These code modifications were created by Robert Vivrette for the Unofficial
// Newsletter of Delphi Users on January 6th, 1997.

interface

uses
  Windows, Classes, Graphics, Controls, Consts;

type
  TMaskedImage = class(TGraphicControl)
  private
    FPicture: TPicture;
    FAutoSize: Boolean;
    FStretch: Boolean;
    FCenter: Boolean;
    FReserved: Byte;
    FMask: TPicture;                                  // Added
    function GetCanvas: TCanvas;
    procedure PictureChanged(Sender: TObject);
    procedure SetAutoSize(Value: Boolean);
    procedure SetCenter(Value: Boolean);
    procedure SetPicture(Value: TPicture);
    procedure SetStretch(Value: Boolean);
    procedure SetMask(Value: TPicture);               // Added
  protected
    function GetPalette: HPALETTE; override;
    procedure Paint; override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    property Canvas: TCanvas read GetCanvas;
  published
    property Align;
    property AutoSize: Boolean read FAutoSize write SetAutoSize default False;
    property Center: Boolean read FCenter write SetCenter default False;
    property DragCursor;
    property DragMode;
    property Enabled;
    property Mask: TPicture read FMask write SetMask; // Added
    property ParentShowHint;
    property Picture: TPicture read FPicture write SetPicture;
    property PopupMenu;
    property ShowHint;
    property Stretch: Boolean read FStretch write SetStretch default False;
    property Visible;
    property OnClick;
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnStartDrag;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Additional', [TMaskedImage]);
end;

constructor TMaskedImage.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ControlStyle := ControlStyle + [csReplicatable] - [csOpaque];   // Changed
  FPicture := TPicture.Create;
  FPicture.OnChange := PictureChanged;
  FMask := TPicture.Create;                                 // Added
  Height := 105;
  Width := 105;
end;

destructor TMaskedImage.Destroy;
begin
  FMask.Free;                                               // Added
  FPicture.Free;
  inherited Destroy;
end;

function TMaskedImage.GetPalette: HPALETTE;
begin
  Result := 0;
  if FPicture.Graphic is TBitmap then
    Result := TBitmap(FPicture.Graphic).Palette;
end;

procedure TMaskedImage.Paint;
var
  Dest: TRect;
begin
  if csDesigning in ComponentState then
    with inherited Canvas do
    begin
      Pen.Style := psDash;
      Brush.Style := bsClear;
      Rectangle(0, 0, Width, Height);
    end;
  if Stretch then
    Dest := ClientRect
  else if Center then
    Dest := Bounds((Width - Picture.Width) div 2, (Height - Picture.Height) div 2,
      Picture.Width, Picture.Height)
  else
    Dest := Rect(0, 0, Picture.Width, Picture.Height);
  // Modified portion of Paint method to handle masking
  with inherited Canvas do
    begin
      // Set CopyMode to srcAnd. While painting the black and white mask, this will
      // have the effect of punching out a section of the form's canvas. The area
      // of the mask that is black will cause the form to become black and the area
      // of the mask that is white will leave the form alone. With srcAnd, black
      // added to any color becomes black. White added to any color remains that
color.
      CopyMode := srcAnd;
      StretchDraw(Dest, Mask.Graphic);
      // Set CopyMode to srcPaint. While painting the final image, this has the
      // effect of mathematically combining the colors on the form with the colors
      // in the image. Since we have punched out a section of black on the form,
      // the colors of the image that are in that area are simply copied to the form.
      // But the other areas of the form are left untouched because the image has
      // black in those areas. With srcPaint, black plus any color is equal to that
      // color. White plus any color is equal to white.
      CopyMode := srcPaint;
      StretchDraw(Dest, Picture.Graphic);
    end;
end;

function TMaskedImage.GetCanvas: TCanvas;
var
  Bitmap: TBitmap;
begin
  if Picture.Graphic = nil then
  begin
    Bitmap := TBitmap.Create;
    try
      Bitmap.Width := Width;
      Bitmap.Height := Height;
      Picture.Graphic := Bitmap;
    finally
      Bitmap.Free;
    end;
  end;
  if Picture.Graphic is TBitmap then
    Result := TBitmap(Picture.Graphic).Canvas
  else
    raise EInvalidOperation.CreateRes(SImageCanvasNeedsBitmap);
end;

procedure TMaskedImage.SetAutoSize(Value: Boolean);
begin
  FAutoSize := Value;
  PictureChanged(Self);
end;

procedure TMaskedImage.SetCenter(Value: Boolean);
begin
  if FCenter <> Value then
  begin
    FCenter := Value;
    PictureChanged(Self);
  end;
end;

procedure TMaskedImage.SetPicture(Value: TPicture);
```

```
begin
  FPicture.Assign(Value);
end;

procedure TMaskedImage.SetMask(Value: TPicture);      // Added
begin                                                 // Added
  FMask.Assign(Value);                                // Added
end;

procedure TMaskedImage.SetStretch(Value: Boolean);
begin
  if Value <> FStretch then
  begin
    FStretch := Value;
    PictureChanged(Self);
  end;
end;

procedure TMaskedImage.PictureChanged(Sender: TObject);
begin
  if AutoSize and (Picture.Width > 0) and (Picture.Height > 0) then
    SetBounds(Left, Top, Picture.Width, Picture.Height);
  Invalidate;
end;

end.
```

Return to Article

Return to Component Cookbook

Return to Front Page