While working through some email a few days back, I read a very nice letter from gentleman in England who was relating all the various things he and his company was doing with Delphi. It was quite encouraging to hear about the extent of Delphi's influence around the globe as well as the unique and inovative things that are being done with it. There are many companies and individuals out there that are not convinced that Delphi is sufficiently powerful to produce real applications. As all of you know, this is complete hogwash. It then dawned on me then, that this newsletter would be an excellent forum for sharing some of these experiences.

As a result, I will prepare a special section in the next newsletter that will do just that. I invite all of you out there to write me a short email (to the above CompuServe address) detailing any unique or innovative program or component you have developed using Delphi. I will go through them all and select a number of them to include in the January issue of UNDU. If this goes over well, I may make it a permanent part of future issues. A few guidelines however:

1. Please keep your email "concise". Notice I did not say "short". Use enough words to describe the things you are working on (or have already worked on) but don't run on about unrelated matters. Keep to the point.
2. Some companies may be sensitive of telling the world about their programming projects or plans. Please consider this when you send something in, because I would rather not be snagged by the FBI or CIA for revealing national security leaks (by the way... does anyone know if Delphi has made much of a show in the Federal Government?).
3. Indicate whether or not you mind having your email address included for reader inquiries. Whenever I mention things that I am doing in the newsletter, I always get a few requests for additional information. Including your email address would help readers know where to turn for additional information.

I look forward to seeing what everyone can put together!

- Robert

Unit File: CHECK1.PAS

```
unit Check1;

interface

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

type
  TForm1 = class(TForm)
    CheckBox1: TCheckBox;
    CheckBox2: TCheckBox;
    CheckBox3: TCheckBox;
    CheckBox4: TCheckBox;
    CheckBox5: TCheckBox;
    CheckBox6: TCheckBox;
    CheckBox7: TCheckBox;
    CheckBox8: TCheckBox;
          Label1: TLabel;
    CheckBox9: TCheckBox;
    CheckBox10: TCheckBox;
    CheckBox11: TCheckBox;
    CheckBox12: TCheckBox;
    CheckBox13: TCheckBox;
    CheckBox14: TCheckBox;
    CheckBox15: TCheckBox;
          CheckBox16: TCheckBox;
    Label2: TLabel;
          Panel1: TPanel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
          Edit4: TEdit;
    Label3: TLabel;
    Label4: TLabel;
    Edit5: TEdit;
    Edit6: TEdit;
    Label5: TLabel;
    Label6: TLabel;
    Label8: TLabel;
    Label9: TLabel;
    Label10: TLabel;
    Label11: TLabel;
          procedure FormActivate(Sender: TObject);
          procedure UpdateChecks(Sender: TObject);
          procedure CheckBox9Click(Sender: TObject);
          procedure UpdateEdits(Sender: TObject);
   private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
   TheNumber : Byte;
   TheEdit : Byte;
implementation

{$R *.DFM}
```

```
Const
   Zero = 1;
   One = 2;
   Two = 4;
   Three = 8;
   Four = 16;
   Five = 32;
   Six = 64;
   Seven = 128;


procedure UpdateMimics;
begin
          {Sorts the Contents of the Field to the corrent Checkboxes}
          {Typically, this code would be attached to a TDataSource OnDataChange
Event}
          with Form1 do begin
                       Checkbox9.Checked :=  Boolean(TheNumber and Zero);
                       Checkbox10.Checked := Boolean(TheNumber and One);
                       Checkbox11.Checked := Boolean(TheNumber and Two);
                       Checkbox12.Checked := Boolean(TheNumber and Three);
                       Checkbox13.Checked := Boolean(TheNumber and Four);
                       Checkbox14.Checked := Boolean(TheNumber and Five);
                       Checkbox15.Checked := Boolean(TheNumber and Six);
                       Checkbox16.Checked := Boolean(TheNumber and Seven);
          end;
end;

procedure UpdateEditMimics;
begin
   with Form1 do begin
          {Sorts the Contents of the Field to the corrent EditField}
          {Typically, this code would be attached to a TDataSource OnDataChange
Event}
          Edit4.Text := IntToStr(TheEdit and 3);
          Edit5.Text := IntToStr(TheEdit shr 2 and 3);
          Edit6.Text := IntToStr(TheEdit shr 4 and 3);
   end;
end;

procedure TForm1.FormActivate(Sender: TObject);
begin
   TheNumber := 0;
   TheEdit := 0;
end;

procedure TForm1.UpdateChecks(Sender: TObject);
begin
   with Sender as TCheckbox do begin
          if Checked = True then
                 TheNumber := TheNumber or Tag
          else
                 TheNumber := TheNumber and not Tag;

          Label1.Caption := IntToStr(TheNumber);
          UpdateMimics;
   end;
end;

procedure TForm1.CheckBox9Click(Sender: TObject);
begin
   CheckBox1.SetFocus;
```

```
end;

procedure TForm1.UpdateEdits(Sender: TObject);
var
TheValue : Byte;

begin
    with Sender as TEdit do begin
           if Text = '' then Text := '0';
           if StrToInt(Text) > 3 then Text := '0';

           TheValue := StrToInt(Text);
           TheEdit:= TheEdit and not ((TheEdit shr Tag and 3) shl Tag) or TheValue
shl Tag;
           Label4.Caption := IntToStr(TheEdit);
           UpdateEditMimics;
    end;

end;

end.
```

Tips & Tricks

Return to Front Page

DBImage's Cruelest Cut of All

By Gene Fowler - Internet: acorios@cello.gina.calstate.edu

As you know, Database Desktop doesn't allow editing of the BLOB Memo or BLOB Graphic fields. I made a pair of "hand held calculators" to use as outriggers for DBD: MemoEdit and PictEdit. In PictEdit I use a DBImage to access the BLOB Graphic field.

All the direct editing you do is with the three Clipboard functions, Copy, Cut, and Paste. I have the setup buttons above the image, and below it I have [View Clipbrd] and [Full Image Editor]. (The last ought to be Editors because I have an InputQueryEx box, with a ComboBox in place of the Edit, for BMP paint program and JPG/GIF converter, etc.).

Anyway, I found that the Cut produced a haywire result. The field and the Clipboard were both cleared, but no copy was stored on the Clipboard before the field was cleared.

Well, my tool was a "quick and dirty" work-bench tool. I looked at the source code and it looked like it ought to work (though I have little database experience). So, I made my editing panel explain the need for care about the Clipboard since it got bombed on a Cut.

I looked at all the bug lists and faq archives around and nobody seemed to be bothered by, or even aware of, this. Then, I read the Txt for a Borland Patch ...which required a different date-time stamp than my copy had. It listed this bug as one repaired.

That didn't do me any good. But it gave me the confidence to assume something WAS wrong with that simple, seemingly clean routine.

```
procedure TDBImage.CutToClipboard;
begin
  if Picture.Graphic <> nil then
    if FDataLink.Edit then
      begin
        CopyToClipboard;
        Picture.Graphic := nil;
     end;
end;
```

The "if FDataLink.Edit then" was the ONLY removeable, piece. So, I pulled it out. A not-very-delicate strategy, but... The CopyToClipboard worked fine, and the image was deleted. The bug seemed swatted.

Then, I shifted records, to have the delete take, and shifted back. Like the proverbial cat, ...my image was back as though never gone. Ahhhh, the FDataLink.Edit was a function and, obviously now, initiated a mode change so the deletion could take. It had to go back in. The trick was to place it after the Copy and before the deletion.

```
procedure TDBImage.CutToClipboard;
begin
  if Picture.Graphic <> nil then
    begin
      CopyToClipboard;
      if FDatalink.edit then
        Picture.Graphic := nil;
    end;
end;
```

That did the trick! Please email me at the above address if you have questions...

Return to Tips & Tricks

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #10 - December 12th, 1995

Playing Wave File Resources From Delphi

by Dr. Adrian Bottoms - CIS: 100435,2330

I have recently been writing a Delphi program designed to help students learn the Japanese Hiragana and Katakana representations of the phonemes. The program initially runs in a training mode by displaying a bitmap of the kana and saying the kana. It then enters a testing mode where it displays the bitmap and then, after a delay, says the kana. The student then presses buttons to claim they got it right or confess that they got it wrong. I plan on releasing this as shareware and would appreciate any feedback on your interest. I have several ideas for the registered version that add to the basic program. Such as animated bitmaps that show the correct stroke order for each kana; modules that teach the numbers and times and dates. Support for other languages would be possible too, for example Thai, Cyrillics, Morse Code (?) and so on.

The development versions of my program used bitmaps and wave from files residing on the hard disc. Since there are more than 100 bitmaps for each of hiragana and katakana and half that number of wave files there is a large number of files to manage. This gives problems with DOS filing systems. Since most of these files are small they waste a great deal of space by occupying large clusters. This is paticularly true for those who are fortunate to have large discs. To get around this, and to make it easier to do configuration management and manage installation I wanted to be able load the BMP and WAV files into the executable file's resources and display and play them from there. This way I need only ship a single executable file.

I created a BITMAPS.RC file using a text editor. An extract of it looks like:

```
AH      BITMAP    BITMAPS\HIRAGANA\AH.BMP
AK      BITMAP    BITMAPS\KATAKANA\AK.BMP
BAH     BITMAP    BITMAPS\HIRAGANA\BAH.BMP
BAK     BITMAP    BITMAPS\KATAKANA\BAK.BMP
BEH     BITMAP    BITMAPS\HIRAGANA\BEH.BMP
BEK     BITMAP    BITMAPS\KATAKANA\BEK.BMP
BIH     BITMAP    BITMAPS\HIRAGANA\BIH.BMP
BIK     BITMAP    BITMAPS\KATAKANA\BIK.BMP
BOH     BITMAP    BITMAPS\HIRAGANA\BOH.BMP
```

I also created a resource file for the wave files in the same way. An extract of that looks like:

```
A         WAVE    WAVES\A.WAV
BA        WAVE    WAVES\BA.WAV
BE        WAVE    WAVES\BE.WAV
BI        WAVE    WAVES\BI.WAV
BO        WAVE    WAVES\BO.WAV
```

These were compiled into Windows resource files using the Borland Resource Compiler (BRC) in \ DELPHI\BIN. The resources files were included in the project after the resource file that Delphi manages itself:

```
{$R *.DFM}
{$R BITMAPS.RES}                          { Include bitmap resource file     }
{$R WAVES.RES}                            { Include sound resource file    }
```

As an aside it is important not to mess with Delphi's resource file since any changes you make will be trashed the next time Delphi writes the resource file. By the way the "*" in *.DFM means the name of this unit and not all .DFM files as would be the case in a DOS command line for example.

Including these resource files in my KANA.EXE obviously increased the size of the executable but it did reduce the disc space used by lots of Kb.

Drawing the bitmaps from a resource was straightforward. I was using a TImage component for the

bitmaps from files. The following code was used to load and display them from a resource.

```
PROCEDURE TMainForm.ShowKana;
    {   Show the bitmap image of the kana that's been selected.
    {   This is done by loading the image from the resources
    {   built into the executable file.
VAR
    KanaName    : Array [0..3] OF Char;     { Ctring type string }
BEGIN
    StrPCopy (KanaName, StrTrim(KanaList[KNum]) + KanaSet);
    KanaBmp.Handle := LoadBitmap(HInstance, KanaName);
    KanaImage.Picture.Graphic := KanaBmp;
    DeleteObject(KanaBmp.Handle);
    KanaImage.Visible := TRUE;
END;    { ShowKana }
```

KanaBmp is a TBitMap which is created in the FormCreate method and destroyed when the form is destroyed.

Playing the sounds from a resource took a little more effort. To start with I was using a TMediaPlayer control to play the .WAV files but could not see any way of getting it to play a wave file from a resource. I managed to overcome this by making direct calls to the Windows Multimedia API. A side benefit of this was that I could remove the TMediaPlayer component and save myself a lot of space. I had also had a problem with the TMediaPlayer. One of the testers for this program did not have a sound card installed. They were using the Microsoft PC Loudspeaker driver which seemed to work without any problems. However my program remained mute on this system. Using the method below overcame this problem.

There is some set up code that is executed in the FormCreate method that initialises some pointers and loads MMSYSTEM.DLL. It goes:

```
UNIT Main;

{$N+}

INTERFACE
    .
    .
    .
CONST
    { MMSYSTEM.H definitions converted to Pascal                              }
    SND_SYNC        = $0000;
    SND_ASYNC       = $0001;
    SND_NODEFAULT   = $0002;
    SND_MEMORY      = $0004;
    .
    .
    .
VAR
    .
    .
    .
    DLLHandle       : THandle;
    PlaySound       : Function (lpszSoundName: PChar; uFlags: Word) : Bool;
    .
    .
    .
PROCEDURE TMainForm.FormCreate(Sender: TObject);
    {   This procedure is invoked when the main form is created.
    {   It initialises everything ready for the student to get started.
BEGIN
```

```
    DLLHandle := LoadLibrary('MMSYSTEM.DLL');
    IF DLLhandle < 32 THEN BEGIN
        MessageDlg('Failed to load MMSYSTEM.DLL', mtError, [mbOK] , 0);
        Close;                              { Blow us away }
        END
    ELSE
        @PlaySound := GetProcAddress(DLLHandle, 'sndPlaySound');
    .
    .
    .
END:      { Of Procedure FormCreate }
```

The procedure which plays the wave file resource is then:

```
PROCEDURE TMainForm.Say (Name : String) ;
    {   Play the "Name" wave resource.
VAR
    Asciz       : Array [0..255] OF Char;
    lpRes       : PChar;
    hRes        : THandle;
    hResInfo    : THandle;
BEGIN
  { Find the WAVE resource and play it }
  StrPCopy(Asciz, StrTrim(Name));          { Convert to Ctring }
  hResInfo := FindResource(HInstance, Asciz, 'WAVE');
  IF hResInfo <> 0 THEN BEGIN
    hRes := LoadResource(HInstance, hResInfo);
    IF hRes <> 0 THEN BEGIN
      lpRes := LockResource(hRes);
      PlaySound(lpRes, SND_SYNC OR SND_MEMORY);
      UnlockResource(hRes);
      FreeResource(hRes);
    END;    { IF hRes }
  END;    { IF hResInfo }
END;    { Of Procedure Say }
```

Notice that the sound is played synchronously (SND_SYNC) so the program waits for the sound playing to complete beore continuing. I have not found (yet) a way of playing the sound asynchronously and have the program resynch with the end of playing. It doesn't look very safe to Unlock and Free the resource while it is still playing!

There is some final clean up code that unloads the MMSYSTEM.DLL:

```
PROCEDURE TMainForm.FormDestroy(Sender: TObject);
BEGIN
  IF DLLHandle >= 32 THEN FreeLibrary(DLLHandle);
END;
```

I should confess that I am a bit of a tiro when it comes to Windows programming. If any of you have any suggestions to make that improve the code I shall be glad to hear them.

Dr. Adrian Bottoms
XDT Computer Consultants
The Old Barn
College Farmhouse
North Road
Cromwell
Nottinghamshire NG23 6JE
UK

Return to Tips & Tricks
Return to Front Page

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.

Issue #4 - May 24, 1995

Issue #5 - June 26, 1995

Issue #6 - July 25, 1995

Issue #7 - August 31, 1995

Return to Front Page

Typically, each issue of the newsletter is posted to three locations. The first is the *Borland Delphi* forum on CompuServe (**GO DELPHI**) in the "Delphi IDE" file section. If you want the issue as soon as it comes out, then this is the place to look. I also put the issue in the *Informant Communications* forum (**GO ICGFORUM**) in the "Delphi Demo/Share" file section at the same time. Lastly, I take the original source material of the issue and package it up and send it off to a gentleman named *Aaron Richardson* who maintains the Delphi Source web site (**http://www.doit.com/delphi/home.html**). He takes these files and converts them to web pages on the site and also posts the Windows *.HLP files on the sites FTP server. If you have questions about UNDU in general, you can contact me at **76416,1373** on CompuServe. If you have questions about his Web version of UNDU, you can contact Aaron at **aaron@doit.com**.

Form File: CHECK1.DFM

```
object Form1: TForm1
  Left = 198
  Top = 101
  Width = 234
  Height = 302
  Caption = 'BitField Demo'
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'System'
  Font.Style = []
  PixelsPerInch = 96
  OnActivate = FormActivate
  TextHeight = 16
  object Label1: TLabel
    Left = 136
    Top = 8
    Width = 49
    Height = 13
    Font.Color = clBlack
    Font.Height = -11
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    ParentFont = False
  end
  object Label2: TLabel
    Left = 32
    Top = 8
    Width = 94
    Height = 13
    Caption = 'CheckBox Value'
    Font.Color = clBlack
    Font.Height = -11
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    ParentFont = False
  end
  object Label3: TLabel
    Left = 24
    Top = 248
    Width = 65
    Height = 16
    Caption = 'Edit Value'
  end
  object Label4: TLabel
    Left = 120
    Top = 248
    Width = 4
    Height = 16
  end
  object Panel1: TPanel
    Left = 16
    Top = 24
    Width = 193
    Height = 220
    BevelOuter = bvLowered
    Caption = 'Panel1'
    TabOrder = 16
    object Label5: TLabel
      Left = 64
      Top = 148
```

```
  Width = 24
  Height = 13
  Caption = 'Edit1'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
end
object Label6: TLabel
  Left = 64
  Top = 172
  Width = 24
  Height = 13
  Caption = 'Edit2'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
end
object Label8: TLabel
  Left = 64
  Top = 196
  Width = 24
  Height = 13
  Caption = 'Edit3'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
end
object Label9: TLabel
  Left = 152
  Top = 150
  Width = 33
  Height = 13
  Caption = 'Mimic1'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
end
object Label10: TLabel
  Left = 152
  Top = 172
  Width = 33
  Height = 13
  Caption = 'Mimic2'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
end
object Label11: TLabel
  Left = 152
  Top = 194
  Width = 33
  Height = 13
  Caption = 'Mimic3'
```

```
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
end
object Edit1: TEdit
  Left = 16
  Top = 144
  Width = 41
  Height = 20
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 0
  Text = '0'
  OnExit = UpdateEdits
end
object Edit2: TEdit
  Tag = 2
  Left = 16
  Top = 168
  Width = 41
  Height = 20
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 1
  Text = '0'
  OnExit = UpdateEdits
end
object Edit3: TEdit
  Tag = 4
  Left = 16
  Top = 192
  Width = 41
  Height = 20
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 2
  Text = '0'
  OnExit = UpdateEdits
end
object Edit4: TEdit
  Left = 104
  Top = 144
  Width = 41
  Height = 20
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 3
  Text = '0'
  OnClick = CheckBox9Click
```

```
    OnEnter = CheckBox9Click
  end
  object Edit5: TEdit
    Left = 104
    Top = 168
    Width = 41
    Height = 20
    Font.Color = clBlack
    Font.Height = -11
    Font.Name = 'MS Sans Serif'
    Font.Style = []
    ParentFont = False
    TabOrder = 4
    Text = '0'
    OnClick = CheckBox9Click
    OnEnter = CheckBox9Click
  end
  object Edit6: TEdit
    Left = 104
    Top = 192
    Width = 41
    Height = 20
    Font.Color = clBlack
    Font.Height = -11
    Font.Name = 'MS Sans Serif'
    Font.Style = []
    ParentFont = False
    TabOrder = 5
    Text = '0'
    OnClick = CheckBox9Click
    OnEnter = CheckBox9Click
  end
end
object CheckBox1: TCheckBox
  Tag = 1
  Left = 32
  Top = 32
  Width = 97
  Height = 17
  Caption = 'CheckBox1'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 0
  OnClick = UpdateChecks
end
object CheckBox2: TCheckBox
  Tag = 2
  Left = 32
  Top = 48
  Width = 97
  Height = 17
  Caption = 'CheckBox2'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 1
  OnClick = UpdateChecks
end
```

```
object CheckBox3: TCheckBox
  Tag = 4
  Left = 32
  Top = 64
  Width = 97
  Height = 17
  Caption = 'CheckBox3'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 2
  OnClick = UpdateChecks
end
object CheckBox4: TCheckBox
  Tag = 8
  Left = 32
  Top = 80
  Width = 97
  Height = 17
  Caption = 'CheckBox4'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 3
  OnClick = UpdateChecks
end
object CheckBox5: TCheckBox
  Tag = 16
  Left = 32
  Top = 96
  Width = 97
  Height = 17
  Caption = 'CheckBox5'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 4
  OnClick = UpdateChecks
end
object CheckBox6: TCheckBox
  Tag = 32
  Left = 32
  Top = 112
  Width = 97
  Height = 17
  Caption = 'CheckBox6'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 5
  OnClick = UpdateChecks
end
object CheckBox7: TCheckBox
  Tag = 64
  Left = 32
```

```
  Top = 128
  Width = 97
  Height = 17
  Caption = 'CheckBox7'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 6
  OnClick = UpdateChecks
end
object CheckBox8: TCheckBox
  Tag = 128
  Left = 32
  Top = 144
  Width = 97
  Height = 17
  Caption = 'CheckBox8'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 7
  OnClick = UpdateChecks
end
object CheckBox9: TCheckBox
  Left = 120
  Top = 32
  Width = 81
  Height = 17
  TabStop = False
  Caption = 'MimicBox1'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 8
  OnClick = CheckBox9Click
  OnEnter = CheckBox9Click
end
object CheckBox10: TCheckBox
  Left = 120
  Top = 48
  Width = 81
  Height = 17
  TabStop = False
  Caption = 'MimicBox2'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 9
  OnClick = CheckBox9Click
  OnEnter = CheckBox9Click
end
object CheckBox11: TCheckBox
  Left = 120
  Top = 64
  Width = 81
```

```
  Height = 17
  TabStop = False
  Caption = 'MimicBox3'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 10
  OnClick = CheckBox9Click
  OnEnter = CheckBox9Click
end
object CheckBox12: TCheckBox
  Left = 120
  Top = 80
  Width = 81
  Height = 17
  TabStop = False
  Caption = 'MimicBox4'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 11
  OnClick = CheckBox9Click
  OnEnter = CheckBox9Click
end
object CheckBox13: TCheckBox
  Left = 120
  Top = 96
  Width = 81
  Height = 17
  TabStop = False
  Caption = 'MimicBox5'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 12
  OnClick = CheckBox9Click
  OnEnter = CheckBox9Click
end
object CheckBox14: TCheckBox
  Left = 120
  Top = 112
  Width = 81
  Height = 17
  TabStop = False
  Caption = 'MimicBox6'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ParentFont = False
  TabOrder = 13
  OnClick = CheckBox9Click
  OnEnter = CheckBox9Click
end
object CheckBox15: TCheckBox
  Left = 120
  Top = 128
```

```
    Width = 81
    Height = 17
    TabStop = False
    Caption = 'MimicBox7'
    Font.Color = clBlack
    Font.Height = -11
    Font.Name = 'MS Sans Serif'
    Font.Style = []
    ParentFont = False
    TabOrder = 14
    OnClick = CheckBox9Click
    OnEnter = CheckBox9Click
  end
  object CheckBox16: TCheckBox
    Left = 120
    Top = 144
    Width = 81
    Height = 17
    TabStop = False
    Caption = 'MimicBox8'
    Font.Color = clBlack
    Font.Height = -11
    Font.Name = 'MS Sans Serif'
    Font.Style = []
    ParentFont = False
    TabOrder = 15
    OnClick = CheckBox9Click
    OnEnter = CheckBox9Click
  end
end
```

Return to Article
Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #10 - December 12th, 1995

A Directory Stack Class

by Brad Atkinson - CIS: 76144,54

Here's a simple directory stack class I've found useful in an application I am working on. This app is picky about what the working directory is when performing its operations, and there are several levels of hierarchy to its directory structure. A DirStack internal to the wrapper class easily keeps things straight, without any participation by the using application, and without the wrapper class concerning itself with the current working directory when invoked.

```
unit DirStack;

interface

Uses Classes, SysUtils;

Type

TDirStack = Class(TStringList)
Public
   Procedure Push(AFullPath : String);
   Function Pop : String;
End;

implementation
{$I+} {I/O errors generate exceptions.}

Var
   PDirStr : PString;
   WorkStr : String;

Procedure TDirStack.Push(AFullPath : String);
Begin
   {Get the current directory.}
   GetDir(0, WorkStr);
   {Change to new directory.}
   ChDir(AFullPath);
   {Push the old dir.}
   Insert(0, WorkStr);
End;

Function TDirStack.Pop : String;
Begin
   Try
      {Get the last dir.}
      ChDir(Strings[0]);
   Finally
      Delete(0);
   End;
End;

end.
```

Return to Component Cookbook

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #10 - December 12th, 1995

Orpheus & Async Professional

Review by Robert Pullan - CIS: 102162,2711

Orpheus

Orpheus was the son of Apollo and the Muse of Music and was said to play the lyre so sweetly that he charmed beasts, trees and rivers. It is a fitting name for the product from TurboPower.

Orpheus is a compilation of VCL components that should be especially useful to those of use using Delphi to manage data. Its main focus is data entry and presentation. In all there are 34 components in the suite. They can be grouped into three basic categories: General, Table-related(think grid) and Data-aware.

TurboPower has been around and in the component-biz for a while, and it shows. Orpheus seems to be a descendant of their C/C++ product Data Entry Workshop. Make no doubt about it, while descended from C/C++, this product is 00% pure Delphi VCL. Without a doubt these and the Asynch Professional for Delphi components are the most complete products I have seen from any Delphi 3rd-party developer. The system design is excellent (they are primarily descendants from Delphi and a single Orpheus controls), documentation is complete and thorough, and they even include help files that can be integrated within Delphi's own help system !

If there are any 3rd party developers out there reading this, buy Orpheus, if for no better reason that to see how to package Delphi components the RIGHT way. I'm not talking about fancy packaging - this comes in a plain unprinted white box with a Dot-Matrix printed mailing label - but a well documented, well though out product.

Orpheus offers many great general components:

1. a tabbed notebook that can optionally destroy and create windows handles as the page looses or gains focus ... thereby becoming a very appealing alternative to the Borland TabbedNotebook component
2. It offers powerful masked edit components, calendars, spinners, a meter, and several other components.

The heart of the system is its masked edit components. You know those components that let you display 123456 as $123,456 or 1234.56, or "10/05/95" as "Oct 5, 1995". These systems work on simple tedit-type data, arrays, tables/grids and databases. The ability to mask input/output ranges from preformmatted simple case-based masking to date and time based formats to programmer-defined, and can make your application really shine. The controls roughly double the number of properties available to native Delphi Tedit/Tdbedit controls, including the ever popular "InputRequired" property.

The product also ships with several good instructional programs which demonstrate the use of these components ... and which are also well documented in the manual. These range from simple address books to very impressive order entry demos. The source code is also included with this product and is a lesson in classes and inheritance.

If you haven't guessed, I LIKE this product. If you are working with data, especially where you would like to limit the opportunities for input error, I am aware of no better product to work with.

Orpheus is available from Delphi Only Tools, Programmers Warehouse and has a suggested retail price of $199, let your fingers to the walking and you will find the street price to be much lower..

ASYNCH PROFESSIONAL for DELPHI

Asynch Professional for Delphi (APD) is another one of those programs from TurboPower. It too benefits mightily from the high standards of "packaging" found in Orpheus ... the dot-matrix label is how you can tell the packages apart.

APD is the first major Delphi component designed to expand design to products with serial

communications capabilities. It too seems to be a descendant of C/C++ products ... which is basically good. The program is 100% Delphi VCL code.

The user can design everything from simple phone dialers to VERY sophisticated comm packages... It is clear that several C/C++ components were used by major Comm program developers in their commercial release products.

The manual starts with a brief explaination of asynchronous communications and how this is handled under Windows. This is one of the best BASIC explainations of the process I have come accross. I like it when I buy a component that works well and get the added bonus that I learn something important.

This is a system designed to allow end users to primarily upload (send to host) and download (retrieve from host) files. Nothing fancy here. The TurboPower BBS has source for a scripting language, can't tell you whether it works or not, but it does seem promising that this is a product which will expand with time.

The communications system support all of the major communications protocols including: Xmodem, Ymodem, Zmodem, Kermit, Compuserve B+ and ASCII. APD comes bundled with a wide range of mainstream modems.

Like Orpheus, APD comes with several neat communications demo programs and source code, and are also documented in the manual.

APD is available from Delphi Only Tools, Programmers Warehouse and has a suggested retail price of $179, let your fingers to the walking and you will find the street price to be much lower..

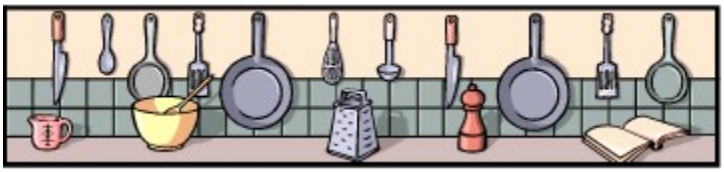
Object	Vendor	OOD Delivery		Delphi Specific			
		Style	Loads	Functionality	Code Examples Manual	Demos	PAS files Included
Orpheus	TurboPower	VCL	Yes	Substantial	5	5	yes
Asynch Pro for Delphi	TurboPower 719.260.6641	VCL DLL	Yes	Substantial	3	3	Yes

Return to Front Page

```
unit Unit1;

interface

uses WinTypes, WinProcs, SysUtils, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, ShellAPI;

type
  TForm1 = class(TForm)
    OpenDialog: TOpenDialog;
    ListBox1: TListBox;
    Panel1: TPanel;
    btnOpen: TButton;
    btnExit: TButton;
    btnClear: TButton;
    procedure FormCreate(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure AppMessage(var Msg: Tmsg; var Handled: Boolean);
    procedure btnExitClick(Sender: TObject);
    procedure btnOpenClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
  private
    { Private declarations }
    procedure LoadListBox(const fName: string);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.LoadListBox(const FName: string);
begin
  if not FileExists(FName) then
    MessageDlg(FName + #10 + 'is an invalid file.', mtError, [mbOK] , 0)
  else
    ListBox1.Items.Add(FName);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  DragAcceptFiles(Form1.Handle, true);
  DragAcceptFiles(Application.Handle, true);
  Application.OnMessage := AppMessage;
end;

procedure TForm1.FormActivate(Sender: TObject);
var
    x : integer;
begin
 Panel1.Caption := 'qty parms: ' + IntToStr(ParamCount);
 if ParamCount > 0 then
 begin
   for x := 1 to ParamCount do
    LoadListBox(ParamStr(x));
 end;
end;
```

```
procedure TForm1.AppMessage(var Msg: Tmsg; var Handled: Boolean);
const
    BufferLength : word = 255;
var
    DroppedFilename : string;
    FileIndex : word;
    QtyDroppedFiles : word;
    pDroppedFilename : array [0..255] of Char;
    DroppedFileLength : word;
begin
  if Msg.Message = WM_DROPFILES then
  begin
    FileIndex := $FFFF;
    QtyDroppedFiles := DragQueryFile(Msg.WParam, FileIndex,
                                     pDroppedFilename, BufferLength);
    Panel1.Caption := 'qty dropped: ' + IntToStr(QtyDroppedFiles);
    for FileIndex := 0 to (QtyDroppedFiles - 1) do
    begin
      DroppedFileLength := DragQueryFile(Msg.WParam, FileIndex,
                                         pDroppedFilename, BufferLength);
      DroppedFilename := StrPas(pDroppedFilename);
      LoadListBox(DroppedFilename);
    end;
    DragFinish(Msg.WParam);
    Handled := true;
  end;
end;

procedure TForm1.btnOpenClick(Sender: TObject);
var
    x : integer;
    FName : string;
begin
  if OpenDialog.Execute then
  begin
    Panel1.Caption := 'qty selected: ' + IntToStr(OpenDialog.Files.Count);
    for x := 1 to OpenDialog.Files.Count do
    begin
      FName := OpenDialog.Files.Strings[x - 1];
      OpenDialog.HistoryList.Add(FName);
      LoadListBox(FName);
    end;
  end
  else
    Panel1.Caption := 'none selected';
end;

procedure TForm1.btnExitClick(Sender: TObject);
begin
     close;
end;

procedure TForm1.btnClearClick(Sender: TObject);
begin
  Listbox1.Items.Clear;
  Panel1.Caption := '';
end;

end.
```

Return to Drag & Drop Article
Return to Front Page



The Component Cookbook

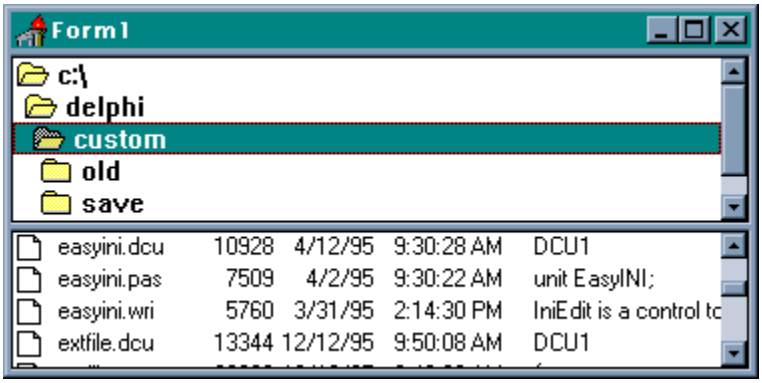
by Robert Vivrette

I have received a number of interesting component projects. The first is a brief little snippet of code that modifies the behavior of the TStatusBar component presented here a few issues back. The second is a component that implements a directory stack, so that directory names can be pushed to and popped off of the stack in sequence.

Extending the StatusBar Resource Gauge

Implementing a Directory Stack

Lastly, Mark Summerfield has sent in a interesting modification to the TFileListBox component. Rather than just showing the file names, it also allows you to see the size, date, time, and even comments extracted from the first line of the file. A very capable component!



Extending the FileListBox

Return to Front Page

```
{-------------------------------------------------------------------}
{ EXTFILE - Extended FileListbox: shows size, date & time of files  }
{ v. 1.00 July, 21 1995                                             }
{-------------------------------------------------------------------}
{ Copyright Enrico Lodolo                                           }
{ via F.Bolognese 27/3 - 440129 Bologna - Italy                     }
{ CIS 100275,1255 - Internet ldlc18k1@bo.nettuno.it                 }
{-------------------------------------------------------------------}

{*********************************************************************}
{ Extended to optionally show descriptive text where possible, and titles     }
{ embedded in rich text files.                                                }
{ Relaid out.                                                                 }
{ v 1.02 20/10/95                                                             }
{ Removed .doc file type since the code to extract titles/first lines was     }
{ bugged and necessary time/info for fix is not available.                    }
{ v 1.03 17/11/95                                                             }
{ $Header: Extended FileListBox extfile/001/003$                              }
{*********************************************************************}
{ Desc additions copyright © Mark Summerfield 1995                            }
{ email: msummerf@fdgroup.co.uk                                               }
{*********************************************************************}

unit ExtFile ;

interface

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

type
    TFile = file of Char ;

    TExtFileListBox = class( TFileListBox )
    private
        FShowSize : Boolean ;
        FShowDate : Boolean ;
        FShowTime : Boolean ;
        FShowDesc : Boolean ;
        FShowRTFDesc : Boolean ;
        FShowDescAlways : Boolean ;
        FSizePos : Integer ;
        FDatePos : Integer ;
        FTimePos : Integer ;
        FDescPos : Integer ;
        FNameWd : Integer ;
        FSizeWd : Integer ;
        FDateWd : Integer ;
        FTimeWd : Integer ;
        FDescWd : Integer ;
        procedure SetShowSize( Value : Boolean ) ;
        procedure SetShowDate( Value : Boolean ) ;
        procedure SetShowTime( Value : Boolean ) ;
        procedure SetShowDesc( Value : Boolean ) ;
        procedure SetShowRTFDesc( Value : Boolean ) ;
        procedure SetShowDescAlways( Value : Boolean ) ;
        procedure SetSizePos( Value : Integer ) ;
        procedure SetDatePos( Value : Integer ) ;
        procedure SetTimePos( Value : Integer ) ;
        procedure SetDescPos( Value : Integer ) ;
```

```
        function GetFileDesc( Filename : string ) : string ;
        function GetDescGeneric( var fp : TFile ; var p, q : Integer ; TitleByte :
Word ) : string ;
        function GetDescRtf( var fp : TFile ; var p, q : Integer ) : string ;
        function IsCharPrintable( c : Char ) : boolean ;
    protected
        procedure SetWidths ;
        procedure DrawItem( Index : Integer ; Rect : TRect ; State :
TOwnerDrawState ) ; override ;
    public
        constructor Create( AOwner : TComponent ) ; override ;
    published
        property ShowSize : Boolean read FShowSize write SetShowSize default True ;
        property ShowDate : Boolean read FShowDate write SetShowDate default True ;
        property ShowTime : Boolean read FShowTime write SetShowTime default True ;
        property ShowDesc : Boolean read FShowDesc write SetShowDesc default True ;
        property ShowRTFDesc : Boolean read FShowRTFDesc write SetShowRTFDesc
default False ;
        property ShowDescAlways : Boolean read FShowDescAlways write
SetShowDescAlways default False ;
        property SizePos : Integer read FSizePos write SetSizePos default -1 ;
        property DatePos : Integer read FDatePos write SetDatePos default -1 ;
        property TimePos : Integer read FTimePos write SetTimePos default -1 ;
        property DescPos : Integer read FDescPos write SetDescPos default -1 ;
    end ;

procedure Register ;

{******************************************************************************}

implementation

const
    DefMaxDescWd = 64 ;  {If you change this then change the "FDescWd :=
TextWidth("}
                         {line in SetWidths so that the number of 'x's is equal to
DefMaxDescWd.}
    KnownFiles = '.htm.txt.asc.bat.ini.hpj' ;
                        {These are generally plain text. If we don't have special}
                        {processing we'll just show the first line.}
    SpecialKnownFiles = '.rtf.wps.wri' ;
                        {Add to this string if you add specific processing for}
                        {other file types in GetFileDesc.}
                        {.rtf =>     Rich Text Format title.}
                        {.wps =>     Works2 first line.}
                        {.wri =>     Write first line.}

{******************************************************************************}

procedure TExtFileListBox.SetShowSize( Value : Boolean ) ;
begin
    if Value <> FShowSize then
        begin
            FShowSize := Value ;
            Update ;
        end ;
end ;

procedure TExtFileListBox.SetShowDate( Value : Boolean ) ;
begin
    if Value <> FShowDate then
        begin
            FShowDate := Value ;
```

```
            Update ;
        end ;
end ;

procedure TExtFileListBox.SetShowTime( Value : Boolean ) ;
begin
    if Value <> FShowTime then
        begin
            FShowTime := Value ;
            Update ;
        end ;
end ;

procedure TExtFileListBox.SetShowDesc( Value : Boolean ) ;
begin
    if Value <> FShowDesc then
        begin
            FShowDesc := Value ;
            Update ;
        end ;
end ;

procedure TExtFileListBox.SetShowRTFDesc( Value : Boolean ) ;
begin
    if Value <> FShowRTFDesc then
        begin
            FShowRTFDesc := Value ;
            Update ;
        end ;
end ;

procedure TExtFileListBox.SetShowDescAlways( Value : Boolean ) ;
begin
    if Value <> FShowDescAlways then
        begin
            FShowDescAlways := Value ;
            Update ;
        end ;
end ;

{*****************************************************************************}

procedure TExtFileListBox.SetSizePos( Value : Integer ) ;
begin
    if Value <> FSizePos then
        begin
            FSizePos := Value ;
            Update ;
        end ;
end ;

procedure TExtFileListBox.SetDatePos( Value : Integer ) ;
begin
    if Value <> FDatePos then
        begin
            FDatePos := Value ;
            Update ;
        end ;
end ;

procedure TExtFileListBox.SetTimePos( Value : Integer ) ;
begin
    if Value <> FTimePos then
```

```
        begin
            FTimePos := Value ;
            Update ;
        end ;
end ;

procedure TExtFileListBox.SetDescPos( Value : Integer ) ;
begin
    if Value <> FDescPos then
        begin
            FDescPos := Value ;
            Update ;
        end ;
end ;

{*********************************************************************}
{ Creates the component and sets the defaults }

constructor TExtFileListBox.Create( AOwner : TComponent ) ;
begin
    inherited Create( AOwner ) ;
    FShowSize := True ;
    FShowDate := True ;
    FShowTime := True ;
    FShowDesc := True ;
    FShowRTFDesc := False ;
    FShowDescAlways := False ;
    FSizePos := -1 ;
    FDatePos := -1 ;
    FTimePos := -1 ;
    FDescPos := -1 ;
end ;

{*********************************************************************}
{ Calculates the max. widths of name, size, date and time }

procedure TExtFileListBox.SetWidths ;
begin
    with Canvas do
        begin
            FNameWd := TextWidth( 'aaaaaaaa.mmm' ) ;
            FSizeWd := TextWidth( '888888888' ) ;
            FDateWd := TextWidth( '888/88/88' ) ;
            FTimeWd := TextWidth( '888.88.88' ) ;
            {If you change the following constant, change DefMaxDescWd to match.}
            FDescWd :=
TextWidth( 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx' ) ;
        end ;
end ;

{*********************************************************************}
{ Draws a line of the ListBox }

procedure TExtFileListBox.DrawItem( Index : Integer ; Rect : TRect ; State :
TOwnerDrawState ) ;
var
    Bitmap : TBitmap ;
    Offset : Integer ;
    OldAlign : Word ;
    SR : TSearchRec ;
    DT : TDateTime ;
```

```
    ThisFile : string ;
begin
    inherited DrawItem( Index, Rect, State ) ;
    with Canvas do
        begin
            SetWidths ;
            Offset := Rect.Left + FNameWd ;
            if ( FShowGlyphs = True ) and ( Bitmap <> nil ) then
                begin
                    Bitmap := TBitmap( Items.Objects[ Index ] ) ;
                    Offset := Offset + Bitmap.Width + 6 ;
                end ;
            { Retrieves Size, Date and Time of the current file }
            if Directory[ Length( Directory ) ] = '\' then
                ThisFile := Directory + Items[ Index ]
            else
                ThisFile := Directory + '\' + Items[ Index ] ;
            FindFirst( ThisFile, faAnyFile, SR ) ;
            DT := FileDateToDateTime( SR.Time ) ;
            { Right alignes the text }
            OldAlign := SetTextAlign( Handle, ta_right ) ;
            if FShowSize then
                begin
                    if FSizePos = -1 then
                        Offset := Offset + FSizeWd
                    else
                        Offset := FSizePos ;
                    TextOut( Offset, Rect.Top, IntToStr( SR.Size ) ) ;
                end ;
            if FShowDate then
                begin
                    if FDatePos = -1 then
                        Offset := Offset + FDateWd
                    else
                        Offset := FDatePos ;
                    try
                        TextOut( Offset, Rect.Top, DateToStr( DT ) ) ;
                    finally
                    end ;
                end ;
            if FShowTime then
                begin
                    if FTimePos = -1 then
                        Offset := Offset + FTimeWd
                    else
                        Offset := FTimePos ;
                    try
                        TextOut( Offset, Rect.Top, TimeToStr( DT ) ) ;
                    finally
                    end ;
                end ;
            if FShowDesc then
                begin
                    SetTextAlign( Handle, ta_left ) ;
                    if FDescPos = -1 then
                        Inc( Offset, 6 )
                    else
                        Offset := FDescPos ;
                    try
                        TextOut( Offset, Rect.Top, GetFileDesc( ThisFile ) ) ;
                    finally
                    end ;
                end ;
```

```
            { Sets the text alignment to the original value }
            SetTextAlign( Handle, OldAlign ) ;
        end ;
end ;

{*************************************************************************}
{For KnownFiles this function grabs the first 255 chars from the file and     }
{tries to extract a meaningful descriptive string from this, or at least      }
{extracting the first line. We only read 255 chars, because reading every file}
{in a directory takes time and we don't want the control to be so slow that   }
{its unusable.}
{However, in the case of SpecialKnownFiles, we actually search the file for}
{embedded titles. For rich text files this can be *very* slow, because the}
{title can appear quite a way into the file, it isn't at a fixed point as it}
{is with Word 6 and Write files. This is why we can optionally skip descs for}
{rtf files.}
{Basically what it does is extract the first non-blank line of text files,}
{and the <TITLE> string from HTM files, (providing they appear in the first}
{255 chars.}
{In all other cases, if ShowDescAlways is false a blank string is returned}
{with no time-consuming file reading; otherwise the first printable chars}
{are returned, if any.}

function TExtFileListBox.GetFileDesc( Filename : string ) : string ;
var
    fp : TFile ;
    s : string ;
    sx : string ;
    temp : string ;
    i, p, q : Integer ;
    ext : string ;
begin
    {Assume we can't find a descriptive string in the chunk we've grabbed.}
    result := '' ;
    p := 0 ;
    q := 0 ;
    if Filename = '' then
        Exit ;
    ext := LowerCase( ExtractFileExt( Filename ) ) ;
    if ( not ShowDescAlways ) and
       ( ( Pos( ext, KnownFiles + SpecialKnownFiles ) = 0 ) or
         ( ( ext = '.rtf' ) and ( not ShowRTFDesc ) ) ) then
        Exit ;
    try
        AssignFile( fp, Filename ) ;
        Reset( fp ) ;
        s := '' ;
        i := 1 ;
        if ( ext = '.rtf' ) and ShowRTFDesc then
            s := GetDescRtf( fp, p, q )
        else
        if ext = '.wps' then
          s := GetDescGeneric( fp, p, q, 256 )
        else
        if ext = '.wri' then
            s := GetDescGeneric( fp, p, q, 128 )
        else
            begin
                while ( not eof( fp ) ) and ( i < 255 ) do
                    begin
                        Read( fp, s[ i ] ) ;
                        Inc( s[ 0 ] ) ;
```

```
                    Inc( i ) ;
                end ;
            end ;
    finally
        CloseFile( fp ) ;
    end ;
    if ( Pos( ext, SpecialKnownFiles ) = 0 ) or
       ( ( ext = '.rtf' ) and ( not ShowRTFDesc ) ) then
        begin
            {For certain file types we can look for specific descriptive
strings...}
            sx := LowerCase( s ) ;
            if ext = '.htm' then
                begin
                    p := Pos( '<title>', sx ) ;
                    if p <> 0 then
                        begin
                            Inc( p, 7 ) ;
                            q := Pos( '</title', sx ) ;
                        end
                    else
                        begin
                            p := Pos( '<h', sx ) ;
                            if p <> 0 then
                                begin
                                    temp := Copy( sx, p, 4 ) ;
                                    Insert( '/', temp, 2 ) ;
                                    Inc( p, 4 ) ;
                                    q := Pos( temp, sx ) ;
                                end ;
                        end ;
                end ;
            {If you add processing for other file types, add their extensions to
KnownFiles.}
            {No explicit if ext = '.txt', or if ext = '.asc', since the following
default}
            {behaviour does what we want, specifically it just gets the first line
of text,}
            {skipping leading blank lines.}
            {Not a known type of file or no luck with the known types so do our
best anyway.}
            if p <= 0 then
                begin
                    p := 1 ;
                    while ( not IsCharPrintable( sx[ p ] ) and
                          ( p < 255 ) ) do
                        Inc( p ) ;
                    if p = 255 then
                        p := 1 ;
                end ;
            {We found the beginning but not the end.}
            if q <= 0 then
                begin
                    q := p + 1 ;
                    while ( IsCharPrintable( sx[ q ] ) and
                          ( q < 255 ) and
                          ( not ( sx[ q ] in [ #13, #10 ] ) ) ) do
                        Inc( q ) ;
                end ;
        end ;
    {Now populate result.}
    i := q - p ;
    if i > DefMaxDescWd then
```

```
        i := DefMaxDescWd ;
    result := Copy( s, p, i ) ;
end ;

{******************************************************************************}

function TExtFileListBox.IsCharPrintable( c : Char ) : boolean ;
begin
    if IsCharAlphaNumeric( c ) or
       ( c in [ ' ', '!', '@', '#', '$', '%', '^', '&', '*', '(', ')', '-', '_',
       '=', '+', '[', '{', ']', '}', '''', '"', ';', ':', '\', '|', '\', '|',
       ',', '<', '.', '>', '/', '?', '~', '`' ] ) then
        result := true
    else
        result := false ;
end ;


{******************************************************************************}

function TExtFileListBox.GetDescGeneric( var fp : TFile ; var p, q : Integer ;
TitleByte : Word ) : string ;
var
    c : Char ;
begin
    p := 0 ;
    q := 1 ;
    result := '' ;
    if FileSize( fp ) < ( TitleByte + 2 ) then
        begin
            while not eof( fp ) do
                begin
                    Read( fp, c ) ;
                    if IsCharPrintable( c ) then
                        begin
                            result[ q ] := c ;
                            result[ 0 ] := Char( q ) ;
                            Inc( q ) ;
                        end ;
                end ;
        end
    else
        begin
            Seek( fp, TitleByte ) ;
            p := 0 ;
            q := 1 ;
            while not eof( fp ) do
                begin
                    Read( fp, c ) ;
                    if IsCharPrintable( c ) then
                        begin
                            p := 1 ;
                            result[ q ] := c ;
                            Inc( q ) ;
                            result[ 0 ] := Char( q ) ;
                        end
                    else
                    if p <> 0 then
                        break ;
                end ;
            p := 1 ;
        end ;
```

```
end ;

{*****************************************************************************}

function TExtFileListBox.GetDescRtf( var fp : TFile ; var p, q : Integer ) : string
;
const
    Limit = 8192 ;
    {The Limit is how many bytes we are prepared to read in search of the title.}
    {Because an RTF file's title is not at a fixed position it can occur quite}
    {deep within a file. Clearly then, the further we are prepared to look, the}
    {more likely we are to find the title, and the slower the search will be.}
    {Remember also that not all RTF files have titles, in which case we search}
    {Limit bytes in vain.}
var
    State : Integer ;
    i : integer ;
    c : Char ;
begin
    p := 0 ;
    q := 11 ;
    i := 0 ;
    State := 0 ;
    result := '(untitled)' ;
    while ( not eof( fp ) ) and ( i < Limit ) and ( State <> 8 ) do
        begin
            Read( fp, c ) ;
            Inc( i ) ;
            case State of
                0 : if c = '\' then State := 1 ;
                1 : if c = 't' then State := 2 else State := 0 ;
                2 : if c = 'i' then State := 3 else State := 0 ;
                3 : if c = 't' then State := 4 else State := 0 ;
                4 : if c = 'l' then State := 5 else State := 0 ;
                5 : if c = 'e' then State := 6 else State := 0 ;
                6 : if c = ' ' then
                        begin
                            State := 7 ;
                            i := 0 ;
                            q := 1 ;
                            result := '' ;
                        end
                    else State := 0 ;
                7 : begin
                        if c = '}' then
                            State := 8
                        else
                            begin
                                result[ q ] := c ;
                                result[ 0 ] := Char( q ) ;
                                Inc( q ) ;
                            end ;
                    end ;
            end ;
        end ;
end ;

{*****************************************************************************}

procedure Register ;

begin
```

```
    RegisterComponents( 'System', [ TExtFileListBox ] ) ;
end ;

end.
```

Return to Component Cookbook
Return to Front Page

A Little Help With PChar's

by Robert Vivrette - CIS: 76416,1373

A few weeks back, a gentleman named Jean-fabien Connault (CIS:100745,233) wrote me a short letter indicating a strange behavior in Delphi. Here is the code he sent me:

```
var
  BufferWinDir: PChar;
  MaxBuf : Integer;
begin
  MaxBuf := 144;
  getMem(bufferWinDir,MaxBuf);
  BufferWinDir := '';
  MessageDlg('*** It''s before GetWindowsDirectory ***', mtInformation, [mbOK], 0);
  GetWindowsDirectory(BufferWinDir,MaxBuf);
  MessageDlg('*** It''s after GetWindowsDirectory ***', mtInformation, [mbOK], 0);
end;
```

At first glance, I could see nothing wrong with this. However, when the code executed, it did indeed have very strange behavior. The first MessageDlg call produced a normal dialog box with the appropriate text. However after executing the GetWindowsDirectory API call, the second MessageDlg call **totally** flipped out. It presented a dialog box that completely filled the screen with the OK button occupying about 95% of the screen real-estate. Clicking on the button made the whole screen appear to "click down" and then the program ended normally.

After digging through it with the debugger, I realized what the problem was...

The line:

```
BufferWinDir := '';
```

was not doing what he thought it was. His intention, of course, was to assign a null string to BufferWinDir. However, what he was doing here was pointing BufferWinDir to a null somewhere else in memory. Let me explain a bit:

When you do a GetMem, windows requests a block of memory and returns an address pointing to that memory. This address is stored in BufferWinDir. Let's presume the address returned is 4807:0F80. Therefore, BufferWinDir (a pointer to an array of characters) is now the 4-byte value 4807:0F80. When the compiler gets to the line where you are assigning a null, it says "Create a Null string in memory and put the address of that Null in BufferWinDir". Let's say the compiler chooses 3DDF:0040 as the location of the null (which it did in a test I did). By doing this assignment, BufferWinDir now points at this null character instead of the array of characters previously reserved. Then, when the GetWindowsDirectory procedure is called, Windows assigns the returned value to the bytes starting at where BufferWinDir is located. This includes the null character originally pointed to, plus a number of additional bytes that could be anything. As a result, the characters are writing over memory outside of the allocated space, destroying other variables or code. In this case, the characters overwrote some element of the DialogBox object, either Code or the Objects properties.

The correct way to assign a null string (or any other string for that matter) to a pchar is to use StrPCopy as follows:

```
StrPCopy(BufferWinDir,'');
```

This function takes the listed string of characters (in this case a null string) and copies them to the memory location pointed to by the first parameter. PChars can sometimes be a bit tricky to work with. I hope this helps a bit!

Return to Tips & Tricks

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #10 - December 12th, 1995

Note from the Editor:

Last issue, I ran this article by Steve Griffiths but accidentally forgot to include the source code to his demonstration program. I am therefore repeating the article in its entirety, with the appropriate links to the source below. I appologize for any confusion this caused! ***- Robert***

Using Integer Fields to Store Multiple Data Elements in Tables

by Steve Griffiths - CIS: 102523,27

I spend a lot of my time writing state reporting database applications, for example EMS, Fire and Police incident reports. In the past I was limited to using Foxpro (sorry!) or Paradox. Although I could get the job done in a reasonable time, I was never happy with the appearance or speed of the finished product. In addition, adding a 3.5 Meg runtime to a FoxPro app tends to make installation a real pain!.

Delphi has allowed me to become a 'real' programmer again - it is easier to code a Paradox app in Delphi than it is using Paradox directly, and the resources and components available provide an excellent user interface. (I use and modify the InfoPower Library a lot.)

One thing that all the state reports have in common is an enormous amount of data - literally hundrds of fields. In an early iteration of one of the forms, I ended up with three linked master tables - 600 odd fields before even thinking about the detail tables. Obviously some rethinking was necessary!

The bulk of the fields was taken up with either checkboxes (Yes/No) and limited answers (1 of 10, Yes / No Maybe). By using Delphi's bit manipulation operators in conjuction with Paradox long (32 bit) Field types, I am able to store information for 31 Checkboxes, 15 one of four types, or 7 one of fifteen types in a single field. (The last bit of the field is not used as Paradox integers are signed). Using this technique has allowed me to fit my master data into a single table.

The Demo

On the left hand side of the form are 8 checkboxes and 3 edit fields. On the right hand side are corresponding objects that mimic the left hand objects - checking a checkbox on the left will check it's mate on the right. The edit fields can contain 0 - 3, and again, the partner will update to the same value. The data for all the checkboxes is contained in an 8 bit integer variable (TheNumber), the value of which is shown on the top of the form. Similarily, the data for the Edit Fields is contained in another 8 bit integer (TheEdit), whose value is shown on the bottom.

Checking a checkbox calls the UpDateChecks function, which updates TheNumber. The function in turn calls the UpdateMimics function which uses TheNumber to determine the status of the mimic checkboxes.

When an EditField is changed the UpdateEdits function is called which updates the value of TheEdit. UpdateEdits then calls UpdateEditMimics which uses TheEdit to update the mimic EditFields.

When using the code with a table, the code for the UpdateChecks and UpdateEdits functions would be placed in the Tables BeforePost Event, and the UpdateMimics / UpdateEditMimics Code would go in the DataSources OnDataChange Event. (In a non-demo situation, the update procedures would refer to the original objects).

Now, let's see how it works...

Checkboxes

Binary Revisited... A checkbox has a true or false value. This is a logical value that can be treated as 1 bit. Therefore, as long as we can get at the individual bits of the whole, an 8 bit (unsigned) number can contain the status of 8 checkboxes.

Unfortunately, Delphi does not appear to allow direct binary representation (if I am missing something tell me!) so we must use numbers to represent the bit position. Here is the bit representation of an 8 bit number..

```
 128      64      32      16      8       4       2       1
Bit 7   Bit 6   Bit 5   Bit 4   Bit 3   Bit 2   Bit 1   Bit 0
```

To keep the code short I have placed the number representing the bit value of each of the checkboxes into the Tag property of each checkbox. Below is the code for UpdateChecks:

```
procedure TForm1.UpdateChecks(Sender: TObject);
begin
  with Sender as TCheckbox do
    begin
      if Checked = True then
        TheNumber := TheNumber or Tag
      else
        TheNumber := TheNumber and not Tag;
      Label1.Caption := IntToStr(TheNumber);
      UpdateMimics;
    end;
end;
```

By **or**-ing TheNumber with the Tag Value of any given checkbox we are garaunteed that the bit value for that checkbox will be true. Conversly, the **and not** operator will replace that bit with a false. Below is the Code for UpdateMimics:

```
Const  {represents the bit number as an integer}
  Zero  = 1;
  One   = 2;
  Two   = 4;
  Three = 8;
  Four  = 16;
  Five  = 32;
  Six   = 64;
  Seven = 128;

procedure UpdateMimics;
begin
  {Sorts the Contents of the Field to the corrent Checkboxes}
  {Typically, this code would be attached to a TDataSource OnDataChange Event}
  with Form1 do
    begin
      Checkbox9.Checked  := Boolean(TheNumber and Zero);
      Checkbox10.Checked := Boolean(TheNumber and One);
      Checkbox11.Checked := Boolean(TheNumber and Two);
      Checkbox12.Checked := Boolean(TheNumber and Three);
      Checkbox13.Checked := Boolean(TheNumber and Four);
      Checkbox14.Checked := Boolean(TheNumber and Five);
      Checkbox15.Checked := Boolean(TheNumber and Six);
      Checkbox16.Checked := Boolean(TheNumber and Seven);
    end;
end;
```

As you can see, there is one entry for each Checkbox. Firstly, the checkbox bit value is **and**-ed against the number. The result is cast as a boolean and used to set the checked property of the mimic.

Edit Boxes

This is a little more complex but is explainable. Because our example allow an entry from 0 to 3, we can establish by looking at bits one and two of the bit representation chart that this range of numbers can be stored as two bits (3 = 1 + 2) ... Bits 0 and 1.

By using the SHR (Shift Right) and SHL (Shift Left) operators the value of an edit box can be moved to the bit 0 and 1 positions, compared against the number 3 which represents bits 0 and 1 both being set, and dealt with from there. In this case, the tag property of each edit field represents the number of bits

that the number must be shifted to reach the bit 0 and bit 1 positions. Here is the code for the UpdateEdits function:

```
procedure TForm1.UpdateEdits(Sender: TObject);
var
  TheValue : Byte;
begin
  with Sender as TEdit do
    begin
      if Text = '' then Text := '0';
      if StrToInt(Text) > 3 then Text := '0';
      TheValue := StrToInt(Text);
      TheEdit:= TheEdit and not ((TheEdit shr Tag and 3) shl Tag) or TheValue shl
Tag;
      Label4.Caption := IntToStr(TheEdit);
      UpdateEditMimics;
    end;
end;
```

The meat of the function is all in one line. Here is how it breaks down:

Firstly the contents of TheEdit are shifted right by Tag positions. This aligns the relevent 2 bits with bits 0 and 1. The result is then **and**-ed with 3 to return the previous value of only those bits. This value is then shifted left back to the original position. By using the and not operator the two bits representing the contents of the edit field are set to 0. The contents of TheValue (the new value of the Edit Field) are shifted left by tag positions to aligh the bits with there proper place in the number, and the two numbers are **or**-ed, which places the value of TheValue into its proper place. Here is the code for UpdateEditMimics:

```
procedure UpdateEditMimics;
begin
  with Form1 do
    begin
      {Sorts the Contents of the Field to the corrent EditField}
      {Typically, this code would be attached to a TDataSource OnDataChange Event}
      Edit4.Text := IntToStr(TheEdit and 3);
      Edit5.Text := IntToStr(TheEdit shr 2 and 3);
      Edit6.Text := IntToStr(TheEdit shr 4 and 3);
  end;
end;
```

As with UpdateMimics, there is 1 line per object. For each EditField, TheEdit is shifted right by the number of positions necessary to align the data with bits 0 and 1, and then anded with 3 (bits 0 and 1 set) to ignore the other bits. Both the Update functions can be written more elegantly, but are done this way fror clarity.

The Editfield range can be changed by assigning more bits per field - 3 bits allows 0 to 8 etc... and changing the shift values accordingly. To seriously save tablespace with 1 of n choice fields, assign the itemsindex property of a stringlist to the table instead of the contents.

Feel free to e-mail if you have questions.

Demo Program Form File

Demo Program Unit Source

Return to Tips & Tricks

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #10 - December 12th, 1995

Drag and Drop from FileManager

by Eric Kundl - CIS: 71560,1373

Delphi allows dragging and dropping within the application easy enough, most components have a dragmode property. But on a small app I was writing I wanted to be able to go into FileManager and drop a filename on the app and have the app start up with the dropped file...OR... if the app was already running, to grab a handful of files (in a greedy sort of way) from FileManager and drop the whole truckload on the apps form. I have seen it done in other apps, how hard could it really be?

After a lot of digging through the Delphi Help I found it was not very hard at all. Here is what I have discovered.

The ability to drag and drop onto a running app is possible through the Windows API. The Delphi provided SHELLAPI unit has the 3 functions needed allow drag-drop functionality. They are:

DragAcceptFiles	Registers whether a window accepts dropped files
DragQueryFile	Retrieves the filename of a dropped file
DragFinish	Releases memory allocated for dropping files

There is also a function not specifically needed to accept files:

DragQueryPoint	Retrieves the mouse position when a file is dropped

In addition, there is a windows message that your app needs to respond to that is sent when you release the mouse button while dragging: WM_DROPFILES.

First, add SHELLAPI to the units list of your form. Next, your app needs to tell Windows that the form is accepting dropped files. The FormCreate Event of your form would need the following code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DragAcceptFiles(Form1.Handle, true);
  Application.OnMessage := AppMessage;
end;
```

Here the first line calls the DragAcceptFiles function, passing the Handle of the form, and the boolean true that sets up drag-drop for the form. The second line allows your App to have its own message handler. In this case you are going to write a procedure named AppMessage, that will trap a Windows message before Windows itself processes it. In your case, you want to trap the WM_DROPFILES message.

Note that the DragAcceptFiles function call uses the Handle of the Form. If you want to have your app accept dropped files when it is minimized, you must add another call to DragAcceptFiles and pass the applications handle. ie: `DragAcceptFiles(Application.Handle, true);`

Thats all there is to setting things up. Next you need to write your own message handler, in this case named AppMessage.

The OnMessage event occurs when your application receives a Windows message. By creating an OnMessage event handler in your application, you can intercept specific windows messages and handle them yourself. Any message that you do not handle yourself is dispatched and Windows will handle it.

You need to write this procedure yourself, there is no event on the Object Inspector to click on that will cause Delphi to create it for you. So you need to create this in the implementation section of the forms code. And don't forget to add the procedure header to the object definition.

The procedure header would look like:

```
procedure AppMessage(var Msg: Tmsg; var Handled: Boolean);
```

Your message handler procedure would look something like this:

```
procedure TForm1.AppMessage(var Msg: Tmsg; var Handled: Boolean);
const
    BufferLength : word = 255;
var
    DroppedFilename : string;
    FileIndex : word;
    QtyDroppedFiles : word;
    pDroppedFilename : array [0..255] of Char;
    DroppedFileLength : word;
begin
  if Msg.Message = WM_DROPFILES then
  begin
    FileIndex := $FFFF;
    QtyDroppedFiles := DragQueryFile(Msg.WParam, FileIndex,
                                     pDroppedFilename, BufferLength);
    for FileIndex := 0 to (QtyDroppedFiles - 1) do
    begin
      DroppedFileLength := DragQueryFile(Msg.WParam, FileIndex,
                                         pDroppedFilename, BufferLength);
      DroppedFilename := StrPas(pDroppedFilename);

      ..... do something with DroppedFilename ....

    end;
    DragFinish(Msg.WParam);
    Handled := true;
  end;
end;
```

In this code, only the Msg.Message WM_DROPFILES is intercepted, which occurs when you drop one or more files from FileManager onto the window (or app, if you included a DragAcceptFiles for the Application.Handle).

If something is dropped, then you call DragQueryFile to process what was dropped, and before you leave, call DragFinish to free up memory used and set Handled to true to let Windows know you used the message yourself.

The Msg.WParam parameter is the Word parameter passed to the application in the wParam parameter of the WM_DROPFILES message.

The DragQueryFile function has 2 uses, based on the value of the second parameter. If FileIndex is a -1 ($FFFF), then it returns the qty of files dropped. If the value of the FileIndex parameter is between zero and the total number of files dropped, DragQueryFile copies the filename corresponding to that value to the buffer pointed to by the pDroppedFilename parameter. Note that the FileIndex values are offset from 0, so the first filename is FileIndex = 0 and the last filename is FileIndex = (qty of files dropped) minus 1.

The third parameter is the buffer that filenames will be placed into. Note that it is not a pascal string, but since we are dealing with the Windows API, is a null terminated string. The fourth parameter is the length of the filename buffer we are providing.

And that is it. Fairly simple, although it took a number of hours of searching books, and particularly the Delphi On-Line help. A few things I have noticed:

1. When dropping files, the DroppedFilename is the complete path, not just the filename.ext
2. It is possible to drag and drop just a directory. So if you are expecting filenames, you have to check for existence yourself.
3. The filenames come in uppercased.

I have included an example program that shows how to get filenames into the app by:

1. dropping filename(s) from filemanager onto the running app, as described above.
2. dropping filename(s) from filemanager onto the minimized apps icon.

3. dropping a filename from filemanager onto the non-running apps filename also in filemanager.
4. running the app and specifying the filenames as parameters on the run command.
5. using the OpenDialog to select filename(s) that are passed in a stringlist.

Anything dragged and dropped is dumped into a listbox.

As a bonus to all this digging, I found that the OpenDialog component allows:

1. selecting more than one file: set the property Options|ofAllowMultiSelect to true.
2. keeping a list of all previously selected files for the user to reselect from: set the property FileEditStyle to fsComboBox.
3. remembering the path of the last selection, and starting the next selection from this path: set the property Options|ofNoChangeDir to false.
   and what may be a bug: if you select multiple files on the root directory, the filenames have an extra backslash(?).

So there you have the sum total of a week of my evenings. With just a small amount of code, your apps can quickly become FileManager friendly.

Drag & Drop Form
Drag & Drop Source

Return to Tips & Tricks
Return to Front Page

UNDU Readers Choice Awards

Because of the simple and elegant nature of creating components in Delphi, there has been a tidalwave of 3rd party products available on CompuServe and the Internet. It is now getting to the point that so many new ones appear each day, that I no longer have the time to look through them all. I think this is unfortunate, because there are a number of great products out there.

In order to rectify this situation, and to help Delphi developers decide which 3rd party tools would best help them, I am instituting the **UNDU Readers Choice Awards**. These awards will serve as a simple rating system of those 3rd Party products that users find the most indispensible.

Please feel free to vote on any Professional/Shareware/Freeware components or other Delphi Add-Ins. For each product, simply rate them on *a scale from 1 to 10*. A rating of '1' indicates you have a very low opinion of the products usablility or capabilities. A rating of 10 would indicate that the product is indispensible and no Delphi developer should be without it.

To cast your vote, simply send me an email (**CompuServe:** 76416,1373, or **Internet:** RobertV@ix.netcom.com) and list the products that you have had experience with along with the rating (1 to 10) that you would give for each. Please only vote for those products that you have used. All votes will be kept confidential, and will serve only to provide a ranking system for the products voted on. Feel free to add any brief comments to your ballot about particular products. When I print the results of the voting, I may include a few excerpts of these comments for some of the more outstanding products. If you happen to know where you obtained the product (i.e. on CompuServe, or a particular Web site, or for sale elsewhere), you may feel free to include this information as well.

I will be maintaining a database of the results and will print updated rankings in all future issues. As a result, there really is no 'deadline' as such for the ballots, as all new votes will be inserted into this database. However, please do not send multiple votes for a single product.

Hopefully the information obtained through these awards will help Delphi Developers to best utilize the available tools currently on the market.

Return to Front Page

```
object Form1: TForm1
  Left = 192
  Top = 102
  Width = 435
  Height = 300
  Caption = 'ekDrop'
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'System'
  Font.Style = []
  PixelsPerInch = 96
  Position = poDefault
  OnActivate = FormActivate
  OnCreate = FormCreate
  TextHeight = 16
  object ListBox1: TListBox
    Left = 0
    Top = 45
    Width = 427
    Height = 226
    TabStop = False
    Align = alClient
    IntegralHeight = True
    ItemHeight = 16
    TabOrder = 0
  end
  object Panel1: TPanel
    Left = 0
    Top = 0
    Width = 427
    Height = 45
    Align = alTop
    Alignment = taRightJustify
    Caption = 'this is the caption'
    TabOrder = 1
    object btnOpen: TButton
      Left = 4
      Top = 4
      Width = 57
      Height = 33
      Caption = 'Open'
      TabOrder = 0
      OnClick = btnOpenClick
    end
    object btnExit: TButton
      Left = 68
      Top = 4
      Width = 57
      Height = 33
      Caption = 'Exit'
      TabOrder = 1
      OnClick = btnExitClick
    end
    object btnClear: TButton
      Left = 132
      Top = 4
      Width = 57
      Height = 33
      Caption = 'Clear'
      TabOrder = 2
      OnClick = btnClearClick
    end
```

```
  end
  object OpenDialog: TOpenDialog
    FileEditStyle = fsComboBox
    Filter = 'All files (*.*)|*.*'
    Options = [ofAllowMultiSelect, ofPathMustExist, ofFileMustExist]
    Title = 'Open Dialog'
    Left = 396
    Top = 4
  end
end
```

Return to Drag & Drop Article
Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #10 - December 12th, 1995

Status Bar Resource Gauge Modification

by Jeremy Coleman - INTERNET:jeremy_coleman@signas.dpa.act.gov.au

Robert: I really appreciated your status bar component, and have modified it slightly so that the bar changes color depending on the amount of available resources (like the Microsoft SysMeter utility). Modification is as follows:

```
if ShowResources then
begin
     ResGauge.Progress := GetFreeSystemResources(GFSR_SYSTEMRESOURCES);
     if (60 <= ResGauge.Progress) then
        ResGauge.ForeColor := clLime
     else
        if (30 <= ResGauge.Progress) then
           ResGauge.ForeColor := clYellow
        else
           ResGauge.ForeColor := clRed;
end;
```

Return to Component Cookbook

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #10 - December 12th, 1995

Delphi Tips - Keeping Application Size to a Minimum & Changing Icons at Runtime

by Paul Harding - CIS: 100046,2604

My company uses about half a dozen applications that are written in Delphi, but tends to use them only one at a time. By gathering together all the .PAS files and all the foms into one project, I've compiled the whole lot into an executable called "TARGET.EXE" ('cos my firm's name is Target Couriers Ltd). Then, by passing a parameter on the command line, I can get the project to decide which form to run.

Here is a code snippet that shows how the project decides between two of the forms, one for Vehicle Volumes and the other is a little utility to test disk drives across the network:

```
{this lives in the project source...}
if UpperCase(ParamStr(1)) = 'VEHICLEVOLUMES' then
  begin
    Application.Title := 'Veh. Volumes';
    Application.CreateForm(TFrmVolumes, FrmVolumes);
  end;
if UpperCase(ParamStr(1)) = 'DRIVETEST' then
  begin
    Application.Title := 'Drive Test';
    Application.CreateForm(TFrmTestDiskDrives, FrmTestDiskDrives);
  end;
{and at the end of the If..end constructs, is the usual...}
 Application.Run;
```

When the program runs, (for example: "C:\MYPROGS\TARGET.EXE VehicleVolumes") the application's icon needs to change. This is done by using "LoadFromFile" to set the application's property "Icon":

```
if UpperCase(ParamStr(1)) = 'VEHICLEVOLUMES' then
  begin
    Application.Icon.LoadfromFile('C:\myprogs\VehVols.ico');
   {rest of code as above.....}
  end;

if UpperCase(ParamStr(1)) = 'DRIVETEST' then
  begin
    Application.Icon.LoadfromFile('C:\myprogs\Drive.ico');
   {rest of code as above.....}
  end;
```

Even though these two examples above are functionally entirely different, by compiling them together (and indeed the 4 or 5 other programs too) I can make an executable with a size only a little larger than the size of each separate program (if compiled independantly). This seems quite handy, and is a classic case of the sum of the parts being far less than the whole!

Return to Tips & Tricks

Return to Front Page

If you'd like to see The Edits, ftp://ftp.coriolis.com/delphi and look for DBDXpand.ZIP. If it is alone, grab it. If there is also DBDXpnd2.ZIP take that instead because the webmeister hasn't upgraded the main file from the emergency one. Or grab both and see the bug as well as the fix.