The Unofficial Newsletter of Delphi Users - Issue #16 - September 1996

Another month zooms by and I think I am back on my publishing schedule. The article submissions were down a bit this time so I decided to spend some time putting together some really cool Tips & Tricks for this issue. I had a lot of fun researching and preparing these techniques and I hope everyone enjoys them!

If you have a technique that you really like and would like to share it with other Delphi Developers, please send it in! I know there are many of you out there who are saying *"I don't have anything worth printing…"* Hogwash! Some of the best stuff I have seen are from beginners. It doesn't have to be anything fancy and it doesn't need to be overly complex. Most of the tips included in this months issue involve less than a dozen lines of code. In fact, the most elegant programming solutions are the ones in the fewest lines of code.

I enjoy seeing the work that Delphi programmers put together. Keep up the good work!

- Robert

A Super-Expanded Tips & Tricks Section!

Core Concepts With Delphi - Enumerated Types

The Component Cookbook

Extending The INI Component

UNDU Prizes!

UNDU Subscriber List

Index of Past Issues

Where To Find UNDU

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 #1 - March 15, 1995

- What You Can Do
- Component Design
- Currency Edit Component
- Sample Application
- The Bug Hunter Report
- About The Editor
- SpeedBar And The ComponentPalette
- Resource Name Case Sensitivity
- Lockups While Linking
- Saving Files In The Image Editor
- File Peek Application

Issue #2 - April 1, 1995

- Books On The Way
- Making A Splash Screen
- Linking Lockup Revisited
- Problem With The CurrEdit Component
- Return Value of the ExtractFileExt Function
- When Things Go Wrong
- Zoom Panel Component

Issue #3 - May 1, 1995

- Articles
- Books
- Connecting To Microsoft Access
- Cooking Up Components
- Copying Records in a Table
- CurrEdit Modifications by Bob Osborn
- CurrEdit Modifications by Massimo Ottavini
- CurrEdit Modifications by Thorsten Suhr
- Creating A Floating Palette
- What's Hidden In Delphi's About Box?
- Modifications To CurrEdit

Periodicals
Progress Bar Bug
Publications Available
Real Type Property Bug
TIni File Example
Tips & Tricks
Unit Ordering Bug
When Things Go Wrong

Issue #4 - May 24, 1995

Cooking Up Components
Food For Thought - Custom Cursors
Why Are Delphi EXE's So Big?
Passing An Event
Publications Available
Running From A CD
Starting Off Minimized
StatusBar Component
TDBGrid Bug
Tips & Tricks
When Things Go Wrong

Issue #5 - June 26, 1995

Connecting To A Database
Cooking Up Components
DateEdit Component
Delphi Power Toolkit
Faster String Loading
Font Viewer
Image Editor Bugs
Internet Addresses
Loading A Bitmap
Object Alignment Bug
Second Helping - Custom Cursors
StrToTime Function Bug
The Aquarium
Tips & Tricks
What's New
When Things Go Wrong

Issue #6 - July 25, 1995

A Call For Standards
Borland Visual Solutions Pack - Review
Changing a Minimized Applications Title
Component Create - Review
Counting Components On A Form
Cooking Up Components
Debug Box Component
Dynamic Connections To A DLL
Finding A Component By Name
Something Completely Unrelated - TVHost
Status Bar Component

The Loaded Method
Tips & Tricks
What's In Print

Issue #7 - August 31, 1995

ChartFX Article
Component Cookbook
Compression Shareware Component
Corrected DebugBox Source
Crystal Reports - Review
DBase On The Fly
Debug Box Article
Faster String Loading
Formula One - Review
Gupta SQL Windows
Header Converter
Light Lib Press Release
Limiting Form Size
OLE Amigos!
Product Announcements
Product Reviews
Sending Messages
Study Group Schedule
The Beginners Corner
Tips & Tricks
Wallpaper
What's In Print

Issue #8 - October 10, 1995

Annotating A Help System
Core Concepts In Delphi
Creating DLL's
Delphi Articles Recently Printed
Delphi Informant Special Offers
Delphi World Tour
Getting A List Of All Running Programs
How To Use Code Examples
Keyboard Macros in the IDE
The Beginners Corner
Tips & Tricks
Using Delphi To Perform QuickSorts

Issue #9 - November 9, 1995

Using Integer Fields to Store Multiple Data Elements in Tables
Core Concepts In Delphi
Delphi Internet Sites
Book Review - Developing Windows Apps Using Delphi
Object Constructors
QSort Component
The Component Cookbook
TSlideBar Component
TCurrEdit Component

The Delphi Magazine
Tips & Tricks
Using Sample Applications

Issue #10 - December 12, 1995

A Directory Stack Component
A Little Help With PChars
An Extended FileListBox Component
Application Size & Icon Tip
DBImage Discussion
Drag & Drop from File Manager
Modifying the Resource Gauge in TStatusBar
Playing Wave Files from a Resource
Review of Orpheus and ASync Professional
The Component Cookbook
Tips & Tricks
UNDU Readers Choice Awards
Using Integer Fields to Store Multiple Data Elements in Tables

Issue #11 - January 18th, 1996

Core Concepts With Delphi - Part I
Core Concepts With Delphi - Part II
Dynamic Delegation
Data-Aware DateEdit Component
ExtFileListBox Component
DBExtender Product Announcement
Dynamic Form Creation
Finding Run-Time Errors
Selecting Objects in the Delphi IDE
The Beginners Corner
The Delphi Magazine
Top Ten Tips For Delphi
The Component Cookbook
Tips & Tricks
The UNDU Awards

Issue #12 - February 23rd, 1996

The Beginners Corner
Delphi Projects
Marketing Your Components
An LED Component
A 3D Progress Bar
Common Strings Functions
Checking if your application is running already
AutoRepeat for SpeedButtons
Form and Component Creation Tip
Detecting a CD-ROM Drive
Drawing Metafiles in Delphi
Shazam Review
Product Announcement - Dr. Bob's Delphi Experts
Book Review - Instant Delphi Programming
Tips & Tricks

The Component Cookbook

Issue #13 - May 1st, 1996

Core Concepts - Sorting
Delphi Information Connection
Creating Resource-Only DLL's
Quick Reports
TIFIMG Product Announcement

Issue #14 - June 1st, 1996

A 3-D Component
An Animation Component
A Bug In TGauge
The Component Cookbook
A Look At Cross Tabs
New Book - Delphi In Depth
New Book - The Revolutionary Guide to Delphi 2
Making the Enter Key Work Like the Tab Key
Jumping Straight to Form Level
Making Menu Items Work Like Radio Buttons
Modifying The System Menu
Products & Reviews
The Beginners Corner
The UNDU Awards
Tips & Tricks

Issue #15 - August 1st, 1996

UNDU - A Work In Progress…
UNDU Prizes!
The UNDU Subscriber List
Core Concepts With Delphi - Parameter Passing
Delphi Programmers Book Shelf
Component Cookbook
Tips & Tricks
How to 'Catch'Keys
Working with String Grids
Coloring Columns in a Grid
Solving a DLL problem
Reducing Memory Requirements
Creating an AutoDialer component

Return to Front Page

Where To Find UNDU

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

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

2. *Borland's* Delphi forum on CompuServe (**GO DELPHI**) in the "Delphi IDE" file section. This forum will only hold the issues in Windows HLP format.
3. *Informant Communications* forum also on CompuServe (**GO ICGFORUM**) in the "Delphi 3rd Party" file section. Again, this forum will only hold issues in the Windows HLP format.

Tips & Tricks

This month marks a significant expansion to the **Tips & Tricks** section. Each month I will be bringing you the best tips I can scrape up either through contributions from readers or on my own.

Do you want to prevent your program from having multiple copies running at the same time? In the 16-bit version of Delphi, this was a fairly easy task. Moving to a 32-bit platform makes the process a bit more complicated. Learn how to accomplish the same effect in the tip on Limiting Multiple Instances Of a Program in Delphi 2.0.

Sometimes you need to provide an interface for users to drag a selection rectangle around objects. If so, check out the article on How to Draw a Rubber-Banding Line! Do you want something a bit more complex? How about an *animated* selection rectangle, described in the section on Marching Ants!

Oftentimes when utilizing these techniques and other involving the use of mouse-manipulation of objects, you might have a need to restrict where the mouse can go. This is very easily accomplished and is covered in the tip on How to Restrict the Mouse Cursor.

Do you need to provide users with the ability to select colors? One option is the TColorDialog which comes with Delphi. You could also use the technique discussed in How to make a Color ComboBox.

Sometimes you may have a need to create a pop-up menu programmatically. Some of you may not be aware that there is A Better Way to Create Menu Items!

You probably know how to do a Splash Screen… But how about a Splash Screen with a Time Delay?

More exciting Tips & Tricks next month!

Return to Front Page

The Component Cookbook

Menu Buttons!

by Pedro Agulló, pagullo@ctv.es

Maybe you've seen those buttons that show a popup menu when clicked? I wanted to use that kind of element to manage a list of bookmarks for an editor - the one I include as part of the IDE for the scripting language I use in my applications. I was going to use three of these buttons: the first one was to be used to add new bookmarks, showing the existing ones checked. The second button would show a list of existing bookmarks and let you remove them. The third button would be used to go to a given bookmark. You can see the idea in action below:

I ended up constructing a component instead of doing all the housekeeping by hand for each button. I used TSpeedButton as the ancestor, and called it *TSpeedBtnMenu*.

Requirements

After thinking a bit, I concluded that the functionality I needed was the following:

1. It would be nice to manipulate the items shown in the popup menu via a TStrings object, which has a predefined property editor.
2. It should be possible to use the menu to show multiple selected items… I needed to show a list of all available bookmarks with the assigned ones checked.
3. It should be possible to modify the menu items before showing the menu.
4. The program should be notified when the end-user makes a selection.
5. It would be nice to show a title explaining what the menu is for, because showing only a list of numbers was not very informative. Figure 1 shows a menu button with a title -the first menu item.

In order to provide this functionality, I ended up adding five properties and two events to those provided by TSpeedButton, plus three procedures:

1. *Items*: the items you want to show as menu items. Use *BeforeShow* to add, remove or modify them just before they are shown.
2. *ItemIndex*: the index of the last selected item, -1 if none. Read-only.
3. *Title*: if you want to show a title for the menu put it here. '' for no title.
4. *ShowChecked*: if this is True, the menu will show as checked the selected item/s.
5. *MultiSelect*: if this is True, the menu will remember all the selections since the last time you assigned a value to this property. An easy way to clear all selections is to make `MultiSelect := MultiSelect`.
6. *BeforeShow*: event triggered just before the menu is shown. Use it to change the items you want to show.

7. *OnSelect*: event triggered when the user selects a menu item. Use *ItemIndex* to get the index of the selected item. DO NOT use OnClick in place of OnSelect.

These were the three functions and procedures added to work with multiple selection:

1. procedure *Select*(ItemIndex: index), selects the item with the given index -it will appear checked the next time the menu pops up.
2. procedure *DeSelect*(ItemIndex : Integer).
3. function *Selected*(ItemIndex : Integer): Boolean, returns true if the item with the given item is selected -checked.

The Source Code

Here's the source code for the component. If you want to get a TBitBtn descendant with the same capabilities, make a copy of all the source code, and substitute *TSpeedBtnMenu* by *TBitBtnMenu.* It works: I just don't want to make this larger. By the way, remember to modify the Register procedure to register TBitBtnMenu too.

About The Author

Pedro Agulló Soliveres is a Software Engineer which specializes in RAD and Internet/Intranet development. He is currently working on C++, Java, Delphi and Intrabuilder applications in the Valencia-Murcia area in Spain. You can contact him via e-mail at pagullo@ctv.es.

P.S.

It would be nice to contact people using Borland's IntraBuilder here in Spain. Whether you are doing it or not, feel free to e-mail me. All suggestions are welcome.

Return to Front Page

UNDU Prizes!

As mentioned in last issue I will be giving away 1 or 2 Delphi related products each month to a randomly chosen contributor to UNDU.
The winners this month are Alan G. Labouseur for his article on *Enumerated Types* and also Pedro Agulló for his article on *Menu Buttons*.
Alan's prize is a copy of **Delphi-In-Depth**, a new book from Osborne/McGraw Hill covering advanced Delphi 2.0 techiques. Pedro wins a copy of the **1995 Delphi Informant Works CD** that holds electronic versions of all issues of Delphi Informant for 1995 plus tons of extra's!
Thanks to all the contributors for making UNDU a success. Keep those articles & tips coming!

Return to Front Page

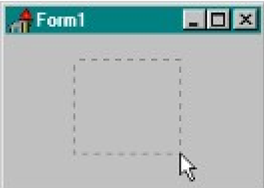
UNDU Subscriber List

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 reader's email addresses and when a new issue is released, I will fire off a batch mailing to notify everyone that it is available.

This is what you need to do to get on the subscriber list… Simply send me an email to my CompuServe address (**`76416,1373`**) 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 me an email with the words **UNSUBSCRIBE UNDU**. If you are sending mail from the Internet, the address is **`76416.1373@cis.com`**

That's all there is to it!

Return to Front Page



How to Draw a Rubber-Banding Line

by Robert Vivrette - CIS: 76416,1373

A Rubber-Banding line is a technique you often see in Windows. For example, you may be looking at an open folder and want to drag a selection rectangle around a few files. As it turns out, this is quite a simple technique in Delphi.

All that is required is to provide a handler for a form's *MouseDown* and *MouseMove* events as follows:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
                               Shift: TShiftState; X, Y: Integer);
begin
  Canvas.Pen.Mode := pmXOr;
  Canvas.Pen.Style := psDot;
  X1 := X; Y1 := Y; X2 := X; Y2 := Y;
end;

procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState;
                               X, Y: Integer);
begin
  if ssLeft in Shift then
    begin
      Canvas.PolyLine([Point(X1,Y1),Point(X2,Y1),Point(X2,Y2)]);
      Canvas.PolyLine([Point(X1,Y1),Point(X1,Y2),Point(X2,Y2)]);
      X2 := X; Y2 := Y;
      Canvas.PolyLine([Point(X1,Y1),Point(X2,Y1),Point(X2,Y2)]);
      Canvas.PolyLine([Point(X1,Y1),Point(X1,Y2),Point(X2,Y2)]);
    end;
end;
```

MouseDown first sets the Form's pen mode to *pmXOr* and its style to *psDot*. Drawing in an XOr mode provides two unique capabilities. First, XOr mode will always show up regardless of the color combinations you have selected. Second, XOr lines can erase themselves! More on that a little later…

Next, we setup 4 private variables: X1,Y1,X2, & Y2. You should define all of these in the **Private** section of the form's definition as Integer type. X1 & Y1 will hold the X,Y coordinates of where the mouse initially goes down. X2 & Y2 will hold the X,Y coordinates as the mouse moves.

Now, all that is left is to do the **MouseMove**. First, I use **PolyLine** to draw from X1,Y1 to the right and then down to X2,Y2. Then with a second PolyLine statement I draw from X1,Y1 down and then right to X2,Y2. I could have draw the rectangle all in one statement, but the dot pattern looks better this way as the mouse moves.

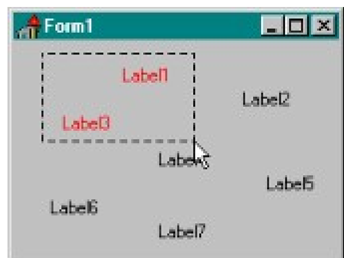
Next, I update X2 and Y2 to the current mouse position, and then redo the same two PolyLine commands again. That's it!

The key to how this works is the XOr command. If you draw a line XOr'ed on the screen and then immediately draw another XOr'ed line over it, the line disappears and the screen is restored to the condition it was before you drew the line. Therefore, when the mouse moves, you simply draw the old rectangle (erasing it) and then draw the new one.

If you wanted the rectangle to disappear when you brought the mouse up, all you would need to do is add a **MouseUp** handler that simply repeated the 2 PolyLine commands again. When you are all done, the two corners are held in (X1,Y1) and (X2,Y2).

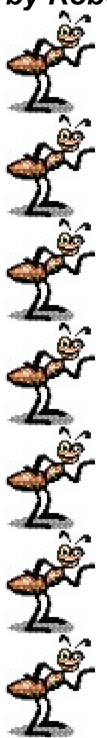
Want something a bit more complex? Try the Marching Ants example!

Return to Tips & Tricks
Return to Front Page



How to Draw Marching Ants

by Robert Vivrette - CIS: 76416,1373



Marching Ants (sometimes also called a *Marquee Rectangle*) is essentially an *animated* Rubber-Banding line. Most high-end drawing packages will have some kind of animated selection rectangle. It is generally not implemented in most programs simply because it is a little more work that a normal Rubber-Banding Line.

Before I show you how it is done, let me describe the behavior. First, the user clicks down the left mouse button and holds it down. As the mouse is then dragged away from this point, an animated selection rectangle appears between the two points. The animation looks much like ants marching single file. The dot pattern forming the rectangle slowly rotates clockwise. The animation functions while the mouse is moving and continues even after the mouse has been released. The picture above shows a sample application implementing Marching Ants, but of course you can't see the animation in the line.

This extension of a Rubber-Banding line is just a little more complex. The source code is provided below and is fairly well commented. The key difference in this is that we are not using **XOr** any more, but rather we are doing our own line drawing routine with **LineDDA**.

Most of you may not be aware of the Windows LineDDA API call, so let me talk about it briefly. Essentially, LineDDA allows you to draw a line between two points, but for each pixel that needs to be drawn, Windows will call a user-defined function that you provide. This function can do most anything. For example, instead of drawing a pixel each time the function is called, you might choose to draw a small circle every 5th time. Or, you could change the color of every pixel to create a rainbow effect. You can get some pretty wild line patterns, and your imagination is the only limit.

In the Marching Ants example, all I do is draw a simple dotted line. The dotted line is formed by drawing 3 clear pixels and then 5 black pixels. This means that out of every 8 calls to the function, the first 3 will produce a clear pixel on the screen and the last 5 will produce black pixels. Since Windows is fetching these colors in sequence as it attempts to draw the line, the effect is a dotted line. The animation effect is created by drawing the line over and over again, but shifting the starting pixel pattern one pixel each time it starts.

The sample application shows how you can use an animated selection rectangle like this to select objects on a form (in this case Labels). When a rectangular area is selected and the mouse goes up, all labels within that area are highlighted Red.

The only thing I don't like about my solution is the method used to erase prior lines. It works fine and is only a few lines of code, but it 'feels' wrong to me. If anyone comes up with an improvement to the RemoveTheRect routine, please let me know and I will publish the revised code next issue.

Complete Source for Marching Ants

Return to Tips & Tricks

Return to Front Page

Marching Ants Source Code

To re-create this application, simply create a new application, drop a Timer and 7 Labels on the form. Then paste the following text over all the existing code in the Forms source code unit. Compile and enjoy!

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
                            Shift: TShiftState; X, Y: Integer);
    procedure FormMouseMove(Sender: TObject; Shift: TShiftState;
                            X,Y: Integer);
    procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
                          Shift: TShiftState; X, Y: Integer);
  private
    X1,Y1,X2,Y2 : Integer;
    procedure RemoveTheRect;
    procedure DrawTheRect;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  Counter : Byte;
  CounterStart : Byte;
  Looper : LongInt;

implementation

{$R *.DFM}

procedure MovingDots(X,Y: Integer; TheCanvas: TCanvas); stdcall;
begin
  Inc(Looper);
  Counter := Counter shl 1;              // Shift the bit left one
  if Counter = 0 then Counter := 1;      // If it shifts off left, reset it
  if (Counter and 224) > 0 then          // Are any of the left 3 bits set?
    TheCanvas.Pixels[X,Y] := Form1.Color // Erase the pixel
  else
    TheCanvas.Pixels[X,Y] := clBlack;    // Draw the pixel
end;

function NormalizeRect(R: TRect): TRect;
begin
```

```
  // This routine normalizes a rectangle. It makes sure that the Left,Top
  // coords are always above and to the left of the Bottom,Right coords.
  with R do
    if Left > Right then
      if Top > Bottom then
        Result := Rect(Right,Bottom,Left,Top)
      else
        Result := Rect(Right,Top,Left,Bottom)
    else
      if Top > Bottom then
        Result := Rect(Left,Bottom,Right,Top)
      else
        Result := Rect(Left,Top,Right,Bottom);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  X1 := 0; Y1 := 0;
  X2 := 0; Y2 := 0;
  Canvas.Pen.Color := Color;
  Canvas.Brush.Color := Color;
  CounterStart := 128;
  Timer1.Interval := 100;
  Timer1.Enabled := True;
  Looper := 0;
end;

procedure TForm1.RemoveTheRect;
var
  R : TRect;
begin
  R := NormalizeRect(Rect(X1,Y1,X2,Y2));  // Rectangle might be flipped
  InflateRect(R,1,1);                     // Make the rectangle 1 pixel larger
  InvalidateRect(Handle,@R,True);         // Mark the area as invalid
  InflateRect(R,-2,-2);                   // Now shrink the rectangle 2 pixels
  ValidateRect(Handle,@R);                // And validate this new rectangle.
  // This leaves a 2 pixel band all the way around
  // the rectangle that will be erased & redrawn
  UpdateWindow(Handle);
end;

procedure TForm1.DrawTheRect;
begin
  // Determines starting pixel color of Rect
  Counter := CounterStart;
  // Use LineDDA to draw each of the 4 edges of the rectangle
  LineDDA(X1,Y1,X2,Y1,@MovingDots,LongInt(Canvas));
  LineDDA(X2,Y1,X2,Y2,@MovingDots,LongInt(Canvas));
  LineDDA(X2,Y2,X1,Y2,@MovingDots,LongInt(Canvas));
  LineDDA(X1,Y2,X1,Y1,@MovingDots,LongInt(Canvas));
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  CounterStart := CounterStart shr 2;            // Shl 1 will move rect slower
  if CounterStart = 0 then CounterStart := 128; // If bit is lost, reset it
  DrawTheRect;                                   // Draw the rectangle
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
                               Shift: TShiftState; X, Y: Integer);
begin
  RemoveTheRect;                                // Erase any existing rectangle
```

```
  X1 := X; Y1 := Y; X2 := X; Y2 := Y;
end;

procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState;
                               X, Y: Integer);
begin
  if ssLeft in Shift then
    begin
      RemoveTheRect;         // Erase any existing rectangle
      X2 := X; Y2 := Y;      // Save the new corner where the mouse is
      DrawTheRect;           // Draw the Rect now... don't wait for the timer!
    end;
end;

procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
                             Shift: TShiftState; X, Y: Integer);
var
  R1,R2,R3 : TRect;
  a        : Integer;
begin
  // Color all labels red that are in the rectangle
  For a := 0 to ControlCount-1 do
    if Controls[a] is TLabel then
      begin
        R1 := (Controls[a] as TLabel).BoundsRect;
        R2 := NormalizeRect(Rect(X1,Y1,X2,Y2));
        if IntersectRect(R3,R1,R2) then
          (Controls[a] as TLabel).Font.Color := clRed
        else
          (Controls[a] as TLabel).Font.Color := clWindowText;
      end;
end;

end.
```

Return to Article

Return to Front Page

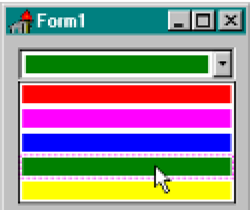
```
{ Copyright (c) Pedro Agulló Soliveres, 1996.
  All rights reserved.

This is FREEWARE.
I do like to receive postcards. If you find it nice to send me a postcard, e-mail me
at pagullo@ctv.es, please, and I'll send you my last address.}

unit Btnmenu;

interface

uses
  Buttons, Classes, Menus, Controls;

type

TSpeedBtnMenu = class(TSpeedButton)
  private
    FItems : TStrings; {We'll use FItems.Objects as a flag to know if
                       an item is selected. Typecasting will be needed!}
    FTitle : String;
    FMenu : TPopupMenu;
    FSelectedItemIndex : Integer; { The index of the last selected item }
    FShowChecked : Boolean;
    FMultiSelect : Boolean;
    FOnSelect : TNotifyEvent;
    FBeforeShow : TNotifyEvent;
    procedure MenuClicked(sender: TObject);
    procedure SetItems(s: TStrings);
    procedure SetMultiSelect(b: Boolean);
    function  Count: Integer;
  protected
    procedure MouseUp(Button: TMouseButton; Shift: TShiftState;
                      X, Y: Integer); override;
  public
    constructor Create(parent: TComponent); override;
    destructor  Destroy; override;
    procedure   Select(i: Integer);
    procedure   DeSelect(i: Integer);
    function    Selected(i: Integer): Boolean;
  published
    property MultiSelect: Boolean read FMultiSelect
                               write SetMultiSelect default False;
    property Title: String read FTitle write FTitle;
    property Items: TStrings read FItems write SetItems;
    property ItemIndex: Integer read FSelectedItemIndex default -1;
    property ShowChecked: Boolean read FShowChecked
                               write FShowChecked default False;
    property OnSelect: TNotifyEvent read FOnSelect write FOnSelect;
    property BeforeShow: TNotifyEvent read FBeforeShow write FBeforeShow;
end;

procedure Register;

implementation

uses
  WinTypes;

{ TSpeedBtnMenu }
procedure TSpeedBtnMenu.MouseUp(Button: TMouseButton; Shift: TShiftState; X, Y:
Integer);
```

```
var
  p : TPoint;
  i : INteger;
  item : TMenuItem;
begin
  inherited MouseUp(Button,Shift, X, Y );

  { It's easier to free the menu and then recreate it each time }
  FMenu.Free;
  FMenu := nil;

  if Assigned(FBeforeShow) then FBeforeShow(self);

  if Count > 0 then
    begin
      FMenu := TPopupMenu.Create(self);
      { If we want to show a menu title we'll have to append it to the top
        of the menu, followed by a separator }
      if FTitle <> '' then
        begin
          item := TMenuItem.Create(FMenu);
          item.Caption := Title;
          FMenu.Items.Add(item);
          item := TMenuItem.Create(FMenu);
          item.Caption := '-';
          FMenu.Items.Add(item);
        end;
      { We have to append the menu items to the menu }
      for i := 0 to Count -1 do
        begin
          item := TMenuItem.Create( FMenu );
          item.Caption := FItems[i];
          item.OnClick := MenuClicked;
          if ((i = ItemIndex) and (ShowChecked)) or
             (MultiSelect and Selected(i) ) then
            item.Checked := True;
          FMenu.Items.Add(item);
        end;
      { Let's show the popup menu just below the button }
      p.X := 0;
      p.Y := Self.Height;
      p := ClientToScreen(p);
      FMenu.Popup(p.X,p.Y-1);
    end;
end;

function  TSpeedBtnMenu.Count : Integer;
begin
  Result := FItems.Count;
end;

constructor TSpeedBtnMenu.Create(parent: TComponent);
begin
  inherited Create(parent);
  FItems := TStringList.Create;
  FMultiSelect := False;
  FMenu := nil;
  FSelectedItemIndex := -1;
  FTitle := '';
end;

destructor  TSpeedBtnMenu.Destroy;
begin
```

```
  FItems.Free;
  inherited Destroy;
end;

procedure TSpeedBtnMenu.MenuClicked(sender: TObject);
begin
  FSelectedItemIndex := FMenu.Items.IndexOf(sender as TMenuItem);
  if Title <> '' then Dec(FSelectedItemIndex, 2);
  if MultiSelect then Select(FSelectedItemIndex);
  if Assigned(FOnSelect) then FOnSelect(self);
end;

procedure TSpeedBtnMenu.SetItems(s: TStrings);
begin
  FItems.Assign(s);
end;

procedure TSpeedBtnMenu.SetMultiSelect(b: Boolean);
var
  i : Integer;
begin
  if ItemIndex >= 0 then Select(ItemIndex);
  if b then
    FMultiSelect := True
  else
    begin
      FMultiSelect := False;
      { We reset the selected item list each time we assign a value
        to the MultiSelect property. An easy way to clear the selected
        item list is to set MultiSelect := True }
      for i := 0 to Count - 1 do DeSelect(i);
    end;
end;

function TSpeedBtnMenu.Selected(i : Integer ): Boolean;
begin
  if( i >= 0 ) and ( i < Count ) then
    Result := (FItems.Objects[i] = TObject(1))
  else
    Result := False;
end;

procedure TSpeedBtnMenu.Select(i: Integer);
begin
  if (i >= 0) and (i < Count) then FItems.Objects[i] := TObject(1);
end;

procedure TSpeedBtnMenu.DeSelect(i: Integer);
begin
  if (i >= 0) and (i < Count) then FItems.Objects[i] := TObject(0);
end;

procedure Register;
begin
  RegisterComponents('Samples',[TSpeedBtnMenu]);
end;

end.
```

[Return to Component Cookbook]

[Return to Front Page]



Creating a Color ComboBox

by Robert Vivrette - CIS: 76416,1373

Delphi's TColorDialog is a wonderful tool, but there are times when you may want to allow a user to select only a limited set of colors. This was what prompted this technique of creating a Color ComboBox. Basically I am using a normal ComboBox, but I am using its OwnerDraw functionality to paint the cells as I wish. In this case, I want the items to be painted different colors. Here is how it is done!
First, drop a ComboBox on a form and set its Style property to csOwnerDraw. Now double-click on the form itself and add the following code to the Form's OnCreate handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with ComboBox1.Items do
    begin
      Add(IntToStr(clRed));
      Add(IntToStr(clFuchsia));
      Add(IntToStr(clBlue));
      Add(IntToStr(clGreen));
      Add(IntToStr(clYellow));
    end;
end;
```

All I am doing here is to add 5 items to the ComboBox. Each item is a string representation of a color constant. For example, clFuchsia becomes '16711935'.

Next click once on the ComboBox, go to the Events tab and double-click on the OnDrawItem event. Type in the following code:

```
procedure TForm1.ComboBox1DrawItem(Control: TWinControl; Index: Integer;
                                   Rect: TRect; State: TOwnerDrawState);
var
  R : TRect;
begin
  R := Rect;
  with Control as TComboBox,Canvas do
    begin
      Brush.Color := clWhite;
      FillRect(R);
      InflateRect(R,-2,-2);
      Brush.Color := StrToInt(Items[Index]);
      FillRect(R);
    end;
end;
```

This procedure is called every time the ComboBox needs to draw an item in its list. For each item, we first fill the background of the item with White using FillRect, then we shrink the rectangle we are using by 2 pixels on each side and fill it again with the color of the item. That's all there is to it!

Return to Tips & Tricks

Return to Front Page

A Better Way to Create Menu Items

by Robert Vivrette - CIS: 76416,1373

Have you ever had to construct a menu dynamically at runtime? If so, you may have code in your project that looks something like this…

```
PopupMenu1 := TPopupMenu.Create(Self);
Item := TMenuItem.Create(PopupMenu1);
Item.Caption := 'First Menu';
Item.OnClick := MenuItem1Click;
PopupMenu1.Items.Add(Item);
Item := TMenuItem.Create(PopupMenu1);
Item.Caption := 'Second Menu';
Item.OnClick := MenuItem2Click;
PopupMenu1.Items.Add(Item);
Item := TMenuItem.Create(PopupMenu1);
Item.Caption := 'Third Menu';
Item.OnClick := MenuItem3Click;
PopupMenu1.Items.Add(Item);
Item := TMenuItem.Create(PopupMenu1);
Item.Caption := '-';
PopupMenu1.Items.Add(Item);
Item := TMenuItem.Create(PopupMenu1);
Item.Caption := 'Fourth Menu';
Item.OnClick := MenuItem4Click;
PopupMenu1.Items.Add(Item);
```

There is a faster way! Use the **NewItem** and **NewLine** functions to create them all in one pass like this:

```
PopupMenu1 := TPopupMenu.Create(Self);
with PopUpMenu1.Items do
  begin
    Add(NewItem('First Menu',0,False,True,MenuItem1Click,0,'MenuItem1'));
    Add(NewItem('Second Menu',0,False,True,MenuItem2Click,0,'MenuItem2'));
    Add(NewItem('Third Menu',0,False,True,MenuItem3Click,0,'MenuItem3'));
    Add(NewLine);                         // Adds a separator bar
    Add(NewItem('Fourth Menu',0,False,True,MenuItem4Click,0,'MenuItem4'));
  end;
```

Return to Tips & Tricks

Return to Front Page

Creating a Splash Screen

by Robert Vivrette - CIS: 76416,1373

Borland has indicated in its Tech Sheets how to create a simple splash screen, so many of you are already aware of how its done. Just in case you don't, follow these steps:

First, add a new form to your project. Name the form SplashScreen, and save the unit as SPLASH.PAS. Set the Form's BorderStyle to bsNone. Next, drop a TPanel on the Form, set it's Align property to alAlignClient and set its Bevel and BorderStyle properties however you wish. Lastly, drop a TImage on the form and set it's Align property also to alAlignClient. Then Load the TImage with your graphic.

To make the splash screen appear when the application launches, you need to modify your application source code (the DPR file). Go to the View menu and pick Project Source. Make the following revisions to cause the Splash Screen to appear before the first form.

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Splash in 'Splash.pas' {SplashScreen};

{$R *.RES}

begin
  try
    SplashScreen := TSplashScreen.Create(Application);
    SplashScreen.Show;
    SplashScreen.Update;
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    SplashScreen.Close;
  finally
    SplashScreen.Free;
  end;
  Application.Run;
end.
```

But wait… there's more! Try out the SplashScreen with a Delay!

Return to Tips & Tricks

Return to Front Page

Creating a Splash Screen With A Delay

by Robert Vivrette - CIS: 76416,1373

The purpose of a SplashScreen is twofold. It's primary purpose is to announce your application, version number, etc. The second is to conceal the delay required for the rest of your application to complete loading and to appear on the screen.

Ideally, a SplashScreen should appear immediately when the user clicks the application icon, and not disappear until the main form of the app is on the screen and ready to accept input. Now, there isn't much we can do to control how fast it appears… we have it appearing as quickly as it can because it is in the project file and is loading before any other part of the program.

However, what if your application only takes a second or two to load, but you still want a splash screen. With the solution provided before, the SplashScreen would appear and disappear too rapidly.

Hence this modification to implement a **minimum** amount of time that the SplashScreen should stay on the screen. It is quite simple actually, just take the code from the prior article on creating a SplashScreen, and add a timer to the SplashScreen form. Set its interval to 1000 times the number of seconds you want the form to be visible (3-4 seconds is good, so a value of 3000-4000 would be ideal). Make sure it's Enabled value is True. Then double-click on the Timer's OnTimer event and add the following code:

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

Lastly, go back to the form in the object inspector and double-click on the OnCloseQuery event. Add the following code:

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

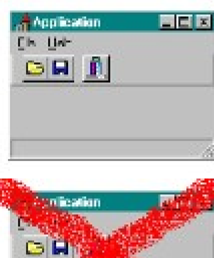
Here is what this does: First, since the timer is enabled initially, it will start counting once the form is created. Once the timer expires its first time, it is disabled. As a result, it will only run once through its interval and then stop. The OnCloseQuery event is used to tell Windows whether the form can be closed or not. If the CanClose variable returns True, the form will be allowed to close, and if it returns False, it won't be. All we have to do is return the opposite of the Timer's Enabled state. If the timer is still running through its first delay, don't allow the form to close yet. As soon as the timer is done, it will be disabled, which will allow the form to close.

The last thing we have to do is modify the project code just a bit to pay attention to the CloseQuery property. Add the following code just before the `SplashScreen.Close` command in the project source:

```
repeat
  Application.ProcessMessages;
until SplashScreen.CloseQuery;
```

When the application gets to this point, it will stay in this repeat loop until the SplashScreen's CloseQuery call permits it to continue. If your application takes longer to load then the SplashScreen's delay setting, then the SplashScreen will close immediately when it gets to this point.

Return to Tips & Tricks
Return to Front Page



Limiting Multiple Instances Of Your Program

by Robert Vivrette - CIS: 76416,1373

In Delphi 1.0 and other 16-bit programming environments it was quite simple to determine if another instance of your application was already running. The variable **hprevinst** held a value that defined if another copy was running, and by examining this value you could prevent a second copy from launching. With the 32-bit version of Delphi, the problem runs a little deeper and cannot be solved this way. 32-bit applications all run in their own address space and so you could have multiple copies of an app running and each would think it was the only one. There are occasions however when you may want to restrict multiple instances of your program. The project below demonstrates how this is done.
Essentially, this technique scans the Windows environment looking for other applications that might be duplicate copies. When you launch the application, it uses the **EnumWindows** function along with a user-defined callback function to look at each window currently present. For each window, we look to see if it has the same class name and same window title as the application that is attempting to launch. We should expect to see at least one window that matches this criteria… namely the application that is performing the test (i.e. it is seeing itself). We will always see this one because the application has already been created by the time we conduct the test.

If there are any additional windows present that match this search criteria, then we can assume that a copy of the application is already running and we will terminate the one that is making this test. As a side effect of how I programmed this, you can modify the **AllowedInstances** constant to permit, say, two or three copies of the program to run before preventing others. Not that many people will ever do this, but it is there nonetheless.

Delphi applications will have a class of 'TApplication' unless it has been changed somehow. Their name will be the name of the program, not the name of the main form. For example, if your project source said '`Program Project1`' at the top, the applications 'name' or window title would be '`Project1`'. This behavior exists because Delphi applications create a phantom window that is the 'Application'. The main form is a child window of this application window.

If we find that a copy of the application is running, it would be appropriate to bring it to the top of the Window Z-order so that the user can see it. Remember, the user might not be aware that another copy is running so bringing it to the top will make this clear. If this step was not done, an application could be minimized and the user would be clicking away on the application icon trying to launch a copy and nothing would be happening.

```
program Project1;

uses
  Forms, Windows, SysUtils,
```

```
  unit1 in 'unit1.pas' {Form1};

{$R *.RES}

const
  AllowedInstances = 1;

var
  MyAppName   : Array[0..255] of Char;
  MyClassName : Array[0..255] of Char;
  NumFound    : Integer;
  LastFound   : HWnd;
  MyPopup     : HWnd;

function LookAtAllWindows(Handle: HWND; Temp: LongInt): BOOL; stdcall;
var
  WindowName : Array[0..255] of Char;
  ClassName  : Array[0..255] of Char;
begin
  // Go get the windows class name
  if GetClassName(Handle,ClassName,SizeOf(ClassName)) > 0 then
    // Is the window class the same?
    if StrComp(ClassName,MyClassName) = 0 then
      // Get its window caption
      if GetWindowText(Handle,WindowName,SizeOf(WindowName)) > 0 then
        // Does this have the same window title?
        if StrComp(WindowName,MyAppName)=0 then
          begin
            inc(NumFound);
            // Are the handles different?
            if Handle <> Application.Handle then
              // Save it so we can bring it to the top later.
              LastFound := Handle;
          end;
end;

begin
  NumFound := 0; LastFound := 0;
  // First, determine what this application's name is
  GetWindowText(Application.Handle,MyAppName,SizeOf(MyAppName));
  // Now determine the class name for this application
  GetClassName(Application.Handle,MyClassName,SizeOf(MyClassName));
  // Now count how many others out there are Delphi apps with this title
  EnumWindows(@LookAtAllWindows,0);
  if NumFound > AllowedInstances then
    // There is another instance running, bring it to the front!
    begin
      MyPopup := GetLastActivePopup(LastFound);
      // Bring it to the top in the ZOrder
      BringWindowToTop(LastFound);
      // Is the window iconized?
      if IsIconic(MyPopup) then
        // Restore it to its original position
        ShowWindow(MyPopup,SW_RESTORE)
      else
        // Bring it to the front
        SetForegroundWindow(MyPopup);
    end
  else
    // None running - allow this instance to continue
    begin
      // This is the code that normally would be in the project source
      Application.Initialize;
```

```
      Application.CreateForm(TForm1, Form1);
      Application.Run;
    end
end.
```

Return to Tips & Tricks

Return to Front Page

Core Concepts with Delphi - Enumerated Types

by Alan G. Labouseur

Greetings and good day! The topic for today is enumerated types. Excited? We'll define the term, look at some examples of enumerated types, and then use them in a program to demonstrate some of their attributes. (There will be a few pointers along the way about functions and Delphi's case statement, along with a galactic principle of programming, so stay alert!)

You may be wondering just what exactly an "enumerated type" is. Well, let's look at the term itself. First, there's the "enumerated" part. That must mean that we're listing, itemizing, tallying, ticking off, specifying, or well . . . enumerating something. Then there's the "type" part. "Type" is a key word in Delphi representing the area of our programs or subprograms where we declare some data-type related information to the compiler. Delphi has several built-in types: char, string, integer, byte, etc. An enumerated type is where we get to make up our own data type! Really, it can be anything we want. We just have to list, or enumerate, every legal element of our new, made-up type.

A classic example of this is an enumerated type representing the days of the week. On Earth, there are seven legal values for this data type. The declaration to the compiler for such a type would look like this:

```
type DayOfWeek = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

Once we do this, we can use DayOfWeek just as if Delphi had it in there from the moment we tore off the shrink wrap. Of course we'll need a variable of this type so we can store and retrieve values. The syntax for that declaration is the same as if we were declaring any built-in data type.

```
var today: DayOfWeek;
```

If we had declared the variable "today" as an integer, we could store values like 7 and 5280 into it. But we declared "today" as a DayOfWeek. So the only values we can assign to it are those listed (enumerated) in it's type declaration. For example,

```
today := Mon;
```

is a perfectly legal assignment. While

```
today := 5280;
```

is not. There is no 5280 in the enumerated list of legal values for this type. Delphi will flag this type mismatch error when you try to compile it.

Now that we can assign values to variables of our new type, how do we retrieve them? This is where the fun starts. While Delphi is nice enough to give us the facility to make up any wacky data types that we want to, it isn't nice enough to supply us with the facility to print them out. This is not really Delphi's fault, however. If you think of writeln as a procedure, and remember that procedures require type declarations of their parameters (you did read the last issue, didn't you?), then it logically follows that writeln could only be available for the data types known in advance: the built-in data types. (Fascinating...) What this means is that we'll have to construct an output routine of our own. Here it is.

```
Function WordDay( TargetDay: DayOfWeek ): string;
```

```
var
   dayStr: string;
Begin
   case TargetDay of
      Mon : dayStr := 'Monday';
      Tue : dayStr := 'Tuesday';
      Wed : dayStr := 'Wednesday';
      Thu : dayStr := 'Thursday';
      Fri : dayStr := 'Friday';
      Sat : dayStr := 'Saturday';
      Sun : dayStr := 'Sunday';
   else
      dayStr := 'Unknown';
   end; { case }
   result := dayStr;   { Note the implicit identifier here. }
End; { Procedure WordDay }
```

I built this output function to accept as a parameter the day to print. Note that it is of the data type "DayOfWeek". The local variable "dayStr" will be used to hold the text I want returned. I used a case statement to map all possible values to appropriate strings. This string value is returned using the "result" construct of functions. Since writeln handles string variables quite well, we'll be able to see DayOfWeek values.

(Aside Number One: We have yet to discuss case statements here in Core Concepts. I'm sorry to throw this at you without background. Think of it as homework. Note the nice structure of Delphi's case statement and the wonderful feature of the else clause. It should be impossible for us to ever get an "Unknown" result. But don't count on it. Programs get complex quickly, and things that you never anticipated will happen. So plan for them and program defensively. It's better to have your user ask you how come "Unknown" comes after "Sunday" than have your program blow up and crash the system because of a runtime error.)

(Aside Number Two: Since a function always needs to return a value (of the type specified in the function declaration), Delphi provides us with a convenient mechanism to do this. Every function has an implicitly declared identifier called "result". You can use this implicitly declared identifier to store the result you want returned from the function. It's quite nice for several reasons, my favorite being that you can insure that there is one and only one exit point from your subprogram. That is a galactic principle of good programming: one way in and one way out.)

But wait! There's more! Delphi provides some special functions for the manipulation of enumerated (and other) types. One such beast is the successor function. The key word "succ" signifies to the compiler the function that will return to you the next element of the enumerated type. For example,

```
today := succ(Mon);
writeln( WordDay(today) );
```

results in the output of "Tuesday". The converse of succ is the "pred" function. (Predecessor.)

```
today := pred(Thu);
writeln( WordDay(today) );
```

The output here is "Wednesday". You cannot wrap around. pred(Mon) and succ(Sun) will yield range errors. So be careful. There is another function that will give you the element's zero-based index in the enumerated list. It's called "ord". It stands for "ordinal value". The ordinal value of Mon is 0. The ord of Sun is 6.

Here's a program that works though what we've discussed here today. Let's go through it and examine the output.

```
01. program Sets;

02. uses WinCrt;
```

```
03. type
04.    DayOfWeek = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
05. var
06.    today: DayOfWeek;

07. Function WordDay( TargetDay: DayOfWeek ): string;
08. var
09.    dayStr: string;
10. Begin
11.    case TargetDay of
12.       Mon : dayStr := 'Monday';
13.       Tue : dayStr := 'Tuesday';
14.       Wed : dayStr := 'Wednesday';
15.       Thu : dayStr := 'Thursday';
16.       Fri : dayStr := 'Friday';
17.       Sat : dayStr := 'Saturday';
18.       Sun : dayStr := 'Sunday';
19.    else
20.       dayStr := 'Unknown';
21.    end; { case }
22.    result := dayStr;  { Note the implicit identifier here. }
23. End; { Procedure WordDay }

24. Begin  { Program }
25.    { Let's check out pred and succ... }
26.    today := Mon;
27.    writeln( WordDay(today) );
28.    today := succ( today );
29.    writeln( WordDay(today) );
30.    today := pred( today );
31.    writeln( WordDay(today) );
32.    { ... and test the ordinal limits. }
33.    writeln( ord(Mon) );
34.    writeln( ord(Sun) );
35. End.  { Program }
```

In line 4 we declare the DayOfWeek enumerated type. In line 6 we create the global variable of that type.

Line 7 begins our function to map DayOfWeek values into printable strings. (Note that the data type of the return value is to be a string.) We are receiving a DayOfWeek value as a parameter and assigning it the identifier "TargetDay". But before we can return anything, we first must declare a local variable ("dayStr", on line 9) to hold our return value until the end of the function. (See galactic principle, above.) On line 11 we begin the case statement which assigns the appropriate string value to dayStr. On the off and unplanned chance that we got a DayOfWeek that wasn't accounted for in the body of the case statement, we will deftly handle it with the else clause on line 19 and subsequent assignment on line 20. (This is the part that impresses family and friends.) Line 22 stores the string value into the implicitly-declared result identifier and the function returns this value.

Line 24 marks the start of the program itself. First we assign the value 'Mon' to the global variable of type DayOfWeek called "today". On line 27 we call the built-in writeln procedure and pass to it as a parameter the value returned from our WordDay function. All is cool since WordDay returns a string and writeln likes strings. So WordDay is called, maps 'Mon' as a DayOfWeek to 'Monday' as a string, and returns that value to writeln. 'Monday' is written to the screen.

We advance one day in the week on line 28 as "today" is assigned the successor of it's own current value. The successor to 'Mon' is 'Tue'. Now when we writeln the result of WordDay again, 'Tuesday' is the output.

Line 30 moves us back to 'Mon' as a result of our call to the predecessor function. Line 31 proves it by once again invoking WordDay to show us the string representation of the DayOfWeek value stored in

"today".

After getting bored with pred and succ we move on to line 33 and the ord function. Here we are asking to writeln the ordinal value of 'Mon'. It is important to note that enumerated types in Delphi, as well as most of its other structures, are zero-based. That is to say that the first element is zero, not one. Realizing that, it's no surprise that the output from line 33 is 0. Care to guess at the output of line 34? If you guessed 6, you're right.

Line 35 ends the program, and this discussion too. Well, almost. Before I go, I'd like to take this opportunity to point out one of those cool things about computer science and Delphi and why things are the way they are. Recall that the Boolean data type consists of True and False. Those are the only two values. True, in math and in binary code, is represented by 1, false by 0. Consider this enumerated type declaration for Boolean:

```
type Boolean = ( False, True );
```

So the ordinal value of False is 0, and ord(True) is 1. Pretty neat, huh?

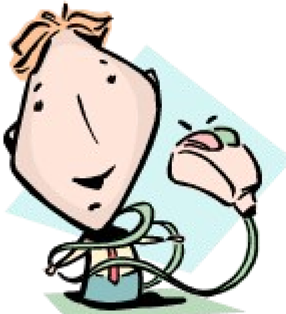
Enumerated types are an important asset in your programming arsenal. They allow you to more accurately model real-world data and expand the constructs of Delphi. And your program code is more readable and intuitive when you use them. So think of some fun examples and practice with enumerated types.

I hope this has been helpful in providing some core information upon which we can build in the near future. I'll have a "putting it all together" article in an upcoming issue. Stay tuned. In the mean time, feel free, encouraged even, to e-mail me with any comments, questions, or suggestions. Thank you and goodnight.

About the Author

Alan G. Labouseur is Vice President and co-owner of AlphaPoint Systems, Inc., a custom programming consultancy located in Brewster, NY. Alan has a Masters degree in Computer Science and has been developing custom software for ten years. He is currently working on Delphi applications for all flavors of Windows on Netware and NT networks for clients in the NY-NJ-CT area. You can reach Alan through e-mail at AGL007@IX.NETCOM.COM or 70312,2726 on CompuServe.

Return to Front Page



How to Restrict The Mouse

by Robert Vivrette - CIS: 76416,1373

A mouse can be an unruly creature and sometimes it is necessary to fence him in! Using the Windows API function **ClipCursor**, it is possible to restrict the movement of the mouse to a specific rectangular region on the screen. Using the Rubber Banding tip mentioned earlier as an example, I have added 3 lines of code to the MouseDown procedure as follows:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton; Shift:
TShiftState; X, Y: Integer);
var
  R : TRect;
begin
  Canvas.Pen.Mode := pmXOr;
  Canvas.Pen.Style := psDot;
  X1 := X; Y1 := Y; X2 := X; Y2 := Y;
  R := BoundsRect;
  InflateRect(R,-30,-30);
  ClipCursor(@R);
end;
```

First, I make a copy of the BoundsRect property of the form and store it in the TRect variable 'R'. Then I use InflateRect to shrink the size of the rectangle by 30 pixels on each side. Then you simply pass this TRect into the ClipCursor routine. When the user presses the mouse down over the form, the ClipCursor routine then establishes a rectangle that will prevent the mouse from leaving the form (actually it can't get any closer than 30 pixels to any edge of the form).

Of course we can't leave the program like that, we need to have some mechanism to release this restriction. All we do is add a MouseDown handler as follows.

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  ClipCursor(Nil);
end;
```

Calling ClipCursor with a NIL parameter releases any mouse clipping that may be in effect. Make sure you do this, or you won't be able to regain control of your mouse!

Return to Tips & Tricks

Return to Front Page

Extended Ini-Component

by Ferdinand Soethe - F.SOETHE@OLN.comlink.apc.org

When writing an application, it's always important to me to not force my users to repeatedly set options that could be stored and restored by the application. To accomplish this 'preservation of state', I have always made extensive use of INI files. And it's always bothered me that I had to write a lot of code just to store and restore the current values of menu items, checkboxes etc.

So I have finally written a generalized solution to this problem. The new class, TExtIniF, is derived from Borland's TIniFile and inherits the basic mechanisms of INI file processing. On top of that it adds a mechanism to register interface objects such as menu items or checkboxes and procedures to store and restore their stored properties. To simplify things even further, I have added procedures that automatically store and restore all registered objects with just one call.

How does it work?

The idea behind TExtIniF is quite simple. On creation it will attach to an INI file and wait for you to register interface objects. When you do (by calling RegisterObject) it stores the object in the object property of an internal TStringList (FRegObjects).

Once objects are registered they can be stored and restored with just a call to StoreObjectStates and RestoreObjectStates. Internally these methods will simply process all objects in the FRegObjects lists and call the methods StoreObjectState (note there is no trailing 's') and RestoreObjectState.

The tricky part is the processing within these methods. To be able to deal with all kinds of different interface objects, StoreObjectState and RestoreObjectState use Delphi's Run Time Type Info-System (RTTI) to determine the type of an object and act accordingly. Let's look at a piece of code to get a better understanding:

```
if (obj is TCheckBox) then
  begin
    {Checkboxes: store checked state}
     with (obj as TCheckBox) do
       writeBool(IniSection,Name,checked);
  end
else
  ...
```

In this snippet from StoreObject we use **is** to check wether the current object (obj) is a TCheckBox. If that is the case we use **as** to cast *obj* like an object of that type in the following with statement and execute whatever methods we need to store the property we want. In this particular case, we use the inherited method WriteBool to store the checked-property of our object.

Do we need with?

Although it looks rather complicated, I couldn't find any better way than using a **with** statement. That's because there are three references to our object and without the bracket each of them would have needed to do an **as** conversion. So instead of three simple lines you would have something like:

```
(obj as TCheckBox).checked:= ReadBool(IniSection,
                                      (obj as TCheckBox).Name,
                                      (obj as TCheckBox).checked);
```

Sections and Keys

Looking at the code above, you can already see, that the class uses the name of an object as a key for it's INI entry. That's convenient because each component has a unique name anyway, so you don't have to think one up.

The section where an entry is stored depends on how the object was registered. When you pass an empty string to RegisterObject, TExtIniF will always store the entry in the default section. The default

section-name is represented by the IniSection-property and set to the default-value 'Options' on creation.

If you pass something other than an empty string, this object will always be stored and restored from the section name you have given.

Changing the IniSection-property

You can change this property any time, but watch out for the results of doing this at different times:

> changing it before storing or restoring an object will make all objects that were registered without INI-section be stored and restored to/from a different section in the IniFile. This is probably the most useful option.
> changing it between a restore- and a store-operation will copy all keys to a new or different INI section.
> changing it between a store- and a restore-operation will save all keys to the current Ini-Section and reload them from a new or different one.

AutoStore-property

Set AutoStore to true, to make TExtIniF take care of saving all object states when it is destroyed. That way you never have to call StoreObjects yourself, just free the TExtIniF-object and all is fine! .

Storing and restoring individual Objects

Sometimes you may want to store or restore the value of an object apart from all the other. I.e. to allow users to save the setting of an object at a certain point in time. In this case you can call the methods StoreObjectState and RestoreObjectState on their own. Just watch handling of the INI-Section in this case:

If you pass an empty string for INI-Section, the object's state will be stored and restored in/from the current default section. If you pass a string, the object will be stored in a section of that name, even if you registered it with a different name!! To store or restore an object in the section that is was registered for, use the GetIniSection-method to find the proper section name. I.e.

```
with myExtIniF do begin
  RestoreObjectState(myField,GetIniSection(myField));
end;
```

By the way… GetIniSection returns FIniSection if you use it on unregistered objects.

Processing unregistered objects

RestoreObjectState and StoreObjectState can also handle unregistered objects. Just call them with the object to quickly save or restore their state.

Unregistering an object

Use the UnRegisterObject method to remove an object from the list.

Why not a component?

Everything peace of code for Delphi seems to be made a component these days. And while I appreciate components for visible objects, I just can't see any value in cluttering my forms with icons for invisible objects. Apart from that I'm always annoyed when I have to recompile my library just to try out some new component. So TExtIniF is not a component but just a class.

Possible problems

One possible problem occurs, if objects are destroyed before StoreObjectStates is called. I'm checking with

```
(obj.classinfo <> nil)
```

If an object exists, since Delphi doesn't reset an object variable to nil when the object is freed. Any

suggestions for a better solution would be much appreciated but so far this has worked fine.

Source for EXTINIF.PAS
Return to Front Page

EXTINI Source

```
{************************************************************
 *  TExtIniF class                                          *
 *  created and copyright by Ferdinand Soethe 1996          *
 *  (email: f.soethe@oln.comlink.apc.org)                   *
 *                                                          *
 *  You may use this source code in your applications       *
 *  (a word of credit would be nice) but you must not       *
 *  resell it as part of a library or publish it in paper-  *
 *  or electronic form without asking my permission.          *
 *                                                          *
 *  TExtIniF extends Delphi's TIniFile to simplify saving a  *
 *  components state to an INI-File. After creation you can  *
 *  register any number of components and save and retrieve  *
 *  their settings with just one call to StoreObjectStates.  *
 *                                                          *
 ************************************************************}

unit extINIF;

interface

uses
  {unfortunately we need to include a units of classes
    that we want to be able to store}
  IniFiles, Classes, Forms, StdCtrls, FileCtrl, Menus,
  SysUtils, TabNotBK;

type
  EExtIniFError = class(Exception);
  TExtIniF = class(TIniFile)
  private
    { Private-Deklarationen }
    {store all objects states before TExtIniF is destroyed}
    FAutoStore: boolean;
    {list of all registered objects}
    FRegObjects: TStringList;
    {Name of [section] where values are stored}
    FIniSection: String;
  protected
    { Protected-Deklarationen }
  public
    { Public-Deklarationen }
    constructor create(IniFName: TFileName);
    destructor destroy; override;
    {find the ini section for a registered object}
    function GetIniSection(obj: TObject): string;
    {Add a component to the list of objects}
    procedure RegisterObject(obj: TObject; INISection: string);
    {Remove a component to the list of objects}
    procedure UnRegisterObject(obj: TObject; INISection: string);
    {Retrieve the setting of a single Object}
    procedure ReStoreObjectState(obj: TObject; INISection: string);
    {Restore states of all registered objects}
    procedure RestoreObjectStates;
    {Restore states of all registered objects}
    procedure StoreObjectState(obj: TObject; INISection: string);
    {Store state of a single object}
    procedure StoreObjectStates;
    {Store states of all registered objects}
  published
```

```
    { Published-Deklarationen }
    property AutoStore: boolean read FAutoStore write FAutoStore;
    property IniSection: string read FIniSection write FIniSection;
  end;

implementation

function ExtractFileBaseName(FullName: string): string;
{return the whole path without the extension}
var
  endPos: integer;
begin
  result := FullName;
  endPos := length(result);
  repeat
    dec(endPos);
  until result[endPos] = '.';
  delete(result,endPos,maxInt);
end;

{ find the section string to a registered object
   if not registered or section string is empty
   return default value}
function TExtIniF.GetIniSection(obj: TObject): string;
var
  index: integer;
begin
  index:= FRegObjects.indexOfObject(obj);
  if ( index > -1 ) then
    begin
      result:= FRegObjects.strings[index];
      if result = '' then
        result:= FIniSection;
    end
  else
    result:= FIniSection;
end; {GetIniSection}

constructor TExtIniF.create(IniFName: TFileName);
begin
  {if you don't pass your own name for the ini-File, it will be the name
   of your exe-file with the extension '*.INI'}
  if (IniFName = '') then
    IniFName:= ExtractFileBaseName(application.exename)+'.ini';
  inherited create(IniFName);
  FRegObjects:= TStringList.Create;
  FIniSection:= 'Options';
end;

destructor TExtIniF.Destroy;
begin
  {If AutoStore is set, values are stored
   before TExtIniF-Object is destroyed}
  if FAutoStore then StoreObjectStates;
  FRegObjects.destroy;
  inherited destroy;
end;

{ Add an object to the list of monitored objects. If you pass an empty string
   for INISection, the default value will apply and no name will be stored}
procedure TExtIniF.RegisterObject(obj: TObject; INISection: string);
begin
  {check if object is already registered}
```

```
  if (FRegObjects.indexOfObject(obj) = -1) then
    FRegObjects.addObject(INISection,obj);
end;

{ Remove an object from the list of monitored objects.}
procedure TExtIniF.UnRegisterObject(obj: TObject; INISection: string);
var
  index: integer;
begin
  index:= FRegObjects.indexOfObject(obj);
  if (index > -1) then FRegObjects.delete(index);
end;

{ Restores the name of an object from the INI-File
   Note: When there is no entry in the INI-File, the object's value
          is not changed.}
procedure TExtIniF.ReStoreObjectState(obj: TObject; INISection: string);
var
  strBuf: string;
begin
  if ( INISection = '' ) then INISection:= FIniSection;
  {the next lines check for the type of object and
   restore whatever property we would like to store of that object
   if you make changes here you will need to make changes in
   StoreObjectState as well!!!}
  if (obj.classInfo <> nil ) then
    begin
      if (obj is TCheckBox) then with (obj as TCheckBox) do
        {Checkboxes: restore checked state}
        checked:= ReadBool(INISection,Name,checked)
      else if (obj is TEdit) then with (obj as TEdit) do
        {Editfield: restore text}
        text:= ReadString(INISection,Name,text)
      else if (obj is TMenuItem) then with (obj as TMenuItem) do
        {Menuitem: restore checked state}
        checked:= ReadBool(INISection,Name,checked)
      else if (obj is TTabbedNoteBook) then with (obj as TTabbedNoteBook) do
        {Notebook: restore open Tab}
        pageIndex:= ReadInteger(INISection,Name,pageIndex)
      else if (obj is TDriveComboBox) then with (obj as TDriveComboBox) do
        begin
         {DriveCombo: restore selected drive}
         strBuf := ReadString(INISection,Name,Drive);
         Drive := strBuf[1];
        end
      else if (obj is TDirectoryListBox) then with (obj as TDirectoryListBox) do
        {DirectoryList: restore current directory}
        Directory:= ReadString(INISection,Name,Directory);
    end
  else
    raise EExtIniFError.create('This object is not supported!');
end;

{ Restores the state of all registered objects from the INI-File}
procedure TExtIniF.RestoreObjectStates;
var
  objNo: integer;
begin
  {iterate through all registered objects}
  for objNo:= 0 to FRegObjects.count - 1 do
    ReStoreObjectState(FRegObjects.objects[objNo],FRegObjects.strings[objNo]);
end;
```

```
{ Stores the state of an object to the INI-File}
procedure TExtIniF.StoreObjectState(obj: TObject; INISection: string);
var
    strBuf: string;
begin
  if ( INISection = '' ) then INISection:= FIniSection;
  {the next lines check for the type of object and
   store whatever property we would like to store of that object
   if you make changes here you will need to make changes in
   ReStoreObjectState as well!!!}
  if (obj.classInfo <> nil ) then
    begin
      if (obj is TCheckBox) then with (obj as TCheckBox) do
        {Checkboxes: store checked state}
        writeBool(INISection,Name,checked)
      else if (obj is TEdit) then with (obj as TEdit) do
        {Editfield: store text}
        writeString(INISection,Name,text)
      else if (obj is TMenuItem) then with (obj as TMenuItem) do
        {Menuitem: restore checked state}
        writeBool(INISection,Name,checked)
      else if (obj is TTabbedNoteBook) then with (obj as TTabbedNoteBook) do
        {Notebook: restore open Tab}
        writeInteger(INISection,Name,pageIndex)
      else if (obj is TDriveComboBox) then with (obj as TDriveComboBox) do
        {DriveCombo: restore selected drive}
        writeString(INISection,Name,Drive)
      else if (obj is TDirectoryListBox) then with (obj as TDirectoryListBox) do
        {DirectoryList: restore current directory}
        writeString(INISection,Name,Directory)
      else
        raise EExtIniFError.create('This object is not supported!');
    end;
end;

{ Stores the state of all registered objects to the INI-File}
procedure TExtIniF.StoreObjectStates;
var
  objNo: integer;
begin
  for objNo:= 0 to FRegObjects.count - 1 do
    StoreObjectState(FRegObjects.objects[objNo],FRegObjects.strings[objNo]);
end;

end.
```

Return to Article

Return to Front Page