The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Well, I guess there ***were*** people listening out there! The UNDU Awards ballots are streaming in... I thank you all very much. I must say however, that the number of votes is sufficiently great now to warrant some kind of program that will track and report them all. I am within a few days of finishing this application and getting the data all in, but I did not want to hold up the issue another week. As a result, I will be reporting the results next issue (right... we've heard ***that*** before!). Sorry about that folks!

Over the past several months, there has been steadily increasing interest for UNDU on the Internet. The web site that originally housed the newsletter seems to have lost some of its momentum, and I have arranged for the newsletter to have a permanent home on the web. The only thing that is holding me back right now is the process of converting the old issues to HTML web pages. The new home is on the Informant Communications web site at **http://www.informant.com**. Sometime within the next month, you should start seeing UNDU issues materializing on that site and I should know an exact address to the issues by next issue. The newsletters will also be available on their FTP site as the usual Windows Help files.

If anyone has suggestions about how to improve UNDU on the internet, by all means drop me a line!

- Robert

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Tips & Tricks

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Tip For Edit Boxes

by Richard Marks - Internet: rmarks@cix.compulink.co.uk

You may have come across the disappearing caret trick! Often when you have a series of Edit Boxes you want to use the OnExit event to do some validation. Sometimes, though, your OnExit routine may need to display a messagebox. When you do this you'll find that after closing the message box, the caret (or cursor) has disappeared!

To see this bug/feature of Delphi, just create 3 or 4 edit boxes, and then in one of them put the code:

```
procedure TForm1.Edit1Exit(Sender: TObject);
begin
  showmessage('The caret will now disappear');
end;
```

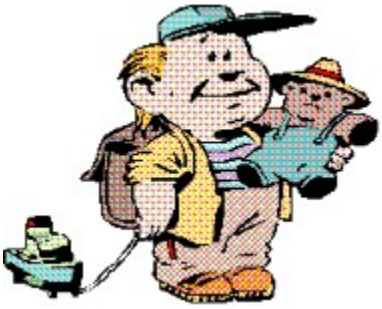
Run it and see what happens. The cure is to get Windows to move the focus to the next control and then straight back again. Here is a piece of code which illustrates how to do it.

```
procedure TForm1.Edit1Exit(Sender: TObject);
begin
  showmessage('Your caret will not disappear this time');
  postmessage(handle,WM_NEXTDLGCTL,0,0);
  postmessage(handle,WM_NEXTDLGCTL,1,0);
end;
```

Return to Tips & Tricks

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996



The Beginners Corner - The Art of Dialog - Part I

Paul H. Comitz - CIS: 102565,3167

The other day I saw someone use a calculator to find the sine of Pi. This might be expected from a lawyer or a stockbroker. Problem is I work at an engineering company. I hope no one ever asks us to do long division during a battery shortage.

It occurred to me that we're starting to use software the same way. We depend on our tools so heavily that we forget, or never even learn, basic skills. With tools like Delphi we're faced with an interesting challenge: use the tools to our full advantage, without becoming so dependent on the tools that we can't express ourselves. Would Perlman be Perlman if he could only play Stradivari???

This months ***Beginners Corner*** tries to find that elusive middle ground. If you look under the "Dialogs" Tab on the Delphi Components Palette, you'll find some extremely powerful and useful Dialog boxes. These common dialogs are File Open, File Save, Font Dialog, Color Dialog, Print, Printer Setup, Find, and Replace. If you need something quickly, with a standard look and feel, these "tools" can't be beat. You can insert one of these dialogs into you application in minutes. A clear an obvious advantage gained by using products like Delphi.

As powerful as these tools are, suppose you need dialogs, that have a different look and feel? It DOES happen. I am using Delphi to create dialogs for just such a project out in the "real" world. We're not necessarily doing anything differently with our dialog boxes, but our user has HMI requirements (the politically correct term for MMI - Man Machine Interface) that are derived from a legacy system. Our user wants their new dialogs to look like their legacy dialogs.

Fortunately, Delphi provides the individual components necessary to create your own dialogs. Our project is to create an Open File dialog using the individual components supplied by Delphi. We'll capitalize on the power of Delphi, without being constrained by the built-ins.

Step 1: Assemble Your Tools

You'll need the following to create your own Open File Dialog:

```
TFileListBox
TDirectoryListBox
TLabel
TDriveComboBox
TEdit
TFilterComboBox
```

TFileListBox, TDirectoryListBox, TDriveComboBox, and TFilterComboBox are located under the System tab on the component palette. TLabel and TEdit are located on the standard tab. Since this is a project column this month, I won't explain what each of these are. However, if you don't know, stop right now and read the online help for each of these components. It'll take you all of 10 minutes and you'll actually know what your doing when your done. Please, please, please, don't drag and drop without knowing what and why. By knowing what your doing your contributing to the overall good karma of the universe. Stop and read NOW.

Plunk all these components down on a form. Go ahead and play with the placement of the components. It's OK, nothing bad will happen. Open File dialogs are usually kind of oblong, longer than they are high with two distinct columns of information. One column usually contains an Edit Box to display the file name, a File List Box and a Filter Combo Box. The other column usually contains a label that displays the path and file name, a Directory Combo Box, and a Drive Combo Box. Rather than try to visualize this, go ahead an display the Delphi File|Open Project Dialog.

I decided to make my dialogs tall and skinny. I like mine like this because my users often use long paths on network volumes with long IDs. By making the dialog box a bit taller I can stack the Edit Box, the Filter Combo Box, and the Drive Combo Box, on top of one another. I can make loner - the entire width of the dialog box. Even though my dialog boxes are skinnier, since the the individual components are longer, I can display longer path names and volume names. Those of you who are using Windows 95 with an inherited DOS file structure know what I mean. For example my Windows 95 path name for this project is **C:\windows\desktop\delphi\skinny\PCDialog.dpr**. Clearly our old style dialogs weren't meant for paths like this!

Step 2: Build the Team

Once you've laid out your components it's simple to get them to work together. Most of this work is done by setting properties on the Object Inspector. If you run this project right now, you'll notice that the Drive Combo Box and the Directory List Box are already partly functional. You can click on the Directory List Box and move through your directory structure. You can click on the Drive Combo Box and select whatever storage devices are resident on your system. Notice that you can even select network volumes and devices. However, the components are not aware of one another - they don't work together. Start integrating the team as follows:

A. Select the Drive Combo Box on your form. On the Object Inspector select the "DirList" property. Select DirectoryListBox1 from the list of allowable components that Delphi has thoughtfully provided for you. Now run the program again. The Directory List Box changes to display the directory structure of the storage device you select in the Drive Combo Box.
B. Select the Directory List Box on your form. On the Object Inspector select the "FileList" property. Select (you guessed it) FileListBox1. This causes the files form the directory the user selects to be displayed in the File List Box. Go ahead and run the program. The Drive Combo Box, the File List Box, and the Directory List Box are all "aware" of each other.
C. Select the File List Box on your form. On the Object Inspector select the "FileEdit" Property. Select Edit1 from the list that Delphi has once again so thoughtfully provided. This ties the Directory List Box to the Edit box you previously placed on your form. If you run the program now, you'll note that clicking a file in the file list box causes the file to be displayed in the Edit box.

Coming in Part II

So far we've allowed the user to select existing files by clicking the storage device in the Drive Combo Box, a directory in the Directory List Box, and a File in the File List Box. Next month we'll expand the functionality by using the ApplyFilePath method to enable the user to manually enter a file and a path directly into the Edit Box. We'll also show how to use the Filter Combo Box to tailor our dialog to specific file types. Finally, we'll show how to open and use the file the user selects. In the meantime, read the online help, and experiment, experiment, experiment!!!

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

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 #7 - August 31, 1995

Issue #8 - October 10, 1995

Issue #9 - November 9, 1995

Issue #10 - December 12, 1995

Issue #11 - January 18th, 1996

Issue #12 - Feburary 23rd, 1996 (This issue)

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 3rd Party" file section at the same time.

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Drawing Windows Metafiles in Delphi

by Grahame S. Marsh - Internet: grahame.s.marsh@corp.courtaulds.co.uk

In this article I will show you how I set about drawing Windows Metafiles in Delphi. The need to write Metafiles came about from a project which, in part, generated some graphical output. My first solution was to simply create a bitmap, draw the required output on the bitmap's canvas, and then save the bitmap to a file. Something like this fragment:

```
var
  MyBitmap: TBitmap;
.
.
.

MyBitmap := TBitmap.Create;
with MyBitmap do
  try
    Width := 100;
    Height := 100;
    with Canvas do
      begin
        Rectangle(0, 0, 100, 100);
        MoveTo(0, 0);
        LineTo(100, 100);
        MoveTo(0, 100);
        LineTo(100, 0);
      end;
    SaveToFile('test.bmp')
  finally
    Free;
  end;
```

This humble example generates an output file 11078 bytes long. The output from my real application was generating bitmaps hundreds of kbytes in size. In order to be more user-disc-space-friendly I decided to look at Windows Metafiles as a space-saving alternative.

For the uninitiated, a Windows Metafile is a collection of GDI (graphics device interface) functions that describe an image. They should in theory take up less space and should be more device-independant than bitmaps. Practically, it is not necessary to understand how the Metafile is stored, only that certain key commands are needed to save and load the Metafiles.

The Metafile has been encapsulated in Delphi with the TMetafile object. This contains methods to load and save Metafiles; the TCanvas object allows you to draw a Metafile on it and the Clipboard unit recognises a Metafile allowing you to copy and paste Metafiles from the Clipboard. *But*, and this is a pretty big but, the TMetafile object does not include a canvas - you cannot draw on it yourself. Having said this, in such a categorical manner, I do get nervous, these things always do seem to be possible with Delphi: The expert comes along and says, "Oh, yes, you do it this way.....", and your carefully crafted hundred lines or so of code get reduced to two or three. Anyway I shall put my head on the block, and show how I overcame the (apparent) limitation of not being able to draw onto a TMetafile.

The GDI has a number of functions used to play with Metafiles which include:

```
function CreateMetafile(Filename: PChar): THandle;
function CloseMetafile(DC: THandle): THandle;
function DeleteMetafile(MF: THandle): Bool;
```

The CreateMetafile function takes a filename as a parameter, or nil to generate a Metafile in memory only. The function returns a device context that you can use to draw with. The CloseMetafile function saves the file (if used) and returns a handle to the Metafile object in memory. Finally, the DeleteMetafile function removes the Metafile from memory. So these can be used in the following way:

```
var
  DC: THandle;
.
.
.
DC := CreateMetafile('test1.wmf');
Rectangle(DC, 0, 0, 100, 100);
MoveTo(DC, 0, 0);
LineTo(DC, 100, 100);
MoveTo(DC, 0, 100);
LineTo(DC, 100, 0);
DeleteMetafile(CloseMetafile(DC));
```

This produces a Metafile on disc which is a satisfying 78 bytes long (the fact that it will be using a whole allocation block on disc and so will actually be occupying far more space is another story, but at least, the size reduction is clear in my example).

There are two problems with this approach. The first is that I have used GDI functions to do the drawing, whereas I would ideally want to use a TCanvas object to do it. Secondly, if you try to load this Metafile using the TMetafile.LoadFromFile method it protests, telling you that the Metafile format is invalid. This arises because the TMetafile object only recognises Metafiles with an Aldus format record header on the front; it does not recognise a "traditional" Metafile created this way. To overcome these two problems is quite simple. If the device context is put into the TCanvas handle property, then drawing using the Canvas takes place into the Metafile:

```
var
  MyCanvas: TCanvas;
  MyMetafile: TMetafile;
.
.
.
MyCanvas := TCanvas.Create;
try
  MyCanvas.Handle := CreateMetafile(nil);
```

The handle property correctly disposes of whatever device context was linked to it before taking on our Metafile device context. You now can draw on the Metafile canvas:

```
with MyCanvas do
  begin
    Rectangle(0, 0, 100, 100);
    MoveTo(0, 0);
    LineTo(100, 100);
    MoveTo(0, 100);
    LineTo(100, 0)
  end;
```

To overcome the second problem I found it easiest to let a TMetafile object do the saving to file for me - it correctly puts the Aldus header record on the Metafile. This is accomplished by putting the memory handle returned by the CloseMetafile function into the handle property of a TMetafile:

```
.
```

```
.
.
  MyMetafile := TMetafile.Create;
  with MyMetafile do
    try
      Handle := CloseMetafile(MyCanvas.Handle);
      SaveToFile('test2.wmf');
    finally
      Free;
    end;
finally
  MyCanvas.Free
end;
```

The TMetafile object correctly includes a call to DeleteMetafile when freeing the object so there is no need for us to call it. This generates a file 278 bytes in size; reflecting the Aldus record header and also the many GDI calls that the TCanvas object makes on your behalf which you don't see in your code. But even using the TCanvas object the savings is still certainly splended and worthwhile.

The Metafile generated by this means could not be read by all applications, notably the windows clipboard just put a single dot on the screen (I would guess that it was my object drawn very small). Other applications would draw it much larger. So I also found it necessary to set the Metafile's width, height and inch properties to get the Metafile to play correctly in a reliable fashion.

It appeared to me that I should encapsulate the drawing Metafile in a new object, a descendant of TMetafile. The complete source code for this is given below. The TDrawMetafile object operates in two modes, a drawing and a use mode. The protected boolean property *Drawing* reflects this, showing whether the object currently expects to be drawn on, or whether the resultant Metafile is made available for use. You cannot do both simultaneously. Whenever the object is put into drawing mode the Metafile canvas starts off blanked, so you cannot draw something, use the Metafile, draw a bit more, use it etc. When created the TDrawMetafile starts off in draw mode. I also provide two methods, Open and Close which put the DrawMetafile onto draw and use mode respectively. The constructor, Create, takes two parameters, the width and height of the image. So to use my well-worn example:

```
var
  MyDrawMetafile: TDrawMetafile;
.
.
.
MyDrawMetafile := TDrawMetafile.Create(100, 100);
with MyDrawMetafile do
  try
    with Canvas do
      begin
        Rectangle(0, 0, 100, 100);
        MoveTo(0, 0);
        LineTo(100, 100);
        MoveTo(0, 100);
        LineTo(100, 0);
      end;
    Close;
    SaveToFile('test3.wmf')
  finally
    Free;
  end;
```

To give a real-world example of a Metafile in use. The application I mentioned earlier is taking UK digital terrain data and drawing it as 3D view. A typical image stored as a bitmap occupied **382,438** bytes, whereas the same image stored as a Metafile occupied only **24,118** bytes. I rest my case.

The source code given here represents bare-bones functionality for a metafile drawing object. I would

welcome suggestion for enhancements and additions to improve it beyond its basic simplicity.
DrawMetaFile Source Code

Return to Tips & Tricks
Return to Front Page

Delphi Projects

A few issues back, I mentioned that I would like to hear from software developers who are doing work professionally with Delphi. It is very interesting to see the kinds of things that are being done, and occasionally, it could spark additional creative juices for new projects. If you are doing something interesting or unique with Delphi at work, please write and let me know. Every issue I will print a few of the pieces I receive...

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Shazam Review

Review by Robert Pullan - CIS:102162,2711

Double your pleasure double your fun ...

It has been my practice to review two related components per article, but I am making a point to focus upon one component in this article which is really two components in one and one that deserves your full attention.

Shazam is a Delphi component of considerable merit and worthy of serious consideration. Recently it has been upgraded to V2.0 and has matured nicely. In its most basic sense, Shazam is a query engine which can produces outstanding SQL calls... and it does a better job than the Visual Query Builder (VQB) component in the $2k Delphi C/S version. Those SQL calls can be used to power a very nice tabular report writer, or be fed to other data aware components (eg. Light Lib's Business Graphics or other products) to provide an interactive capability heretofore unavailable without tremendous effort. Additionally, unlike other report writers available to Delphi users, Shazam makes its power available to both end users and developers.

SHAZAM the Query engine

Shazam lacks the fancy visual table linking icons provided by VQB, but its interface is simple, straightforward and effective. More importantly, where the rubber meets the road, comparing Shazam to VQB, is like comparing a supercharged Ferrari race car to a stock Chevy.

To use the interface, after selecting data tables, one simply drags fields and relational information onto a two simple grids. Users can sort, group, hide, and filter fields quite powerfully. The result is clear and awesomely powerful SQL commands, and if that isn't enough Shazam AUTOMATICALLY configures its SQL calls for Paradox, xBase, Oracle, Sybase,MS SQL Server, Interbase, Informix and Watcom dialects. The Query Properties offers ways to create additional variants, if these standard dialects aren't sufficient to your needs!

Shazam is such a good SQL generator, I have stopped using VQB and use Shazam exclusively - even when the report writer is not involved. While this piece of Shazam is overshadowed by the report writer, it is in fact the real heart of this component! Without the awesome SQL query power made easy, the report writer would merely be a very nice component that allows end user access.

Not only can one link tables within a database, but also different format tables (eg. xBase and Paradox) from different databases (eg. different BDE aliases) at the same time! Shazam does depend entirely upon the TQuery component and BDE to perform this magic, so don't think it will do something BDE can't.

SHAZAM the Report Writer

This feature can add significant appeal to projects by adding an end user, ad hoc tabular reporting option to your app. The Shazam component contains a four notebook tabs.

1. The first tab creates a list of tables from which the report will draw data and selects the datafields and how they are to be displayed. This is done by dragging fieldnames from an outline box to a grid. My experience with untrained end users suggests this step is quite simple and almost intuitive.
2. The second tab provides a grid upon which relationships between the tables. End users had some difficulty in grasping the relational notions (a challenge faced anyone in their first exposure to RDMS concepts), but found the interface simple to use once they understood relational keys.
3. The third tab displays the needed SQL code. This tab can be hidden if one is delivering an end user product. I make frequent cut & paste use of this tab to create SQL requests for other apps.
4. The final tab displays the resultant report in WYSIWIG format. Output can be in printed reports or one can save the results in a Paradox or xBase or two ASCII file formats.

All of these Tabs, and much of Shazam's look and behavior, are controlled by more than 60 properties and 20 methods which makes Shazam very flexible and configurable for the developer. Beware that the Shazam Report Writer will NOT work with Apollo or other BDE replacements that don't use BDE or work with TQuery.

The reports a user can create are nicely polished and obviously have the potential to get as sophisticated as one can make an SQL query. Shazam lacks many of the output formatting features of Quick Reports (Borland has added the 32-bit release of Quick Reports to Delphi 2.0), but fancy screens is not what Shazam strives to achieve ... and powerful SQL queries is not what Quick Reports (either 16-bit or 32-bit variants) will be known for either. Both products actually compliment each other.

If you prefer, Shazam works wonderfully as a more or less conventional report printer. One can create reports with Shazam a design-time and run these reports at run-time with no more end user involvement than causing an eventhandler to fire Shazam and a report.

Shazamware not only ships clearly written HLP files, but have taken the time to separate developer help from end user help (and even include end user help in RTF format). The demo programs and reports included with the program are also stand outs in clarity, and do this product justice.

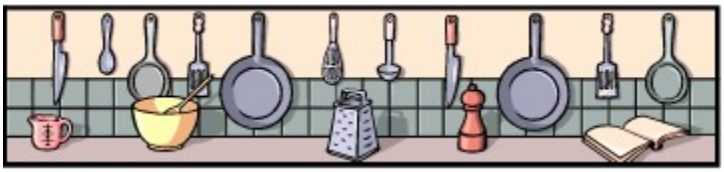
Zig Ziglar, the ultimate sales guru has preached "Sell the sizzle, not the steak" . The sizzle Shazam produces is the kind of "sizzle" that can add big value-added to your apps with very little effort. The steak here is the query engine (however, Sominex would love to bottle a sleeping potion as powerful as a discussion of SQL calls), people WILL notice the report writer.

All in all, I would put Shazam in a "must have" category if you are creating database apps that require tabular reporting. Shazamware has taken the time to do it right and is promising a 32-bit compliant DCU/VCL variant shortly after the release of Delphi 2.

Shazam is available through both the CIS Shareware forum and ZAC catalogs in three formats: a simple compiled stand alone app (USD 39), a 100% Delphi DCU (USD 79) or the Delphi DCU with source code (USD 138). There is a demo version available for download on the CIS Delphi forum, get it, try it.

Return to Front Page

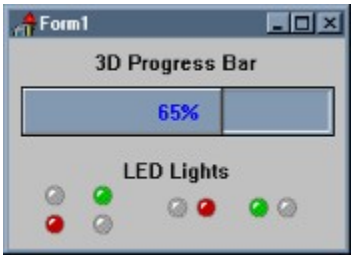
The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996



The Component Cookbook

An Auto-Repeating SpeedButton **- by Grahame S. Marsh**

General String Functions - **by Lars Posthuma**
3D-Progress Bar and LED Lights - **by Lars Posthuma**



Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Form & Component Creation Tips

by Duncan Campbell - Internet: duncan@helix.net

When dynamically creating forms (rather than using the Delphi default of creating all forms at application startup), people often do the following:

```
Procedure frmMain.ShowAnotherForm;
var
  AnotherForm: TAnotherForm;
begin
  AnotherForm := TAnotherForm.Create(Application);
  If AnotherForm.ShowModal = mrOk then
    {..other code to do after form is closed..}
  AnotherForm.Free;
end;
```

There is a quicker and neater way of doing it:

```
Procedure frmMain.ShowAnotherForm;
begin
  with TAnotherForm.Create(Application) do
    try
      If ShowModal = mrOK then
        {..other code to do after form is closed..}
    finally
      Free;
    end;
end;
```

This may look like more typing, but you are guaranteeing that you free up the memory used by the form in the **try..finally** block - the "finally" section is *always* executed. A better example of this occurs in the case where you want to use a TTable variable:

```
Procedure frmMain.TableCompExample;
begin
  with TTable.Create(nil) do
    try
      DatabaseName := 'SomeDataBase';
      TableName := 'SomeTable';
      Open;
      {..Do something..}
    finally
      Close;
      Free;
    end;
end;
```

In this second case, not only are we guaranteeing that the memory used by the TTable component is released, but we are also ensuring that the Table itelf is closed as well. I have found that if I always use this format of coding, I create tighter, less-buggy code and automatically use the **try..except** block which otherwise can be tedious to add to your code.

Return to Tips & Tricks

Return to Front Page

Detecting A CD-ROM Drive

By Ewart J.H. Nijburg - CIS:100662,2621

I needed a routine that would determine if a drive was CD-ROM or not... this was what came out. It isn't beautiful, but it does the trick! The code is compatible with the 32-bit version of Delphi.

```
Function IsCDrom(Drv : Char):BOOLEAN;
Var
  CDR   : string;
  cnt   : byte;
  Bx,cx : word;

  Procedure CDR_GET_DRIVE_COUNT (var COUNT, FIRST: word); assembler;
  asm
    mov ax, 1500h
    xor bx, bx
    int $2f           {CDROM_Interrupt}
    les di, COUNT
    mov es:[di], bx
    les di, FIRST
    mov es:[di], cx
  end;

begin
  IsCDROM := false;
  CDR := '';
  CDR_GET_DRIVE_COUNT(bx,cx);
  if BX > 0 then
    for cnt := 0 to (bx-1) do
      CDR := CDR + char(Cx + Byte('A') + cnt);
  IsCDROM := (Pos(upcase(Drv),CDR) > 0);
end;
```

Return to Tips & Tricks

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Book Review - Instant Delphi Programming by Dave Jewell

Review by Paul Harding CIS: 100046,2604

ISBN1-874416-57-5 / Wrox Press / Priced at $24.95USA / $34.95CAN or £22.99U.K.

Instant Delphi Programming is a clear, well written book of an intermediate level. The chapters are nicely labelled at the top corner of the page for easy retrieval, and the pages are well laid out. Code examples are clear, being in mono spaced font and highlighted out against a grey background.

It starts with the basics of the Delphi IDE, discusses event driven programming, and how to add code and start your first project. Dave Jewell moves through the subject matter quickly, and soon the reader has his sleeves rolled up with a useful application: An Icon Viewer.

By chapter 2 (Developing a User Interface) the book is running at full tilt, with more and more real projects being developed so that the concepts are introduced and explored in real applications. Chapter 3 concentrates on event handling, with the same no nonsense style leading us through the rest of the book. Through the rest of the book major topics are covered, with a good balance between design, database topics, menus, debugging and an introduction to component design leading to the appendix which helps Visual Basic developers make the leap. Each chapter ends with some exercises, which really do need to be done for maximum benefit.

Throughout the book are good programming examples, with quite an emphasis on making use of the Windows API, but there is no single chapter on this topic alone. The 3.5" diskettes supplied have all the code on them, although typing the code in oneself is always a preferred method of learning. My main criticism is that not enough time is spent on the Pascal language, although I suppose it would be better to buy another book dedicated to this topic alone, vast as it is. So Dave Jewell may have decided not to write a book about the language, and just concentrate his efforts on the Delphi environment and its use. He has done an excellent job of this, and I recommend this book.

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Delphi Projects

by Stephen B. Kirby - Internet: zenith@houston.va.gov

The Department of Veterans Affairs is using Delphi to develope a GUI front-end to its existing database. The VA is currently using a database written in the M language (formerly MUMPS). The VA has developed components that use WINSOCK to make remote procedure calls to our database. Currently, the software is primarily in beta testing, and there is only two or three national packages developed by the VA that are almost ready for distribution. I'm currently testing one of these packages locally...It's a Medical Imaging package developed at our Washington DC developement sites.

The Houston VA Medical Center is one of several sites testing the Delphi components (known as Remote Procedure Call Broker or RPC Broker). We're in the process of developing several packages locally using the RPC Broker.

Feel free to write if you have any questions.

Return to Delphi Projects

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Am I Running Already?

By Paul Harding - CIS:100046,2604

How to establish if your program is already running.

I have an "AutoDial" form which gets user input for the name of an address book entry. The address database is searched for that entry and on pressing the <Enter> key the modem is dialled for that company's phone number, and the form is closed.

The biggest problem with this form is its load time - even on a fast machine it takes a while to load and display the form, and as the form is in use lots of times each day, it really does need to be snappy. So what I did was to minimize the form when it's just been used instead of closing the form. Then I had to establish a way of restoring the form the next time the program is run. The program's type declaration is like this...

```
type
   TAutoDialDlg = class(TForm)
   {and lots more...}
```

This tells us that the class of the window is a TAutoDialDlg, which we are going to need when we want to find out if the program is already running. So in the project source, two variables need to be declared...

```
var
   handle: LongInt; {to store the window handle for the form if it is minimized}
   ClassName: PChar; {this is going to be TAutoDialDlg, or whatever, see your
form's type}
```

And still in the project source, where normally the form would be created, the following code is used to either create the form if the program is being run for the first time, or to restore the form if it's minimized...

```
ClassName := 'TAutoDialDlg';
handle := FindWindow(ClassName, NIL); {Windows API}
if handle = 0 then {Auto Dial is not already running, so let's create its form...}
   {this line already exists in the project source}
   Application.CreateForm(TAutoDialDlg, AutoDialDlg)
else
  begin
    showWindow(handle, SW_RESTORE); {Windows API}
    halt;   {now we have restored the auto dial form, quit this program}
  end;
```

The Windows API routine FindWindow returns a handle to the window (zero if not found) and takes two parameters, the ClassName and the WindowName, and either one can be NIL. Another Windows API routine ShowWindow is used to restore the form, and takes the form's handle as its first parameter, and a

constant, being SW_RESTORE in this case, as its second parameter.

Finally, there has to be a way of minimizing the form rather than closing it, so instead of the Close; command this code does the trick:

```
WindowState := wsMinimized;
```

Every time the program is run, which in my example is caused by a hot key combination (I'm a keyboard junkie and don't like to reach for the mouse when a hotkey will do), the program determines either to load the form (slow) or to restore the form if its minimized, so speeding up its appearance on my desktop.

Return to Front Page
Return to Tips & Tricks

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Delphi Projects

by Martin Holland - CIS:100012,110

My name is Martin Holland and I am a biochemist working in a hospital in Wolverhampton, England. My interest in computing stemmed from an earlier interest in electronics which is the reason one of my applications has been to connect an analyser to a laboratory computer. Because of Robert's request for industry accomplishments using Delphi, I though I might put together a brief description of some of the things I have worked on.

SNOOPY

This sits in the background and at predetermined times deletes unwanted files (which you specify) and warns if memory or disk space is getting low. Its a watchdog for secretaries so they can call the computer support guys.

DIALLER

I suppose most people have written this one but I can never remember my colleagues' telephone numbers so I did one too.

CONWAPI

This is for creating contract of employment documents. Instead of wading though a Word for Windows document and filling in the details, the user fills in a simple form. The information is then plugged into the document using Windows For Workgroups application interface calls. Makes life simple.

WAV & MIDI player.

I wrote this using a component from UNDU (I think). It searches for WAV and MID files and allows you to play them.

DECRYPT

This encodes text (using a key) so you can send a coded message by email. You can also decode messages you receive if you know the key. This is useful if you don't want to kill your email programme every time you leave the office.

INTERFACE

This receives test results from an analyser, allows editing and merging with a worksheet and uploading to a laboratory computer.

In addition I am working on a series of small programmes that a biochemist might want to use during the course of his work. Examples are: Chi Squared test, non-linear curve fit, interpretation of data to diagnose the cause of ascites or anaemia and a way of saying how precise a method must be for a given drug assay knowing the dose interval and half life. These are only the tip of the iceberg and I am sure I shall be programming many more, but due to Delphi's power, I can turn out a simple application in about half an hour now.

Return to Delphi Projects

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Marketing Your Components

by Robert Pullan - CIS:102162,2711

As a product reviewer for UNDU, I have had the opportunity to review a lot of components. I would like to share some general commentary on component packaging with the developers who want to produce excellent products. My goal here is not to beat up upon those firms who provide us with the many wonderful third party components, but hopefully to share some observations.

Folks... **Stop Selling Your Products Short!** Most component demos / evaluation copies I look at are so incompletely "packaged" that they are worthless... and in many cases encourage potential buyers to avoid your work. When I refer to packaging, I am not referring to fancy boxes or printed material, but rather what is included with a product or demo (Demo EXEs, Documentation, HLP files, etc.).

Documentation

As time has moved on I have seen a disturbing trend away from good documentation. IMHO this not only makes understanding a component more difficult for the user, but also substantially reduces the chances that your demo/evaluation version will result in a sale. It amazes me that firms will try to produce wonderful demos, but fail to properly explain the how and why of their products.

Typically I examine ten to twenty Delphi components a month and have been doing so for most of the existence of Delphi. On a first exposure, I give each component three or four minutes to show me what its got. If I can't figure it out, if it doesn't have adequate docs, if the docs are not immediately clear, or if the only source of documentation is to play "go fish" in a HLP file, they make an immediate appearance in my "recycle bin". My point is: if a focused and experienced user will only tolerate a few minutes of confusion to gain an impression, what will a more casual user tolerate? Why risk the many hours spent developing your opus on unclear docs ?

As a reviewer, I can say that there is a very high correlation between well documented products and products worthy of review in this or other publications. As a developer, I can also confirm there is a virtually perfect correlation between well documented products and products my company, Lighthouse Technologies Inc, purchases. I am not saying good docs necessarily make a good product, but rather good products almost always have good docs.

Well conceived and written documentation is also a cost saver. Well written docs are a one time effort and help minimize calls to customer support from customers. Every component needs three levels of documentation. It needs:

Paper/TXT/WRI Based "Getting Started" Instructions

I have yet to see a component that doesn't have some quirks that are not immediately obvious to your new user. The TXT/WRI level "Getting Started" focused documentation is needed because new users need more help, and HLP files (even if well written) require new users to play hypertext "go fish" until they find the right combination of information to achieve functionality. While it is

especially nice to have printed and bound documentation (and not unreasonable to expect in products selling for more than $100 or so), it is amazing what high-quality output can be achieved with a WRI formatted document.

Clear Demo Program

I suggest simple sample programs which would not only allow the user to compile your program in action, but also explore how it interacts with other key components. These programs should be over commented, especially whenever there is a special piece of information (eg. component will not work if design-time active=true). Try to remember that your code is clear to you, but it may not be clear to your users. Sample programs abound, but IMHO most are poorly commented. If your component uses file paths or aliases, or somehow uses the database components, place file path or BDE alias information in the header of the main PAS file. I strongly recommend using the demo data Borland includes with Delphi as a database. This will not only reduce file size, but provide a great basis for comparisons.

***.HLP File Level Documentation**

Once users have mastered the basics of the component, they want something that can help them explore the soul of your opus. Help files are an excellent way to deliver this kind of knowledge. It is also critical to include examples of code for the reason that someone is looking a feature up in HLP is because they don't understand how it works. A snippet of code can reveal much.

I strongly recommend that commercial grade components include ALL three forms, because each operates at a different and necessary level. "Getting started" should be focused upon installing and a very simple example of how to configure and use the component. The Demo Program should be perhaps a bit more advanced, with an aim to show off neat features. The HLP files should aimed a user who has basic proficiency and needs details.

Mainstream companies that do an outstanding job in this include TurboPower (Orhpeus and Asynch Pro for Delphi) and InfoPower. One would also be well advised to check out TtaDBMRO from Tamarack or Shazam by Shazamware Solutions both are examples of excellent WRI-based documentation produced by smaller companies. The key to good documentation is not fancy printing to books, but good clear easy to understand language.

End-User Documentation

For components which offer significant functionality, the need for documentation extends beyond the needs of the programmer. The end user needs clear HLP files. Absent ASCII or RTF help prototypes or end-user HLP files, means the component developer is depending upon the programmer to do a good job representing the features of the component to the end user. The risk here is if the component is right, but the programmer does an inadequate job of documenting your product end users will be unhappy, and guess who gets blamed ...

Source-Code Option

I know this issue often cuts to the guts of component developers. However, this is why copyright laws exist. Many companies, like my firm, will not use components that lack source code - with rare exceptions. The underlying reason is these components become part of our deliverables. If a component behaves unexpectedly, our customers expect us to fix the problem and don't want us to point fingers elsewhere. Developers can ill afford to include components that place our products in a position of jeopardy.

There is also an internal illogic to the resistance to providing source code. The reason firms choose to buy versus build a component is that the buyer doesn't have the resources or can more productively use resources elsewhere. Few buyers are trying to "steal" your most innovative ideas.

Lack of a source code option also suggests something else. It suggests the developer hasn't really copyrighted the code. If this is so, is the product or developer substantial enough to become involved with. The cost of a copyright is extremely inexpensive.

Conclusion

I realize most components are written by programmers, not marketers. But everyone who expects to prosper in the third party market needs to understand that marketing is an important consideration. Buyers are interested in your component, because it does something they don't know how to do or prefer not to take the time to do. The easier you make it for the user to understand what you have done, the greater the chance you will make a sale.

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

AutoRepeat for SpeedButtons

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

Please find below the source code of a component which gives a speedbutton an autorepeat operation. The TDBNavigator has this feature and I have found it useful to have autorepeating buttons on a number of applications. The extensions to TSpeedButton are not complex and I would hope that the property names are intuitive. I also enclose the .RC code for the bitmap I use in the component palette.

```
{ Component derived directly from TSpeedButton - uses a timer to provide
  extra clicks if the button is pressed and help depressed by the mouse. }

unit RptSpd;

interface

uses
  Buttons, Classes, Controls, ExtCtrls;

type
  TTimerRange = 1..$ffff;
  TRptSpeedButton = class(TSpeedButton)
  private
    fAutoRepeat  : boolean;
    fCount  : word;
    fInitPause,
    fRepeatPause : TTimerRange;
    fRepeatTimer : TTimer;
    procedure TimerExpired(Sender: TObject);
  protected
    procedure MouseDown (Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
override;
    procedure MouseUp (Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
override;
  public
    constructor Create (AOwner : TComponent); override;
    destructor  Destroy; override;
  published
    property   AutoRepeat : boolean read fAutoRepeat write fAutoRepeat default
true;
    property   InitialPause : TTimerRange read fInitPause write fInitPause default
400;
    property   RepeatPause : TTimerRange read fRepeatPause write fRepeatPause
default 100;
    property   Count : word read fCount;
  end;

{properties:
  AutoRepeat : boolean   - does the autorepeat function operate
  InitialPause : word    - time in mS before first repeat occurs
  RepeatPause : word- time in mS between subsequent repeats
  Count : word - number of repeats (including initial press)}

implementation

constructor TRptSpeedButton.Create (AOwner : TComponent);
begin
  inherited Create (AOwner);
  AutoRepeat := true;
```

```
  InitialPause := 400;
  RepeatPause := 100;
end;

destructor TRptSpeedButton.Destroy;
begin
  fRepeatTimer.Free;
  inherited Destroy;
end;

procedure TRptSpeedButton.MouseDown(Button: TMouseButton; Shift: TShiftState; X, Y:
Integer);
begin
  inherited MouseDown (Button, Shift, X, Y);
  if fAutoRepeat then
    begin
      if not Assigned (fRepeatTimer) then fRepeatTimer := TTimer.Create(Self);
      fCount := 1;
      with fRepeatTimer do
        begin
          OnTimer := TimerExpired;
          Interval := fInitPause;
          Enabled  := true;
        end;
    end;
end;

procedure TRptSpeedButton.MouseUp(Button: TMouseButton; Shift: TShiftState; X, Y:
Integer);
begin
  inherited MouseUp (Button, Shift, X, Y);
  fRepeatTimer.Free;
  fRepeatTimer := nil;
end;

procedure TRptSpeedButton.TimerExpired(Sender: TObject);
begin
  fRepeatTimer.Interval := fRepeatPause;
  if (fState = bsDown) and MouseCapture then
    try
      Click;
      inc (fCount);
    except
      fRepeatTimer.Enabled := false;
      raise;
    end;
end;

end.
```

Readers can save the following data as a text file, called **RPTSPD.RC** and compile it into a component resource using the *Borland Resource Compiler* (included with Delphi). Simply go into the directory with the file and type **C:\DELPHI\BIN\BRCC RPTSPD.RC** and the resource compiler will generate a **RPTSPD.RES** file. Rename it to **RPTSPD.DCR** and place it in the same directory as the source code unit listed above. When you compile this unit into your component library, it will use this bitmap on the component palette.

```
TRPTSPEEDBUTTON BITMAP
{
 '42 4D 96 01 00 00 00 00 00 00 76 00 00 00 28 00'
 '00 00 18 00 00 00 18 00 00 00 01 00 04 00 00 00'
 '00 00 20 01 00 00 00 00 00 00 00 00 00 00 00 00'
```

```
  '00 00 10 00 00 00 00 00 00 00 00 00 80 00 00 80'
  '00 00 00 80 80 00 80 00 00 00 80 00 80 00 80 80'
  '00 00 C0 C0 C0 00 80 80 80 00 00 00 FF 00 00 FF'
  '00 00 00 FF FF 00 FF 00 00 00 FF 00 FF 00 FF FF'
  '00 00 FF FF FF 00 77 77 77 77 77 77 77 77 77 77'
  '77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77'
  '77 77 77 77 77 77 77 77 77 77 77 70 00 00 00 00'
  '00 00 00 00 07 77 77 70 78 88 88 88 88 88 88 88'
  '07 77 77 70 F7 77 77 00 77 8B 80 78 07 77 77 70'
  'F7 77 77 0B 08 B0 00 78 07 77 77 70 F7 70 07 07'
  'B0 80 77 78 07 77 77 70 F7 70 F0 0B 00 B8 07 78'
  '07 77 77 70 F7 70 BF 07 0B 8B 80 78 07 77 77 70'
  'F8 00 F0 0B 70 B0 00 78 07 77 77 70 F7 0F B0 B7'
  'B7 08 07 78 07 77 77 70 F7 70 FB 0B 00 0B 80 78'
  '07 77 77 70 F7 70 BF B0 B0 B8 B8 08 07 77 77 70'
  'F0 00 F0 00 7B 0B 00 08 07 77 77 70 F0 BF BF 07'
  'B7 B0 77 78 07 77 77 70 F7 0B FB F0 70 00 77 78'
  '07 77 77 70 F7 0F BF BF 07 77 77 78 07 77 77 70'
  'F7 70 00 00 08 77 77 78 07 77 77 70 FF FF FF FF'
  'FF FF FF F7 07 77 77 70 00 00 00 00 00 00 00 00'
  '07 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77'
  '77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77'
  '77 77 77 77 77 77'
}
```

Return to the Component Cookbook
Return to Front Page

```
unit Percnt3D;
(*
   TPercnt3D by Lars Posthuma; December 26, 1995.
   Copyright 1995, Lars Posthuma.
   All rights reserved.

   This source code may be freely distributed and used. The author
   accepts no responsibility for its use or misuse.
   No warranties whatsoever are offered for this unit.

   If you make any changes to this source code please inform me at:
   LPosthuma@COL.IB.COM.
*)

interface

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

type
  TPercnt3DOrientation = (BarHorizontal,BarVertical);

  TPercnt3D = class(TCustomPanel)
    private
      { Private declarations }
      fProgress    : Integer;
      fMinValue    : Integer;
      fMaxValue    : Integer;
      fShowText    : Boolean;

      fOrientation : TPercnt3DOrientation;
      fHeight      : Integer;
      fWidth       : Integer;
      fValueChange : TNotifyEvent;

      procedure SetBounds(Left,Top,fWidth,fHeight: integer); override;
      procedure SetHeight(value: Integer); virtual;
      procedure SetWidth(value: Integer); virtual;

      procedure SetMaxValue(value: Integer); virtual;
      procedure SetMinValue(value: Integer); virtual;
      procedure SetProgress(value: Integer); virtual;
      procedure SetOrientation(value: TPercnt3DOrientation);
      procedure SetShowText(value: Boolean);
      function GetPercentDone: Longint;
    protected
      { Protected declarations }
      procedure Paint; override;
    public
      { Public declarations }
      constructor Create(AOwner: TComponent); override;
      destructor Destroy; override;
      procedure AddProgress(Value: Integer);
      property PercentDone: Longint read GetPercentDone;
      procedure SetMinMaxValue(Minvalue,MaxValue: Integer);
    published
      { Published declarations }
      property Align;
      property Cursor;
      property Color default clBtnFace;
```

```
      property Enabled;
      property Font;
      property Height default 25;
      property Width default 100;
      property MaxValue: Integer
               read fMaxValue write SetMaxValue
               default 100;
      property MinValue: Integer
               read fMinValue write SetMinValue
               default 0;
      property Progress: Integer
               read fProgress write SetProgress
               default 0;
      property ShowText: Boolean
               read fShowText write SetShowText
               default True;
      property Orientation: TPercnt3DOrientation            {}
               read fOrientation write SetOrientation
               default BarHorizontal;
      property OnValueChange: TNotifyEvent                  {Userdefined Method}
               read fValueChange write fValueChange;
      property Visible;
      property Hint;
      property ParentColor;
      property ParentFont;
      property ParentShowHint;
      property ShowHint;
      property Tag;

      property OnClick;
      property OnDragDrop;
      property OnDragOver;
      property OnEndDrag;
      property OnMouseDown;
      property OnMouseMove;
      property OnMouseUp;
  end;

procedure Register;

implementation

constructor TPercnt3D.Create(AOwner: TComponent);
begin
 inherited Create(AOwner);
 Color       := clBtnFace;                      {Set initial (default) values}
 Height      := 25;
 Width       := 100;
 fOrientation := BarHorizontal;
 Font.Color  := clBlue;
 Caption     := ' ';
 fMinValue   := 0;
 fMaxValue   := 100;
 fProgress   := 0;
 fShowText   := True;
end;

destructor TPercnt3D.Destroy;
begin
 inherited Destroy
end;

procedure TPercnt3D.SetHeight(value: integer);
```

```
begin
 if value <> fHeight then begin
   fHeight:= value;
   SetBounds(Left,Top,Width,fHeight);
   Invalidate;
 end
end;

procedure TPercnt3D.SetWidth(value: integer);
begin
 if value <> fWidth then begin
   fWidth:= value;
   SetBounds(Left,Top,fWidth,Height);
   Invalidate;
 end
end;

procedure TPercnt3D.SetBounds(Left,Top,fWidth,fHeight : integer);
 Procedure SwapWH(Var Width, Height: Integer);
 Var
  TmpInt: Integer;
 begin
  TmpInt:= Width;
  Width := Height;
  Height:= TmpInt;
 end;
 Procedure SetMinDims(Var XValue,YValue: Integer; XValueMin,YValueMin: Integer);
 begin
  if XValue < XValueMin
   then XValue:= XValueMin;
  if YValue < YValueMin
   then YValue:= YValueMin;
 end;
begin
 case fOrientation of
   BarHorizontal: begin
                   if fHeight > fWidth
                     then SwapWH(fWidth,fHeight);
                   SetMinDims(fWidth,fHeight,50,20);
                  end;
   BarVertical  : begin
                   if fWidth > fHeight
                     then SwapWH(fWidth,fHeight);
                   SetMinDims(fWidth,fHeight,20,50);
                  end;
 end;
 inherited SetBounds(Left,Top,fWidth,fHeight);
end;

procedure TPercnt3D.SetOrientation(value : TPercnt3DOrientation);
Var
 x: Integer;
begin
 if value <> fOrientation then begin
   fOrientation:= value;
   SetBounds(Left,Top,Height,Width);                        {Swap Width/Height}
   Invalidate;
 end
end;

procedure TPercnt3D.SetMaxValue(value: integer);
begin
 if value <> fMaxValue then begin
```

```
   fMaxValue:= value;
   Invalidate;
 end
end;

procedure TPercnt3D.SetMinValue(value: integer);
begin
 if value <> fMinValue then begin
   fMinValue:= value;
   Invalidate;
 end
end;

procedure TPercnt3D.SetMinMaxValue(MinValue, MaxValue: integer);
begin
 fMinValue:= MinValue;
 fMaxValue:= MaxValue;
 fProgress:= 0;
 Repaint;                                          { Always Repaint }
end;

{ This function solves for x in the equation "x is y% of z". }
function SolveForX(Y, Z: Longint): Integer;
begin
 SolveForX:= Trunc( Z * (Y * 0.01) );
end;

{ This function solves for y in the equation "x is y% of z". }
function SolveForY(X, Z: Longint): Integer;
begin
 if Z = 0
   then SolveForY:= 0
   else SolveForY:= Trunc( (X * 100) / Z );
end;


function TPercnt3D.GetPercentDone: Longint;
begin
 GetPercentDone:= SolveForY(fProgress - fMinValue, fMaxValue - fMinValue);
end;

procedure TPercnt3D.Paint;
var
 TheImage: TBitmap;
 FillSize: Longint;
 W,H,X,Y : Integer;
 TheText : string;
begin
 with Canvas do begin
   TheImage:= TBitmap.Create;
   try
     TheImage.Height:= Height;
     TheImage.Width := Width;
     with TheImage.Canvas do begin
       Brush.Color:= Color;
       with ClientRect do begin
         { Paint the background }
         { Select Black Pen to outline Window }
         Pen.Style:= psSolid;
         Pen.Width:= 1;
         Pen.Color:= clBlack;
```

```
{ Bounding rectangle in black }
Rectangle(Left,Top,Right,Bottom);

{ Draw the inner bevel }
Pen.Color:= clGray;
Rectangle(Left + 3, Top + 3, Right - 3, Bottom - 3);
Pen.Color:= clWhite;
MoveTo(Left + 4, Bottom - 4);
LineTo(Right - 4, Bottom - 4);
LineTo(Right - 4, Top + 2);

{ Draw the 3D Percent stuff }
{ Outline the Percent Bar in black }
Pen.Color:= clBlack;
if Orientation = BarHorizontal
  then w:= Right - Left { + 1; }
  else w:= Bottom - Top;
FillSize:= SolveForX(PercentDone, W);
if FillSize > 0 then begin
  case orientation of
   BarHorizontal: begin
                   Rectangle(Left,Top,FillSize,Bottom);

                   { Draw the 3D Percent stuff }
                   { UpperRight, LowerRight, LowerLeft }
                   Pen.Color:= clGray;
                   Pen.Width:= 2;
                   MoveTo(FillSize - 2, Top + 2);
                   LineTo(FillSize - 2, Bottom - 2);
                   LineTo(Left + 2, Bottom - 2);

                   { LowerLeft, UpperLeft, UpperRight }
                   Pen.Color:= clWhite;
                   Pen.Width:= 1;
                   MoveTo(Left + 1, Bottom - 3);
                   LineTo(Left + 1, Top + 1);
                   LineTo(FillSize - 2, Top + 1);
                  end;
   BarVertical:   begin
                   FillSize:= Height - FillSize;
                   Rectangle(Left,FillSize,Right,Bottom);

                   { Draw the 3D Percent stuff }
                   { LowerLeft, UpperLeft, UpperRight }
                   Pen.Color:= clGray;
                   Pen.Width:= 2;
                   MoveTo(Left + 2, FillSize + 2);
                   LineTo(Right - 2, FillSize + 2);
                   LineTo(Right - 2, Bottom - 2);

                   { UpperRight, LowerRight, LowerLeft }
                   Pen.Color:= clWhite;
                   Pen.Width:= 1;
                   MoveTo(Left + 1,FillSize + 2);
                   LineTo(Left + 1,Bottom - 2);
                   LineTo(Right - 2,Bottom - 2);
                  end;
  end;
end;
if ShowText = True then begin
  Brush.Style:= bsClear;
  Font       := Self.Font;
```

```
            Font.Color := Self.Font.Color;
            TheText:= Format('%d%%', [PercentDone]);
            X:= (Right - Left + 1 - TextWidth(TheText)) div 2;
            Y:= (Bottom - Top + 1 - TextHeight(TheText)) div 2;
            TextRect(ClientRect, X, Y, TheText);
          end;
        end;
      end;
      Canvas.CopyMode:= cmSrcCopy;
      Canvas.Draw(0,0,TheImage);
      finally
        TheImage.Destroy;
    end;
  end;
end;

procedure TPercnt3D.SetProgress(value: Integer);
begin
 if (fProgress <> value) and (value >= fMinValue) and (value <= fMaxValue) then
begin
   fProgress:= value;
   Invalidate;
 end;
end;

procedure TPercnt3D.AddProgress(value: Integer);
begin
 Progress:= fProgress + value;
 Refresh;
end;

procedure TPercnt3D.SetShowText(value: Boolean);
begin
 if value <> fShowText then begin
   fShowText:= value;
   Refresh;
 end;
end;

procedure Register;
begin
 RegisterComponents('Lars', [TPercnt3D]);
end;

end.
```

Return to The Component Cookbook
Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Product Announcement - Dr. Bob's Experts for Delphi 1.0

by Bob Swart - CIS: 100434,2072

Advanced and Integrated Expert Support for 16-bit Delphi Development

Dr.Bob's Collection of Delphi 1.0 Experts consists of a dozen Delphi IDE experts in three different forms (18 total experts, with two invisible experts to "group" them together in the Help-menu):

```
S  F  P  (Standard, Form or Project Expert)
X  -  -  FileOpen
X  -  -  SysUtils Info
X  -  -  Resource Info
X  -  -  Project Info
X  -  X  DLL Source
X  -  X  Component
X  X  -  Splash Form
X  X  -  Table Form Expert (with Marco Cantu)
X  -  -  DLL Expert Loader
X  -  -  16- to 32-bit Resource Converter
X  X  X  C DLL Header Converter 1.03
```

Three more experts are used to give the package a more complete look (see for yourself)...

Overview

The 12 IDE Experts are all combined in one DRBOB.DLL of only 660304 bytes. This may seem a big DLL, but consider the fact that it actually holds 18 expert classes as well as over a dozen form definitions! The installation is performed by INSTALL.EXE that copies DRBOB.DLL to the C:\DELPHI\DRBOB directory and adds this Expert Library to the [experts] section in the DELPHI.INI file.

Dr.Bob's Expert About Screen...

This About Splash screen (standard Expert) was designed using the Splash Form Expert, and shows up at start-up and close down of Delphi (if the DRBOB.DLL is installed). Registered users (see below on details about registering DRBOB.DLL) will be enabled to remove this Splash screen. If you select this form from the Help menu, then you can close if by clicking on it with the left mouse button.

Dr.Bob's FileOpen Expert...

The FileOpen standard Expert consists of a simple OpenDialog to let you open just about any file in the Delphi IDE. Quite handy if you want to look at .INT files, or any other files than the .DPR files without having to go to a lot of trouble with the current Project dialog. Note that the file is opened as project, so any loaded files in the IDE will be closed (you will be asked if you want to save the contents). So, you can also use this expert to open up any .PAS files and be able to instantly recompile them.

Dr.Bob's SysUtils Info Expert...

The SysUtils Info standard Expert, developed for the upcoming book The Revolutionary Guide to Delphi 2.0 from WROX Press, will show a modeless form with information regarding the current currency, date and time information.

Dr.Bob's Resource Info Expert...

The Resource Info standard Expert will show a modeless always-on-top form with information regarding the current state of available System, User and GDI resources, as well as the available high and conventional memory. There is now also a "mark" (left) and "used" (right) button. Clicking on the Mark-button will make a snapshot of the current resource and memory values, while clicking on Used-button

will show the difference between the current state and the marked state. This way, you can easily identify the resource and memory requirements of your applications (just "mark" before a run, and click on "used" during running the application) as well as identify possible resource leakage (if "used" is not zero after a run).

Dr.Bob's Project Info Expert...

The Project Information standard Expert will show a modeless dialog with information about the current project, including the number and names of the units and forms in the project. With the Expand-button, all units can be opened at once, while the Reduce-button will close all units and forms (to bring down resource usage, a good thing to do right before another run). Note that this dialog is already ViCiouS-aware (my upcoming Version Control System for Delphi). If ViCiouS is installed, then the ViCiouS-button will be enabled and the Project Information dialog itself will automatically be shown when a new project is loaded. Also, using the ViCiouS-button we can activate ViCiouS with the current project information. Stay tuned for more information about ViCiouS...

Dr.Bob's DLL Source Expert...

This DLL Source Generator standard and project Expert, first developed in an article for issue #3 of The Delphi Magazine, will give you a lot of options to generate a DLL source skeleton file which is opened in the Delphi IDE as new project (to be compiled immediately).

Dr.Bob's Component Expert...

The Component standard and project Expert will enable you to enter the information for a new component, just like the default Component Expert of Delphi, but with one significant additional feature: the resulting source file is immediately opened as new project in the Delphi IDE. So we don't have to re-open it to be able to compile it. This will save you time and frustration (I've been there).

Dr.Bob's Splash Form Expert...

The Splash Form standard and form Expert generates nice Splash screens for your projects. All you need to do is select a nice bitmap, and the form will be created automatically. Double-click on the bitmap to get a preview of what the Splash screen will look like. Note that the project main .DPR file is automatically modified to make sure the Splash screen is created and shown when your application starts, and destroyed when your main form is activated (so the Splash screen will only be active during the time your application loads before your main form is shown).

Marco & Dr.Bob's Table Form Expert...

The Table Form standard and form expert is written by Marco Cant and Dr.Bob during a seminar for the UK-DDG (Delphi Developers Group). Thanks again for Joanna and Phil for giving us the opportunity to get the best out of this "live" session. If you want to know how the experts works internally, then check out issue #7 of The Delphi Magazine where we've written down the implementation details in an article How To Write Your Own Database Expert. The Table Form Expert will guide you through four pages that ask you to pick an alias, a table, field from the table, and finally offer you a possibility to alter the names (labels) for these fields that will be created on the resulting form. Just like the Database Form Expert that is part of Delphi itself, only this time you're able to give each field its own label.

Dr.Bob's Dynamic DLL Expert Loader...

The DrBob.DLL comes with a total of 12 DLL Experts. This standard Dynamic Expert Loader Expert offers you the ability to load other DLL experts (like EXPTDEMO.DLL, if not already loaded). Note that if s standard DLL expert is loaded this way, the Help menu won't get updated. So it will only work correctly for Form and Project DLL Experts that get loaded by the Dynamic Expert Loaded. For details, check out the upcoming book The Revolutionary Guide to Delphi 2.0 again.

Dr.Bob's 16- to 32-bit Resource Converter...

The standard 16- to 32-bit Resource Converter Expert is able to convert 16-bit binary resource files which contain only bitmaps (.DCR files and .RES files with only bitmaps for example) to 32-bits resource files.

Quite handy if you plan to upgrade to Delphi 2.0, which we all will do shortly, right?

Dr.Bob's C DLL Header Converter ...

The final standard, form and project expert is the well-known HeadConv C DLL Header Converter Expert. This expert will assist in converting C DLL header files to Delphi implicit or explicit import units. The version inside DRBOB.DLL will already be able to generate both implicit and explicit import units (the previous unregistered version of HeadConv 1.02 was only able to generate implicit import units). (current registered users of HeadConv will get a discount to the registration fee - see your e-mail)

Registration

Usage only: SWREG id#: 9866 (US$ 25.00) - you'll get a helpfile with info to get rid of the Splash screen.

Source code: SWREG id#: 9867 (US$ 50.00) - you'll get the helpfile and the full source code.

Return to Front Page

The Unofficial Newsletter of Delphi Users - Issue #12 - February 23rd, 1996

Evolving a Twenty-First Century Text Editor: A Project Description

by Gene Fowler Internet: acorios@cello.gina.calstate.edu

I recently read again Robert's December call for descriptions of projects undertaken using Delphi. I haven't seen what the call has pulled in yet, but certainly look forward to doing so. And I think I have a project to throw on the table that's bound to provoke inventive thinking in any Delphi-armed, program-wiring dreamer.

PocketPad* is an MDI text editor. that is about as minimal as a description gets. But even what it is (the concept) is evolving. I began it the day I received and installed Delphi, in March 1995. I had been prototyping the innards in BPW7.

I guess you know why I was only prototyping it. An MDI Notepad was a contradiction in terms. All the multiline edits (memos) had to share the (roughly) 32KB heap in the instance's single dGroup.

Delphi, as you know, allows 32KB to each memo installed. The slight of hand occurs in TCustomMemo's code, and Microsoft's Edit code is fooled into thinking the instance's heap is where it ain't. PocketPad became a real project. 32KB is not much room. But I'd written DOS-based tools I'd wire in. One of these, TxtPager SPLITs and reJOINS large files into two or more .PT? files. I've written in multiple loading from both the FileOpenDialog and the command line on startup.

PocketPad, then, has been my test bed for learning Delphi. In the beginning, it was to be a better Notepad. It's been several other things. Now, it's an email jockey's dream machine. By late Fall, I saw that with the ubiquitous (web) browsers for display and printing and email for its global linking, a text editor with a little prepping would be Everyman's (generic term, includes women) e-word processor. In dedicated word processors, the user has the problem of not being sure how what he is typing will export "as 0-127 text."

You'll like the fact that the topics in the on-line Help are both help for the user and shop talk for Delphi laborers. As I said, it's a test bed. I work out the innards for what I guess will one day be a dozen or more components or, in some instances, possibly are now out in the cybersea floating about. I do a whole lot of direct wiring and you can pretty well see it (you've got to have a little xray vision) as you see what I do. That's why I think this description belongs here.

Sorry about no Source code. It's a mass and something of a mess. More importantly, while it is straight-away code, no brilliant miracle based secret algorithms, ...I've many forms, everything interwired, and I have units I've written separately and I've even rewritten in the heart of Borland's routines, as in Dialogs.PAS. When I did InputQueryEx (an InputQuery with a History), I pulled the IQ function out into Ex's own unit and reworked it. But when I put Captions on MessageDlg, I had to go down into the innards, into CreateMessageDialog(), and snip wires and solder new ones in.

Those are two things, then, modeled in PocketPad. MessageDlgs have captions counterpointed to messages while keeping the 3D look and the little stop sign or large exclamation point. And InputQuerys can have history lists.

But here are a few of the items that are truly Delphi design uses and not just power code. I won't go into Tools at all. And most of everything else is left for you to discover. Oh, one comment on reusability. I evolved this from Borland's TextEdit demo. Uh huh! Use that xray vision to see the "energies" between what was and what is.

1) I use the ready-made FileOpenDialog rather than building one from the pieces components Delphi has on hand. The separate Filters combo has Strings ItemsIndex. But in the common dialog, there seems no way to get at it. My latest triumph (over the last week) includes emulating Winword's giving you back the filter you had at the last load (in the current work session) instead of the default one. The dialog keeps the Dir and Drive, more or less, because they read and set the hardware. But the filter snaps back to the default. I faked access to the the interior Index. I read the loaded file's extension, and if I have a filter I make that current. If I don't, I make *.* the filter.
2) I have eight boilerplates, and each has a dialog for setting the content on the fly. Now I have a

panel that lists shows them and clicking on the label gets the reset dialog. Boilerplates are used with functions like my menu-driven batch typing. I suspect the boilerplate system, like the Toolbox (on Tools menu, but not the only tool), a miniature ProgMan, could be captured in a component.

3) I just spoke of the text editor, browser, and email forming a cheap and ultimately very powerful e-word processor. You've no doubt already wondered about typing in HTML tags, perhaps using a web editor. Well, you can do that. But there's a difference between "editor" and "editing." A web editor is lousy to write email in, stopping to fill out forms for building tags and all that.

- Forget about using html as typographers' marks to design static pages. Use it as punctuation and write your text. In Notepad, you'll want a kind of shorthand notation for where begin and end tags will go. Then, you go back and fill in.
- But I have layers of batch typing of html punctuation in PocketPad. It starts with a very simple system on the Edit menu, with ^S getting <|> and ^X getting </|>. Use the boilerplate for the tag innards. Then, I have HTML and Txt Keys menus. For building anchors and such, I do not use a form to fill out. I use a series of InputQuerys so you write the elements in order ...as you write!
- I have a tool for bringing up your browser for wysiwyg reading. If you have an .htm in the active editor that will be in the browser. Otherwise. the last one you had in it from this tool. Of course you can change that when firing the tool.

I certainly hope I'm steering you toward examining this as the test bed project it is, not just noting features and liking or disliking the end product. Instead of noting what's there, ask what I'm doing and why.

An example. I have MessageDlgs with the glyphs pulled off the buttons. Why would I do that? The answer lies in what is going on before I come on the scene. In this case, it is that same FileopenDialog. It's internal error handling brings up a message box that's 3D, like MessageDlg and not like MessageBox, but without glyphs on the buttons.

If you've tried to think something through and it just plain doesn't make sense, feel free to ask me about it. But don't assume I always know and understand why I do what I do....

* ftp://ftp.coriolis.com/pub/Shareware/pcketpad.zip collects a copy of PocketPad. It is Shareware, but while it is 1.x no registration is required. It is 1.2 as I write this.

Return to Delphi Projects

Return to Front Page

```
unit DrawMeta;

{--- make a drawable metafile ---}
{     Grahame Marsh Jan 1996     }

interface

uses
  Graphics, Forms, WinProcs;

type
  TDrawMetaFile = class(TMetaFile)
  private
    FCanvas : TCanvas;
    FDrawing : boolean;
    procedure SetDrawing (State : boolean);
  protected
    property Drawing : boolean read FDrawing write SetDrawing;
  public
    constructor Create (aWidth, aHeight : integer);
    destructor Destroy; override;
    procedure Open;
    procedure Close;
  published
    property Canvas : TCanvas read FCanvas write FCanvas;
  end;

implementation

{ create our drawing metafile, take the drawing width and height to those }
{ given in the parameters, set the inch property to that found in         }
{ screens.pixelsperinch and finally open for drawing                      }

constructor TDrawMetafile.Create (aWidth, aHeight : integer);
begin
  inherited Create;
  Inch := Screen.PixelsPerInch;
  Width := aWidth;
  Height := aHeight;
  Open;
end;

{ before destroying the metafile ensure that it is closed, this means     }
{ that any canvas is destroyed                                            }

destructor TDrawMetafile.Destroy;
begin
  Close;
  inherited Destroy;
end;

{ The key method: puts the metafile into draw and use modes depending  on }
{ the boolean parameter. When true the metafile goes into draw mode by    }
{ putting a CreateMetafile device context into a Canvas handle property   }
{ the canvas can now be drawn on. When false the metafile goes into use   }
{ mode by putting the handle returned from the CloseMetafile call into    }
{ the metafile's handle property. Note that the width, height and inch    }
{ properties are preserved over this assignment. Finally the canvas is    }
{ freed.                                                                  }
```

```
procedure TDrawMetafile.SetDrawing (State : boolean);
var
  KeepInch,
  KeepWidth,
  KeepHeight : integer;
begin
  if State <> FDrawing then
    begin
      FDrawing := State;
      if Drawing then
        begin
          FCanvas := TCanvas.Create;
          FCanvas.Handle := CreateMetafile(nil);
        end
      else
        begin
          KeepWidth := Width;
          KeepHeight := Height;
          KeepInch := Inch;
          Handle := CloseMetafile(FCanvas.Handle);
          Width := KeepWidth;
          Height := KeepHeight;
          Inch := KeepInch;
          FCanvas.Free;
        end;
    end;
end;

procedure TDrawMetafile.Open;
begin
  Drawing := true
end;

procedure TDrawMetafile.Close;
begin
  Drawing := false
end;

end.
```

Return to MetaFile Article
Return to Front Page

```
unit LedLight;
(*
   TLedLight by Lars Posthuma; December 26, 1995.
   Copyright 1995, Lars Posthuma.
   All rights reserved.

   This source code may be freely distributed and used. The author
   accepts no responsibility for its use or misuse.
   No warranties whatsoever are offered for this unit.

   If you make any changes to this source code please inform me at:
   LPosthuma@COL.IB.COM.
*)



{$R LedLight.res}

interface

uses
  WinTypes, WinProcs, Classes, Graphics, Controls, ExtCtrls, Forms, SysUtils;

Const
  MinLedSize = 8;
  MaxLedSize = 30;

type
  String5 = String[5];

  TLedLightOrientation = (LedHorizontal,LedVertical);
  TLedLightState       = (LedOff,LedOn);

  TLedLight = class(TCustomPanel)
    private
      { Private declarations }
      fBorderWidth : Integer;
      fOrientation : TLedLightOrientation;
      fState       : TLedLightState;
      fLedSize     : Integer;
      fLedSpacing  : Integer;
      fHeight      : Integer;
      fWidth       : Integer;
      fValueChange : TNotifyEvent;
      fCtl3D       : Boolean;
      procedure PaintLeds(Led1,Led2: String5);
      procedure SetBorderWidth   (value: Integer); virtual;
      procedure SetBounds(Left,Top,fWidth,fHeight: integer); override;
      procedure SetCtl3D(value : boolean); virtual;
      procedure SetHeight(value: Integer); virtual;
      procedure SetOrientation(value: TLedLightOrientation);
      procedure SetLedSize(value: Integer);
      procedure SetLedSpacing(value : Integer);
      procedure SetState(value: TLedLightState); virtual;
      procedure SetWidth(value: Integer); virtual;
    protected
      { Protected declarations }
      procedure Paint; override;
    public
      { Public declarations }
      constructor Create(AOwner: TComponent); override;
      destructor Destroy; override;
```

```
      procedure ChangeState; virtual;
    published
      { Published declarations }
      property Align;
      property BevelInner;
      property BevelOuter;
      property BevelWidth;
      property Cursor;
      property Enabled;
      property Color default clBtnFace;
      property Height: Integer                              {Height of Outer
Limits}
               read fHeight write SetHeight
               default 17;
      property Width : Integer                              {Width of Outer
Limits}
               read fWidth write SetWidth
               default 36;
      property Orientation: TLedLightOrientation            {}
               read fOrientation write SetOrientation
               default LedHorizontal;
      property LedSize: Integer                             {True Size of Leds}
               read fLedSize write SetLedSize
               default 13;
      property LedSpacing: Integer                          {Space between the
Leds}
               read fLedSpacing write SetLedSpacing
               default 6;
      property State: TLedLightState                        {Indicates Leds On or
Off}
               read fState write SetState
               default LedOff;
      property BorderStyle;
      property BorderWidth: Integer
               read fBorderWidth write SetBorderWidth
               default 1;
      property Ctl3D: Boolean
               read fCtl3D write SetCtl3D
               default False;
      property OnValueChange: TNotifyEvent                  {Userdefined Method}
               read fValueChange write fValueChange;
      property Visible;
      property Hint;
      property ParentShowHint;
      property ShowHint;
      property Tag;

      property OnClick;
      property OnDragDrop;
      property OnDragOver;
      property OnEndDrag;
      property OnMouseDown;
      property OnMouseMove;
      property OnMouseUp;
  end;

procedure Register;

implementation

constructor TLedLight.Create(AOwner: TComponent);
begin
 inherited Create(AOwner);
```

```
 Color       := clBtnFace;                          {Set initial (default) values}
 fLedSize    := 13;
 fLedSpacing := 6;
 fHeight     := 17;
 fWidth      := 36;
 fBorderWidth:= 1;
 fOrientation:= LedHorizontal;
 fState      := LedOff;
 fCtl3D      := False;
 BevelInner  := bvNone;
 BevelOuter  := bvNone;
 Caption     := ' ';
end;

destructor TLedLight.Destroy;
begin
 inherited Destroy
end;

procedure TLedLight.SetHeight(value: integer);
begin
 if value <> fHeight then begin
   fHeight:= value;
   if fHeight < MinLedSize + 4
     then fHeight:= MinLedSize + 4;
   if fHeight > MaxLedSize + 4
     then fHeight:= MaxLedSize + 4;

   fLedSize:= fHeight - 4;
   SetBounds(Left,Top,fWidth,fHeight);
   Invalidate;
 end
end;

procedure TLedLight.SetWidth(value: integer);
Var
 MyWidth: Integer;
begin
 if value <> fWidth then begin
   fWidth:= value;
   MyWidth:= ((MinLedSize + 2) * 2) + LedSpacing;
   if fWidth < MyWidth
     then fWidth:= MyWidth;
   MyWidth:= ((MaxLedSize + 2) * 2) + LedSpacing;
   if fWidth > MyWidth
     then fWidth:= MyWidth;

   fLedSize:= (fWidth - 4 - LedSpacing) div 2;
   SetBounds(Left,Top,fWidth,fHeight);
   Invalidate;
 end
end;

procedure TLedLight.SetBounds(Left,Top,fWidth,fHeight : integer);
begin
 case fOrientation of
   LedHorizontal: begin
                    fWidth := (fLedSize + 2) * 2 + fLedSpacing;
                    fHeight:= fLedSize + 4;
                  end;
   LedVertical  : begin
                    fWidth := fLedSize + 4;
                    fHeight:= (fLedSize + 2) * 2 + fLedSpacing;
```

```
                  end;
 end;
 inherited SetBounds(Left,Top,fWidth,fHeight);
end;

procedure TLedLight.SetLedSize(value : Integer);
begin
 if value <> fLedSize then begin
   fLedSize:= value;
   if fLedSize < MinLedSize
     then fLedSize:= MinLedSize;
   if fLedSize > MaxLedSize
     then fLedSize:= MaxLedSize;
   if LedSpacing > fLedSize
     then LedSpacing:= fLedSize;

   SetBounds(Left,Top,fWidth,fHeight);
   Invalidate;
 end
end;

procedure TLedLight.SetLedSpacing(value : Integer);
begin
 if value <> fLedSpacing then begin
   fLedSpacing:= value;
   if fLedSpacing < 2                                    {Valid Range
[2..fLedSize}
     then fLedSpacing:= 2;
   if fLedSpacing > LedSize
     then fLedSpacing:= LedSize;
   if (fLedSpacing mod 2) <> 0                           {Only Even values
allowed}
     then Inc(fLedSpacing);

   SetBounds(Left,Top,fWidth,fHeight);                   { testen zonder f.. }
   Invalidate;
 end
end;

procedure TLedLight.SetOrientation(value : TLedLightOrientation);
begin
 if value <> fOrientation then begin
   fOrientation:= value;
   SetBounds(Left,Top,Height,Width);                     {Swap Width/Height}
   Invalidate;
 end
end;

procedure TLedLight.SetBorderWidth(value : Integer);
begin
 if value <> fBorderWidth then begin
   fBorderWidth:= value;
   if fBorderWidth < 0                                   {Valid Range: [0..2]}
     then fBorderWidth:= 1;
   if fBorderWidth > 2
     then fBorderWidth:= 2;
   Invalidate;
 end
end;

procedure TLedLight.SetState(value : TLedLightState);
begin
 if value <> fState then begin
```

```
   fState:= value;
   if Assigned(fValueChange)                                    {Call Userdefined
Method}
     then OnValueChange(Self);
   Invalidate;
 end
end;

procedure TLedLight.SetCtl3D (value : boolean);
begin
 if value <> fCtl3D then begin
   fCtl3D:= value;
   Invalidate;
 end
end;

procedure TLedLight.ChangeState;
begin
 if fState = LedOff
   then fState:= LedOn
   else fState:= LedOff;
 if Assigned(fValueChange)                                    {Call Userdefined
Method}
   then OnValueChange(Self);
 Invalidate;
end;

procedure TLedLight.Paint;
begin
 inherited Paint;
 with Canvas do begin
   Brush.Color:= Color;
   with ClientRect do begin
     if Ctl3D = True then begin                               {Give 3d look}
       Pen.Color:= clBtnShadow;
       MoveTo(Left + 1, Bottom - 3);
       LineTo(Left + 1, Top + 1);
       LineTo(Right - 2, Top + 1);
       MoveTo(Left, Bottom - 1);
       LineTo(Right-1, Bottom - 1);
       LineTo(Right-1, Top - 1)
     end;
     Case fState of
       LedOff: PaintLeds('Green','Gray');
       LedOn : PaintLeds('Gray','Red');
     end;
   end;
 end;
end;

procedure TLedLight.PaintLeds(Led1,Led2: String5);
var
 BMP1, BMP2  : TBitmap;
 TheLed      : Array[0..5] of Char;
 MyOffs      : Integer;
 DRect, SRect: TRect;
begin
 (* Create the Leds from Resource file.
    Dynamic creation by means of Old-fashioned API call *)
 StrPCopy(TheLed,Led1);
 BMP1       := TBitmap.Create;
 BMP1.Handle:= LoadBitmap(hInstance, TheLed);
```

```
 StrPCopy(TheLed,Led2);
 BMP2       := TBitmap.Create;
 BMP2.Handle:= LoadBitmap(hInstance, TheLed);
 Try
   (* If BorderStyle is bsSingle the leds are somewhat shifted; to fix this
      "problem" paint the leds at a somewhat smaller offset *)
   MyOffs:= 2;
   if BorderStyle = bsSingle
     then Dec(MyOffs);

   (* Create Source Rectangle; used for both Leds *)
   SRect:= Rect(0,0,13,13);
   MyOffs:= 2;
   if BorderStyle = bsSingle
     then Dec(MyOffs);

   (* Create Destination Rectangle; used for drawing first Led *)
   DRect:= Rect(MyOffs,MyOffs,(MyOffs + LedSize),(MyOffs + LedSize));

   (* Draw first Led *)
   Canvas.BrushCopy(DRect,BMP1,SRect,clBtnFace);

   (* Create Destination Rectangle; used for drawing Second Led *)
   case fOrientation of
     LedHorizontal: DRect:= Rect((Width - 2 - LedSize),MyOffs,(Width - 2),(MyOffs +
LedSize));
     LedVertical  : DRect:= Rect(MyOffs,(MyOffs + LedSize + LedSpacing),(2 +
LedSize),(2 + (2 * LedSize) + LedSpacing));
   end;

   (* Draw second Led *)
   Canvas.BrushCopy(DRect,BMP2,SRect,clBtnFace);
   Finally
     BMP1.Free;
     BMP2.Free;
 end;
end;

procedure Register;
begin
 RegisterComponents('Lars', [TLedLight]);
end;

end.
```

Return to The Component Cookbook
Return to Front Page

```
Unit Strings;

(*
   TStrings by Lars Posthuma; December 26, 1995.
   Copyright 1995, Lars Posthuma.
   All rights reserved.

   Some of these functions have become obsolete due to the extended
   functionality of some Delphi Units but since this is a plain conversion
   from a former BP7 unit to Delphi I didn't bother to remove certain
   duplicate functions. This code has not yet been cleaned and/or
   optimized. To Clipper Programmers some of these functions will look very
   familiar!

   This source code may be freely distributed and used. The author
   accepts no responsibility for its use or misuse.
   No warranties whatsoever are offered for this unit.

   If you make any changes to this source code please inform me at:
   LPosthuma@COL.IB.COM.
*)

Interface

Uses
 SysUtils, Classes;

type
  TStrings = class(TComponent)
    private
    protected

    public
      Function AT(InStr: String; SearchStr: String): Byte;
      Function FILL(InStr: String; Pos, Len: Byte; Ch: Char): String;
      Function INTtoSTR(InVal: LongInt): String;
      Function LEFT(InStr: String; Len: Byte): String;
      Function LOWER(InStr: String): String;
      Function PADC(InStr: String; Len: Byte; Ch: Char): String;
      Function PADL(InStr: String; Len: Byte; Ch: Char): String;
      Function PADR(InStr: String; Len: Byte; Ch: Char): String;
      Function RAT(InStr: String; SearchStr: String): Byte;
      Function REALtoSTR(InVal: Real; Decimals: Byte): String;
      Function REPLICATE(RepliStr: String; NoTimes: Byte): String;
      Function RIGHT(InStr: String; Len: Byte): String;
      Function SPACE(Len: Byte): String;
      procedure SplitString(const InStr: String; SplitAt: Char; var
LeftStr,RightStr: String);
      Function STRtoINT(InStr: String): Integer;
      Function STRtoLONGINT(InStr: String): LongInt;
      Function STRtoREAL(InStr: String): Real;
      Function STRTRAN(InStr: String; SearchStr, ReplaceStr: String; MatchCase:
Boolean;
                       Start, Count: Byte): String;
      Function STUFF(InStr: String; Pos, Len: Byte; StuffStr: String): String;
      Function SUBSTR(InStr: String; Pos, Len: Byte): String;
      Function TrailingChar(const Value: String; Trailer: Char): String;
      Function TrailingBackSlash(const Value: String): String;
      Function TRIMALL(InStr: String): String;
      Function TRIML(InStr: String): String;
      Function TRIMR(InStr: String): String;
```

```
      Function UPPER(InStr: String): String;
      Function UPPEREVERYFIRST(InStr: String): String;
    published
  end;

procedure Register;

(*
Function AT(InStr: String; SearchStr: String): Byte;
{ Returns the position of the first occurrence found for SearchStr in InStr;
  returns 0 if SearchStr has not been found. }

Function FILL(InStr: String; Pos, Len: Byte; Ch: Char): String;
{ Returns a string in which from a position (Pos) for a length of (Len)
  characters have been replaced by (Ch). }

Function INTtoSTR(InVal: LongInt): String;
{ Converts an integer (InVal) of type Byte, Word, ShortInt, Integer or
  LongInt to its String equivalent. }

Function LEFT(InStr: String; Len: Byte): String;
{ Returns the leftmost (len) characters of (InStr.) }

Function LOWER(InStr: String): String;
{ Returns lowercase characters for (InStr). }

Function PADC(InStr: String; Len: Byte; Ch: Char): String;
{ Returns a string of Length (len) in which InStr was left and right padded
  with character (Ch). }

Function PADL(InStr: String; Len: Byte; Ch: Char): String;
{ Returns a string of Length (len) in which InStr was left padded with
  character (Ch). }

Function PADR(InStr: String; Len: Byte; Ch: Char): String;
{ Returns a string of Length (len) in which InStr was right padded with
  character (Ch). }

Function RAT(InStr: String; SearchStr: String): Byte;
{ Returns the position of the right most found occurrence for SearchStr in
  InStr; returns 0 if SearchStr has not been found. }

Function REPLICATE(RepliStr: String; NoTimes: Byte): String;
{ Return a string containing a repetition of Number of times (NoTimes)
  of (RepliStr). }

Function RIGHT(InStr: String; Len: Byte): String;
{ Returns the rightmost (Len) characters of (Instr). }

Function SPACE(Len: Byte): String;
{ Returns a string containing (Len) Spaces. }

procedure SplitString(const InStr: String; SplitAt: Char; var LeftStr,RightStr:
String);
{ splits InStr at char SplitAt and returns two strings LeftStr and RightStr}

Function STRTRAN(InStr: String; SearchStr, ReplaceStr: String; MatchCase: Boolean;
                 Start, Count: Byte): String;
{ Returns a string in which (SearchStr) has been replaced by (ReplaceStr)
  from occurence number (start) a maximum of (Count) times. }

Function STRtoINT(InStr: String): Integer;
```

```
{ Converts a string containing a value to its Integer equivalent. }

Function STRtoLONGINT(InStr: String): LongInt;
{ Converts a string containing a value to its LongInt equivalent. }

Function STRtoREAL(InStr: String): Real;
{ Converts a string containing a value to its Real equivalent. }

Function STUFF(InStr: String; Pos, Len: Byte; StuffStr: String): String;
{ Returns a string in which at Position (Pos) for a length of (Len)
  characters have been replaced by (StuffStr). }

Function SUBSTR(InStr: String; Pos, Len: Byte): String;
{ Returns a string which is a subset of (InStr) starting at Position (Pos)
  for a  length of (Len). }

Function TrailingChar(const Value: String; Trailer: Char): String;
{ Returns a string which has Trailer as a trailing Char the character. }

Function TrailingBackSlash(const Value: String): String;
{ Returns a string which has a BackSlash as the trailing character. }

Function TRIMALL(InStr: String): String;
{ Returns a string of which all leading and trailing spaces have been
  removed. }

Function TRIML(InStr: String): String;
{ Returns a string of which all leading spaces have been removed. }

Function TRIMR(InStr: String): String;
{ Returns a string of which all trailing spaces have been removed. }

Function UPPER(InStr: String): String;
{ Returns Uppercase characters for (InStr). }

Function UPPEREVERYFIRST(InStr: String): String;
{ Returns Uppercase character for every first character of each word
  of (InStr). }

Function REALtoSTR(InVal: Real; Decimals: Byte): String;
{ Converts a Real value (InVal) to its String equivalent. }
*)

Implementation

Function TStrings.AT(InStr: String; SearchStr: String): Byte;
Begin
 AT:= Pos(SearchStr,InStr);
End;


Function TStrings.FILL(InStr: String; Pos, Len: Byte; Ch: Char): String;
Begin
 Result:= Stuff(InStr,Pos,Len,Replicate(Ch,Len));
End;


Function TStrings.INTtoSTR(InVal: LongInt): String;
Var
 TempStr : String;
Begin
  STR(InVal,TempStr);
  INTtoSTR:= TempStr;
```

```
End;


Function TStrings.LEFT(InStr: String; Len: Byte): String;
Begin
  LEFT:= Copy(InStr,1,Len);
End;


Function TStrings.LOWER(InStr: String): String;
Var
 i  : Byte;
Begin
  for i:= 1 to Length(InStr) do begin
    if InStr[i] in ['A'..'Z']
      then InStr[i]:= CHR(ORD(InStr[i]) + 32);
  end;
  LOWER:= InStr;
End;


Function TStrings.PADC(InStr: String; Len: Byte; Ch: Char): String;
Var
 NoPAD  : Integer;
 TempStr: string;
Begin
  FillChar(TempStr[1],Len,Ch);
  TempStr[0]:= CHR(Len);
  NoPad    := Length(InStr);
  If NoPAD <= Len
    then Move(InStr[1],TempStr[((Len - NoPAD) DIV 2) + 1],NoPAD)
    else Move(InStr[((NoPAD - Len) DIV 2) + 1],TempStr[1],Len);
  PADC:= TempStr;
End;


Function TStrings.PADL(InStr: String; Len: Byte; Ch: Char): String;
Var
 TempStr  : String;
 LenInStr : Byte;
Begin
  FillChar(TempStr[1],Len,Ch);
  TempStr[0]:= CHR(Len);
  LenInStr  := Length(InStr);
  If LenInStr <= Len
    then Move(InStr[1],TempStr[Succ(Len - LenInStr)],LenInStr)
    else Move(InStr[1],TempStr[1],Len);
  PADL:= TempStr;
End;


Function TStrings.PADR(InStr: String; Len: Byte; Ch: Char): String;
var
  TempStr  : String;
begin
  FillChar(TempStr[1],Len,Ch);
  TempStr[0]:= CHR(Len);
  if Length(InStr) <= Len
    then Move(InStr[1],TempStr[1],Length(InStr))
    else Move(InStr[1],TempStr[1],Len);
  PADR:= TempStr;
end;
```

```
Function TStrings.RAT(InStr: String; SearchStr: String): Byte;
Var
 Quit    : Boolean;
 Pos,Len : Integer;
Begin
 RAT := 0;
 Quit:= False;
 Len := Length(SearchStr);
 Pos := Length(InStr) - Len;
 while Quit = False do begin
   if Copy(InStr,Pos,Len) = SearchStr then begin
     RAT := Pos;
     Quit:= True;
   end;
   if Pos = 1 then begin
     Quit:= True;
   end;
   Dec(Pos,1);
 end;
End;


Function TStrings.REPLICATE(RepliStr: String; NoTimes: Byte): String;
Var
 i      : Byte;
 TempStr: String;
Begin
  TempStr:='';
  For i:= 1 To NoTimes Do TempStr:= TempStr + RepliStr;
  REPLICATE:= TempStr;
End;


Function TStrings.RIGHT(InStr: String; Len: Byte): String;
Begin
  RIGHT:= Copy(InStr,(Length(InStr) - Len),Len);
End;


Function TStrings.SPACE(Len: Byte): String;
Var
 TempStr : String;
Begin
 Fillchar(TempStr[1],Len,' ');
 TempStr[0]:= CHR(Len);
 SPACE     := TempStr;
End;


procedure TStrings.SplitString(const InStr: String; SplitAt: Char; var
LeftStr,RightStr: String);
var
 n: Integer;
begin
 n:= Pos(SplitAt,InStr);
 if n = 0 then begin
   LeftStr := InStr;
   RightStr:= '';
 end
 else begin
   LeftStr := Copy(InStr,1,(n - 1));
   RightStr:= Copy(InStr,(n + 1),Length(InStr) - n);
```

```
 end;
end;


Function TStrings.STRtoINT(InStr: String): Integer;
Var
 Temp,Code : integer;
Begin
  STRtoINT := 0;
  Val(InStr,Temp,Code);
  if Code = 0
    then STRtoINT:= Temp;
End;


Function TStrings.STRtoLONGINT(InStr: String): LongInt;
Var
 Temp,Code : integer;
Begin
  STRtoLONGINT := 0;
  Val(InStr,Temp,Code);
  if Code = 0
    then STRtoLONGINT:= Temp;
End;


Function TStrings.STRtoREAL(InStr: String): Real;
var
  Code : Integer;
  Temp : Real;
begin
  STRtoREAL:= 0;
  If Copy(InStr,1,1)='.'
    then InStr:= '0' + InStr;
  If (Copy(InStr,1,1)='-') and (Copy(InStr,2,1)='.')
    then Insert('0',InStr,2);
  If InStr[length(InStr)] = '.'
    then Delete(InStr,length(InStr),1);
  Val(InStr,Temp,Code);
  if Code = 0
    then STRtoREAL:= Temp;
end;


Function TStrings.STRTRAN(InStr: String; SearchStr, ReplaceStr: String;
                MatchCase: Boolean; Start, Count: Byte): String;
Var
 i,Pos,LenS,LenR : Integer;
 TempStr         : String;
Begin
 TempStr:= InStr;
 if MatchCase = False then begin
   TempStr   := UPPER(TempStr);
   SearchStr := UPPER(SearchStr);
   ReplaceStr:= UPPER(ReplaceStr);
 end;
 LenS   := Length(SearchStr);
 LenR   := Length(ReplaceStr);
 for i:= 1 to (Start - 1) do begin
   Pos    := AT(TempStr,SearchStr);
   TempStr:= STUFF(TempStr,Pos,LenS,PADR('þ',LenS,'þ'));
 end;
 i  := 1;
```

```
 Pos:= AT(TempStr,SearchStr);
 while (Pos <> 0) AND (i < Count) do begin
   InStr  := STUFF(InStr,Pos,LenS,ReplaceStr);
   TempStr:= STUFF(TempStr,Pos,LenS,PADR('þ',LenR,'þ'));
   Pos    := AT(TempStr,SearchStr);
   Inc(i,1);
 end;
End;


Function TStrings.STUFF(InStr: String; Pos, Len: Byte; StuffStr: String): String;
Begin
 Delete(InStr,Pos,Len);
 Insert(StuffStr,InStr,Pos);
 STUFF:= InStr;
End;


Function TStrings.SUBSTR(InStr: String; Pos,Len: Byte): String;
Begin
  SUBSTR:= Copy(InStr,Pos,Len);
End;


Function TStrings.TrailingChar(const Value: String; Trailer:Char): String;
begin
 Result:=Value;
 if Copy(Value,Length(Value),1) <> Trailer
   then Result:= Result + Trailer;
end;


Function TStrings.TrailingBackSlash(const Value: String): String;
begin
 Result:=Value;
 if Copy(Value,Length(Value),1) <> '\'
   then Result:= Value + '\';
end;


Function TStrings.TRIMALL(InStr: String): String;
Begin
  TRIMALL:= TRIML(TRIMR(InStr));
End;


Function TStrings.TRIML(InStr: String): String;
Var
 Pos: Integer;
Begin
  Pos:= 1;
  While InStr[Pos] = ' ' do begin
    Inc(Pos,1);
  end;
  TRIML:= SUBSTR(InStr,Pos,(Length(InStr) - Pos + 1));
End;


Function TStrings.TRIMR(InStr: String): String;
Var
 Pos: Integer;
Begin
  Pos:= Length(InStr);
```

```
  While InStr[Pos] = ' ' do begin
    Dec(Pos,1);
  end;
  InStr[0]:= CHR(Pos);
  TRIMR   := InStr;
End;


Function TStrings.UPPER(InStr: String): String;
Var
 i : Integer;
Begin
  for i:= 1 to Length(InStr) do InStr[i]:= UpCase(InStr[i]);
  UPPER:= InStr;
End;


Function TStrings.UPPEREVERYFIRST(InStr: String): String;
Var
 Pos     : Integer;
 TempStr : String;
Begin
  TempStr   := TRIMR(InStr);
  TempStr[1]:= UpCase(TempStr[1]);

  Pos:= RAT(TempStr,' ');
  While Pos <> 0 do begin
    TempStr   := STUFF(TempStr,Pos,1,'þ');
    Inc(Pos,1);
    InStr[Pos]:= UpCase(InStr[Pos]);
    Pos       := RAT(TempStr,' ');
  end;
  UPPEREVERYFIRST:= InStr;
End;


Function TStrings.REALtoSTR(InVal : Real; Decimals : Byte): String;
Var
 Width   : Byte;
 IntPart : Real;
 TempStr : String;
Begin
 IntPart := Int(InVal);
 STR(Int(InVal):40:0,TempStr);
 Width:= Length(TRIML(TempStr));

 STR(InVal:Width:Decimals,TempStr);
 REALtoSTR:= TempStr;
End;


procedure Register;
begin
  RegisterComponents('System', [TStrings]);
end;

END.
```

Return to The Component Cookbook
Return to Front Page